Cost-Effective Large-Scale Digital Twins Notification System with Prioritization Consideration

by

Mira Vrbaski

Thesis submitted to the University of Ottawa
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Electrical and Computer Engineering

Examining Committee

The following served on the Examining Committee for this thesis.

External Member: Omar Abdul Wahab
Professor, Department of Computer Engineering and Software Engineering
Polytechnique Montreal

Carleton Member: Sreeraman Rajan
Professor, Department of Systems and Computer Engineering
Carleton University

Internal Member(s): Amiya Nayak
Professor, School of Electrical Engineering& Computer Science (SEECS)
University of Ottawa

Nancy Samaan
Professor, School of Electrical Engineering & Computer Science(SEECS)
University of Ottawa

Supervisor(s): Miodrag Bolic
Professor, School of Electrical Engineering & Computer Science (SEECS)
University of Ottawa

Shikharesh Majumdar
Professor, Department Systems and Computer Engineering
Carleton University

Declaration of Authorship

I hereby certify that this thesis is entirely my own original work except where otherwise indicated. I am aware of the University of Ottawa regulations concerning plagiarism, including those regarding consequent disciplinary actions. Any use of the works of any other author, in any form, is properly acknowledged at their point of use.

Abstract

Large-Scale Digital Twins Notification System (LSDTNS) monitors a Digital Twin (DT) cluster for a predefined critical state, and once it detects such a state, it sends a Notification Event (NE) to a predefined recipient. Additionally, the time from producing the DT's Complex Event (CE) to sending an alarm has to be less than a predefined deadline. However, addressing scalability and multi-objectives, such as deployment cost, resource utilization, and meeting the deadline, on top of process scheduling, presents a complex challenge. Therefore, this thesis presents a complex methodology consisting of three contributions that address system scalability, multi-objectivity and scheduling of CE processes using Reinforcement Learning (RL).

The first contribution proposes the IoT Notification System Architecture based on a micro-service-based notification methodology that allows for running and seamlessly switching between various CE reasoning algorithms. Our proposed IoT Notification System architecture addresses the scalability issue in state-of-the-art CE Recognition systems.

The second contribution proposes a novel methodology for multi-objective optimization for cloud provisioning (MOOP). MOOP is the first work dealing with multi-optimization objectives for microservice notification applications, where the notification load is variable and depends on the results of previous microservices subtasks. MOOP provides a multi-objective mathematical cloud resource deployment model and demonstrates effectiveness through the case study.

Finally, the thesis presents a Scheduler for large-scale Critical Notification applications based on a Deep Reinforcement Learning (SCN-DRL) scheduling approach for LSDTNS using RL. SCN-DRL is the first work dealing with multi-objective optimization for critical microservice notification applications using RL. During the performance evaluation, SCN-DRL demonstrates better performance than state-of-the-art heuristics. SCN-DRL shows steady performance when the notification workload increases from 10% to 90%. In addition, SCN-DRL, tested with three neural networks, shows that it is resilient to sudden container resources drop by 10%. Such resilience to resource container failures is an important attribute of a distributed system.

Acknowledgements

Completing this Ph.D. journey while managing the responsibilities of work and motherhood has been an incredible challenge, and I am grateful to have had the support and encouragement of so many people along the way.

First and foremost, I want to express my deepest gratitude to my family. To my dear children, **Anna** and **Marko Vrbaski**: your patience and understanding during late nights of studying and research are a testament to your remarkable maturity. You are my greatest motivation. To my husband, **Srdjan Vrbaski**: your unwavering belief in me and your constant support have been my anchor through this demanding journey. Thank you for stepping in whenever I needed time to focus on my work. To my father Professor **Zivota Sesic**, who was my greatest inspiration to persist, and to my mother **Jelena Sesic** for her constant love and support throughout my whole life.

I am immensely thankful to my supervisors, Professor **Miodrag Bolic** and Professor **Shikharesh Majumdar**, for not only guiding me through the academic intricacies but also for understanding and accommodating the challenges of a working mother. Your mentorship and flexibility have made this pursuit possible.

I would like to express my heartfelt gratitude to the members of my advisory committee: Professors **Nancy Samaan**, **Sreeraman Rajan**, **Amiya Nayak** and **Omar Abdul Wahab**, for their insightful feedback, constructive criticism, and expert guidance that significantly enriched the quality of this work.

I am thankful, to my colleagues at **Nokia** and **MMIST**, your camaraderie and understanding when I had to balance work commitments and research are greatly appreciated. Your collaborative spirit made my dual responsibilities manageable.

I am deeply grateful to **Sue Ackerman** (R&D Director at Nokia), **Dave Senior** (Senior Director and Global Product Development Leader at Nokia), and **Peter Harrison** (Network Analytics practice Head, Service Portfolio Sales, Network Cognitive Services at Nokia) for recognizing the importance of continued learning and for allowing me the flexibility to balance my work commitments with my academic pursuits. Your support not only facilitated my research but also fostered an environment where growth is encouraged and valued.

I would also like to extend my appreciation to the entire leadership team at Nokia for fostering a culture that values personal and professional development. Your understanding of the challenges faced by individuals pursuing advanced education while working has been invaluable.

I am also thankful for the support and camaraderie of my colleagues at Nokia. Your teamwork and encouragement have made the intricate dance of work and academia more manageable and enjoyable.

To my friends who have stood by me through the ups and downs, offering words of encouragement and a listening ear, your presence has been a source of strength.

I am indebted to the participants of my study, who shared their time and experiences, making my research meaningful and insightful.

Lastly, to every working mother who understands the juggling act of career, education, and family, you inspire me. Your strength and determination are a reminder that challenges can be overcome with perseverance and a strong support system.

This thesis is not just a culmination of academic effort; it represents a chapter of my life where I proved to myself that dreams can be realized against all odds.

Thank you All.

Mira Vrbaski
August 2023

Table of Contents

List of Tables

List of Figures

Acronyms

ANN - Artificial Neural Network
AI - Artificial Intelligence
API - Application Program Interface
AWS - Amazon Web Services
BC - Behaviour Cloning
BHGE - Baker Hughes from General Electronics
BN - Bayesina Networks
CE - Complex Event
CEC - Complex Event Criteria
CER - Complex Event Reasoning
CEP - Complex Event Processing
CES - Complex Event System
CN - Communication
CNS - Critical Notification System
CI - Computational Intelligence
CPU - Central Processing Unit
D - Deadline
DBN - Dynamics Bayesian Networks
DC-MC-RTS - Docker containerized mixed-critically real-time-system
DD - Digital Twin Data
DL - Deep Learning
DRL - Deep Reinforcement Learning
DNN - Deep Neural Network DT - Digital Twin
DTPCA - Digital Twin Priority Controller Algorithm
DTRAS - Digital Twin Resource Allocation Scheduler
E - Event
ES - Event Stream
FDS - Flexible Defferable System
HCA - Hierarchical Cluster Analysis
HL - Harware Level
IL - Intuitive Learning
IoT - Internet of Things
IP - Internet Protocol
LLA - Local Linear Embedding
LJF - Long Job First
LSDTNS - Cost-Effective Large-Scale Digital Twins Notification System

LTU - Linear Threshold Unit
FCFS - First Come, First Served
JBD - Job Before Deadline
MAP - Map Posterior
ME - Monitoring Entity
MO - Monitoring Object
ML - Machine Learning
MLE - Maximum like-hood estimation
MLP - Multi-Layer Perception
MOO - Multi-Objective Optimization MOOP - Multi-Objective Optimization for Cloud Provisioning
NE - Notification Event
NJBD - Notification Job Before Deadline
NFC - Near field communication
NN- Neural Network
NN1 - Neural Network 1; a small compact neural network
NN2 - Neural Network 2; a compact neural network
NN3 - Neural Network 3; a small convolution neural network
NMS - Network monitoring System
NLP - Natural language processing
NRT- Non Real Time
OSL - Operating System Level
PCA - Principal Component Analysis
PG - Policy Gradient
PE - Physical Entity
PMO - Physical Monitoring Object
PP - Probabilistic Programming
RAM - Random Access Memory
RDD - Resilient Distributed Dataset
RL - Reinforcement Learning
RFID - Radio frequency identification
REST - Representational State Transfer
REST API - REST Application Program Interface
RESTful API an interface that two computer systems use to exchange information securely over the internet
RPM - Remote Patient Monitoring
RT - Real-Time
RQ - Research Question

SJF - Small Job First
SL - Supervised Learning
SLO - Service Level Objective
SNC-DRL - Large-Scale Digital Twins Notification System
SQL - Structured Query Language
t-SBE - t-distributed Stochastic Neighbour Embedding
VM - Virtual Machine
VE - Virtual Entity
WIFI-Wireless Fidelity
UID - Unique Identifier
USL - Unsupervised Learning
URL - A Uniform Resource Locator

Chapter 1

Introduction

1.1 Overview

As the availability and affordability of sensors continue to increase, we are witnessing an enormous expansion of sensor networks deployed for various applications that include smart healthcare solutions, smart homes, environmental monitoring, and smart cities [15]. Deployed sensors across the Internet are essential components of the *Internet of Things* (IoT) systems. IoT systems ranging from machine-type communications to mission-critical communications (such as autonomous driving cars and remote surgery) pose unprecedented challenges in capacity, latency, reliability, and scalability [16].

One of the main aspects of IoT Notification systems (IoT NS) is to monitor the external environment and send a notification when a specific, observed complex event occurs in the environment. The Complex Event (CE) combines observed multiple primitive events, acquired from IoT systems sensor data, over time. IoT NS operates by observing a set of primitive events in external environments, interpreting and combining them to identify a higher level of CEs [1]. An essential part of the system under observation is to recognize CEs that match a given set of Complex Event Criteria (CEC). However, those primitive events are uncertain by their nature and can create CEs that are even more uncertain. Therefore, if the probability of the CE is not taken into consideration, the reliability of the IoT Notification system is at significant risk.

The IoT Notification system sensors observe objects in a limited area. In this dissertation, the monitored area that contains a set of sensors is called the *Monitoring Entity* (ME) and the observed object is the *Monitored Object* (MO). For example, a prison often has

hundreds of prisoner cells. A similar set of sensors could be deployed in each prisoner cell to monitor one prisoner. Each prisoner represents a , and a prisoner cell represents a ME. In prison, the IoT Notification system monitors several monitoring objects, which requires the system to scale and process a large volume of critical data.

Furthermore, the Monitoring Entity definition presented above is similar to IBM's definition of a *Digital Twin* (DT). A digital twin is a virtual representation of a physical object or system across its lifecycle, using real-time data to enable understanding, learning, and reasoning [18]. Therefore, further in this document, we will use Digital Twin instead of ME; ME is used in our previous published work.

Bigdata complements IoT systems [43]. More precisely, IoT systems produce an enormous amount of data; IoT Systems generated data are Bigdata.

1.2 Research Challenges and Motivations

1.2.1 Research Challenges

The Cost-Effective Large-Scale Digital Twins Notification System (LSDTNS) with prioritization consideration is a complex multi-domain problem that requires a holistic approach for problem analysis and solution. The current state-of-the-art is only looking into problems from the single-domain point of view. In the following text, we will explain each domain that contributes to the whole research challenge and opens tremendous research opportunities while presenting many challenges.

Large-Scale Digital Twins Notification System (LSDTNS) definition:
LSDTNS is a critical monitoring system that must monitor n Digital Twins, where n can be a large number, order of a few hundred. Each Digital Twin (DT) has m IoT devices connected through the network. IoT devices generate primitive Events (E) and send them in specified time intervals over the network, which generates a sequential array of events known as an Event Stream (ES). The LSDTNS systems have a notification *deadline (D)*, which is the latest time the system should respond with a notification event. For each DT, the system combines m E into a Complex Event (CE) and then applies a Complex Event Reasoning (CER) algorithm by applying Complex Event Criteria (CEC) to detect a Critical Event (CE). When a CE is detected, the system has to send a Notification Event (NE) to predefined recipient before the deadline D.

Figure 1.1 shows an LSDTNS example: a cluster of n DTs, where each DT has a set of primitive Events (E_i *where* $i = 1, ..n$) that compose a Complex Event (CE). LSDTNS monitors n DT to recognize when Complex Event Criteria (CEC) are reached, and, once that happens, it generates a Notification Event (NE).

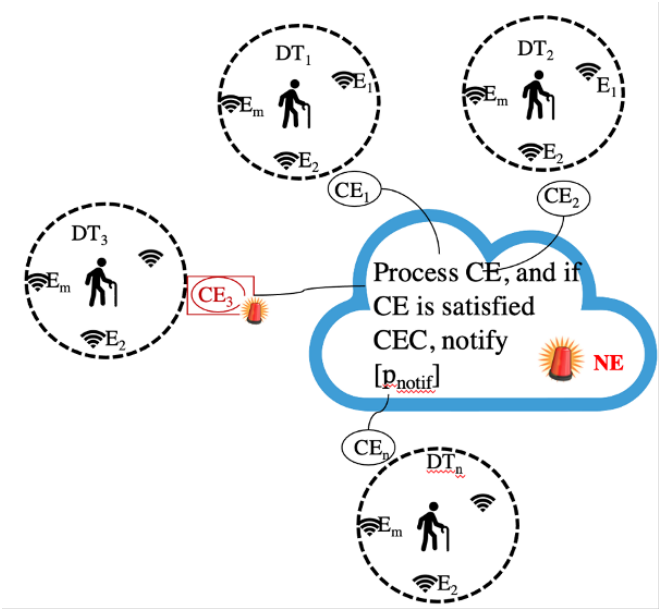


Figure 1.1: LSDTNS example.

To better illustrate the LSDTNS concept, the "prisoner" use case example will be presented: The goal of the prisoner use case example is to develop an IoT Notification system to improve inmates' safety by detecting events that correspond to attempted suicides. Monitoring the prisoners is done using small radars that can detect activities (active or not), posture (lying, sitting or standing) and estimate their breathing rate every 30 seconds. Detection and estimation are done using signal processing algorithms developed in our laboratory. These sensors' accuracy could be much higher; the current values are in the 80% range. The objective is to reliably detect that the person is not moving and not breathing, then to alarm the officers [1]. We envision this system being placed in every prison cell, resulting in many sensors sending data continuously [2]. A prisoner presents a Monitoring Entity, also known as a Digital Twin (DT).

A DT critical state is the estimated state of the monitoring object that requires generation notification or the alarm. A critical state of a DT, the estimated state of the monitoring object that requires generating notification, is defined in the system and requires that the

IoT Notification system is continuously monitored. For example, a prisoner is in a critical state if they attempt suicide. The system can only recognize the attempt through a complex event recognition of detected primitive events (for example, in the prisoner use case: activities, posture, and breathing rate).

The LSDTNS systems have a notification deadline: For example, in the prisoner use case [1], the system's response to the deadline must be less than 3 minutes, which is one of the system requirements [1].

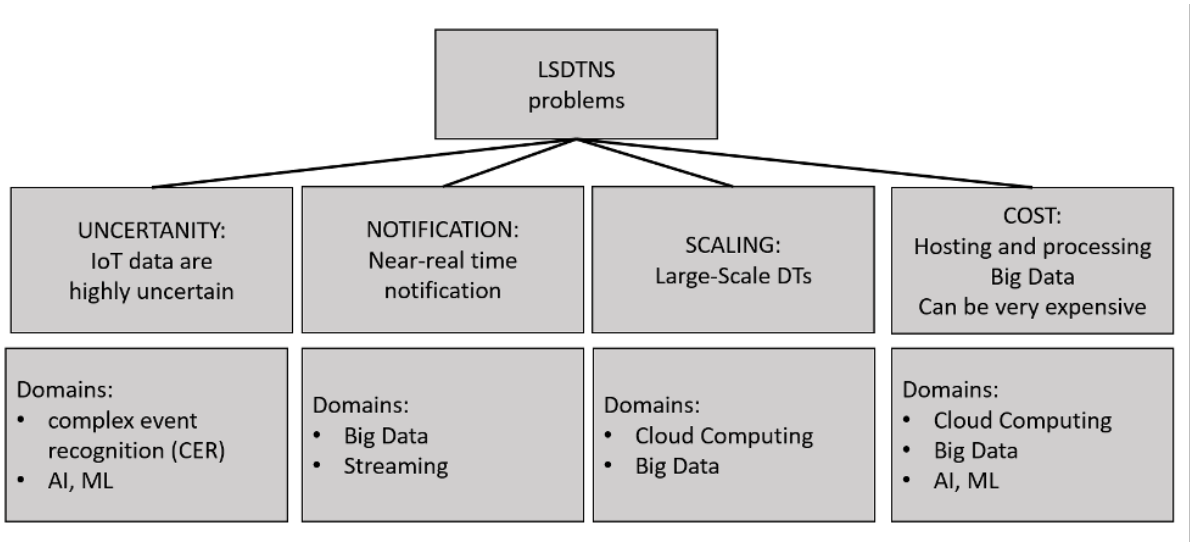


Figure 1.2: Large Scale Digital Twins Notification System challenges and the list of the research domains contributing to the current art.

LSDTNS is a complex system facing multiple challenges, crossing multiple research domains. Figure 1.2 presents LSDTNS challenges and research domains that this dissertation focuses on. The challenges are the following:

1. *Uncertainty*, the system must be able to address uncertainty in the system through complex event recognition;
2. *Notification*, the system must meet the IoT Notification system deadline;
3. *Scalability*, the system must address running data-intensive jobs;
4. *Cost optimization*, the system must consider cloud resource cost optimization.

The research domains that have been researched during the dissertation work include:

1. Complex Event Recognition (CER)
2. Bigdata
3. Cloud Computing
4. Multi-objective optimization (MOO)
5. Artificial Intelligence (AI) and Machine Learning (ML)

LSDTNS has to address *Uncertainty* in the system; some events in LSDTNS are inferred from other events, and uncertainty is propagated from parent events to the inferred events, contributing to overall system uncertainty. Most LSDTNS monitor Complex Events (CE), a complex relationship of primitive events produced by IoT devices and collected in LSDTNS systems. An essential part of the system under observation is recognizing complex events that match a given set of Complex Event Criteria (CEC), where CEC is a function of m primitive events. Therefore, the DT critical state detection is achieved with CEC [2].

CER is a research domain that deals with the recognition of complex events from large-scale data streams, which can be inherently uncertain due to various factors such as noise, incompleteness, and ambiguity. Therefore, CER algorithms must account for uncertainty to recognize and effectively interpret events in real-world scenarios. There are two main directions for handling uncertainty in the CER domain: Complex event programming, rule-based solutions, and probabilistic programming models, which can assign probabilities to different events based on the available evidence and the level of confidence in the evidence. CER aims to enable computers to detect and understand complex events in real time automatically. More information on the CER state-of-the-art can be found in Chapter 2.

In this dissertation, the focus is not on contributing to the CER algorithms but on addressing scalability issues in state-of-the-art CER systems, which is even more required for large-scale systems such as LSDTNS. In addition, CER is an emerging research domain where new algorithms keep emerging fast. Because of that, LSDTNS focuses on providing an architecture that allows for easy and quick application of the new algorithms to LSDTNS with minimal interruptions to the existing system when needed, which will also translate to lowering the system maintenance cost.

LSDTNS generates a large amount of data and must address *Bigdata* problems. The LSDTNS must monitor n Digital Twins, where n can be a large number, order of a few

hundred. Each Digital Twin has m IoT devices connected through the network. IoT devices generate primitive Events (E) and send them in specified time intervals over the network, which generates a sequential array of events known as an Event Stream (ES). Therefore, the system must be scalable and capable of processing and storing a large number of DTs' data. More information on the Big Data state-of-the-art can be found in Chapter 2.

LSDTNS is required to process a massive volume of data streams. The streamed data arrives from Monitored Objects (MO), contributing to virtual Digital Twins representation, where each MO has a set of sensors that stream data in time intervals. Therefore, a highly scalable solution is critically required to handle such data effectively. In addition, this implies that selecting appropriate scalable *Cloud Computing architecture* is crucial.

LSDTNS running data-intensive jobs is *costly*. Running data-intensive jobs requires large-scale parallel processing engines; a *Cloud Computing* environment provides the required processing power. Even though the processing is done on a Cloud, the cost can still be significant. Therefore, optimizing the usage cost of cloud resources for running data-intensive jobs is very important. Cloud service providers, such as Amazon AWS, Microsoft Azure, IBM, etc., allow users to outsource the hosting of applications and services to a cloud using clusters of virtual machine instances. Cloud service providers charge a service usage cost to cloud service users. The service usage cost is based on the hourly usage rate of the virtual machine instances running in the service provider cloud [44].

LSDTNS must address *Near Real-Time Notification* problems. The LSDTNS systems have a *notification deadline (D)*, which is the latest time the system must respond with a notification event. For example, in the prisoner use case, the system must detect an event that corresponds to attempted suicides, and the system must respond by the required deadline specified in the system requirements.

LSDTNS must address *Multi-optimization objective* problems. The LSDTNS system has multiple objectives: *deadline (D)*, *cost* and *resource usability* to address simultaneously.

LSDTNS should learn directly from experience; the latest achievements in *ML and AI* should be explored.

In the end, each DT has a set of additional information on the DT; we will call the set of additional information the DT Profile (profile). For example, a DT profile can contain additional information on a monitored prisoner: the length of the prisoner's sentence, medical condition, psychological state, and similar personal data. Based on the profile, we can determine the critical state of the monitored DT.

1.2.2 Short Overview of the State-of-the-art

This section will provide a short overview of the state-of-the-art, and more details can be found in Chapter 2. As we already mentioned in the previous section, LSDTNS is a complex system facing multiple challenges crossing multiple research domains. In our state-of-the-art research through multiple domains we find that research challenges in one domain might have a solution in another domain and vice versa. Figure 1.3 shows LSDTNS data flow analysis and presents our attempt to organize researched domains based on the data flow. The LSDTNS's data flow analysis consists of three stages. The first stage is the data originator, which defines the components that originate data, such as IoT sensor data (covered in Section 2.1) and Digital Twin (covered in Section 2.2). The second stage is the data transport. We acknowledged that this is an unimportant component, and we will not focus on it in this dissertation. The third component is data processing. In this component, we looked for the following answers:

- What kind of data is processed? Section 2.4 provides an answer to the question and covers the Bigdata domain.
- Where is data processed? Section 2.5.5 provides an answer to the question and covers the Cloud computing domain.
- How data processing is architected? Section 2.6 provides an answer to the question and talks about architecture.
- What kind of algorithms are used to process data? Section 2.3 provides an answer to the question and talks about the Reinforcement Learning algorithm.

In their brief survey paper on Complex Event Recognition (CER), Alvizos et al. [8] highlighted certain limitations observed in the current state-of-the-art. They observed that distributed processing for probabilistic complex event streams is still in its early stages of development. Probabilistic Programming (PP), a foundational machine learning technology, is preferred over Complex Event Modeling (CEM) [13], with techniques such as Bayesian Networks and Markov Chains being prominent examples.

In summary, probabilistic programming holds promise for addressing uncertainty. However, a key challenge in the probabilistic programming approach is that achieving higher accuracy, especially with algorithms like Dynamic Bayesian networks capable of modeling complex probabilistic systems, often requires longer processing times and more computational resources. Hence, the strategic selection of probabilistic programming algorithms

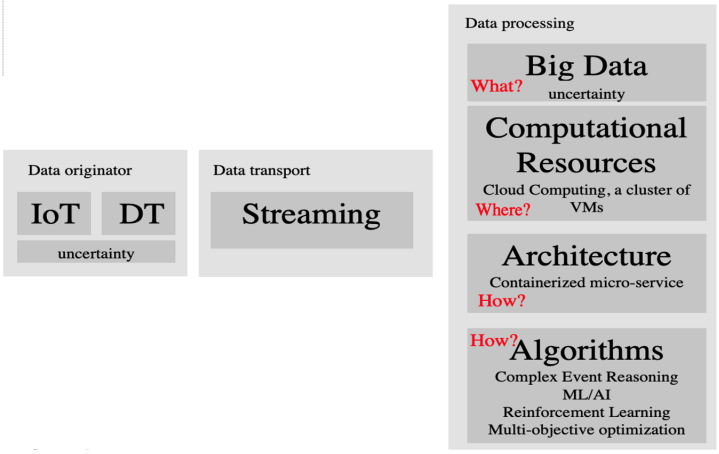


Figure 1.3: LSDTNS data flow analysis consists of three components: a) data originator, b) data transportation, and c) data processing.

becomes pivotal, aligning them with the required accuracy and processing time constraints. Furthermore, as highlighted in the survey [8], all observed Complex Event Recognition (CER) systems seem to exhibit scalability issues, emphasizing the necessity for a novel architectural approach.

In Section 2.6 we look into two architectural approaches: microservice and monolithic. The monolithic architecture is a traditional software development architectural style, where all components are built as a single code base and deployed as a single installation file. The micro-service architecture is a newly adopted architectural development style, where the application comprises small autonomous services developed for a specific domain.

IoT systems and DT produce huge and complex data known as Bigdata. Bigdata is impossible for traditional software and traditional warehouses to process, where the Cloud computing domain is developed to help. with Bigdata processing. Bigdata can be structured, unstructured or semi-structured from different sources, such as machines, humans, and information collected from nature [31]. More information is presented in Section 2.4. Big Data solutions mainly run in Cloud computing environments (more on the topic can be found in Sections 2.5 and 2.5.5. Cloud computing is a set of aggregating hardware resources, mostly virtualized resources.

In order to better present the Cloud computing domain, we will provide Buyya et al. definition [45], where Cloud is a parallel and distributed computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned

and presented as one or more unified computing resources based on service-level agreements (SLA) established through negotiation between the service provider and consumers [45]. Virtualization is the foundation of Cloud computing and is achieved with Virtual Machines (VM). Virtualization technologies are an inseparable part of cloud computing that enables multiple virtualized servers to run in isolation on physical machines. Two major server virtualization technologies are hardware-level (HL) and operating-system-level (OSL) virtualization. Service use cost is related to the number of virtualized resources used in a solution. Therefore, the allocation of virtualized resources, particularly OSL (Docker), is one of the focuses of this research. More detail on Cloud Computing is presented in Section 2.5.

Two directions in Cloud computing allow us to process a huge amount of data. The first one uses parallel processing engines (see Section 2.5.5), and the second uses containers (see Section 2.8.0.1).

There are a couple of state-of-the-art solutions that use parallel processing in Cloud computing that we will mention. The work of Akbar et al. [35] is a great attempt and starting point to address the complex event uncertainty in Large-Scale IoT applications. However, the work is a simplified problem that LSDTNS must address. Therefore, their solution needs to address the cost and deadline of the solution. In addition, the Bayesian Network they used in their example is oversimplified, and they mainly only investigate one monitoring object, Digital Twin. More information can be found in Section 2.5.5.1.

Sidhanta et al. [47] also uses the parallel processing cloud computing approach. It presents OptEx: a novel deadline-aware cost optimization model for optimizing the cost of running data-intensive jobs in a Spark environment. The work of Sidhanta et al. [47] provides a novel, deadline-aware cost optimization model for optimizing the cost of running data-intensive jobs in a Spark environment. However, neither solution addresses our requirements. More details can be found in Section 2.5.5.2

Running Bigdata-intensive jobs in a cloud computing environment requires large-scale parallel processing engines that can be very costly. Optimizing the cost of using cloud resources for data-intensive jobs is a fundamental yet relatively less explored problem [47]. Cloud service providers, such as Amazon AWS, Microsoft Azure, and IBM, allow users to outsource the hosting of applications and services to a cloud using clusters of virtual machine instances. Even though Sidhanta et al. provided a novel deadline-aware cost optimization model, the model does not address our problem statement requirements. Topics on optimizing service usage cost will be discussed in chapters 4 and 5.

Wu et al. [40] contributed significantly to the Container Resources allocation field. Notably, they were focused on the Docker container resource, which is widely used. Their

work proposes a CPU allocation approach, called a flexible deferrable server (FDS) scheduler, that improves the performance of a Docker containerized mixed-criticality real-time system (DC-MC-RTS). DC_MC_RTS defines two container types: RT-Containers and NRT-Containers, where an RT-Container type is a container containing the real-time application, and an NRT-Container type is a container containing a non-real-time application. The proposed FDS has been implemented and is available in the Docker community edition 17.09.0 [84] However, their work needs to provide more granularity to solve the resource challenges.

In section 2.8 we cover scheduling in more detail. Here, we will highlight two: DeepML [65] and DeepML2 [66], where DeepML2 further improves DeepML, significantly contributing to resource allocation using reinforcement learning. Compared with traditional heuristics approaches, both solutions adapt to different conditions, converge quickly, and learn sensible strategies in hindsight. Therefore, we use their learning as groundwork for our SCN-DRL algorithm. However, our objectives are different from DeepML and DeepML2. Instead of focusing on the average slowdown time, we are more interested in on-average job competition based on job priority, where our solution favors the most critical jobs first. In addition, the synthetic data used to train both DeepML and DeepML2 only support two types of jobs based on the job length: short and long, where the ratio is fixed at 80% short and 20% long jobs. In reality, it is expected that this ratio can change over time.

The LDTNS system has multiple objectives: deadline (D), cost, and resource usability to address simultaneously. Most real-world scenarios involve making trade-offs concerning different performance objectives [38]. No single optimal policy in multi-objective problems maximizes all the objectives. Instead, there is a Pareto set (also known as the Pareto front), a set of all Pareto optimal solutions that are non-dominated. The non-dominated solution presents another feasible solution to the observed current solution; it is better than the current one in some objective functions and does not worsen other objective functions. Finding the optimal Pareto set is a complex problem and hard to achieve, so an approximation is often used. More information on the current state-of-the-art in multi-objective optimization can be found in Section 2.7.

1.2.3 Motivation

The proposed research goals are motivated by the need for Critical Large-Scale Notification Systems that can process uncertain data and respond within a notification deadline (D), where D is the latest time the system should respond with a notification event. In addition, the "prisoner" case discussed earlier can be easily extended to other similar use cases. For

example, it monitors babies in the natal care unit, where a quick response based on vital functions requires a fast response or any other critical notification system.

As mentioned in Section 1.2.1, different domains should be considered in solving the problem. On one side, we have highly uncertain data (Complex Event Recognition research domain area, covered in Chapter 2 Section 2.1.1.1) that requires trade with extended processing time and high processing power. Conversely, running data-intensive jobs, such as IoT notification systems, on a large-scale parallel processing engine can be very costly (Chapter 2). The cost of running large-scale parallel processing is an important topic yet a relatively less explored and researched problem [47].

This research aims to develop an intelligent enough solution to trade between DTs' complex event monitoring accuracy and cost-effectiveness of the overall system. However, most DTs at a given time will be in a non-critical state; therefore, by prioritizing DTs based on the critical level, we will be able to trade the accuracy level with the cost.

This dissertation proposes research on the solution for a Cost-Effective Large Scale Notification System for a Digital Twin Cluster, considering the uncertainty and priority of Digital Twins. The thesis proposal considers that IoT systems are highly uncertain, have to process massive amounts of data, and must be able to process and scale a cluster of digital twins. In addition, the following assumption is that not all Digital Twins will require notification messages, have equal priority and require equal sampling frequency for keeping close monitoring.

1.2.4 Research Questions

Based on the challenges discussed in Sections 1.2.2 and 1.2.1, the following research questions (RQs) are raised:

1. RQ1: How to architect LSDTNS that can address the following:
   (a) How can we optimize the architecture of LSDTNS to minimize operational costs, maximize resource utilization, and ensure the system's ability to recognize critical complex events while meeting required deadlines?
   (b) How can we architect a scalable and maintainable CE request processing system for a DT cluster to efficiently handle hundreds of DTs while meeting system deadlines?

2. RQ2: Techniques for selecting the right VM type offered by a Cloud Service provider based on cost, CPU, RAM, and storage are challenging. Additionally, LSDTNS adopts microservice architecture, which increases the number of components in the applications; the problem then becomes even more challenging. Cloud service providers provide a long list of VM types; each provided VM type has CPU, RAM, Storage, and cost. For such a complex deployment system, the following questions need to be answered:
   (a) How to select a VM type that provides the most cost-effective solution?
   (b) Which container cluster deployment will provide the best cloud VM resource utilization?
   (c) How many microservice container instances are required to support the incoming workload?
3. RQ3: How to schedule incoming DT complex event requests in order to satisfy multi-optimization objectives:
   (a) How should the incoming workload be scheduled to deliver a notification on time (by the deadline)?
   (b) How should the incoming workload be scheduled so that cloud resources maximize utilization?
   (c) Can we build a solution that learns to manage the cloud container's resources directly from experience?

1.2.5 Specific Aims

Specific Aims that the dissertation should achieve are:

1. Developing LSDTNS architecture that is cost-effective, scalable, and able to process large amounts of streamed incoming DT data before specified system deadline.
2. Create a methodology that is able to select a VM type based on a multi-objectivity approach that satisfies three objectives: cost, resource utilization and deadline.
3. Create a schedule for incoming large DT complex event requests that satisfy multi-optimization objectives and learn based on experience.

1.2.6 Assumptions

We are making several assumptions for the LSDTNS.

Assumption 1: All DTs have the same number of IoT devices (sensors) and the same DT Profile metadata.

Assumption 2: The system must process n DT CE requests and, if CEC is met, send a Notification Event (NE) message. Of course, not all requests will require a notification event message to be sent. The parameter p_{notif} (network load) presents a percentage of total n requests that will require a notification event message [2] [3], and p_{notif} can take a value in [0,1].

Assumption 3: The size of DTs, n, might change over time, although we assume that change will be minimal. If, for example, prisoner cells in prison are the observed DTs [2], it is unlikely that the number of cells will change drastically over time. What can change is the p_{notif}, the number of CE requests requiring notification event messages to be processed.

Assumption 4:

1. *System Capacity (c)*: refers to the maximum number of digital twins that the system is capable of accommodating. This limit represents the upper bound of the system's capabilities in terms of managing and processing digital twins.

2. *Workload:* presents the number of digital twins that are presented to the system for processing. Specifically, it denotes the number (n) of digital twins that Complex Events (CEs) are tasked with handling within the system. This metric provides insights into the demand placed on the system's resources and processing capabilities.

3. *Notification Workload:* signifies the number of requests that require notifications to be issued.

Assumption 5: Additional information about DT, known as the DT profile, will be considered, and DT profile can increase or decrease the overall DT's priority.

1.3 Summary of Contributions

The research methodology and objectives are highlighted in Figure 1.4 w.r.t the identified research questions (Section 1.2.4). Four research domains are considered in order to attain those objectives. Furthermore, the following major contributions of this dissertation are summarized in the following subsections:

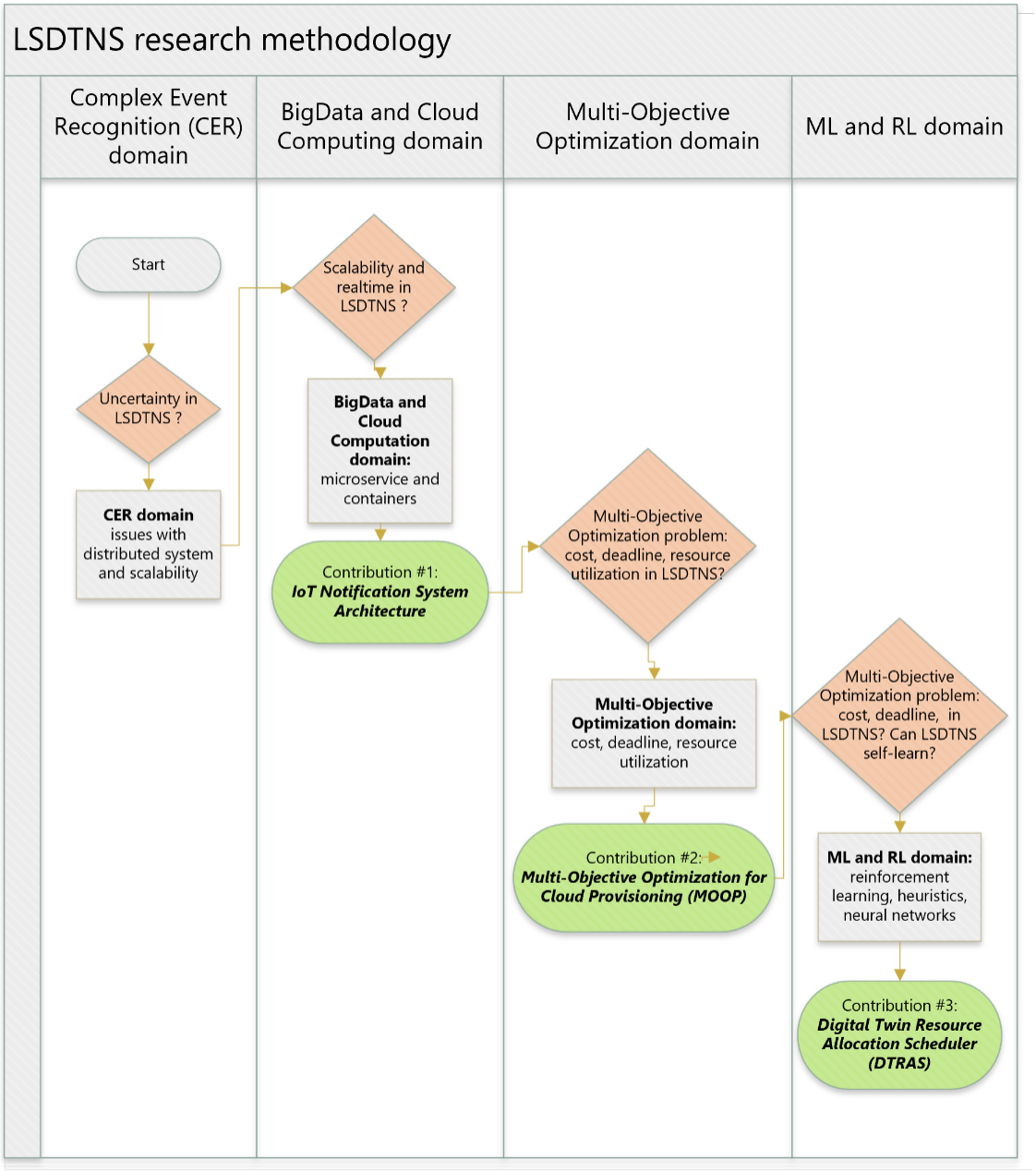


Figure 1.4: Challenges of the Large Scale Digital Twins Notification System and the list of the research domains researched in this thesis to address challenges.

1.3.1 IoT Notification System Architecture

The IoT Notification System architecture covered in our work [3] presents a micro-service-based notification methodology that uses complex event recognition to handle the IoT system's uncertainty. Current works in Processing Uncertain Complex Events are attempts to quantify the system uncertainty by estimating the likelihood of the occurrence of events of interest while taking into account the uncertain data and uncertain parameters of the model of the physical system with rule-based complex event processing or the Bayesian network. However, these methods were implemented as stand-alone software solutions that cannot scale under heavy loads of incoming events. This research proposes a micro-service-based notification methodology [3] that opens the door to scaling IoT uncertain streaming data to many sensors, which cannot be done with the current works.

1.3.2 Multi-Objective Optimization for Cloud Provisioning

Multi-Objective Optimization for cloud Provisioning (MOOP) presents an approach to optimize a large-scale micro-service deployment for a critical notification system. The proposed optimization solution addresses optimization for three objectives: cloud service cost, cloud resource utilization of CPU, RAM and storage, and meeting the system notification deadlines. To the best of our knowledge, this is the first work dealing with multi-objective optimization for microservice notification applications where the notification load is variable and depends on other priority-executed microservices, which are part of the total process request. The presented work analyzes and proposes a Pareto front and optimal list from preselected VM candidates. First, we formulated multi-objective optimization problems for micro-service-based large-scale notification system applications and provided the mathematical deployment model. Then, based on the expected workload and required multi-objective criteria, the thesis proposes a set of algorithms that provide the Pareto front optimal solution for preselecting a Cloud Service provider's VM type.

1.3.3 Digital Twin Resource Allocation Scheduler

Digital Twin Resource Allocation Scheduler (DTRAS) was devised using Deep Reinforcement Learning (DRL). The work presented in this chapter is not limited only to the cluster of Digital Twins. The same concept can be used in a much broader deployment area as a Scheduler for large-scale Critical Notification applications based on Deep Reinforcement Learning (SCN-DRL). SCN-DRL is a Multi-objective Scheduler for a Critical Notification system in a Digital Twin (DT) cluster based on Deep Reinforcement Learning. This

research addresses optimization for three objectives: cloud service cost, cloud resource utilization, and system notification deadline. To the best of our knowledge, this is the first research dealing with multi-optimization objectives for critical microservice notification applications using deep reinforcement learning, where the notification load is variable and depends on the results of the other prior processed microservice. Inspired by recent advances in deep reinforcement learning, we researched how to build a solution that learns to manage the cloud container's resources directly from experience. We evaluated three Neural Network (NN) types in the proposed work. Results show that all three NNs: a) provide feasible performance for scheduling the DT cluster's notification jobs, b) outperform state-of-the-art heuristics, and c) keep steady performance when notification workload increases from 10 to 90%. Furthermore, resilience to resource container failures is a critical component of the distributed system; our proposed research shows that SCN-DRL is resilient to sudden resource drops by up to 10%.

1.4 Thesis Organization

The remainder of this dissertation is organized as follows:

Chapter 2 briefly discusses the fundamental concepts of the Internet of Things, Digital Twins, the Bigdata, Cloud Computing, Uncertainty, cloud computing resources, and reinforcement learning.

Chapter 3 presents our proposed IoT Notification System architecture. The presented architecture is groundwork for further research in this dissertation.

Chapter 4 presents our proposed Multi-objective optimization for cloud provisioning (MOOP). In the end, the chapter presents a case study to demonstrate the proposed techniques.

Chapter 5 presents our proposed Scheduler for the Critical Notification system based on Deep Reinforcement Learning (SCN-DRL). The end of the chapter presents experimental results and a performance discussion of the DTRAS solution.

Chapter 6 concludes this dissertation and discusses future work.

Chapter 2

Background and Literature Review

This dissertation chapter covers the state-of-the-art and is organized into eight sections.

Section 2.1 introduces the concept of Internet of Things and discusses uncertainty in IoT systems. Section 2.2 introduces the Digital Twin concept and covers uncertainty in DT's systems. Section 2.3 introduces Reinforcement Learning, focusing on Policy Gradient and Imitation learning. Section 2.4 introduces Bigdata concepts and covers uncertainty in Bigdata systems. Section 2.5 introduces Bigdata and Cloud Computing. Then, Section 2.6 debates the pros and cons of microservice and monolithic architectures. Section 2.7 covers the state-of-the-art and concerns multi-objective optimization. Finally, Section 2.8 covers the state-of-the-art in scheduling on Clouds.

2.1 Internet of Things (IoT)

Internet of Things (IoT) means "thing" or "object" connected to the Internet and each other. An object (a "thing") can be almost anything: a computer, tablet or smartphone, door lock, home security camera, fitness device, and etc. What is essential is that each object has: a) a unique identification number (UID) and b) an Internet Protocol (IP) address. Next, those objects have to be connected, and this can be achieved via cords, wires, and wireless technologies, including cellular or satellite connections, Bluetooth, and WIFI. In addition, they sometimes use built-in electronic circuitry and radio frequency identification (RFID) or near-field communication (NFC) capabilities. Once objects are connected, the collected object data must be transferred for further processing [7].

IoT and Bigdata concepts go together: An IoT System data is essentially a Bigdata.

To better understand the Bigdata concept, it is helpful to have some historical background. In 2001, Gartner provided the Bigdata definition, which is still the primary definition: Bigdata is data that contains a greater variety arriving in increasing volumes and with ever-higher velocity. The Bigdata attributes variety, volume, and velocity are the three Vs.

Bigdata has significant volume and complex data sets, especially from new data sources, such as social media or IoT systems. These data sets are so voluminous that traditional data processing software cannot manage them. Nevertheless, these massive volumes of data can be used to address business problems that would not have been able to be tackled before.

The use of Bigdata analytics has grown significantly in just the past years. At the same time, the IoT has introduced public awareness and people's imagination about what an entirely interconnected world can offer. For this reason, Bigdata and IoT are almost made for each other. IoT devices will produce, analyze, share, and transfer data in real-time. Without this data, IoT devices would not have the functions and capabilities that have caused them to gain so much worldwide observation.

2.1.1 Uncertainty in IoT systems

As previously noted, IoT notification systems are complex systems that try to observe a physical system by deploying many sensors, such as temperature sensors, cameras and GPS tracking devices. These sensors produce a large amount of raw, uncertain data. It is important to emphasize that only a small subset of that raw data, combined with other collected sensor data and additional information about the monitoring entity, presents a complex event that should be further processed in IoT notification systems. For example, such notifications may be used to alert doctors and nurses of the critical state of their remote patients or to alert dispatchers when a vehicle is stolen. Recognizing the combination of these data events is known as Complex Event Recognition (CER), and such systems are known as Complex Event Systems (CES).

Most existing works have focused on extending rule-based complex event processing by including uncertainty calculations in the solution. For example, in work done by Cugola et al. on TESLA, complex event specification language [10] was further extended to include system uncertainty. There were a few attempts to address uncertainty resulting from event interference using Bayesian network [11] [10] and Hidden Markov models [14]. Wasserkrug et al. [16] propose a mechanism for the event materialization under uncertainty and two

algorithms for specifying probability space based on the Bayesian network and the Monte Carlo sampling algorithm.

2.1.1.1 State-of-the-art: Complex Event Recognition

Based on the short survey provided by Alvizos et al. [8], Complex Event Recognition (CER) systems did not handle Uncertainty until recently; all papers cited in the work were published after 2008. The short survey paper [8] concludes that the fundamental drawback of most systems is the absence of support for constructing hierarchies of CEs. In addition, most systems do not support Uncertainty in the rules defining CEs, and those that support Uncertainty in the rules are based on simplified assumptions. Furthermore, they conclude that making substantial simplifications limits the accuracy in domains with complex dependencies, whereas other solutions face severe underperformance issues, even when approximate inference is employed. Furthermore, their paper states that the distributed processing of probabilistic complex event streams is still in its early stages [8]. In order to continue the discussion with related work, we need to introduce a few terms from the probability programming field. General knowledge presents what we know to hold in the observed domain without considering the details of a particular situation [13]. A probabilistic model encodes general knowledge about a domain in quantitative probabilistic terms [13]. Probabilistic programming is one way to code probabilistic models and interfaces. Bayesian or probabilistic machine learning is the fundamental concept of machine learning and is a way to create a system that helps us make designs in the face of Uncertainty [13]. Evidence is specific information we know about a particular situation [13]. A query asks a question to gather information about the situation [13]. An inference uses a probabilistic model to answer queries with the given evidence [13]. Possible Worlds describe all situations that are considered possible before seeing any evidence. A prior probability distribution is the probability distribution before seeing any evidence. A posterior probability distribution is a distribution after considering evidence [13]. Maximum a posteriori (MAP) estimation chooses the most probable value given observed data and prior Belief. Maximum likelihood estimation (MLE) chooses a value that maximizes the probability of observed data. Alvizos et al. [38] also state that all review CER solutions only support the probability of occurrence of a complex event, and none of them support Maximum of Posteriori (MAP) inference or approximate inference.

In addition, Dhilon et al. [39] research work introduces an edge-computing based Complex Event Processing (CEP) architecture for Remote Patient Monitoring (RPM), which is an essential issue in the context of remote healthcare. The proposed architecture [39] detects complex events that may indicate impending health problems and is performed on

a mobile device that receives data from sensors attached to a patient's body.

2.2 Digital Twin

Digital Twin is a new paradigm, and the definitions vary among scientific circles. One close to our understanding is the IBM definition: a digital twin is a virtual representation of a physical object or system across its lifecycle, using real-time data to enable understanding, learning, and reasoning [18]. The "Digital Twin" concept was introduced in the publication of "Mirror Worlds" by D. Gelernet in 1991. Dr. M. Grieves applied the first idea as a software concept for manufacturing in 2002. The term "Digital Twin" was first introduced by NASA's John Vickers in 2010 [18].

2.2.1 Three-dimensional Digital Twin

In 2003, Dr. M. Grieves proposed the first three-dimension DT, which, by some academic circles, is considered the origin of the DT [19] [20]. As shown in 2.2, the three-dimension DT contains three parts [20]:

1. A Physical Entity (PE) in physical space
2. A Virtual Entity (VE) in virtual space, and
3. A Connection (CN) of data and information that ties the physical and virtual entities together.

The physical entity exists in a physical space with real functions and capabilities. It operates based on specific preplanned behaviours and produces actual outputs. For example, an airplane, a physical entity, can fly, carry cargo, land and, etc. On the other side, the virtual entity consists of models to describe the physical counterpart from different perspectives, such as geometrical dimensions, physical properties, behaviours, etc. The DT is a digital mirror that accompanies the physical entity during its lifetime, characterized by real-time synchronization, accurate mapping, and high reliability. Interactions between the physical and virtual entities are performed via the connection. The physical entity collects and transmits the real-time data to the virtual replica for model updating and calibration. The virtual entity feeds valuable information generated from virtual simulations back to the physical entity to guide and optimize the corresponding physical entity. The DT forms a closed loop between the physical and the virtual space [19].

Figure 2.1: Three-dimension DT Physical Entity (PE), Virtual Entity (VE) and Connection of Data and Information (CN).

2.2.2 Extended five-dimensional Digital Twin

Tao et al. 2018 proposed an extended five-dimension DT [21] by adding DT data and services concepts to the previously mentioned three-dimension DT. The five-dimension DT concept is designed for a broader range of applications and to support data fusion and on-demand usage; a high-reliability modelling method is developed to perfectly and thoroughly reproduce geometries, physical properties, behaviours, and rules of the physical entity.

The five-dimension DT concept is shown in expression 2.1 [21] :

$$M_{DT} = (PE, VE, Ss, DD, CN) \tag{2.1}$$

PE refers to the physical entity existing in the physical space; VE represents the virtual entity consisting of a set of models; Ss stands for the services for both PE and VE; DD is the DT data, and CN is the connection that ties different parts of the DT together. Furthermore, Figure 2.2 shows a five-dimension DT concept model and the model elements' interactions.

2.3 Reinforcement Learning

This section covers *Reinforcement learning (RL)*, one of the most active and promising research areas in artificial intelligence.

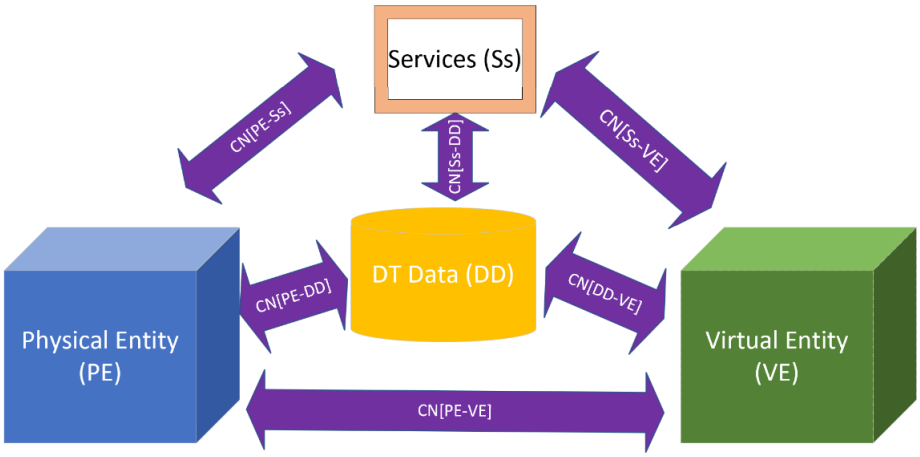


Figure 2.2: Five-dimension DT concept model that consists of file elements: PE, VE, CN, and two model extensions Ss and DD

The well-known RL book, "Reinforcement Learning: An Introduction," published in 1998 by Sutton and Barto [26], effectively defines and explains reinforcement learning as: "learning what to do, how to map situations to actions, to maximize a numerical reward signal. The learner is not told which actions to take but must discover which ones yield the most reward by trying them. In the most interesting and challenging cases, actions may affect the immediate reward and the next situation and, through that, all subsequent rewards. These two characteristics—trial-and-error search and delayed reward—are the two most important distinguishing features of reinforcement learning."

Reinforcement learning (Figure 2.5) consists of four elements:

- Agent,
- Environment,
- Action,
- Reward.

RL is a type of ML that involves training an agent to interact with an environment to learn a specific task. The agent receives feedback in the form of rewards or punishments, and the goal is to maximize the cumulative reward over time. Communication between

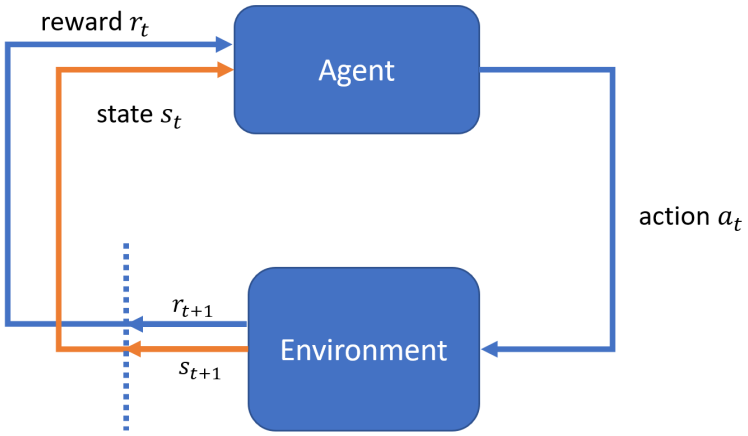


Figure 2.3: Four elements of Reinforcement Learning: agent, environment, action and reward.

agents and the environment is done through set of actions, know as an Action Space. For each action selected by an agent, the environment performs a predefined change in the environment, which brings the environment to a new state. The environment can only be in the predefined set of states, known as a State Space. The environment will validate the new state and reward the action. The agent can be any algorithm, for example, a heuristic or a neural network. The algorithm takes as in input the state of the environment and returns one of the RL system's actions from Action Space.

The learning happens through the actual environment interaction, where the model-free method is selected. The reinforcement learning model-free algorithm can be divided into two main groups: Dynamic Programming and Policy Optimization. Dynamic Programming [8] and Policy Optimization [8].

Dynamic Programming is used to compute the optimal policy, a sequence of actions that maximizes the expected reward over time. Dynamic Programming assumes that the underlying dynamics of the environment are known, and the goal is to find the optimal policy that maximizes the expected cumulative reward over an infinite time horizon. The dynamic programming method involves the optimal policy by solving the optimal action-value function. On the other hand, policy optimization is a process of improving the policy of an RL agent to maximize the expected cumulative reward. The policy is a function that maps the environment's state to an action that the agent should take. The goal of policy optimization is to find the policy that maximizes the expected cumulative reward over time. One of the subgroups of Policy Optimization is the Policy Gradient method that we

will focus on in the thesis.

2.3.1 Policy Gradient Method

The policy gradient is a representation of the policy-based approach. The gradient of the objective given by [26]:

$$\nabla_\theta E_{\pi\theta}[\sum_{t=0}^{\infty} \gamma^t r_t] = E_{\pi\theta}[\nabla_\theta \log \pi(s,a) Q^{\pi\theta}(s,a)] \tag{2.2}$$

In Equation 2.2, $Q(s,a)$ represents the cumulative reward value based on the $\pi\theta$ policy. (s,a) represents the probability of selecting an action a in state s under the current policy. After obtaining the gradient, we use the gradient ascent algorithm to update the parameters of the policy:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta log \pi_\theta(s_t, a_t) v_t \tag{2.3}$$

Equation 2.3 is the parameter updating process, θ is the policy parameter, and α is the learning rate. v_t is an unbiased estimate of $Q(s,a)$.

2.3.2 Imitation Learning

Imitation Learning (IL) methodology aims to speed up RL learning time. The optimization policy is usually learned in traditional reinforcement learning tasks by calculating cumulative rewards. The traditional RL method is simple and straightforward but requires a very long training process and many training data for better performance [66]. After years of development, the IL method solved multi-step decision-making problems well and significantly speed up the training process of reinforcement learning. It also reduces the size of the required training data [14]. IL has been widely applied in robotics, self-driving cars, and other fields [66]. However, the search space for multi-step decision-making in RL tasks is enormous. Learning appropriate decisions before many steps based on cumulative rewards is very difficult. Directly imitating the "state-action" of human experts can significantly ease this problem. This process is known as "*Behaviour Cloning (BC)*". BC produces the samples, which are initially used for machine learning [27]. Building the

initial policy model can be done with a classifier or regression algorithms. Then, such a policy can be used as an initial policy for machine RL. Since the RL methods are based on environmental feedback, the first model will be improved over the learning time to obtain a better policy.

2.3.3 Neural Networks: Convolutional and Fully Connected

Artificial Neural Networks (ANN) are brain-inspired systems that are intended to replicate the organic neural network, such as the human brain, learn. One of the ANN commonly used techniques is *Multi-Layer Perception (MLP)*. MLP is composed of one (pass-through) Input Layer, one or more layers of *Linear Threshold Units (LTU)*, called Hidden Layers, and one final layer of LTUs called the output Layer [86]. When an ANN has two or more hidden layers, it is called a Deep Neural Network (DNN). DNNs have a much higher parameter efficiency than shallow ones; they can model complex functions using exponentially fewer neurons than shallow nets, making them much faster to train.

Further in this text, we will focus on two types of Hidden Layers, used in Chapter 5: Fully Connected (Compound) Neural Networks and Convolutional Neural Networks.

2.4 Bigdata

In the last couple of decades, continuous increases of computational power enabled society to produce an overwhelming data flow. As a result, Bigdata is increasingly becoming available, which led to research that enabled computers to understand Bigdata [32] better.

There are a lot of Bigdata definitions; we will highlight two.

- In 2008, the Journal of Science published that "Bigdata represents the progress of the human cognitive processes, usually includes data sets with sizes beyond the ability of current technology, method and theory to capture, manage, and process the data within a tolerable elapsed time".
- The definition given by the Gartner says: "Bigdata are high-volume, high-velocity, and/or high-variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization".

Bigdata is so huge and complex that traditional software and traditional warehouses can't work and process it. Bigdata can be structured, unstructured or semi-structured from different sources, such as machines, humans, and collected information from nature [31].

2.5 Cloud Computing and Bigdata

Bigdata and Cloud Computing are both the fast-moving technologies. Cloud Computing is associated with provisioning of computing infrastructure and Bigdata is associated with processing methods for all kinds of resources. Both technologies are developing in parallel, and they are interconnected. Furthermore, some new cloud-based technologies have to be adopted because of dealing with Bigdata for concurrent processing [32].

Clouds supply a set of aggregating hardware resources on demand for providing a large amount of computer power. The main idea behind Cloud Computing is delivering computing as a utility. Therefore, the Cloud computing business model provides a cloud computing service [45].

To better understand the Cloud Computing concept, we can briefly look into hydropower distribution for analogy, where we, as the electrical energy user, use electrical power in our home and offices, not knowing and wondering how and from where that energy is coming from. The same concept applies to the Cloud, where the user will use computational power when and how much they need, and they will be charged per usage. Conversely, the cloud computing provider will care of hardware, management, electrical energy, building and similar [45].

There are many Cloud Computing definitions; in the next few lines present three:

- Buyya et al. have stated: “Cloud is a parallel and distributed computing system consisting of a collection of inter-connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements (SLA) established through negotiation between the service provider and consumers” [45].
- Vaquero et al. have stated: "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), also allowing for optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which the Infrastructure Provider offers guarantees utilizing customized Service Level Agreements" [45].
- In addition, the report of McKinsey and Co. states that “Clouds are hardware-based services offering to compute, network, and storage capacity where: Hardware management is highly abstracted from the buyer, buyers incur infrastructure costs as variable OPEX, and infrastructure capacity is highly elastic” [45].

2.5.1 Virtualization in Cloud Computing

Virtual Machine (VM) is a software environment that emulates a complete computer system, including hardware components such as the CPU, memory, storage, and network interface. A VM allows multiple operating systems to run on a single physical machine, providing isolation and flexibility. Each VM runs as a separate entity, with its own operating system and applications, as if it were a physical machine. Replacing computational resources with VMs is called virtualization.

VMs are a fundamental building block of cloud computing. They enable cloud service providers to offer their customers the ability to provision and manage computing resources on-demand without the need for physical hardware. Here is the list of four key benefits that VMs bring to cloud computing:

- *Resource Optimization*: Virtualization allows multiple VMs to run on a single physical machine, which enables better utilization of hardware resources. This translates to cost savings for cloud service providers and their customers.
- *Scalability*: Cloud computing is designed to provide elastic scaling of resources, meaning that customers can quickly and easily scale up or down their computing resources as needed. VMs make this possible by providing a flexible and portable platform for running applications.
- *Isolation*: VMs provide a layer of isolation between different applications and customers running on the same physical machine. This helps to prevent one customer from affecting the performance or security of another customer's applications.
- *Easy Migration*: VMs are portable and can be easily moved between physical machines, making it easy for cloud providers to perform maintenance or upgrades without disrupting customer services.

2.5.2 Cloud Computing Infrastructure Resources

Cloud Computing infrastructure can only be achieved with virtualization technologies. Virtualization technology enables multiple virtualized servers to run in isolation on physical machines [40]. Two major types of server virtualization technologies are:

- *hardware-level (HL) virtualization*, such as Xen [11], KVM [33] and VMware ESXi [48], and

- *operating-system-level (OSL) virtualization*, such as LXC [13], Ubuntu LXD [8], BSD Jails [17], Oracle Solaris Zones [10], Windows Containers [87], and Docker [82].

The main difference between HL and OSL virtualization is that, in HL virtualization, each virtualized server runs its operating system and applications, while OSL virtualization, also called containerized virtualization, encapsulates applications as standard OS processes and their dependencies to create containers. OSL containers are also referred to as virtual servers.

2.5.3 Virtualization Containers

In OSL, virtualization containers are managed by the underlying OS kernel, which contributes to low-overhead virtualization. OSL virtualization effectively avoids the overheads of starting and maintaining virtualized servers (i.e., containers), while those overheads are necessary for HL virtualization. Because of the low overhead, OSL virtualization has become the dominant virtualization technology.

One of the popular OSL virtualization solutions is Docker [84]. As a result, we see a trend where many server applications are virtualized Docker containers. Docker uses the resource isolation features of the Linux kernel, such as cgroups and namespaces, to allow multiple containers to run within a single Linux system.

In the state-of-the-art space, it is important to emphasize the significant scientific contributions in the Docker Resource Allocation work by Wu et al. [40] described in the paper "Dynamic CPU Allocation for Docker Containerized Mixed-Criticality Real-Time Systems". Their work proposes a CPU allocation approach called a flexible deferrable server (FDS) scheduler that improves the performance of a Docker containerized mixed-criticality real-time system (DC-MC-RTS). DC-MC-RT defines two different container types: RT-Containers and NRT-Containers, where an RT-Container type is a container containing the real-time application and an NRT-Container type is a container containing a non-real-time application. By allocating the CPU accurately, the timing constraints of RT-Containers can be met since their workloads are known in advance. However, it is difficult to provide performance guarantees to NRT-Containers since Docker uses static CPU allocation methods and the online workloads of NRT-Containers are unpredictable, and they might be changed significantly at the runtime. In particular, FDS first provides available CPU capacity to RT-Containers in order to ensure their timing constraints can be met. Then, the remaining CPU capacity is provided to NRT-Containers dynamically

at runtime to meet their unpredictable online requirements as much as possible. The proposed FDS has been implemented and made available in the Docker community edition 17.09.0 [84].

The dissertation work will focus on both types of virtualization: VMs and containers.

2.5.4 Cloud Computing Resource Selection and Cost

Running data-intensive jobs requires large-scale parallel processing engines that can only be run in a Cloud Computing environment; such a solution can be very costly. Optimizing the cost of running-data-intensive jobs in the Cloud Computing environment, known as cloud resource utilization, is very important research area, yet it is a relatively less explored problem [47].

Cloud service providers, such as Amazon AWS, Microsoft Azure, and IBM, allow users to outsource the hosting of applications and services to a cloud using clusters of VM instances. Cloud service providers charge a service usage cost to cloud service users. The service usage cost is based of the hourly usage rate for the VM instances running in the service provider cloud [47]. The Cloud service provider offers a predefined set of VMs, known as VM types.

Selecting the right VM type offered by a Cloud Service provider based on cost, CPU, RAM, and storage is a challenging task. For example, Microsoft Azure offers over 450 VM types for the user to choose from [46]; selecting the most optimal VM type is challenging.

If microservice architecture is adopted, which increases the number of components in the applications, the problem becomes even more challenging.

Here, we will highlight some of *the research questions* the dissertation addresses in Chapter 4.
For a complex containerized microservice architecture deployed in a public cloud, the following questions need to be answered:

1. Which VM type will provide the most cost-effective solution?
2. Which container cluster deployment will provide the best cloud VM resource utilization?
3. How many microservice container instances are required to support the incoming workload?

2.5.5 Cloud Computing Processing Engines

Expansion in Bigdata has given rise to computational challenges and created research opportunities for techniques and systems required to address them. The size of data has outgrown the capabilities of single machines, and users need new systems to be able to perform computations on multiple nodes. That triggered an explosion of new cluster programming models targeting diverse computing workloads. At first, the focus of those new models was relatively specialized in response to new workloads; for example, MapReduce supported batch processing, Google developed Dremel [49] for interactive SQL queries, and Pregel [50] was developed for iterative graph algorithms.

In the open source community, the Apache Hadoop stack, as well as systems like Storm [50] and Impala [88] are also specialized. Even in the relational database world, the trend has shifted from "one-size-fits-all" systems [51] to databases built for specific purposes.

In 2009, researchers at the University of California, Berkeley [51], concluded that most Bigdata applications need to combine many different processing types. They started the Spark project to design a unified engine for distributed data processing. They noted, that the nature of Bigdata is diverse and unstructured; a typical pipeline will need MapReduce-like code for data loading, SQL-like queries, and iterative machine learning. Specialized engines can thus be given rise to create both complexity and inefficiency; users must combine disparate systems, and some applications cannot be expressed efficiently in any single engine.

Spark has a programming model similar to MapReduce but extends it with a data-sharing abstraction called Resilient Distributed Datasets (RDD). With this extension, Spark can capture a wide range of processing workloads that previously needed different engines, including SQL, streaming, machine learning, and graph processing. In addition, even Spark is written initially in Scala, an object-oriented programming language; today, Spark supports development in four commonly used languages: Scala, Java, Python and R.

In 2013, the Spark project was donated to the Apache Software Foundation and switched its license to Apache 2.0, which enabled fast adoption in both, academic research circles and industry. In February 2014, Spark became a Top-Level Apache Project [31].

2.5.5.1 Real-Time Probabilistic Data Fusion for Large-Scale IoT Applications

This section highlights the state-of-the-art work of Akbar et al. [35]. We selected this work because it addresses real-time probabilistic data fusion in large-scale IoT applications. The work proposes a two-layer architecture for analyzing IoT data. The first layer provides a generic interface using a service-oriented gateway to ingest data from multiple interfaces and IoT systems. Data is stored in a scalable manner and analyzed in real-time to extract high-level events. The second layer is responsible for the probabilistic fusion of these high-level events. Furthermore, in the second layer, their solution processes collected data in the form of complex events, using Bayesian Networks (BN) to address the system uncertainty. Their proposed solution using open-source components is optimized for large-scale applications. They demonstrate their solution in a real-world use case in the domain of intelligent transportation systems, where they analyzed traffic, weather, and social media data streams from Madrid city to predict the probability of congestion in real-time [35].

Akbar et al. also researched the work of Cugola and Margara [11] in the Complex Event Reasoning (CER) domain, and they concluded that processing data in real-time often leads to the design of simplified solutions with no inherent support for handling uncertainty. According to Akbar et el., no work in the literature focuses on the computation of conditional probabilities for real-world problems to construct BNs with Complex Event Processing (CEP). They noted that as the complexity of the Bayesian inference process increases with an increase in the data size and the number of data sources, it is essential to propose a scalable and efficient solution [35].

The architecture proposed by Akbar et al. has two components: data collection and system analysis. The architecture is built from open-source components. The incoming stream data are collected as three independent data streams: Traffic data, Twitter, and weather network data. The traffic data are aggregated into data sets and pushed to the system for processing as an IoT service every 5 minutes. The Twitter data is available through an open API and are pushed into the system. The weather data used is an open API provided by Weather Underground.

The BN model proposed by Akbar et al. consists of five nodes: a) Time, which presents the time of the day, b) Weather, which presents the weather conditions, Large Crowd Concentration (LCC) Event, traffic congestion level, d) Day, presents weekdays or weekend, and e) Congestion, presents congestion level.

The work of Akbar et al. [35] is a great attempt and starting point in addressing the complex event uncertainty in Large-Scale IoT applications. The paper presents a data flow: from the data originator, various sensors, through data transport using Kafka to

data processing using Spark Engine [51]. However, the work is a simplified researched problem compared to LSDTNS objectives. The presented paper still does not address the following questions: 1) monitoring multiple critical Digital Twins simultaneously, 2) providing scalable architecture that allows easy CER code module maintainability, 3) having a multi-objective solution deployed in the cloud environment. In addition, the BN mode used in their research is oversimplified in comparison to the real BN models used in the industry.

2.5.5.2 OptEx: A Deadline-Aware Cost Optimization Model for Spark

This section highlights the work of Sidhanta et al. [47] that present OptEx: a novel deadline-aware cost optimization model for optimizing the cost of running data-intensive jobs in a Spark environment. The OptEx model estimates the cost-optimal cluster composition for running a given Spark job on a cloud under a completion deadline specified in the Service Level Objective (SLO). OptEx is a closed-form job execution model for the Apache Spark processing engine, and it is the first work that analytically models job completion time on Spark. The model can estimate the completion time of a given Spark job on a Cloud concerning the size of the input dataset, the number of iterations, and the number of nodes comprising the underlying cluster. Experimental results demonstrate that OptEx yields a mean relative error of 6% in estimating the job completion time [47]. Furthermore, experiments show that OptEx can correctly estimate the cost-optimal cluster composition for running a given Spark job for specified an SLO deadline. In addition, OptEx achieves an accuracy of 98% [47].

OptEx decomposes a Spark job execution into different phases: the initialization phase, the preparation phase, the variable sharing phase, and the computation phase. Then, following an analytical modelling approach, OptEx expresses the execution time of each phase in terms of the cluster size, number of iterations, the input dataset size, and specific model parameters.

Sidhanta et al. work [47] provides a novel deadline-aware cost optimization model for optimizing the cost of running data-intensive jobs in a Spark environment. However, the model does not address our requirements. We observe that the OptExt solution has not been tested on Spark streaming applications. OptEx solution was tested for simple Spark, non-streaming applications, such as WordCount, Sorting, and PageRank, by varying the input data size and the number of executors. In addition, even though the system is application deadline-aware, it is not a critical notification system. The proposed work is a different research problem than LSDTNS, and the following questions are still not ad-

dressed: 1) monitoring a large number of critical Digital Twins simultaneously, 2) providing a scalable architecture that allows easy CER code module maintainability.

2.5.5.3 dSpark: Deadline-base Resource Allocation for Bigdata Application in Apache Spark

This section highlights the work of Islam et al. [41], which presents dSpark, a lightweight, pluggable resource allocation framework for the Apache Spark processing framework. dSpark, the allocation framework, works from the master node along with an underlying cluster manager. The solution uses two models: a resource allocation model where a cost-effective, deadline based Resource Allocation Schema (RAS) can be built for an application and an application completion time prediction model that predicts the completion time of an application based on the number of executors and application properties.

Figure 2.4 shows a typical Apache Spark cluster. Applications are submitted through a cluster manager to run in the cluster. Spark comes with a default Standalone cluster manager or can be integrated with Apache Mesos or Hadoop Yarn cluster managers to allocate resources among applications. An Apache Spark cluster can be deployed on a cloud Virtual Machines (VM), then, to deploy an application in the cluster, a Resource Allocation Schema needs to be defined. dSpark provides a solution that is cost and application deadline aware: dSpak provides a cost-effective RAS that ensures that an application will be completed before the user-specified deadline [41].

The proposed dSpark Framework architecture contains two modules:

- *Profiler* is controlled by the Resource Allocator and generates the application profile for an application.
- *Resource Allocator* is the main component of the framework, that, based on the application information, user-specific deadline, and VM price produces the Resource Allocation Schema.

dSpark provides a cost and deadline aware resource allocation solution for deploying the multiple applications in Apache Spark. Similar to the OptExt solution, the dSpark solution does not address our requirement. dSpark has also been tested for simple Spark, non-streaming applications, such as WordCount, Sorting, and PageRank, by varying the input data size and number of executors, and not a streaming application.

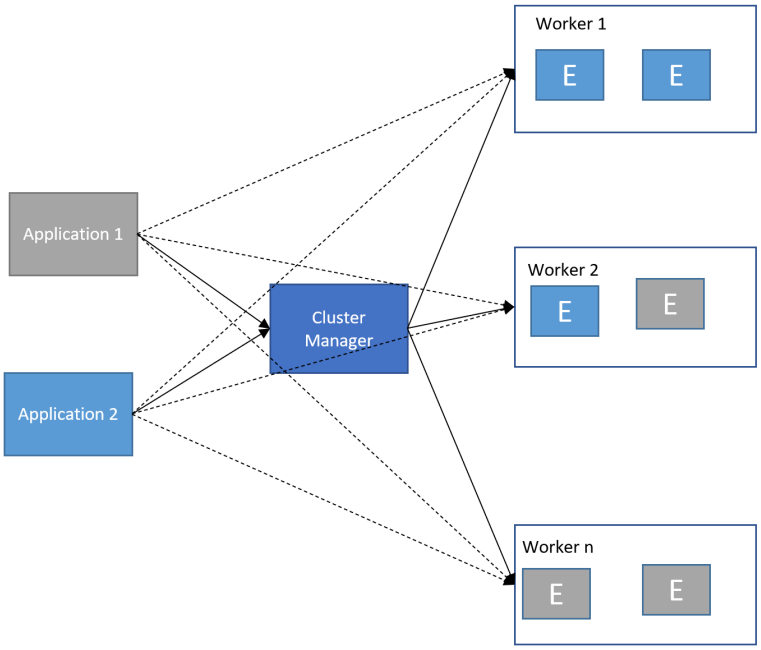


Figure 2.4: A typical Apache Spark cluster, shows two applications that need to be deployed in the Apache Spark cluster

After further Cloud computing processing engines state-of-the-art research, particularly Spark, we concluded that Spark and other processing engines, even though they are developed to process large-scale data, do not satisfy LSDTNS requirements: 1) monitoring a large number of critical Digital Twins simultaneously, 2) providing a scalable architecture that allows easy CER code module maintainability.

2.6 Architecture: Microservice vs Monolithic

Expansion in Bigdata volume also affects the way applications evolve.

Essential requirements for any complex system, which is even more critical for Bigdata systems, are that the system has to be maintainable and remain, over time, malleable to change. The way to achieve this is through modularity. Modularity requires that the source code for a given module should be separated from the code of the rest of the modules. A module also has to have well-defined interfaces for interaction with the other modules. Modularity ensures that the code is understandable, encapsulates functionality, and defines constraints on how different system parts interact [48]. Even for the traditional monolithic architecture, the modular approach is highly recommended.

Haywood [33] analyzed and summarized when a modular monolithic architecture should be used and when a microservice architecture should be deployed. He stated [33] that if an application has to have high scalability and modules are well-defined domains, and there is less concern about the complexity than the scalability, then, the microservices should be chosen over the modular monolithic architectures.

Figure 2.5 shows how scalability and domain complexity affect the decision of which architecture should be selected for a specific application. When selecting the architecture, the question should be asked: "What is most important to optimize in the application?". For example, if a domain is (relatively) simple or can be broken into smaller independent domains, but we have to achieve a high level of scalability, and the application is processing a high volume of data, the micro-service architecture is suitable. On the other hand, if the application domain is complex and expected volumes are bounded (for example, used only for an enterprise), then a modular monolithic architecture still makes more sense [33].

Since a Bigdata project requires working with *vast volumes of data*, and *scalability is critical*, the *micro-service architecture is recommended*. This dissertation research focuses on microservice architecture.

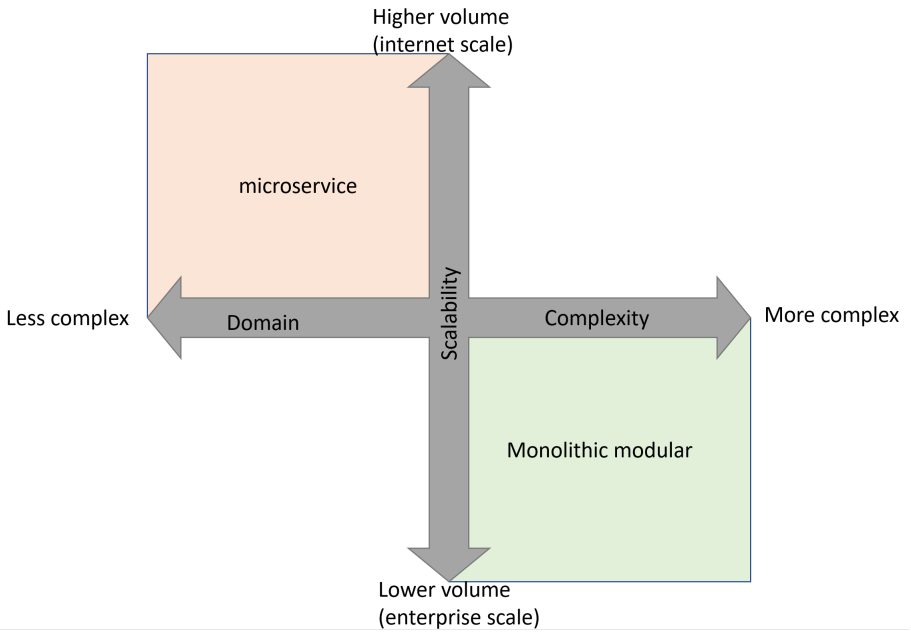


Figure 2.5: Scalability vs. Domain complexity

2.7 Multi-Objective Optimization

As previously mentioned in chapter 1.2.1, the dissertation focuses on solving the multi-objective optimization problem. Any multi-objective optimization problem involves making trade-offs between different objectives. Our focus is on the optimization of the following three objectives:

1. Minimizing the deployment cost
2. Maximizing resource utilization
3. Maximizing the number of jobs to be completed before the deadline.

Multi-objective problems involve trade-offs with respect to different performance metrics. In a multi-objective problem, performance is measured using multiple objectives metrics. Because of that, there is not a single optimal solution, but multiple optimal solutions [52]. For example, one optimal policy may prefer a low-cost but low percentage of jobs completed before the deadline, whereas another optimal policy prefers high-cost but a high percentage of jobs completed before the deadline.

In multi-objective problems, no single optimal policy maximizes all the objectives. Instead, there is a Pareto set (also known as Pareto front); a set of all Pareto optimal solutions that are non-dominated solutions. The non-dominated solution means that there is not another feasible solution better than the current one in some objective function without worsening other objective functions. Finding the optimal Pareto set is a complex problem and hard to achieve, so an approximation is often used.

Equation 2.4 formulates a multi-objective optimization problem [60], where m is the number of objectives, Π is the policy, and $f_i(\Pi)$ is the objective function i:

$$\max_{\Pi} F(\Pi) = \max_{\Pi}[f_1(\Pi), f_2(\Pi), ...f_m(\Pi)], \tag{2.4}$$

Definition: Pareto optimal [60]:
We say policy Π dominates policy Π' if $F(\Pi) \geq F(\Pi')$ and $F(\Pi) \neq F(\Pi')$. A policy Π is Pareto optimal, if, and only if, it is not dominated by any other policy. The set of all such policies is called a Pareto set, and the image of the Pareto set is the objective space, called Pareto front.

Multi-objective optimization research emerges in two research directions:

- *Heuristic techniques.* Some of the multi-objective heuristics solutions are:
  - The work of Frincu et al. [53] is an example of a meta-heuristics scheduling approach in the multi-cloud resource provisioning environment. The multi-objective solution optimizes the following: deployment and run-time cost of the solution, resource load, and application availability in case of failures. The solution researches the Web 2.0 application. The presented solution architecture in the paper does not use containerized micro-services architecture, and they system it is not a critical notification system, and the deadline is not the required objective.
  - The work of Legillon et al. [54] is also an example of meta-heuristics methodology that proposes a model that addresses placing services into Infrastructure as a Service (IaaS) virtual machines from multi cloud providers. The objective of their work is cost minimization. Same as in the previous example, the solution architecture is not containerized micro-services, it is not a critical notification system, and the deadline is not an objective.

  - Guerrero et al., 2017 [55] propose a genetic algorithm approach using the non-dominated Sorting Algorithm II (NSGA-II) to optimize container allocation and elasticity management. The proposed solution is multi-objective and enhances system provisioning, system performance, system failure and network overhead.
  - Guerrero et al., 2018 [56] propose an approach to optimize the deployment of microservice based applications using containers in a multi-cloud architecture. The solution has three multi-optimization objectives: cloud service cost, network latency among microservices, and time to start a new microservice when a provider becomes unavailable.
  - Fan et al. [57] research on microservice scheduling in edge computing. The solution has three multi-optimization objectives: network latency, reliability, and load balancing.
  - Xu et al. [72] formulate the energy efficiency virtual resource allocation for cloud computing as a multi-objective optimization problem (decreasing the number of used server, increasing total dynamic power and increasing utilization of servers). Their researched constraints functions represent the potential problems when VMs are scheduled across different Physical Machines (PMs).
  - Chen [62] proposes a cloud resource allocation method supporting sudden and urgent resource demands. The proposed solution can allocate various resources timely for urgent resource demands. The proposed multi-objective mathematical model considers the minimum performance match distance between virtual machines and physical machines, and the minimum number of physical machines as resource allocation goals.

- *Reinforcement learning.* Some of the multi-objective RL solutions are:
  - Yang et al., 2019 [60] present a new algorithm for multi-objective reinforcement learning (MORL) with linear preferences. The goal of linear preference is to enable few short adaptations to the new task. The MORL aims to learn policies over multiple competing objectives whose preferences are unknown to the RL agent. The proposed solution does not consider micro-service architecture nor deployment in the cloud environment. The work is focused on improving MOO algorithm and exploring RL.
  - Chen et al. [58] present a Multi-Objective Reinforcement Learning (MORL) framework that aims to approximate the Pareto frontier uniformly. The framework has two stages. The soft actor-critic (SAC) algorithm is executed in the first stage to a multi-policy soft actor-critic (MPSAC) algorithm. In the second

stage, the multi–objective covariance matrix adaptation evolution strategy (MO-CMA-ES) is applied to fine-tune the policy-independent parameters. Similar to the previous RL example, the proposed solution does not consider micro-service architecture nor deployment in the cloud environment. The work is focused on improving the MOO algorithm and exploring RL.

- Tian et al. [59] propose an efficient evolutionary learning algorithm to find the Pareto set approximation for a continuous robot control problem. The solution extends the state-of-the-art RL algorithm and presents a novel prediction model to guide the learning process. The work is focused on improving MOO algorithm and exploring RL for continuous robot control.
- The work of Rjoub et al. [71] proposes deep and reinforcement learning-based scheduling approaches to automate the scheduling of large-scale workloads onto cloud computing resources, while reducing resource consumption and task wait times.

Furthermore, research objectives also greatly vary. Some of the research objectives are: network latency [56] [57], high availability [53] and reliability [57], and system performance and system failure [55].

Table 2.1 provides a summary and comparison of various state-of-the-art papers focusing on multi-objective optimization. The table evaluates different characteristics relevant to the requirements of the LSDTNS:

1. *Cloud Containerized Deployment*: Indicates whether the solution employs cloud containerization for deployment.
2. *Micro-Service Observation*: Indicates whether the solution is built using a microservices architecture.
3. *Critical Notification System with Deadline Objective*: Indicates whether the system is designed for a critical notification scenario where meeting deadlines is one of the objectives.

Additionally, the table captures the following aspects:

1. *Multi-Objective Criteria*: Specifies the various objectives the system is built to optimize.

2. *Application Types*: An application type used to validate and demonstrate the effectiveness of the proposed solutions.

The works of Guerrero et al. [55] [56], and Fan et al. [57] are the most similar to our approach, although there are significant differences. All three papers research multi-objective optimization for containerized microservice applications, which is similar to our work. However, all three works focus on different objectives than our research focus. Our research focuses on minimizing the deployment cost, maximizing resource utilization, and maximizing the number of job requests to be completed before the deadline. In addition, the applications observed in Guerrero et al. and Fen et al. works are not critical notification system applications, and, therefore, a new research focus is needed. As previously mentioned, selecting the most optimized Cloud Service provider VM type for containerized microservice application is a challenging task; none of the previously referenced works are trying to answer how to select the best VM type candidates based on multi-objectives and expected load in a large-scale containerized microservice applications.

2.8 Scheduling

Scheduling jobs in cloud computing is a critical task that ensures the efficient utilization of available resources while meeting the requirements and objectives of users. As previously mentioned, LSDTNS system has to fulfill the following multi-optimization objective requirements: a) minimize the deployment cost, b) maximize resource utilization and c) maximize the number of jobs to be completed before the deadline.

Definition: *timing parameters associated with processing a job*,
The timing parameters presented below are used in the dissertation as well as in some state-of-the-art work that will be presented later in the text [76]:

1. $arrival\_time$ – the time at which the process arrives in the ready queue.
2. $completetion\_time$ – the time at which the process completes its execution.
3. $burst\_time$ – time required by the process for CPU execution.
4. $turn\_around\_time$ - time interval from the $arrival\_time$ to the $completetion\_time$.
5. $waiting\_time$ - the time difference between $turn\_around\_time$ and $burst\_time$.

Well-known *CPU Scheduling algorithms* that will be used for the evaluation are [76]:

1. *First Come First Server* ($FCFS$) – an algorithm that schedules jobs in the same order they arrived.
2. *Shortest Job First* (SJB) – an algorithm for which the process (job) with the shortest burst time is scheduled first.
3. *Longest Job First* (LJF) – an algorithm for which the process (job) with the longest burst time is scheduled first.

2.8.0.1 Scheduling containers: state-of-the-art

A recent comprehensive survey [64], published by Ahmad et al. in 2021, focuses on container scheduling techniques. They noted that container scheduling can take different forms depending on the underlying technology. For example, depending on the implementation details, the incoming task can be scheduled directly on a container running on the physical machine (hardware) or on a virtual machine (VM) running on the physical machine (hardware). They also noted that container scheduling has become critical for the cost-efficient operation of modern applications on the cloud due to the diverse nature of the workload and available cloud resources.

A container scheduling problem is an NP-hard problem. There is no polynomial time complexity algorithm to find optimal scheduling for large-size problems. The majority of work examined in the survey [64] fit into one of four categories:

- *Mathematical modeling* is suitable for small-sized problems. Mathematical modeling techniques represent scheduling problems as a collection of equations with specific constraints. One commonly used approach is Integer Linear Programming (ILP), where variables are assigned integer values and linear equations are formulated to represent the problem's constraints [64].
- *Heuristics* algorithms are the majority of solutions, and they are generally of low complexity and generate satisfactory schedules in a reasonable amount of time [64].

- *Meta-heuristics* are population-based optimization algorithms inside the intelligence processes and behaviours arising in nature [64].
- *Machine Learning* algorithms and techniques have not been fully explored for container scheduling [64].

Other work focuses on Kubernetes, providing Kubernetes Scheduling Framework [36] [37] or proposing a value-based market allocation heuristic model for Dockers [14].

2.8.0.2 Scheduling and RL

As previously mentioned, machine learning algorithms and techniques have not been fully explored for container scheduling [64]; further in this section, we will explore the related state-of-the-art in the domain of scheduling and RL. Unfortunately, none of them does investigate scheduling of jobs on containers using RL techniques.

Raub et al. [69] present BigTrustScheduling that is a trust-aware Bigdata task scheduling approach in a cloud computing environment. BigTrustScheduling consists of three stages: 1) VMs' trust level computation, 2) task priority level determination, and 3) trust-aware scheduling. The system used in this research is a Hadoop cluster environment, which is an architecturally different environment from the containerized architecture used in this dissertation. Furthermore, the objectives of the BigTrustScheduling (Bigdata performance and makespean cost) are different from LSDTNS objectives (resource utilization, deadline and resource cost).

Rjoub et al. [70] present Deep Smart Scheduling (DSS) that automatically predicts the Virtual Machines (VMs) to which each incoming big data task should be scheduled; the objectives are to improve the performance of big data analytics and reduce their resource execution cost. DSS combines Deep Reinforcement Learning (DRL) and Long Short-Term Memory (LDTM). The system used in this research is a Hadoop cluster environment and, as such, can not be used in the environment with containerized architecture.

The work of Rajob et al. [71] continues the previous work [70]. It proposes four deep reinforcement learning-based scheduling approaches to automate the process of scheduling large-scale workloads onto cloud computing resources, where the objectives are to reduce resource consumption and task waiting times. The system used in this research is also a Hadoop cluster environment.

Mao et al. [65] present DeepRM, an example solution that translates the problem of packing tasks with multiple resource demands into a learning problem. Their results show

that DeepRM compares to state-of-the-art heuristics. Furthermore, DeepRM adapts to different conditions, converges and learns strategy quickly. They stated that resource management problems are ubiquitous in computer systems and networks. Examples include job scheduling in computing clusters, bitrate adaptation in video streaming, relay selection in Internet telephony, virtual machine placement in cloud computing, congestion control, etc. They also stated that the majority of these problems are solved today using methodically designed heuristics. Furthermore, they suggest that the typical design flow consists of two phases. In phase 1, the designer comes up with clever heuristics for a simplified model of the problem, and after that, in phase 2, the designer manually tests and tunes the heuristics until a satisfying performance are achieved. This process is long and often has to be repeated multiple times.

The DeepRM leverages the Deep Reinforcement Learning (DRL) paradigm. The RL Agent presented in the DeepRM RL system is a policy built using Deep Neural Network (DNN).

Further improvements to the DeepRM solution have been provided by Ye et al. [66]. They introduce the online resource scheduling algorithm DeepRM2 and the offline resource scheduling algorithm DeepRM-Off. Compared to the state-of-the-art DRL algorithm, DeepRM, and heuristic algorithms, their proposed algorithms have higher convergence speed and better scheduling efficiency regarding average slowdown time, job completion time, and rewards.

DeepRM and DeepRM2 significantly contribute to the resource allocation domain using reinforcement learning. In our solution, we use DeepRM ideas and build on top of them. Our objective is different from DeepRM and DeepRM2. Instead of focusing on the average slowdown time, we are more interested in average job completion based on job priority, where our solution favours the most critical jobs first. There are also a few critical differences: Our architecture is a containerized microservice architecture running in a Cloud on VM instances, whereas their software is deployed directly on the hardware. Furthermore, they only observe a single objective, such as average slow down, and our system requires multi-objective consideration.

All presented solutions have in common: they did not evaluate microservices applications deployed in the cloud containers and had not observed the system job completion deadline as the system requirement.

Table 2.2 summarizes and compares previously mentioned state-of-the-art scheduling solutions based on DRL. The comparison table focuses on the following the LSDTNS requirements:a) Is the provided solution deployed as a cloud-containerized application? b) Is the observed application designed as a microservice? c) Is the system designed for

critical notification purposes, with meeting deadlines as one of its objectives? Since our scheduler is a multi-objective scheduler, we also observe the multi-objective criteria the system is built for, and the type of applications used to verify the solution.

Table 2.2 offers a comprehensive summary and comparison of recent state-of-the-art research papers that focus on scheduling through Deep Reinforcement Learning (DRL). This table assesses the following characteristics:

1. *Cloud Containerized Deployment*: This column indicates whether the solution utilizes cloud containerization for its deployment.
2. *Micro-Service Observation*: This column highlights whether the solution employs a microservices architecture.
3. *Critical Notification System with Deadline Objective*: This column denotes whether the system is designed for a critical notification scenario where meeting deadlines is a primary objective.

In addition to these key points, the table also covers the following aspects:

1. *Multi-Objective Criteria*: This column specifies the various objectives the system is built to optimize.
2. *Application Types*: This column details the types of applications used for validation and demonstration of the proposed solutions.

Table 2.1: Multi-objective optimization: comparison with the state-of-the-art

ref	contain.	ms	optim. objectives	notif. app.	application	method	Note
[57]	yes	no	network latency reliability load balancing	no	SockShop	heuristics	Edge computing
[55]	yes	yes	system performance system failure network overhead	no	SockShop	heuristics	Single cloud
[56]	yes	yes	cost network latency time to start new ms	no	SockShop	heuristics	Multi cloud
[53]	no	no	high availability fault tolerance app. cost	no	Web 3.0	heuristics	Multi cloud
[54]	no	no	cost minimization fault tolerance app. cost	no	Web 3.0	heuristics	Multi cloud
[72]	no	no	number of used server dynamic power servers' utilization	no	C++ sim.	heuristics	deployment VMs-PMs
[62]	no	no	distance VM-PM min. number of PMs u	no	CloudSim	heuristics	deployment VMs-PMs
[60]	no	no	not specified	no	DST, FTN	DRL	Generalized algorithm
[58]	no	no	high speed low cost	no	DST, FTN	DRL	Dynamic MORL
[59]	no	no	not specified	no	DST, FTN	DRL	DRL Framework
[71]	no	no	resource consumption task wait time	no	Google Cluster	DRL	process automation

Table 2.2: Scheduling: Comparison of the state-of-the-art

ref	contain.	ms	optim. objectives	notif. app.	application	method	Note
[69]	no	no	bigdata performance makespean cost	no	Google cluster	DRL	Hadoop
[71]	no	no	min. task exe. delay resource utilization	no	Google cluster	DRL	Hadoop
[70]	no	no	cost bigdata performance	no	Google	DRL	Hadoop
[65]	no	no	single optimization objective	no	simulated jobs	DRL	deployment on HW
[66]	no	no	single optimization objective	no	simulated jobs	DRL	deployment on HW

Chapter 3

IoT Notification System Architecture

This chapter of the dissertation proposes a micro-service-based notification methodology using complex event recognition to handle the uncertainty of *LSDTNS*. Note that this proposed solution can be used in any other IoT Notification System.

Part of the work presented in this chapter has been published in two papers:

1. "Complex Event Recognition Notification Methodology for Uncertain IoT Systems Based on Micro-Service Architecture," The IEEE 6th International Conference on Future Internet of Things and Cloud, FiCloud 2018 [2].
2. "A Performance-Driven Micro Services-Based Architecture/System for Analyzing Noisy IoT Data," 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2019 [3].

This chapter addresses the RQ1 research questions discussed in Section 1.2.4 and presents a proposed micro-service-based notification methodology using complex event recognition to handle the uncertainty of *IoT* systems. Our present methodology can address the scalability issue in CER systems, and still keep the flexibility to select from various Complex Event Recognition solutions.

Section 3.1 presents the system overview. Section 3.2 discusses the IoT Notification system architecture, where the following is covered:

- Subsection 3.2.1 discusses IoT Notification system modules,
- Subsection 3.2.2 discusses IoT Complex Event Recognition modules,

- Subsection 3.2.3 discusses IoT Controller Module Functionality,
- Subsection 3.2.4 discusses the IoT Notification System workflow,
- Subsection 3.2.5 discusses the LSDTNS components and their interactions,
- Finally, Subsection 3.2.6 discusses the components of IoT Notification systems.

Section 3.3 demonstrates the feasibility of the proposed IoT Notification system architecture through the Case Study: Prisoner Use Case. Section 3.4 provides discussion.

3.1 System Overview

This dissertation focuses on a digital representation of a physical *IoT* system. The *IoT* system is one of several systems that operate by observing a set of primitive events that happen in external environments, interpreting and combining them to identify a higher level of *Complex Events (CE)* [2]. The physical system consists of *n Physical Monitored Objects (PMO)*. A *PMO* can be, for example, an older adult whose vital signs are monitored or a prisoner in a prison cell where the detection of a suicide attempt is monitored for prevention. Each *PMO* in a digital world is presented as a *DT*, and each is equipped with the same set of m primitive sensor types. Sensors data is collected and processed, which defines a *DT*.

The *IoT* system monitors multiple primitive *Events (E)* in specified time intervals, which generates a sequential array of events known as an *Event Stream (ES)*. An *E* consists of sensor value v, event type *type*, and the time occurrence t (Equation 3.1). The *CE* is a complex relationship of primitive events and can be presented as a function of n primitive events in a time interval ΔT (Equation 3.2).

$$E_i(v_i, type_i, t_i), where\ i = (1, ..., n) \tag{3.1}$$

$$CE = f_{CE}(E_1, E_2, ...E_n) \tag{3.2}$$

$$CEC = f_{CEC}(E_1, E_2, ...E_n) \tag{3.3}$$

An essential part of the observed system is recognizing complex events that match a given set of *Complex Event Criteria (CEC)*, where *CEC* is a static function of n events (Equation 3.3). In the prisoner use case, the *IoT* system monitors the activities of prisoners

in order to detect a prisoner's suicide attempt. A suicidal attempt presents a critical state and high risk for the prisoner that the *IoT system* must monitor. A prisoner cell is equipped with three sensors: an activity sensor that detects the prisoner's activity (moves / does not move), a position sensor that detects the prisoner's posture (sit, stand, lie down) and a breathing sensor that detects when the prisoner breathing (yes/no). The system has to detect the prisoner's critical state to prevent death and then notify when the critical state is detected. Critical state detection is recognized with *CEC* (if the prisoner is inactive, lying down and not breathing). The *DT* 3.4 consists of a unique entity id, for example, prison cell id, and a list of primitive *Event Streams (ES)* (for example, breathing event stream, activity event stream and posture event stream) that are monitored by the *IoT* system.

$$DT(entity\_Id, List < ES >) \tag{3.4}$$

The IoT Notification System consists of *n Digital Twins (DT)*. Each DT has the same set of m sensors that generate streams of primitive data. Each sensor Sk, where $k = 1, ..m$, streams data in equal intervals ΔTk. Please note that sampling of each sensor type Sk can be different sampling rates. In addition, the primitive sensors data is aggregated into a Complex Event (CE) in the time interval ΔT - see Fig. 3.1. Each DT_i will have a corresponding CE_i (the relationship between DT and CE is 1:1).

We can also recognize the IoT Notification System as a *Notification Monitoring System (NMS)*, where *NMS* is a system that operates by observing a set of primitive events that happen in the external environment, interpreting and combining them to identify a higher level of complex events [3]. Once the system recognizes complex events that match the predefined set of *CEC*, the system sends a notification message, also known as an alarm, to a previously predefined recipient. If the notification message is critical, such as protecting someone's life, it needs immediate action. Such a system, where the notification message has critical importance, is a Critical Notification System (CNS). Figure 1.4 shows an example of the *LSDTNS*.

Additionally, the time from producing the *CE* to sending an alarm has to be less than *deadline D*, which is the system's predefined parameter. D presents the latest time the system should respond with a *Notification Event (NE)*.

The *NMS* rarely monitors only one DT; it is more likely to watch thousands of monitoring entities. Therefore, the system has to be able to scale and process a large number of monitoring entities. An essential characteristic of the system is that it monitors the DT for a predefined critical state, and it is responsible, once it detects such a state, for sending a *Notification Event (NE)* to a predefined recipient.

Figure 3.1: An example of CNS system consisting of n DTs, where each DT composes a CE. The CNS monitors to recognize when CEC is met and, once that occurs, sends NE to a predefined recipient for further processing.

3.2 IoT Notification System Architecture

3.2.1 IoT Notification System Modules

This section presents and explains *IoT Notification methodology modules*. As previously defined in the Research Challenges 1.2.1, the system should be modular for a complex system to be maintainable and malleable to change. The system is modular when the source code is organized into logical modules. Furthermore, a module is required to have well-defined interfaces for interaction with the other modules. Modularity ensures that the code is understandable and functionality is well encapsulated. The modular software defines how parts of the system interact [33].

LSDTNS requires a modular solution; the system has to incorporate multiple modules from different complex domains into a single solution (for example, rule-based complex event processing, probabilistic programming and streaming).

LSDTNS requires the following components (see Figure 3.2):

1. A Streaming module,
2. An IoT Controller module,
3. A Complex Event Recognition module,

4. A Notification module.

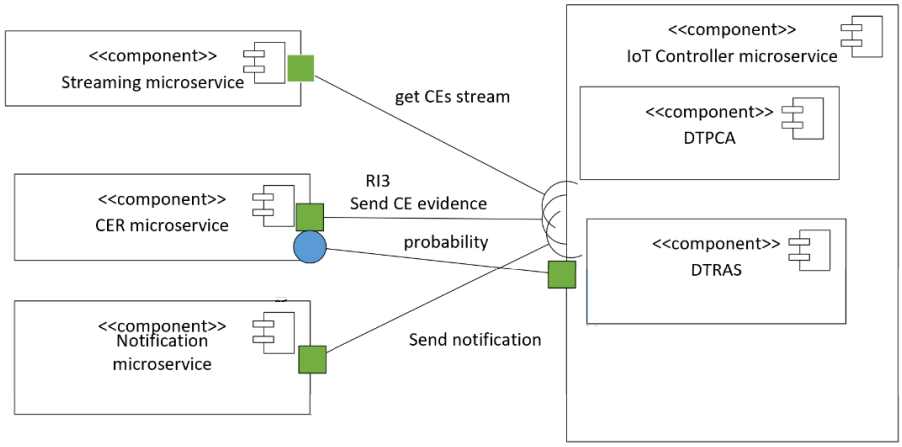


Figure 3.2: LSDTNS Components

The purpose of the *Streaming module* is to prepare raw data for the *Complex Event Recognition (CER)* process and to stream data to one of the monitoring components by performing either coarse filtering in the rule-based complex event module or fine processing in the probabilistic programming module.

The purpose of the *IoT Controller module* is to manage and coordinate other modules in the system. The Controller is the "brain" of the system.

The purpose of the *Complex Event Recognition (CER)* module is to consider the system uncertainty and provide a probability of occurrence for the *CE*. In addition, the module should be able to provide confidence intervals, which are obtained from the variance of the posterior distribution. Further in this document, the *CER* module will be covered in more detail.

The purpose of the *Notification module* is to provide a different type of notification mechanism that the controller module will use to notify clients about the monitored complex event. Based on the probability of the complex event occurrence received from the complex event recognition module, the Controller will decide which notification mechanism type is provided in the notification module to use. Therefore, the notification module is only a provider of different notification mechanisms.

3.2.2 The Complex Event Recognition Module

Uncertainty in *IoT systems* is always present at some level, as shown in Section 2.1.1. The uncertainty in *IoT systems* is a complex problem that must be considered. The Complex Event Recognition module is a crucial component that addresses this complex uncertainty problem in the *LSDTNS*.

The complex event recognition component can be either rule-based complex event recognition or use probabilistic programming, as has been covered previously in Section 2.1.1, Uncertainty in IoT systems. As presented in Section 2.1.1.1, state-of-the-art in Complex Event Recognition and the Dynamic Bayesian network, which are part of the probabilistic programming paradigm, provide an appropriate way of modeling. Probabilistic programming is a paradigm that unifies general-purpose programming with probabilistic modeling. The primary objective is to specify a probabilistic model and perform inference on it. The probabilistic programming field contributes many different algorithms. Some algorithms require less or more computational power and less or more time to estimate posterior distribution. Since selecting suitable algorithms is a domain-specific question, it should be left to the LSDTNS domain designer to select the appropriate algorithms for a specific domain problem. In addition, rule-based complex events and probabilistic programming are fast-growing fields. With all the ongoing effort from open-source communities and commercial enterprises, one can expect continuous solution enhancements (e.g. new probabilistic programming or rule-based reasoning algorithms). Considering that probabilistic programming and rule-based complex event recognition fields are advancing fast, we advise that the component is built so that the implementation can be easily adapted to use a new and better solution without affecting the integration with other modules in the system.

Furthermore, each monitoring object has additional information for the monitoring entity; we will call it the *DT Profile (profile)*. For example, in prison, a *DT* profile can contain additional information on a monitored prisoner: the length of the prisoner's sentence, the medical condition of the prisoner, the physiological state of the prisoner, and similar additional information. Based on the profile, we can determine the *critical state* of the monitored *DT*. The *LSDTNS* component, the *IoT controller*, computes a *DT* processing priority for each monitoring entity.

3.2.3 IoT Controller Module Functionality

The purpose of the *IoT controller module* is to manage and orchestrate other modules in the system. During the system development phase, a list of k predefined *DT* resource

types for the system is created. Each *DT* resource type is mapped with a corresponding containerized module type, encapsulating a complex event recognition technique and/or algorithm. As previously mentioned, selecting the complex event recognition algorithm is a domain-specific problem, and a domain expert will perform the algorithm selection task. The domain expert will select the appropriate algorithm for the corresponding *DT* resource type based on the domain requirements.

The *Digital Twin Priority Controller Algorithm (DTPCA)* and the *Digital Twin Resource Allocation Scheduler (DTRAS)* are subcomponents of the IoT controller module, and we will briefly explain them.

DTPCA calculates the *DTs* priority using the following data: a) a *DT* profile, b) historical *DT* priority data, and c) the current *DT CE* data. Research on DTPCA is an optional component and will not be covered in this dissertation but in future work.

The *Digital Twin Resource Allocation Scheduler (DTRAS)* has to schedule *n DT complex event (CE)* probabilistic processes on a cloud cluster. The objective of *DTRAS* is to perform scheduling so that processing priority, which can be one of *k* predefined *DT* resource types, is given to the higher priority processes first, and so on. The additional requirement is that processes should be completed before the specified system *deadline D*. *DTRAS* will be covered in depth in Chapter 5: Digital Twin Resource Allocation Scheduler (DTRAS).

Table 3.1 shows the system parameter values and two subgroups of the solution. The first group of *DT modules* presents all *DT Controller elements*, and the second subgroup presents a Cloud computer resource where the processing will be deployed.

Table 3.1: DT System parameters

Size	System variable	System subgroup
n	Digital Twins (DT)	DT controller elements
m	Sensors per each DT	DT controller elements
k	Preselected DT priority resources	DT controller elements
c	Cloud clusters	Cloud computing resource
cc	Number of containers in a cluster	Cloud computing resource

3.2.4 IoT Notification System Workflow

This section presents the *IoT Notification system components interaction workflow*. The proposed workflow considers that the IoT system will consist of many digital twins, where

each digital twin will collect the same type of primitive data. As a result, some parts of the workflow will be repeated for each digital twin. The workflow is presented in Figure 3.3 and is explained below:

1. Once the DTPCA calculates and assigns priority types to all n DTs, the Controller will request Kubernetes to start *cc* pods (aka Docker containers) that run a Complex Event Recognition Module.

2. Once all *cc* containers are started, the system is ready to process incoming CE streams from the streaming module.

3. Further steps present time-critical real-time system processing, and the steps will run continuously. Those steps are repeated for each DT_i, where $i = 1, \ldots, n$:

   (a) The streaming module aggregates m individual primitive events for each DT_i into DT_i complex event (CE_i), where $i = 1, ..n$.

   (b) IoT Controller redirects the CE_i stream for DT_i to the assigned DT_i container instance for processing.

   (c) The Digital Twin Resource Allocation Scheduler schedules incoming n DTs' CEs based on priority p of the Kubernetes pods.

   (d) Once all incoming data is processed, the priority is assigned to each DT.

   (e) The notification module will be invoked if the priority is equal to or higher than notification threshold P.

   (f) This step presents a DTPCA priority update and will be invoked when the following events occur:

   - The DT's profile has been changed.
   - The DT's new expected probability is changed.
   - Scheduled update.
   - Manually requested updates.

3.2.5 LSDTNS Components and Their Interactions

This section shows *LSDTNS components*, selected technologies and interaction between components. As previously described, our system is modular, and each module encapsulates a logical, functional modular unit. The following modules were described in the previous section:

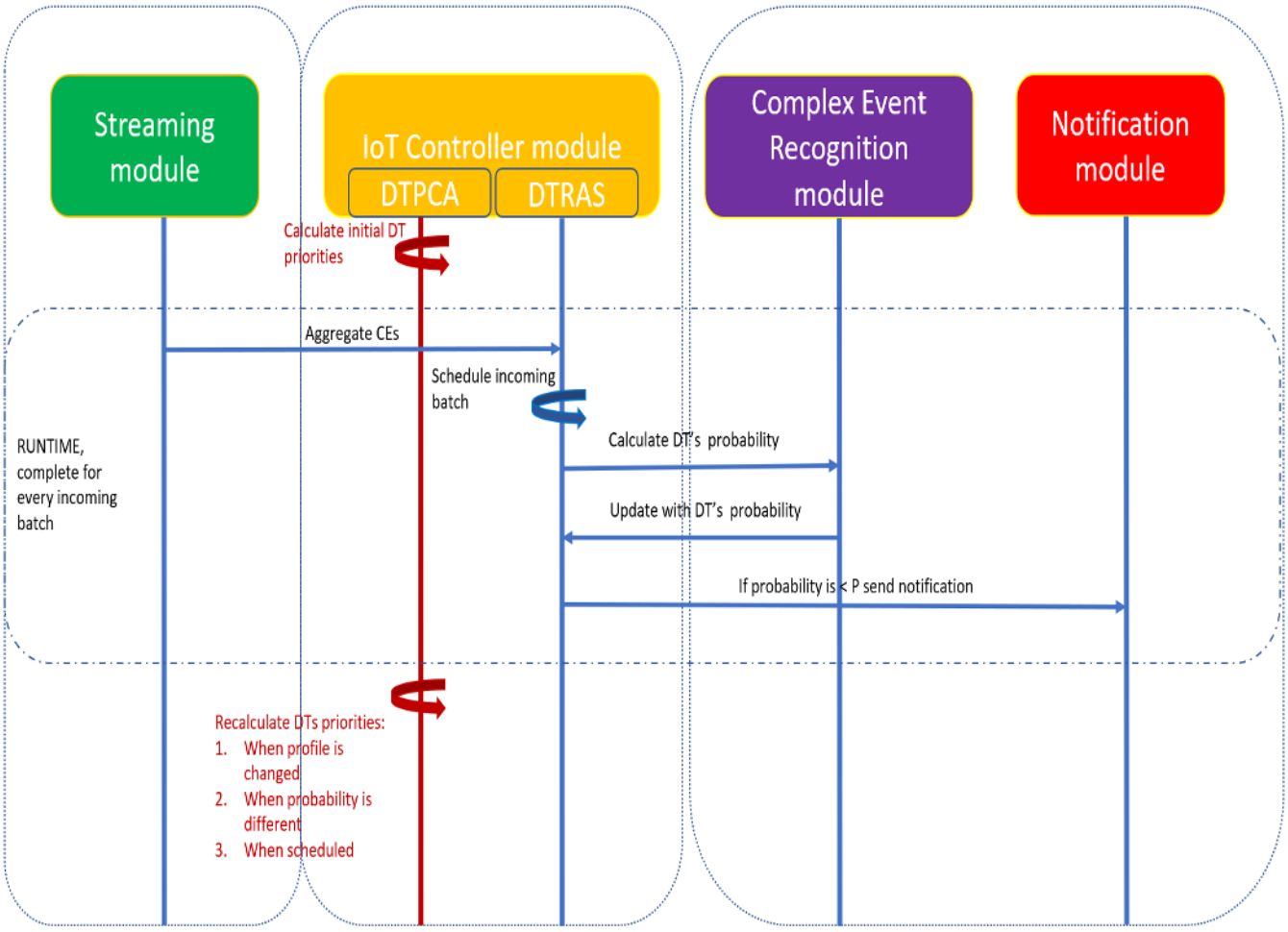


Figure 3.3: IoT Notification System methodology workflow

1. Streaming module,
2. IoT Controller module,
3. Complex Event Recognition module,
4. Notification module.

Furthermore, the section explains why micro-service architecture and REST APIs are selected.

In his work, Haywood analyzed and summarized when *modular monolithic architecture* should be used vs. *micro-service architecture* [33]. It is stated in [33] that if an application has high scalability, modules are well-defined domains, and there is less concern about the complexity than the scalability. Therefore, the *micro-services* should be chosen over the *modular monolithic architectures*.

The IoT notification system needs to process continuous streams of data. The volume of streamed data is in order of n, where n is the number of DT. Therefore, a highly scalable solution is critically required to handle such data volume effectively. Our modules, presented in Section 3.2.1 IoT Notification Methodology modules, are well-defined domains where Domain expertise is required for implementation (for example, rule-based complex event processing and probabilistic programming).

Using the *micro-service* architectural approach requires well-defined *micro-service interfaces*, which allows the refactoring and future modifications of each component for handling technology enhancements and system requirements enhancements over time without interfering with the existing implementations of other *micro-services*. For example, *rule-based complex event* domain, as well as *probabilistic programming*, are fast-growing domains. Open-source communities and commercial enterprises provide continuous improvements in the form of new inference algorithms implemented in probabilistic programming. The system can replace components with the latest improvements using the micro-services approach without interfering with the rest of the system.

An *application program interface (API)* is a set of routines, protocols, and tools for building software applications. An *API* specifies how software components should interact. An *API* can be internal (only known internally in the component) or publicly available, as an external *API*. The publicly available *APIs* should be well documented, and those *APIs* should not change over time. In rare cases, an *API* is deprecated.

Many *API* methodologies can be used for intercommunication. However, based on many years of software development trends, the *REST API* has been selected for the

dissertation proposal. One example of trending is the leading public cloud computing solution, Microsoft Azure, where all Azure App services are RESTful, and the body content is in JSON format [44].

3.2.5.1 Data Model

This section shows the data model used for the solution—the following Figure 3.4 shows model objects. The model contains the following classes:

1. *Complex Event* class encapsulates information about a complex event and consists of the following:
   (a) an id, a unique CE identifier
   (b) a list of events
   (c) a date, presents a time stamp of when the complex event occurred.
2. *Event* class encapsulates event information and consists of a) an id, a unique Event identifier; b) event type, for example, temperature, oxygen; c) a value that corresponding to the sensor value.
3. *Event Type* is an enumeration class that defines a list of Events in the system.
4. *Priority* is an enumeration class that defines a list of priorities in the system.
5. *The Historical Priority* class provides a list of previous priorities for the observed DT.
6. *Digital Twin* class presents a DT in the system and has the following parameters:
   (a) id that is a unique identifier
   (b) priority, which presents the current priority of the DT
7. *Digital Twin Info* class provides additional information about the DT

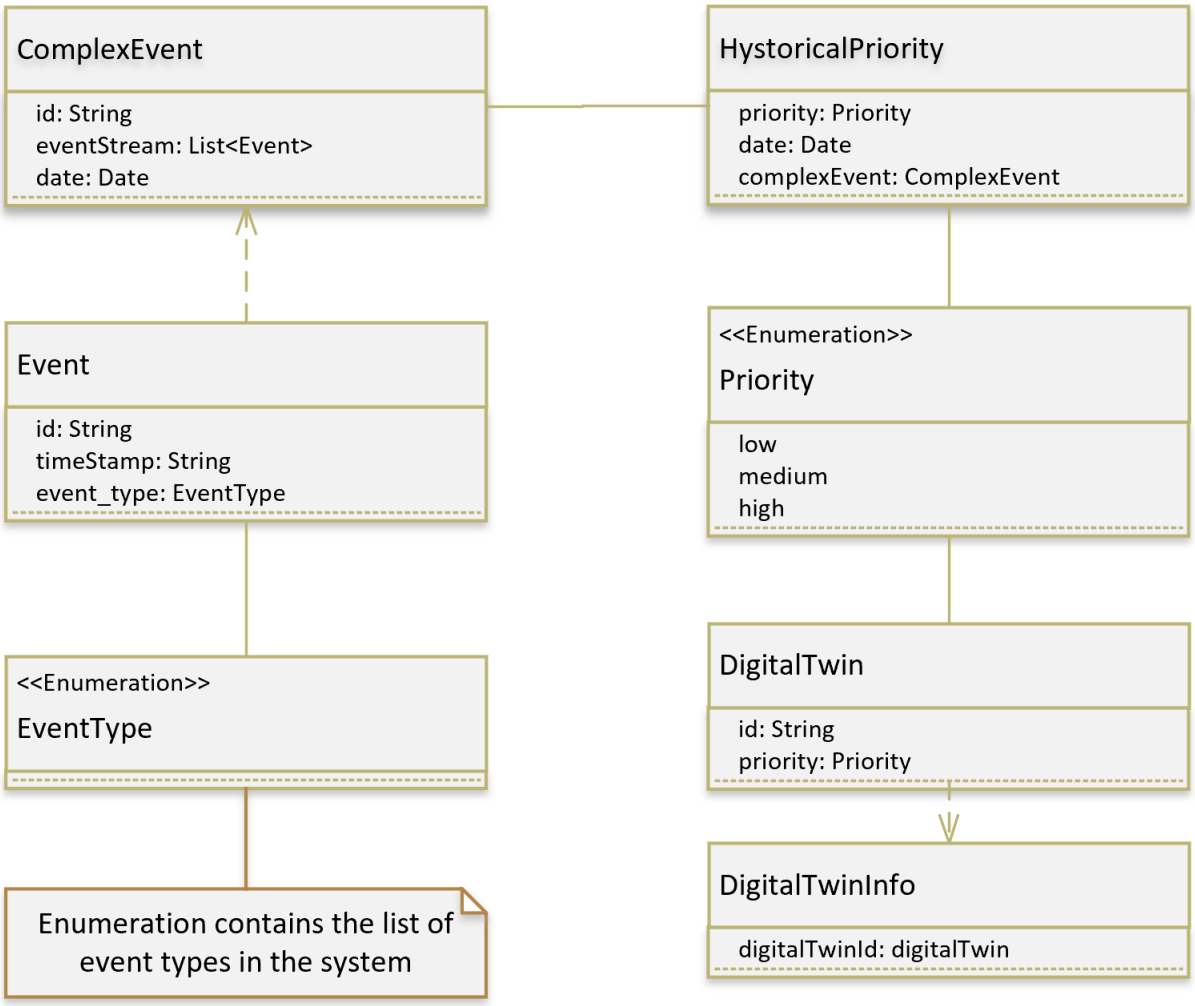


Figure 3.4: Partial Data model

3.2.5.2 LSDTNS Micro-services and APIs

This section explains *LSDTNS micro-services*, their *Representational State Transfer (REST) APIs* and the interaction between micro-services. It is important to emphasize that inter-communication between micro-services is done solely over external micro-services APIs.

The *REST API* is domain-independent, and, as such, it can be used in any *IoT* system. Each *API* specifies the URL of that method, the request method (which can be one of GET, POST, PUT, or DELETE), the data format (we are using JSON since it is the most commonly used and has a small footprint), and body, which presents the data content of the message.

All microservice modules are implemented using the Java programming language and open-source Springboot framework [81]. The modules are deployed inside Docker containers [82].

Figure 3.5 shows the *LSDTNS component diagram*, which will be covered later in the text.

Stream micro-service

The purpose of the *Stream module* (Figure 3.5 stream MS) is to aggregate primitive events into the complex event and continuously send the aggregated complex event data stream for processing to the complex event reasoning module.

Stream service API

The *Stream service API* contains "Register for CE Stream" (Table 3.2), an essential external *API* that allows the *IoT Controller* to register, aka subscribe, to receive *ME CE* streams (Figure 3.5, 5).

Table 3.2: StreamManager service API

Name	URL	Request Method	Format	Produce
Register for CE stream	/streaming/register/<dt-id>	GET	Text	TEXT

IoT Controller Micro-service

The *IoT Controller* has access to the Data Warehouse. The *IoT Controller* invokes and loads the *DT* list from a data warehouse (Figure 3.5, 1). Once the *DT* list is loaded, the *IoT Controller* will calculate the priory for each *TD* using *DTPCA*. The priority can have one of the values from the k-predefined priority list. Two sets of data influence the priority selection:

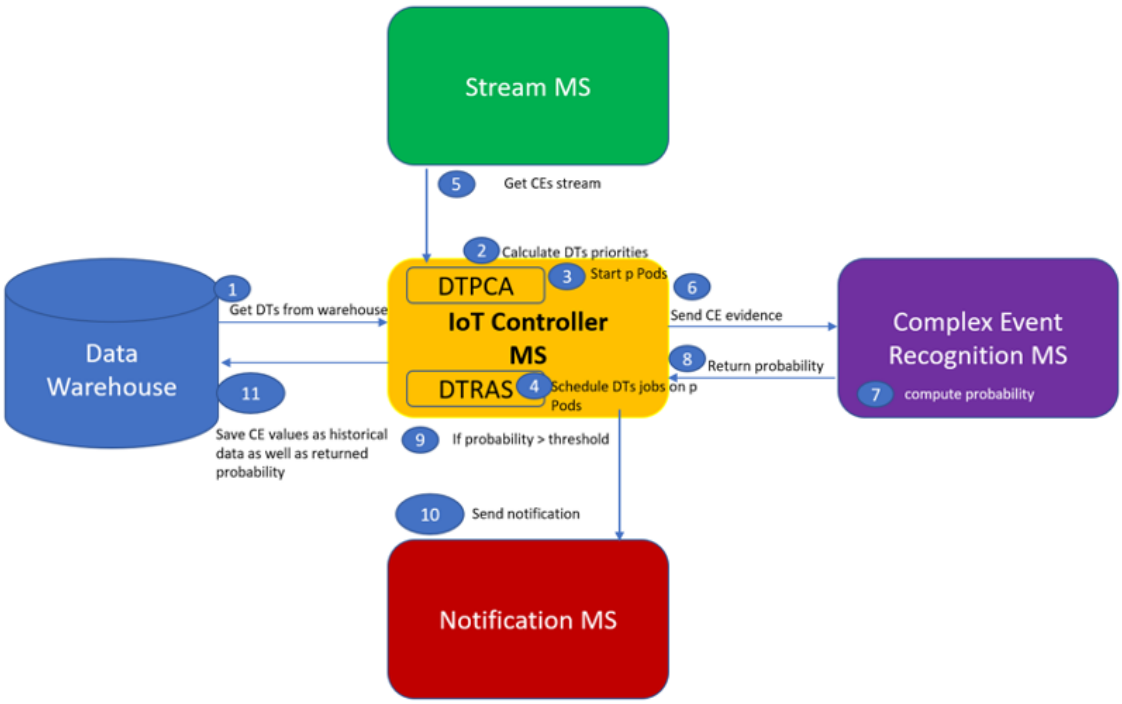


Figure 3.5: LSDTNS component diagram. 1) get the TDs from the data warehouse, 2) DTPCA calculates DTs priorities, 3) start p Pods running CER modules, 4) run DTRAS, schedule DT's jobs to p CER containers, 5) get CE stream from Stream MS, 6) send CE evidence to each CER, 7) each CER container calculates probability based on CE evidence, 8) the CER container returns probability to the IoT controller, 9) IoT controller compares the return probability with the threshold, and, if the probability is greater than the threshold, repeat steps 4- 9, 10) notify the Controller with the CE probability, 11) save CE stream evidence and return probability into the data warehouse.

1. the TD profile that is retrieved from the Data warehouse
2. the historical data for the DT (Figure 3.5, 2)

.

In Step 3, the *IoT Controller* sends a request to Kubernetes to start p containers (Figure 3.5, 3). In Step 4 (Figure 3.5, 4), the *DTRAS* will schedule DT's jobs on p containers. The *IoT Controller* receives CE streams for each DT (Figure 3.5, 5) and redirects them to the corresponding Complex Event container (Figure 3.5, 6). The Complex Event Pod will take CE evidence stream data and calculate the probability of the CE event (Figure 3.5, 7) and return this value to the IoT Controller (Figure 3.5, 8). Once the *IoT Controller* receives probabilities from *CER* containers, the Controller will compare received values with the previously defined threshold for the system (Figure 3.5, 9). For values greater than the threshold, the *IoT controller* will send a message to the notification *MS* (Figure 3.5, 10). To emphasize that once the sensor data streaming is established, the processes defined in Steps 5,6,7,8,9,10 are high priority. In Step 11 (Figure 3.5, 11), the *IoT Controller* sends an update to the data warehouse, and this step has a lower priority than Steps 5-10. The data stored in the Data Warehouse is HistoricalProbability (Figure 3.5), which combines the *CE* sensor data for each DT and probability for the *CE* sensor data. Stored *HistoricalProbability* will be used as historical data to make better predictions in the future.

IoT Controller Service APIs

The *IoT Controller* external *APIs* consist of a set of *APIs*:

1. *IoTControllerManager*, which contains four *APIs* responsible for starting and stopping *ME* containers (Table 3.3)
2. *IoTControllerPriorityManager* contains one *API* responsible for receiving probability from *ME* containers (Table 3.4).

3.2.5.3 CER Micro-service

The primary purpose of the *Complex Event Reasoning (CER) micro-service* is to calculate the probability of the complex event based on the received *CE* evidence (Figure 3.5, 5) and return the probability value to the *IoT Controller* for further processing (Figure 3.5, 7).

Table 3.3: IoTControllerManager service APIs

Name	URL	Request Method	Format	Produce
Start the process for a single DT container	/iot-controller/start/<dt-id>	GET	JSON	TEXT
Start the process for All DT containers	/iot-contrller/start	GET	JSON	TEXT
Stop process for single DT container	/iot-controller/stop/<dt-id>	GET	JSON	TEXT
Stop process for All DT containers	/iot-controller/stop	GET	JSON	TEXT

Table 3.4: IoTControllerPriorityManager service APIs

Name	URL	Request Method	Format	Produce
Update DT CE probability	/IOT-controller-probability/<dt-id>	POST	JSON	HistoricalProbability

As mentioned, each project's list of k *DT* resource types is predefined.We decided to have three profile possibilities: a) high, b) medium and c) low. For each *DT profile*, a corresponding algorithm is chosen. For example, the low profile uses complex event probability and rule-based solution, the medium profile uses Static Bayesian, and the high profile Dynamic Bayesian Algorithm.

Even though the algorithm chosen can be different and correspond to one of the *DT* profiles from the predefined list, they all have the same external *APIs*, presented in Section 3.2.5.4.

The open-source *JBOSS DROOLS fusion Complex Event Processing (CEP) tool* [83] is selected for processing CE of DTs with the low probability being critical. The JBoss is well-known and widely used by the open-source community. DROOLS fusion is one of its components. Complex event processing aims to recognize *CEs*, calculate probability, and respond as soon as possible. Each event type received from the stream micro-service is stored in the corresponding event type stream of the *CEP component*. All those event-type streams are considered when complex event rules are executed. The *JBoss Drools CEP tool* has two possible execution modes: Cloud and Streaming. We chose the second since *IoT*

applications have to process streams of events. All sensors' data is collected independently at specified time intervals and is not synchronized in time. Therefore, the Sliding Time Window functionality of the *CEP tool* becomes very handy. The Sliding Time Window allows the user to write rules that will only match events occurring in the last T time units, where T is definable by the rule designer. This functionality is used to observe different event streams in the time window.

Probabilistic programming (PP) is selected for processing CE of DTs with the medium and high probability to be critical. The purpose of the PP micro-service is to calculate the probability that *DT* will require notification while considering the predefined domain probability model, inference algorithm and evidence provided in the form of streamed sensor data. Probabilistic programming is a suitable methodology to address all uncertainty types mentioned in Section 2.1.1.1. However, it requires domain knowledge to build a probability model and select the most suitable inference algorithm. Once the PP micro-service calculates the probability value, the probability value is sent back to the *IoT Controller* (Figure 3.5, 8). The *IoT Controller* uses the probability value and the predefined reasoning rule to decide further action (Figure 3.5, 9).

Our experimental work, used an *open-source probabilistic programming library, Figaro* [85]. Figaro is a library built using the Scala programming language and can be easily integrated into more extensive programs and frameworks. Figaro probabilistic programming library has several built-in reasoning algorithms that can be applied automatically to new models. The implementation of medium priority uses the static Bayesian network. The algorithm of choice is Variable Elimination, an algorithm for finding the posterior distribution for a variable in an arbitrarily structured Bayesian network [30]. The implementation of the high priority is done using a Dynamic Bayesian network [30].

3.2.5.4 Complex Event Recognition Service APIs

The *Complex Event Recognition external APIs* consist of a set of *APIs*:

1. *CERManger*, which contains two *APIs* responsible for starting and stopping algorithms inside *DT containers* (Table 3.5)
2. *CEREvidenceManager*, which contains one *API* responsible for receiving *CE evidence*, and computing probability (Table 3.6)

.

Table 3.5: CERManager service APIs

Name	URL	Request Method	Format	Produce
Start Algorithm	/cer/start/<dt-id>	POST	JSON	Success or Error
Stop Algorithm	/cer/stop/<dt-id>	POST	JSON	Success or Error

Table 3.6: CEREvidenceManager service APIs

Name	URL	Request Method	Format	Produce
Set CE Evidence	/cer/evidence/<dt-id>	POST	JSON	ComplexEvent

3.2.5.5 Notification Micro-service

The purpose of the *Notification micro-service* is to provide different notification mechanisms such as email, SMS, setting alarms, etc. The notification microservice has one external API (Table 3.7): NotificationManager, that allows the IoT Controller to send the notification message (Figure 3.5, 10).

Table 3.7: NotificationManager service APIs

Name	URL	Request Method	Format	Produce
SendNotification	/notification/<dt-id>	POST	JSON	HistroicalProbability

3.2.6 IoT Notification Systems Components

This section explains the essence of *IoT Notification systems*. It presents the processing steps and workflow, from capturing the raw sensor data to the final system notifications. The notification process can be divided into six steps, as presented in Figure 3.6. Step 1 shows the raw data streams (events) captured in DTs. In Step 2, events are sent over the network to where they will be processed. Step 3 presents an aggregation component that aggregates events into DT complex events. In Step 4, the DT complex events list is passed to the IoT controller MS, where DT complex events are processed. Step 5 presents notification events sent over the network to a notification component used in Step 6. Finally, step 6 produces the notification action, such as an alarm.

IoT Notification systems considered in this research are associated with a *deadline D*. Each step has its execution time $Tstep$. Therefore, the sum of all execution times must be

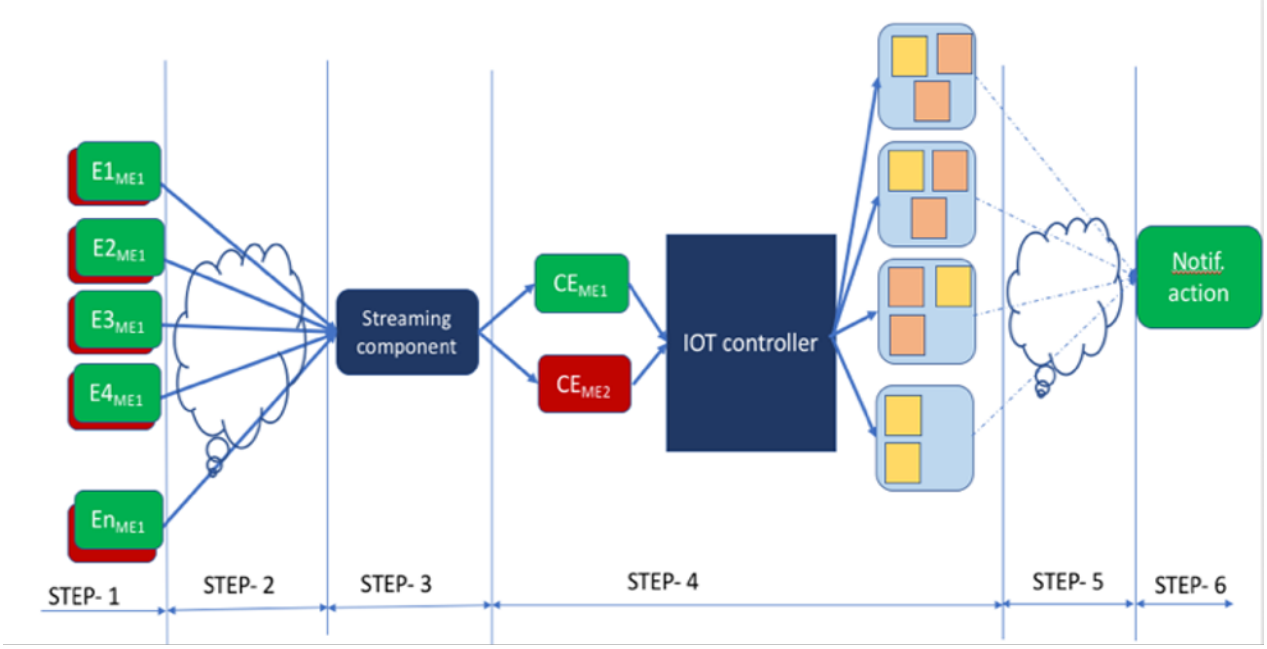


Figure 3.6: IoT Notification systems phase steps

equal to or be less than the D, presented in Equation 3.5.

$$Tstep_i \leq D \ where \ i = 1, 6 \tag{3.5}$$

$$Dstep_i \leq D \ where \ i = 1, 6 \tag{3.6}$$

In order to achieve the overall system deadline, each step has an expected deadline $Dstep_i$ to complete the process. Therefore, the sum of all deadlines should be equal to or less than the system deadline, presented in Equation 3.6.

The following two chapters: Chapter 4 and Chapter 5 will focus on processing in Step-4, presented in Figure 3.5.

3.3 Case Study: Prisoner Use Case

The goal of the prisoner use case example is to develop algorithms to improve inmates' safety by detecting events that correspond to attempted suicides. Monitoring of the prisoners is done using small radars that can detect activities (active or not), posture (lying,

sitting or standing) and estimate breathing rate every 30 sec. Detection and estimation are done using signal processing algorithms developed in our Lab. The accuracy of these methods is modestly low, for example, 80%. The objective is to reliably detect that the person is not moving and not breathing and to alarm the officers using such inaccurate data. We envision that this kind of system can be placed in every prison cell resulting in a large number of sensors sending data continuously [1].

3.3.1 Implementation of Prisoner User Case

3.3.1.1 Data and Streams User Case

In our prototype implementation, we use only two monitoring entities: cell1 and cell2. Cell1 has three sensors: active-1, posture-1 and breathe-1, and the cell2 has the following three corresponding sensors: active-2, posture-2 and breathe-2. Please note that the number of monitoring entities in the real prison will be in the order of hundreds of active prison cells. The active sensor provides two values: "yes" when the prisoner moves and "no" when a prisoner does not. The sensor has 97% accuracy in this example. The posture sensor provides three posture states: STAND - the prisoner is standing (82% accuracy), SIT – the prisoner is sitting (83% accuracy) and LIEDOWN – the prisoner is lying down (80% accuracy). The breathing sensor provides two values: "yes" when a prisoner breathes and "no" when the prisoner does not breathe. The breathing sensor accuracy is 82% for yes and 80% for no.

3.3.1.2 Implementation of Modules

Since all modules were built using Java, a common data model library is shared among most components. Figure3.7 shows the common data model libraries, where EventR, ME and Complex event are non-domain specific, and ActivityEventR, BreatheEventR, PostureEventR and EventType are domain specific implementations, and present data in the prisoner user case.

3.3.1.2.1 Streaming module In our prototype, data records are read from the Excel files and stored in memory. When a process starts the data is streamed to one of the reasoning components. Java parallel threads are used to stream all 6 data streams (cell1: activity-stream1, breathe-stream1, posture-stream1 and cell2: activity-stream2, breathe-stream2, posture-stream2).

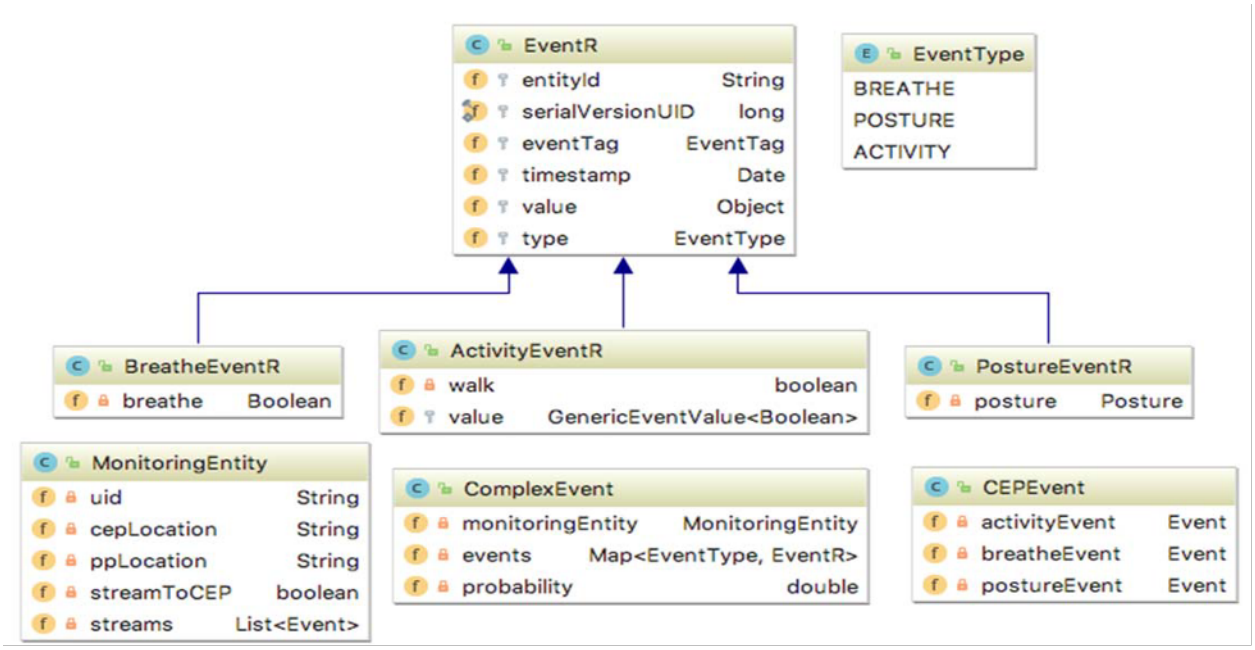


Figure 3.7: Class diagram of common data model library

3.3.1.2.2 Controller module The Controller does not have any domain-specific tasks. All functionality and interfaces with other modules are explained in Section 3.2.5.2.

3.3.1.2.3 CEP module The CEP module consists of REST APIs presented in Section 3.2.5.3, and the core module functionality. The REST APIs are generic for any IoT system, and the core module functionality is domain-dependent and requires a good understanding of the domain problem and of the complex event processing paradigm. We decided to use JBoss Drools Fusion complex event processing [83] for our implementation. As mentioned, we are using Stream mode that allows us to process incoming streams. Figure 3.8 shows CEP process definition, known as Knowledge base.

The data is streamed from the stream module to the CEP module. Once the CEP module receives data, data are continuously added into the corresponding rule base streams. The rules defined in the package rule (Figure 3.8 package) run continuously. Our package only has one rule "Does Prisoner Lay-Down" (Figure 3.9). The rule recognizes the complex event, where the activity event value is no, and the posture event value is LIEDOWN. Furthermore, we expect that all events are less than 30 seconds apart. Once the rule is satisfied, we will update Complex Event and send the notification to the Controller to

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule">
    <kbase name="rules" equalsBehavior="equality"
eventProcessingMode="stream" packages="rules">
        <ksession name="ksession-rules" default="true"
type="stateful" clockType="pseudo"/>
    </kbase>
</kmodule>
```

Figure 3.8: Knowledge module definition

notify the Stream module to redirect streams to the PP module.

```
rule "Does Prisoner Lay-Down"
when
    $cep : CEPEvent()

    $a1 : ActivityEvent() from entry-point ActivityStream
    $p1: PostureEvent() from entry-point PostureStream
    exists ActivityEvent($p1.posture ==Posture.LAY_DOWN, activity ==
false, this after[0s,4s]$p1) from entry-point ActivityStream
then

 $cep.setActivityEvent($a1);
 $cep.setPostureEvent($p1);
 end
```

Figure 3.9: "Does prisoner Lay-Down" rule

3.3.1.2.4 PP module The PP module consists of REST APIs presented in Section 3.2.5.3, and the core module functionality. The REST APIs are generic for any IoT system, and the core module functionality is domain-dependent and requires a good understanding of the domain problem and probabilistic programming and artificial intelligence knowledge. For our implementation, we decided to use Figaro probabilistic programming library [85].

Figure 3.10 presents the Prisoner cell reasoning component. As shown in the figure, three sensor streams (active, posture, and breathe) are continuously coming to the system and those sensor values are probabilistic evidences. An additional, non-sensor-related, information about the prisoner can be added as entry information for providing additional knowledge about the prisoner. For example, if we know the period of the prison sentence, that might change the probability of the prisoner being in the critical state. Figure 3.10 shows that the reasoning component consists of: the model and the inference algorithm. For our use case, we use the Variable Elimination algorithm as the inference algorithm.

Figure 3.11 presents a simplified version of the prisoner cell Bayesian model. The model consists of seven nodes. On the top of the directed acrylic graph is a Prisoner condition

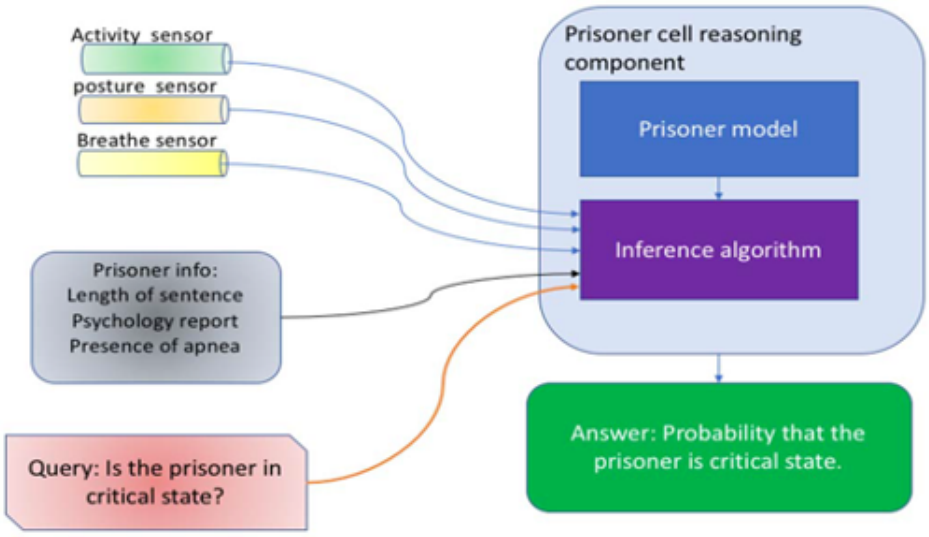
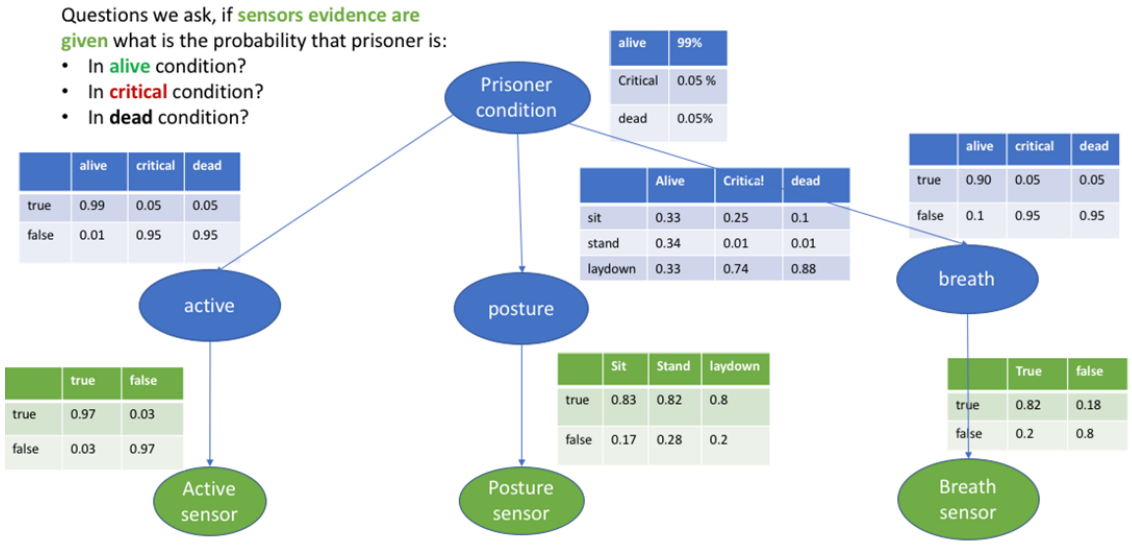


Figure 3.10: Prisoner cell component.



	alive	critical	dead
true	0.99	0.05	0.05
false	0.01	0.95	0.95

	Alive	Critical	dead
sit	0.33	0.25	0.1
stand	0.34	0.01	0.01
laydown	0.33	0.74	0.88

	alive	critical	dead
true	0.90	0.05	0.05
false	0.1	0.95	0.95

	true	false
true	0.97	0.03
false	0.03	0.97

	Sit	Stand	laydown
true	0.83	0.82	0.8
false	0.17	0.28	0.2

	True	false
true	0.82	0.18
false	0.2	0.8

Figure 3.11: Prisoner cell Bayesian model.

node, which defines the prisoner state. The prisoner condition node can have two possible values: noncritical (probability of 99%) and critical (probability of 1%). The prisoner condition node is the common ancestor of the active, posture and breathe nodes. The active node has two values, "yes" with a probability 99% and "no" with a probability of 1%. The posture node has three values: SIT with a probability of 24% for noncritical, and 14% for critical, STANDS with a probability of 63% for noncritical and 1% for critical and LIEDOWN with a probability of 23% for non-critical and 85% for critical. The breathe node has two values: "non-critical" with a probability of 90% and "critical" with a probability of 95%. The model also has three leaf nodes: active sensor, posture sensor and breathing sensor, where each leaf node has a corresponding ancestor node.

It is important to note that all these values are arbitrarily chosen in this thesis. After the experimental data are obtained, these values will be updated.

3.3.2 Results

Table 3.8 presents the execution time of then CEP algorithm shown in Figure 3.9. and PP Bayesian inference algorithms for the model shown in Figure 3.11. Bayesian inference is slower than the CEP algorithm.

Table 3.8: execution time of CEP and PP algorithms

CEP algorithm execution time	PP algorithm execution time
4.47 ms	33500 ms

Table 3.9 shows how probabilities of the prisoner's noncritical and critical state change with sensor evidence. The most critical probability of 97.68% is obtained when the prisoner is not active, does not breathe and is lying. The second critical state, according to our model, with a critical probability of 86.93% is when the prisoner is not active, does not breathe and is sitting.

3.4 Discussion

In this chapter, we presented a novel IoT Notification System Architecture based on a micro-service methodology that uses complex event recognition to handle uncertainty in IoT systems. Our present methodology can address the scalability issue in CER systems, and still keep the flexibility to select from various Complex Event Recognition solutions.

Table 3.9: Activity, posture and breathing probabilities, and complex event probability of critical and non-critical prisoner's condition.

Active	Posture	Breath	Alive	Critical
Yes	Stand	Yes	0.9999	0.0001
No	Stand	Yes	0.9999	0.0001
Yes	Stand	No	0.9997	0.0003
No	Stand	No	0.8468	0.1532
Yes	Sit	Yes	0.9992	0.0008
No	Sit	Yes	0.6839	0.3160
Yes	Sit	No	0.9889	0.0110
No	Sit	No	0.1307	0.8693
Yes	Lie down	Yes	0.9950	0.0049
No	Lie down	Yes	0.2546	0.453
Yes	Lie down	No	0.9338	0.0662
No	Lie down	No	0.0232	0.9768

Section 3.3 evaluates the proposed IoT Notification System Architecture through the Prisoner Use Case. The system orchestrates one of two monitoring techniques based on the estimated risk for the observed person/object. When the risk is low, the system performs coarse filtering of the data stream using Complex Event Processing (CEP), and when the system is under risk, a Bayesian network implemented using Probabilistic Programming (PP) techniques is used. Our tests show that Bayesian inference is slower than CEP, thus justifying our idea that one should use less demanding CEP characterized by a lower computational demand for low-risk monitoring.

The proposed IoT Notification System Architecture, based on the micro-service methodology, presented in this chapter is used as a starting point for further research in Chapters 4 and 5.

Chapter 4

Multi-Objective Optimization Methodology for Cloud Provisioning

This chapter proposes a *Multi-Objective Optimization methodology for cloud Provisioning (MOOP)*.

Part of the work presented in this chapter has been published in a paper: "Multi-objective optimization for cloud provisioning: A case study in large-scale microservice notification applications," The IEEE 10th International Conference on Future Internet of Things and Cloud FiCloud 2022, Rome, Italy [4].

In Chapter 3, we discussed the *IoT Notification system architecture* based on the microservice architecture paradigm. Microservices are deployed in a cloud cluster. Each microservice is deployed in a container, creating a containerized image. In the previous chapter, in Subsection 3.2.6, we presented IoT Notification systems flow with six steps. In this chapter, we will focus on Step-4, presented in Figure 3.5.

This chapter addresses the RQ2 research questions discussed in Section 1.2.4, and presents an optimizing approach for a large-scale microservice deployment in a critical notification system. The presented research focuses on optimization for three objectives:

1. cloud service cost
2. cloud resource utilization of CPU, RAM, and storage
3. meeting the system notification deadlines.

To the best of our knowledge, this is the first work dealing with multi-objective optimization for microservice notification applications where the notification load is variable and depends on other priory executed microservice in request processes. Furthermore, to the best of our knowledge, this is the first work based on multi-objective optimization, which analyses and proposes a Pareto front, an optimal list from preselected VM candidates. First, we formulate multi-objective optimization problems for microservice-based large-scale notification system applications, and then we provide the mathematical deployment model. Then, based on the expected workload and required multi-objective criteria, we propose a set of algorithms that provide the Pareto front optimal solution for preselecting a Cloud Service provider's VM type. Finally, at the end of this chapter, we present a case study demonstrating the use of the proposed technique.

Note: Multi-Objective Optimization (MOO) is considered to be a form of unconstrained optimization, which involves mathematical and computational techniques aimed at finding the minimum or maximum of a function without any constraints or limitations on the variables involved. However, Pareto optimal solutions, used in the proposed MOOP, introduce an element of constrained optimization. These constraints emerge from the trade-offs between different objectives. Each Pareto optimal solution signifies a point where the competing objectives have been balanced to the best possible extent, within the confines of these trade-off constraints.

Section 4.1 presents the system overview. Section 4.2 discusses the IoT Notification System Application. Section 4.3 presents the mathematical model. Section 4.4 covers proposed algorithms. Section 4.5 demonstrates the feasibility of the proposed MOOP through the Case Study. Section 4.6 provides discussion on proposed MOOP solution in this chapter.

4.1 System Overview

Microservice architecture is a new emerging architectural approach that has become increasingly popular in recent years [54]. Microservice-based applications are independent, small, modular services. Those services are named microservices. In addition, containerization is a newly emerging trend: a lightweight virtualization technology where applications deployed inside containers can easily be deployed and scaled up or down. Moreover, containers are suitable for encapsulating and deploying microservices [57]. Furthermore, data-intensive jobs often require large-scale parallel processing engines that can only be run in a cloud computing environment; such a solution can be very costly.

Therefore, optimizing the usage cost of cloud resources for running data-incentive jobs is very important. Cloud service providers, such as Microsoft Azure, Amazon AWS, and IBM, allow users to outsource the hosting of applications and services to a cloud using clusters of virtual machine instances. Cloud service providers charge a service usage cost to cloud service users. The service usage cost is based on the hourly usage rate of the virtual machine instances in the service provider's cloud [34]. Selecting the right VM type offered by a Cloud Service provider based on cost, CPU, RAM, and storage is challenging. For example, Microsoft Azure provides over 450 VM types [75]. Furthermore, the adoption of the microservice architecture increases the number of components in the applications, and because of that, the problem becomes even more challenging. For such a complex deployment system, the following questions need to be answered:

1. Which VM type from the list of offered VMs will provide the most cost-effective solution?
2. Which container cluster deployment will provide the best cloud VM resource utilization?
3. How many microservice container instances are required to support the incoming workload?

This is a multi-objective optimization problem that involves making trade-offs between different objectives. This chapter focuses on optimizing the following three goals: a) minimizing the deployment cost, b) maximizing resource utilization, and c) maximizing the number of job requests to be completed before the deadline.

The main components presented in this chapter are:

1. A formulated multi-objective optimization problem for microservice-based, large-scale notification system applications, Section 4.3,
2. A multi-objective mathematical deployment model of containerized microservices in a cloud cluster, Section 4.3,
3. An algorithm that estimates the required number of microservice containers based on the expected workload, Section 4.4.1
4. An algorithm that selects the Pareto front leading VM type candidates based on multi-objective optimization, Section 4.4.2
5. An algorithm for deploying microservice containers in a cluster where the VM type is preselected, Section 4.4.3.

4.2 The IoT Notification System Application Definition

In this chapter, as previously mentioned, we observe only Step-4 of the IoT Notification system phase steps shown in Figure 3.5. Figure 4.1 presents the Step-4 processing application workflow in the form of *Directed Acryclic Graph* (DAG).

The DAG has two paths. Path `#1` starts from the $START$ node and ends in $END1$, and it consists of the following four nodes: $START$, PP_MS, $NOTIF\_MS$, and $END1$. The PP_MS presents the probabilistic programming process deployed as a containerized microservice. $NOTIF\_MS$ presents the notification process deployed as a containerized microservice. The Path `#1` is the critical path, where the requested *deadline* D has to be granted, and the notification has to be delivered before *deadline* D. After processing the PP_MS based on the results of the processing, the system might require the notification, and the workflow will continue in the $NOTIF\_MS$ microservice.

Path `#2` starts from the $START$ node and ends in $END2$, and it consists of the following four nodes: $START$, PP_MS, DB_MS, and $END2$. Path `#2` is not a critical path since the data record has to be updated in the Database, but no deadline is attached to this process.

Even though the DAG presents the LSDTNS workflow (Figure 4.1), the methodology presented in this chapter can be applied to other similar DAG solutions following the presented methodology.

The application is deployed in three deployment layers that will be covered further in the text:

1. Software and Application layer
2. Platform layer
3. Infrastructure layer

4.2.1 Software and Application layer

4.2.1.1 Application

LSDTNS application is modelled as a DAG since DAG is a valuable and well-known tool for presenting the processing of data flows. A DAG that is characterized as a tuple (ms_set, $ms\_relation$, ms_path), where $ms\_set = \{ms_1, ms_2, \dots ms_t\}$ is a set of t microservices

of the application; $ms\_relation$ is the set of dependencies among microservices; ms_path is a set of paths, where all paths consist of microservices from the start to the end in the execution process.

Figure 4.1: DAG example

One path in ms_path that is considered a critical path is $ms\_path\_critical$. The system requirement is that the execution of the critical path $ms\_path\_critical$ is finished before a given deadline D. Figure 4.1 shows an example of a DAG. The DAG has two paths. $Path$ 1 starts from the START node and ends in END1, and it consists of the following four nodes: $START$, PP_MS, $NOTIF\_MS$, and END1. Microservice ms_{i} is characterized as a tuple $(comp\_res, storage\_res, max\_requests)$, where $comp\_res$ represents the compute resources required for processing a request on the microservice ms_{i}; $storage\_res$ represents the storage resource necessary for processing a microservice request; and, $max\_requests$ is the maximum number of requests for one instance of a microservice ms_{i}. In addition, $preset_{\mathrm{i}}$ is the preceding set of microservices that provides data to microservice ms_{i} to be executed. When there is a microservice $ms_{\mathrm{k}} \in preset_{\mathrm{i}}$ then the relation between microservices can be presented as $(ms_{\mathrm{i}},ms_{\mathrm{k}}) \in ms\_relation$. In other words, ms_{k} needs to be completed before ms_{i} can be executed.

4.2.1.2 The workload from the critical path point of view

This section focuses on the $ms\_path\_critical$ from the workload perspective. The critical path consists of the following microservices $ms\_path\_critical = \{ms_1, ..., ms_{\mathrm{k}}\}$, where k is the number of microservices in the critical path.

Figure 4.2 shows the subcomponents of the critical path's execution time, where:

1. $wait\_time_{\mathrm{i}}$ - is the waiting time for the i^{th} microservice execution process. The wait time is expressed as Equation (4.1), where $arrival\_time_{\mathrm{i}}$ is the job arrival time, and $start\_time_{\mathrm{i}}$ is the scheduled start time.

2. $burst\_time_{\text{i}}$ is the execution time of the i^{th} microservice.

3. ms_nl_{i} is the network latency time between the i^{th} and $(i-1)^{\text{th}}$ microservices' locations.

$$wait\_time_i = start\_time_i - arrival\_time_i \tag{4.1}$$

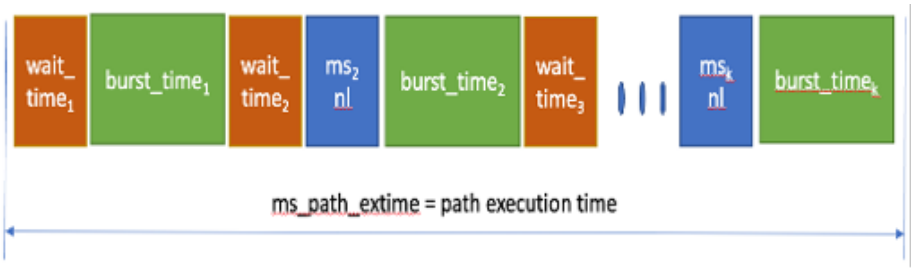


Figure 4.2: Subcomponents of the critical path's execution time

Figure (4.2) shows the components of $ms\_path\_critical$ execution time, denoted by $path\_extime$. Equation (4.3) expresses $path\_extime$, directly proportional to deadline D, where β presents a security coefficient.

$$\begin{aligned} path\_extime = {} & wait\_time_1 + burst\_time_1 + \\ & \sum_{n=2}^{k} (wait\_time_n + burst\_time_n + ms\_nl_n) \end{aligned} \tag{4.2}$$

$$\beta * path\_extime = D, where\ \beta \geq 1 \tag{4.3}$$

As previously noted, only the p_{notif} percentage of total n requests will require notification messages. That means only the p_{notif} percentage of the requests will fully complete the $critical\_ms\_path$. The rest, the $(100 - p_{\text{notif}})$ percentage of total n requests, will complete the execution path in the microservice prior to the notification subtask.

4.2.2 Platform layer - Container-based Architecture

Each microservice in the application layer is encapsulated in a container image, and container images are deployed in the cloud's cluster VM environment. The VMs in the cluster

used for processing the workload are often referred to as worker nodes. Each container image has the following computational resource characteristics: 1) CPU computational resource: The CPU is measured in CPU units, which is 1 vCPU/Core in the cloud environment or 1 hyper-thread on bare metal (when a hypervisor is installed directly on the physical machine, it is called a bare-metal hypervisor), 2) RAM - Memory computational resource: the system requested memory is measured in bytes.

Kubernetes [73], a popular container scheduler and orchestrator, recognizes two types of computation resources: requested and limited. A requested computation's resource is a recourse required to execute the application inside a container. While a limited computation's resource is designed to limit a container, the container will not use more computational resources than the limit, even if resources are available. The following parameters characterize a microservice of type y:

1. $R^{\text{y}}_{\text{cpu}}$–required CPU computational resource for microservice of type y
2. $R^{\text{y}}_{\text{ram}}$ –required RAM storage resource for microservice of type y
3. $R^{\text{y}}_{\text{storage}}$ –required storage for microservice of type y
4. $T^{\text{start}}_{\text{y}}$- time to start container image of microservice of type y

4.2.3 Infrastructure Layer

The dissertation's scope limits the system's deployment on a cloud provided by a single cloud provider in a single location. Investigating multi-cloud vendors or different geographical locations of the same cloud vendor forms a critical direction for future research. At this point in time, the public container offerings only allow a single VM type in the cluster. To be consistent with public container offerings, we will assume that all worker nodes in a cluster must be of the same type (CPU, RAM, storage, and cost).

The Solution infrastructure can be presented as cluster C, where cluster C has a set of v VMs of type c. All VMs in the cluster are of the same VM type. The VMs are characterized as tuple $\{R^{\text{VMc}}_{\text{cpu}}, R^{\text{VMc}}_{\text{ram}}, R^{\text{VMc}}_{\text{storage}}, cost^{\text{VMc}}\}$ where:

1. $R^{\text{VMc}}_{\text{cpu}}$ represents CPU capacity for VM type c
2. $R^{\text{VMc}}_{\text{ram}}$ represents RAM capacity for VM type c
3. $R^{\text{VMc}}_{\text{storage}}$ represents a storage capacity for a VM type c
4. $cost^{\text{VMc}}$ presents a cost per hour for a VM type c

4.3 Mathematical Model: Objectives

The focus of the chapter is multi-objective optimization in a cluster. Objectives can be split into two groups: *static objectives*, which do not change once the system is deployed, and *dynamic objectives*, traditionally known as scheduling optimizations that take effect after deployment is completed during the application run time. The focus is on the following optimization objectives:

1. deadline
2. resource utilization
3. cost.

All three objectives will be further discussed in the following text.

4.3.1 Objective 1: Deadline

As previously noted, the critical path consists of microservices, and the critical microservice path is presented as $critical\_ms\_path = \{ms_1, \dots ms_k\}$. First, the $critical\_ms\_path$ presents a task. Then, microservices are subtasks of the total task. For example, if the critical path consists of two microservices, $pp1\_ms$ and $notif\_ms$, the task will consist of two subtasks (a subtask to process $pp1\_ms$ microservice and a subtask to process $notif\_ms$ microservice). The $critical\_ms\_path$ execution is presented as job task j with the following characteristics:

- T^j– execution time for a task j on the critical path
- T^j_{ms} – execution time of microservice ms for a task j, where $T^j_{ms} = task_{end\_time} - task_{arrival\_time}$
- T_{ms} - ideal time to run a microservice ms, where
  T_{ms}=$task_{end\_time} - task_{start\_time}$
- $T^j_{VM_i,VM_j}$ - the network latency between two microservices deployed on two different VMs: VM_i and VM_j.

- $T^{\text{net}}_{\text{ms}_\text{x},\text{ms}_\text{y}}$ – the network latency between two microservices. If microservices are deployed on two different VMs in the same cluster, the network latency will be greater than 0. However, when microservices ms_x and ms_y are deployed on the same VM, the latency is so small that we assume it will be 0. The network communication latency between two microservices, ms_x and ms_y, is presented as Equation (4.4).

$$T^{net}_{\text{ms}_\text{x},\text{ms}_\text{y}} = T^{j}_{\text{VM}_\text{i},\text{VM}_\text{j}} \tag{4.4}$$

- Note: VMi and VMj present two different VMs in the same cluster, where each VM hosts one of the dependent microservices.

  The task j's execution time is computed as the sum of all sub_task' execution times, where a sub_task is the sum of the microservice execution time and network latency between dependent microservices on the observed critical path (4.5).

$$T^j = \sum_{i=1}^{ms} T^j_i + \sum_{i=1,k=2}^{ms} T^{net}_{ms_i,ms_k}, \quad i < k \wedge i \neq k \tag{4.5}$$

- D^j – deadline for the task j

- $D^j \geq T^j$, a task type with $j's$ execution time, must be less than or equal to the deadline D^j

- $deadline\_ach_\text{j}$ - the number of all tasks of type j that are executed before the deadline D (4.6).

$$dealine\_ach_j = \sum_{i=1}^{k} \begin{cases} 1 & ,if\ D^j \geq T^j_i \\ 0 & ,if\ D^j < T^j_i \end{cases} \tag{4.6}$$

- $avg\_deadline\_ach_j$ - the average number of tasks completed before the deadline (4.17).

$$avg\_deadline\_ach_j = \frac{deadline\_ach_\text{j}}{k} \tag{4.7}$$

- $deadline\_missed_\text{j}$ - a counter for all tasks of type j that missed the deadline D (4.8)

$$deadline\_missed_\text{j} = k - deadline\_ach_\text{j} \tag{4.8}$$

- $avg\_deadline\_missed_{\text{j}}$ - the average number of tasks that missed the deadline D (4.9)

$$avg\_deadline\_missed_{\text{j}} = \frac{deadline\_missed_{\text{j}}}{k} \tag{4.9}$$

4.3.2 Objective 2: Resource Utilization

As mentioned in Section 4.3, resource utilization can be expressed as static and dynamic.

4.3.2.1 Static Utilization

Cluster C has two sets of values: 1) the cluster resource requirements based on the sum of all microservice containers that should be deployed on the cluster and 2) the cluster capacity offering provided by the cloud vendor once the VM type and the number is selected. Resource utilization presents the relation between these two values. In addition, the cluster resource requirements are not a fixed value; the workload is expected to vary, and the cluster can adapt by scaling up and down (adding more or less computational nodes). We denote this utilization as static since once the VM type is selected for deployment, it does not change.

Several parameters are defined next:

- $R^{\text{VMc}}_{\text{storage}}$ - the VM's storage capacity for VM type c
- $R^{\text{VMc}}_{\text{cpu\_req\_min}}$ presents the minimum required CPU (Equation 4.10)
- $R^{\text{VMc}}_{\text{cpu_req_max}}$presents the maximum required CPU (Equation 4.11)
- $R^{\text{VMc}}_{\text{ram\_req\_min}}$ presents the minimum RAM required (Equation 4.13)
- $R^{\text{VMc}}_{\text{ram\_req\_max}}$ presents the maximum required RAM (Equation 4.12)
- $R^{\text{VMc}}_{\text{storage\_req\_min}}$ presents the minimum required storage (Equation 4.14)
- $R^{\text{VMc}}_{\text{storage\_req\_max}}$ presents the maximum required storage (Equation 4.15)
- $R^{\text{VMc}}_{\text{cpu\_req\_used}}$ presents the used CPU
- $R^{\text{VMc}}_{\text{ram\_req\_used}}$ presents the used RAM

- $R^{\text{VMc}}_{\text{storage\_req\_used}}$ presents the used storage

$$R^{\text{VMc}}_{\text{cpu\_req\_min}} = \sum_{ms=1}^{num\_alloc\_ms} R^{\text{VMc}}_{\text{cpu\_req\_min ms}} \tag{4.10}$$

$$R^{\text{VMc}}_{\text{ram\_req\_min}} = \sum_{ms=1}^{num\_allocms} R^{\text{VMc}}_{\text{ram\_req\_min ms}} \tag{4.11}$$

$$R^{\text{VMc}}_{\text{storage\_req\_min}} = \sum_{ms=1}^{num\_alloc\_ms} R^{\text{VMc}}_{\text{storage\_req\_min ms}} \tag{4.12}$$

$$R^{\text{VMc}}_{\text{cpu\_req\_max}} = \sum_{ms=1}^{num\_alloc\_ms} R^{\text{VMc}}_{\text{cpu\_req\_max ms}} \tag{4.13}$$

$$R^{\text{VMc}}_{\text{ram\_req\_max}} = \sum_{ms=1}^{num\_alloc\_ms} R^{\text{VMc}}_{\text{ram\_req\_max ms}} \tag{4.14}$$

$$R^{\text{VMc}}_{\text{storage\_req\_max}} = \sum_{ms=1}^{num\_alloc\_ms} R^{\text{VMc}}_{\text{storage\_req\_max ms}} \tag{4.15}$$

Then, the utilization can be defined as a combination of CPU, RAM, and storage utilization.

CPU utilization (Equation 4.16):

$$U[VMi]_{\text{cpu}} = \frac{R^{\text{VMc}_i}_{\text{cpu\_req\_used}}}{R^{\text{VMc}_i}_{\text{cpu}}} \tag{4.16}$$

has the value is between 0 and 1.

Since $R_{\text{cpu}}^{\text{VMc}_i}$ will not change over time, we will present it as a constant $\beta_{\text{cpu}}^{\text{VMc}_i}$, then (Equation 4.17):

$$U[VMi]_{\text{cpu}} = \beta_{\text{cpu}}^{\text{VMc}_i} * R_{\text{cpu\_req\_used}}^{\text{VMc}_i} \tag{4.17}$$

RAM utilization (4.18):

$$U[VM_i]_{\text{ram}} = \frac{R_{\text{ram\_req\_used}}^{\text{VMc}_i}}{R_{\text{ram}}^{\text{VMc}_i}} \tag{4.18}$$

has the value is between 0 and 1 (4.18).

Same with the RAM utilization, $R_{\text{ram}}^{\text{VMc}_i}$ will not change. We can present it as constant $\gamma_{\text{ram}}^{\text{VMc}_i}$ (Equation 4.19):

$$U[VM_i]_{\text{ram}} = \gamma_{\text{ram}}^{\text{VMc}_i} * R_{\text{ram\_req\_used}}^{\text{VMc}_i} \tag{4.19}$$

Storage utilization (4.20):

$$U[VM_i]_{\text{storage}} = \frac{R_{\text{storage\_req\_used}}^{\text{VMc}_i}}{R_{\text{storage}}^{\text{VMc}_i}} \tag{4.20}$$

, the value is between 0 and 1 (4.20).

As in the case of storage utilization, $R_{\text{storage}}^{\text{VMc}_i}$ will not change. We can present it as constant $\delta_{\text{storage}}^{\text{VMc}_i}$ (Equation 4.21):

$$U[VM_i]_{\text{storage}} = \delta_{\text{storage}}^{\text{VMc}_i} * R_{\text{storage\_req\_used}}^{\text{VMc}_i} \tag{4.21}$$

Overall, the total VM_i utilization is presented (4.22)(4.23) as a sum:

$$U[VM_i] = U[VM_i]_{\text{cpu}} + U[VM_i]_{\text{ram}} + U[VM_i]_{\text{storage}} \tag{4.22}$$

$$\begin{aligned} U[VM_i] = \beta_{\text{cpu}}^{\text{VMc}_i} * R_{\text{cpu\_req\_used}}^{\text{VMc}_i} + \\ \gamma_{\text{ram}}^{\text{VMc}_i} * R_{\text{ram\_req\_used}}^{\text{VMc}_i} + \delta_{\text{storage}}^{\text{VMc}_i} * R_{\text{storage\_req\_used}}^{\text{VMc}_i} \end{aligned} \tag{4.23}$$

The cluster utilization can be described using Equations (4.24) and (4.25):

$$U[VM_j] = \sum_{\mathrm{i=1}}^{\mathrm{total-VMc}} U[VM_i] \tag{4.24}$$

$$\begin{aligned} U[VM_j] = \sum_{\mathrm{i=1}}^{\mathrm{VMc}_j} (\beta_{\mathrm{cpu}}^{\mathrm{VMc}_i} * R_{\mathrm{cpu\_req\_used}}^{\mathrm{VMc}_i} + \\ \gamma_{\mathrm{ram}}^{\mathrm{VMc}_i} * R_{\mathrm{ram\_req\_used}}^{\mathrm{VMc}_i} + \delta_{\mathrm{storage}}^{\mathrm{VMc}_i} * R_{\mathrm{storage\_req\_used}}^{\mathrm{VMc}_i}) \end{aligned} \tag{4.25}$$

4.3.2.2 Dynamic Utilization

Dynamic utilization is a percentage of resources utilized during the system processing time, where, in this case, the resources are deployed containers in a cluster on VMs. This type of utilization is affected by the performance of the scheduling algorithm. Equation (4.26) presents the dynamic utilization:

$$U_D = \sum_{\mathrm{t=0}}^{\mathrm{total\_cycle\_time}} \frac{\sum_{\mathrm{i=1}}^{num\_containers} container\_runjob_i}{num\_containers} \tag{4.26}$$

4.3.3 Objective 3: Cost

Cloud service providers charge a usage cost to cloud service users based on the hourly usage rate of the VM instances running in the service provider's cloud. The cost of service is denoted as $Cost[VM_i]$, where i presents an i^{th} instance of the VM. The overall cost per cluster can be presented as (4.27):

$$Cost[cluster] = \sum_{\mathrm{i=1}}^{\mathrm{total-VMc}} Cost[VM_i] \tag{4.27}$$

4.4 Algorithms

4.4.1 Algorithm: "ms containers' count"

This section presents a method devised for calculating the required number of containers based on: a) the critical ms_path and all microservices that are part of the critical ms_path, and b) the expected workload.

Algorithm 4.1 : ms containers' count

Input: $critical\_path\_ms$, $[ms\_path\_len]$, n
Input: $arrival\_time\_period$, p_{notif}
Output: $[num\_count]$

$init[num\_count]$
for **do** $i \leftarrow k$
    $num\_job = n$
    if ms is notif **then**
        $num\_job = n \times p_{\text{notif}}$
    end if
    $num\_count = \frac{num\_jobs * ms\_path\_len}{arrival\_time\_period} \times \frac{1}{\beta}$
    $add\ num\_count\ to\ [num\_count]$

end for
$return[num\_count]$

The deployment cost is a critical objective. Referring to it by name this algorithm aims to optimize the number of containers deployed in a cluster. The number of containers is directly related to the number of VMs deployed in the cluster, which is related to the total cost. In order to reduce the required number of containers, a new term $arrival\_time\_period$ is introduced. The arrival time of n incoming jobs can be spread over some time period. This time period will be referred to as $arrival\_time\_period$.

The workload CE arrives in the arrival time interval ΔT (covered in Section 3.1). The $total\_cycle\_time$ (presented in Figure 4.3) is equal to the $arrival\_time\_period$ (4.28). The total cycle time must be smaller than or equal to the arrival time interval ΔT (4.29).

$$total\_\text{cycle\_time} = ms\_\text{path\_extime} + arrival\_\text{time\_period} \tag{4.28}$$

$$total\_cycle\_time \leqslant \Delta T \tag{4.29}$$

Figure 4.3 visually presents the relationship among values: $total\_cycle\_time$, $ms_i\_path\_exttime$, $arrival\_time\_period$.

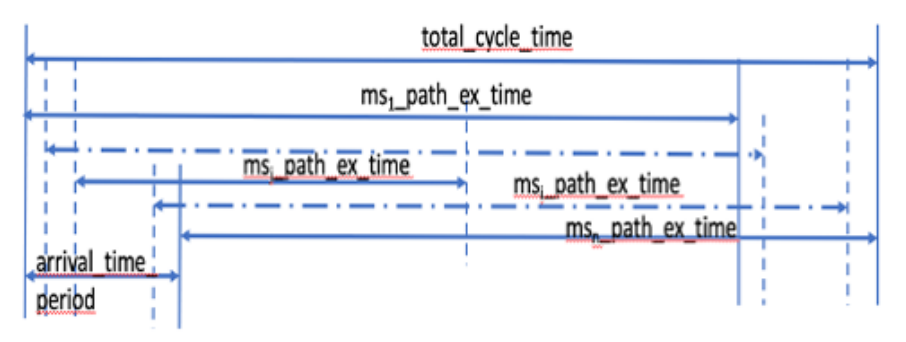


Figure 4.3: Total cycle time

The parameter num_count_i is the number of required containers for a ms_i microservice. The num_count_i is a component of the $critical\_path\_ms$ and is expressed as equation (4.30).

$$num\_count_i = \mathbf{f}(arrival\_time\_period, num\_jobs, \beta, ms\_path\_len) \tag{4.30}$$

Equation (4.31) further describes the computation of num_count_i. Note that the number of containers is directly proportional to the number of jobs (workload) and the microservice execution time (ms_path_len), and inversely proportional to $arrival\_time\_period$ and the security coefficient β.

$$num\_count = (\frac{num\_jobs * ms\_path\_len}{arrival\_time\_period}) * (\frac{1}{\beta}) \tag{4.31}$$

For the regular microservices that are not notification microservices, the total number of jobs (num_jobs) is n, the number of incoming CE. For the notification microservices, the number of jobs is captured with Equation (4.32), where p_{notif} is the percent of total jobs n that will require notification.

$$num\_jobs = n * p_{\text{notif}} \tag{4.32}$$

The definition of the ms containers' count algorithm is presented in Algorithm 4.1. The procedure "getNumCounts" takes the following: a) the *critical_ms_path*, b) the list of *ms_path_len*, c) n, which is a number of incoming CE, d) *arrival_time_period*, e) p_{notif} , and the procedure getNumCounts returns the list of ms containers' counts for each ms on the critical path.

4.4.2 Algorithm: "deployment"

The definition of the deployment algorithm is presented in Algorithm 4.2. The main procedure *deployment* takes the following inputs: a) a list of *num_count*, a list of ms containers' count for each required microservice on the *critical_ms_path* provided by Algorithm 4.1 in Section A, and b) a list of vm candidates. The primary deployment procedure invokes the *vm_deployment* procedure that returns a deployment table or null. If null is returned, the VM is unsuitable for deploying ms containers. Then, the deployment procedure returns the list of deployment tables for each VM candidate. Examples of deployment tables are shown in Tables 4.2 and 4.3.

The procedure *vm_deployment* first checks (*check_vm_validity*) that the VM's CPU, RAM and storage are not smaller than every ms CPU, RAM, or storage on the *critical_ms_path*. If a check does not pass, the return value is null, and the VM is discarded. The [*ms_counters*] are initiated in the next step, and values are set to 0. Further, the *vm_depoyment* procedure invokes the *fill_vm* procedure that fills a VM instance with ms containers. For each assigned ms type to a VM instance, the *ms_counters* for this ms type are increased by one since we are counting deployed ms containers.

4.4.3 Algorithm: "VM Pareto front"

The definition of the $VMParetofront$ algorithm is presented in Algorithm 4.3. This algorithm aims to find a list of VM types that form a Pareto front based on the set multi-objective optimization goals. The procedure $find\_pareto\_optimal$ takes the input data tuple [vm_id, *deadline*, *cost*, *util*] and returns the $pareto\_front$. Based on a multi-objective optimization goal, the *dominance_test* takes two input data tuples, x and y. The return value tells us if x dominated y and vice versa, or if they are not dominated.

Algorithm 4.2 : deployment

```
 1: procedure: deployment
Input: [num_count], [vm_type]
Output: [deploy_vm]
 2: for  do vm_j ← [vm_type]
 3:     vm_df = vm_deployment(vm_j, [num_counts])
 4:     if  then vm_df != null
 5:         add vm_df to [deploy_vm]
 6:     end if
 7: end for
 8:
 1:  procedure:vm_deployment
Input: vm, [vm_count]
Output: vm_df
 2: vm_df ← null
 3: if check_vm_validity != null then
 4:     [ms_counters] = 0
 5:     vm_i|[ms_counts] ← fill_vm([ms_counters]|vm_info|[ms_info]
 6: end if
 7:
 1:  procedure:fill_vm
Input: [ms_count], vm_info, [ms_info]
Output: vm_i
 2:
 3: update [ms_count]
 4: vm_i ← vm_i(vm_id [ms_count] cpu_util ram_util storage_util)
```

Algorithm 4.3 : VM Pareto front

```
1:  procedure: find Pareto optimal
Input: inputData
Output: pareto_front
2:  set pareto_front
3:  cadidate_row = 0
4:  set dominated
5:
6:  while candidate_row = inputData[candidate_row] do
7:      inputData.remove(candidate_row)
8:      row = 0
9:      non_dominated = True
10:
11:     while len(!inputData) != 0&row < len(inputData) do
12:         row ← inputData[row]
13:         if dominance_test(row, candidate_row) ← true then
14:             inputData.remove(row)
15:             dominated.add(row)
16:         else if dominance_test(row, candidate_row) ← false then
17:             non_dominate = False
18:             dominated.add(candiate_raw)
19:             row+ = 1
20:         end if
21:
22:         if non_dominated then                ▷ add non_dominated point to pareto_front
23:             pareto_front.add(candidate_row)
24:             if len(inputData) == 0 then
25:                 break
26:             end if
27:         end if
28:     end while
29: end while
```

```
1:  procedure: dominance_test(x, y)
Input: x|y
Output: boolean
2:
3:  test1 : deadline ← x(deadline) > y(deadline)
4:  test2 : cost ← x(cost) < y(cost)
5:  test3 : util ← x(util) > y(util)                ▷ true if all tests are true
6:  test1 ← test2 ← test3 ← true
```

4.4.4 Fitness Function

The Pareto front presents a set of optimal solutions. In order to get the single best option, the linear weighted sum model will be used. At first, a single optimization problem per cluster is defined, and the fitness function f(cluster) is presented (4.33), where D presents the $avg\_deadline\_ach$, U presents utilization optimization (4.25) and C' (4.34) presents the normalized cost optimization:

$$\begin{aligned} f(cluster) = a_1 * avg\_deadline\_ach + a_2 * U + a_3 * C' \\ Where\ a_1, a_2, a_3 \geq 0,\ and\ a_1 + a_2 + a_3 = 1 \end{aligned} \tag{4.33}$$

Equation (4.34) shows the normalization C' function:

$$C' = \frac{(C - Cmin)}{(Cmax - Cmin)} \tag{4.34}$$

Furthermore, the cluster is expected to scale up and down from the minimum number of VMs (the min number of ms containers) to the maximum number of VMs (the maximum number of ms containers). The fitness function is presented in Equation (4.35).

$$\begin{aligned} f = (\delta_1 * f(min) + \delta_2 * f(max)) \\ where\ \delta_1 + \delta_2 = 1,\ and\ 0 \leq \delta_1, \delta_2 \leq 1 \end{aligned} \tag{4.35}$$

4.5 Case Study

The case study presented in this section analyzes the notification application covered in Section 4.2.1 in Figure 4.1 The pp_ms container requires 5 CPUs and 7 RAMs, and the $notif\_ms$ container requires 1 CPU and 2 RAMs. In addition, the pp_ms job's length is 50 milliseconds, and the $notif\_ms$ job's length is 20 milliseconds. The expected workload is 100 jobs, $arrival\_time$ is 20 milliseconds, and the security coefficient β is 7.2.

In addition, the expected notification workload will vary from $p\_notifmin = 20\%$ to $p\_notifmax = 60\%$. Once Algorithm 4.1 is applied, the number of pp_count is 30 containers for both min and max notification loads, and $notif\_countmin$ is 20, and $notif\_countmax$ is 60 containers.

The list of VMs shown in Table 4.1 has been selected as an example from the Azure price calculator [74]. The cost is presented in cost units; the cost unit, for example, can be

USD or any other currency. Please note that a problem can be larger than our case study example. In order to simplify and demonstrate the problem, we selected only 8 VM types from over 450 possibilities [75].

Table 4.1: The list of VMs

Id	A3	A4	B4	B8	B12	E8	E16	E64
CPU	4	8	4	8	12	8	16	64
RAM	7	14	16	32	48	64	128	504
Storage	285	604	32	64	2	2	2	2
Cost	36	72	22	43	65	92	372	742

Tables 4.2 and 4.3 show two examples of deployment tables returned by Algorithm 4.2. Both Tables 4.2 and 4.3 observed the same VM type E64. The only difference between Table 4.2 and 4.3 is that, in Table 4.2 , the number of required notification containers is 30, whereas in Table 4.3, it is 60.

Table 4.2: A deployment example 1:
VM(cpu = 64, ram=504), pp_ms(cpu=5, ram=8), notif_ms(cpu=1, ram=2), pp_count = 30, notif_conut = 20

VM_id	pp_count	notif_count	cpu_util	ram_util
1	12	4	100	11.11
2	12	4	100	11.11
3	12	4	100	11.11
4	0	8	12.5	3.12

Table 4.3: A deployment example 2:
VM(cpu = 64, ram=504), pp_ms(cpu=5, ram=8), notif_ms(cpu=1, ram=2), pp_count = 30, notif_conut = 60

VM_id	pp_count	notif_count	cpu_util	ram_util
1	12	4	100	11.11
2	12	4	100	11.11
3	12	4	100	11.11
4	0	48	75	19.05

Table 4.4: Show values from Algorithm 4.2. Highlighted values belong to the Pareto front calculated by Algorithm 4.3

VM_id	Cost20n	cpu20	Ram20	Cost60n	Cpu60	Ram60
A4	0.63	0.7125	0.3857	0.76	0.875	0.5714
B8	0.9	0.7125	0.1688	0.97	0.875	0.250
B12	1	0.9444	0.2222	1	0.9722	0.2778
E8	0.43	0.7125	0.0844	0.61	0.875	0.1278
E16	0	0.9659	0.1136	0	0.9375	0.1339
E64	0.37	0.7812	0.0913	0.44	0.9375	0.131

Table 4.5: Fit functions results: for f-min minimal cluster size, f-max, for maximal cluster size, and f-total for minimal and maximum cluster size combined. $a_1 = 0, a_2 = 0.5, a_3 = 0.5, \delta_1 = 0.7, \delta_2 = 0.34$

VM_id	f-min	f-max	f-total (70min, 30 max)
A4	0.42	0.78	0.528
B12	0.897	0.97658	0.920874
E16	0.397	0.38	0.3919

Figure 4.4 and Table 4.4 show the number of required VMs that were calculated using the Algorithm 4.2 *deployment* for min (20%) and max (60%) notification workload. We can notice that for some VM types, the number of containers does not increase with more containers deployed (A4, B8, E8 and E64), and for some, the number of containers increases with increased workload (B12 and E16).

Further results are presented after applying Equation 4.26, (Equation 4.31) with the following parameters $a_1 = 0$,$a_2 = 0.5$,$a_3 = 0.5$, $\delta_1 = 0.7$,$\delta_2 = 0.34$.

Figure 4.5 and Table 4.5 show for each VM type total cost (cost20 and cost60, where cost20 presents 20% and cost60 present 60% notification workload), cpu utilization (cpu20 and cpu60) and ram utilization (ram20 and ram60). The values in the columns Cost20n and Cost60n in Figure 4.5, are 1 - the normalized value of cost value from columns Cost20 and Cost60 in the table. For normalization, Equation 4.35 has been used. After applying Algorithm 4.3 to the data presented in Table 4.5, 20% of the notification workload has three elements in the Pareto front (A4, B12 and E16) and two for 60% of the notification workload (A4 and B12).

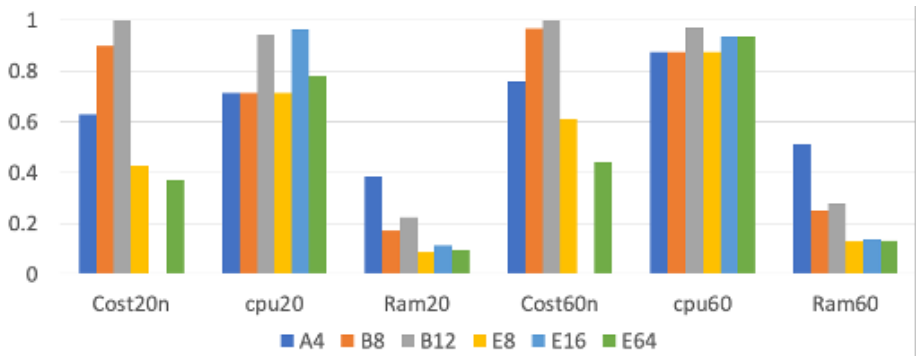


Figure 4.4: Values from Algorithm 4.2, the VMs A4, B12, B16 belong to the Pareto front calculated by Algorithm 4.3

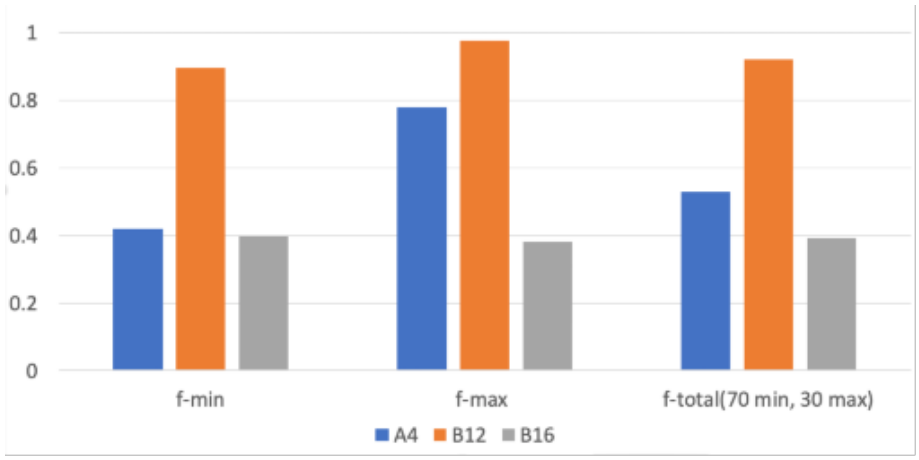


Figure 4.5: Results of fit functions for f-min minimal cluster size, f-max, for maximal cluster size, and f-total for minimal and maximum cluster size combined $a\_1 = 0$,$a\_2 = 0.5$,$a\_3 = 0.5$,$\delta_1 = 0.7$,$\delta_2 = 0.34$

.

4.6 Discussion

This chapter presented an approach to optimize the deployment of a large-scale critical notification system. Our research focuses on multi-optimization objectives for microservice notification applications, where the notification load is variable and depends on the results of previous microservice subtasks.

First, in this chapter, we formulated the multi-objective mathematical deployment model. Then, we presented and demonstrated the methodology through the case study. In step one, the methodology estimates the required number of microservice containers based on the expected workload. Step two selects the Pareto optimal front for leading VMtype candidates based on multi-objective optimization using the results from step one. Finally, step three delivers the deployment model for preselected VM types based on the results from step two.

Chapter 5 concerns on multi-objective optimization for microservice notification applications and will be focused on dynamic optimization and resource allocation scheduling. Chapter 5 operates under the assumption that a deployment map has been provided. This assumption constitutes one of the significant contributions offered by the solution to the Multi-Objective Optimization Problem (MOOP) discussed in the chapter.

Chapter 5

Digital Twin Resource Allocation Scheduler

This dissertation chapter covers the Digital Twin Resource Allocation Scheduler (DTRAS) that was devised using Deep Reinforcement Learning (DRL). The work presented in this chapter is not limited only to the cluster of Digital Twins. The same concept can be used in broader deployments as a Scheduler for a large-scale Critical Notification application based on Deep Reinforcement Learning (SCN-DRL).

Part of the work presented in this chapter has been published in two papers:

1. "SCN-DRL: Scheduler for large-scale Critical Notification applications based on Deep Reinforcement Learning," The IEEE 10th International Conference on Future Internet of Things and Cloud FiCloud 2022, Rome, Italy [5].

2. "Deep Reinforcement Learning solution for Scheduling critical notifications in a Digital Twin cluster", ready for submission [6].

This chapter provides answers to RQ3 research questions discussed in Section 1.2.4. Previously, in Chapter 4, we presented MOOP, a multi-objective optimization solution for provisioning microservice critical notification system applications. MOOP focuses on *static multi-objective optimization*. The following three multi-objectives are optimized: a) cloud service cost, b) cloud resource utilization of CPU, RAM and storage, and c) meeting the system notification deadline. In the previous chapter, we observed that the objectives can be divided into two categories: 1) *static*, which does not change once the system is

deployed, and 2) *dynamic*, which is traditionally known as scheduling optimization and takes effect after deployment is completed, during the application run time.

The work in this chapter addresses the *dynamic optimization* for three objectives: cloud service cost, cloud resource utilization, and system notification deadline. In addition, the work in this chapter provides an answer to the DTRAS discussed in Chapter 3.

Micro-service architecture, presented in Chapter 3, is a popular paradigm. Combined with micro-service containerization, it allows the building of scalable and reliable systems. However, the micro-service architecture has a drawback: selecting a modular microservice architecture increases the number of components and their interactions that must be managed, scheduled, and orchestrated. To the best of our knowledge, the results presented in this dissertation are the first work dealing with multi-objective optimization for microservice notification applications, where the notification load is variable and depends on other priory executed microservice in request processes.

The work presented in this chapter addresses *dynamic optimization* for three objectives: cloud service cost, cloud resource utilization, and system notification deadline. A scheduler for large-scale Critical Notification applications based on Deep Reinforcement Learning (SCN-DRL) is an example solution that translates the containerized microservice cloud cluster scheduling problem into a learning problem, achieved using Deep Reinforcement Learning.

SCN-DRL is a solution based on RL, where the RL Agent is a pre-trained Neural Network (NN) that has been trained using the Policy gradient approach. In this chapter, we present our experimental results with three types of NNs: 1) NN1 is built using the fully connected NN with one hidden layer, 2) NN2 is built using the fully connected NN with three hidden layers, and 3) NN3 is built using the 2D Convolutional NN.

Our research shows that SCN-DRL tested with all three types of NNs:

1. Provides feasible performance for scheduling the DT cluster's notification jobs.
2. Outperforms well-known heuristics schedulers Small Job First (SJF), Long Job First (LJF), and First Come, First Served (FCFS), as well as the heuristics, $First\_VM$ and $Same\_VM$, presented in Section 5.1.3.
3. Keeps steady performance when the $notification\_workload$ p_{notif} increases from 10 to 90%.

4. It gives rise to increases in performance if the *total_system_workload* is 5% smaller than the system capacity c (see Section 1.1).

Furthermore, the research results show that SCN-DRL using all three NNs are resilient to sudden container resources drop by 10% and even show improved performance when the container resources drop by 5%. Containerized microservice resilience to failures is a crucial component of a distributed system, and our researched critical notification system described in this thesis is one of them.

Please refer to Chapter 4, Section 4.2, for the definition of the system used in this chapter. In addition, based on the research presented in Chapter 4, the following have been preselected and used in this research:

1. VM type is deployed in the cloud cluster, where VM info such as cost, CPU, RAM and storage are known.
2. The number of required microservice application containers is known.
3. A container's deployment schema is known.

The remainder of this chapter is organized as follows: Section 5.1 describes the Resource Learning Setup. Section 5.2 discusses the Neural Networks model training. Section 5.3 discusses the workload used for NNs training. Section 5.4 discussed the performance evaluation of Neural Networks. Section 5.5 provides discussion.

5.1 Reinforcement Learning Setup

Reinforcement learning has four essential components:

1. Environment
2. State space
3. Action space
4. Rewards

Figure 5.1 presents the Reinforcement Learning components. Please note that Figure 5.1 shows six different Agents, but only one will be paired with the Environment during the learning. Also, please note that all Agents use the same actions (discussed in Section 5.1.3) to communicate with the Environment. The Agent algorithm performs the selection of action.

5.1.1 Environment

The Environment consists of the following (see Figure 5.1):

1. Cluster
2. Job slots
3. Backlog

5.1.1.1 Cluster

The cluster is built using the deployment schema provided. The deployment schema defines the number of VMs m and, for each VM, provides the type and number of deployed microservice (ms) containers.

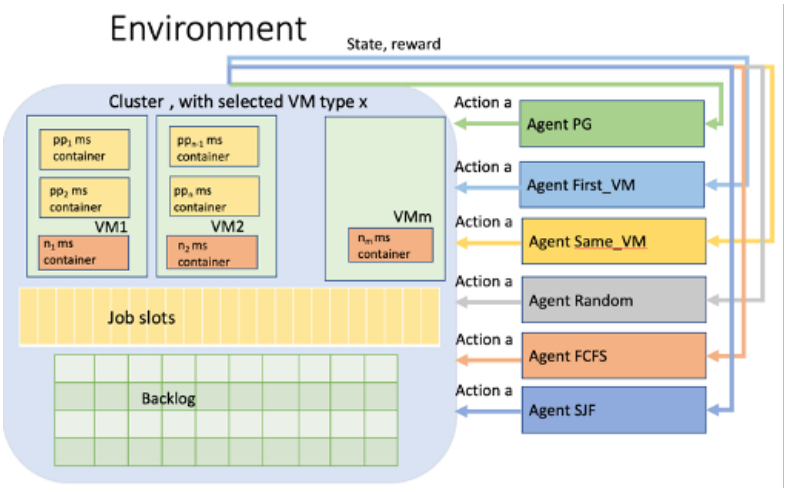


Figure 5.1: Reinforcement learning components

The deployment schema is shown in Table 5.1 for the reinforcement learning environment, the VM type B12 [4]. The deployment schema displays five columns for each VM deployed: 1) vm_id is the position of the VM in the cluster, 2) pp_count is the number of PP_MS deployed on this particular VM, 3) $notif\_count$ is the number of $NOTIF\_MS$ deployed on this VM, 4) cpu_util is the percentage of CPU utilization for this particular VM after ms containers have been deployed, 5) ram_util is RAM utilization percentage of the VM after ms containers have been deployed.

Table 5.1: The deployment schema
VM(cpu = 12, ram=48), pp_ms(cpu=5, ram=8), notif_ms(cpu=1, ram=2), pp_count = 30, notif_conut = 60

VM_id	pp_count	notif_count	cpu_util	ram_util
1	2	2	100	25
2	2	2	100	25
3	2	2	100	25
4	2	2	100	25
5	2	2	100	25
6	2	2	100	25
7	2	2	100	25
8	2	2	100	25
9	2	2	100	25
10	2	2	100	25
11	2	0	83.34	16.67
12	2	0	83.34	16.67
13	2	0	83.34	16.67
14	2	0	83.34	16.67
15	2	0	83.34	16.67

5.1.1.2 Action Space

Initially, we evaluated the commonly known CPU scheduling algorithms (FCFS, SJF, LJF), which showed poor performance due to our system's nature. More information on evaluation of commonly known CPU scheduling algorithms can be found in Section 2.8 , and in our previous work [5].

Communication between Agents and the Environment is done through actions. After analyzing the nature of the *total_system_workload*, we provided a set of 10 recognized actions. For each action selected by an Agent, the Environment performs a predefined change in the Environment, which brings the Environment to a new state, presented in Section 5.1.1.3. The Environment will validate the new state and reward the Action 5.1.2. Table 5.2 provides the list of actions and presents an action number and a description for each action.

Table 5.2: Action Space

Action	Action description
0	Let the system run for one millisecond
1	Schedule all job types based on arrival time (FCFS)
2	Schedule all jobs based on the first to arrive at the backlog
3	Schedule all jobs randomly
4	Schedule notification jobs randomly
5	Schedule notification jobs on the same VM as pp jobs
6	Schedule notification jobs based on arrival time (FCFS only for notification jobs)
7	Schedule notification jobs based on the first to arrive at backlog
8	Schedule notification jobs based on the earlier pp complete time
9	Allocate jobs

5.1.1.3 State Space

The state space, shown in Figure 5.2, is a distinct image. The image consists of five segments:

1. The segment highlighted in *orange colour* presents the number of the "PP_MS" microservice containers deployed in the cluster
2. The segment highlighted in *green colour* presents the number of "NOTIFICATION_MS" microservice containers deployed in the cluster
3. The segment highlighted in *yellow colour* presents the number of the job slots
4. The segment highlighted in *blue colour* presents the number of backlog slots
5. The segment highlighted in *white colour* presents the number of completed jobs

The number in the first two segments represents the cluster resources for both PP_MS and $NOTIF\_MS$ containers. Each rectangle in the first and second segments presents a container deployed in the cluster. Suppose value in the rectangle is greater than 0. In that case, this particular container has a job scheduled and currently running in the container. The number shown in the rectangle presents the elapsed job time. If the number is 0, no job is currently running in that container. The number in a rectangle of the third and fourth segments represents job slots and backlog. The number shown is the job's arrival

45	105	0	0	98	45	0	0	55	34	0	0	0	23	11
122	123	231	0	0	0	0	0	23	66	123	0	0	0	23
34	0	99	98	0	0	0	0	44	0	0	0	0	12	11
34	12	3	4	0	0	0	0	10	11	0	0	0	0	3
32	12	1	5	32	11	10	9	23	34	56	11	0	0	0
5	21	3	4	0	0	0	0	0	0	0	12	11	9	1
0	0	0	0	0	0	0	0	12	34	22	11	1	1	0
55	67	89	92	102	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 5.2: An example of State space

time. If the number is 0, then this job slot or backlog slot is empty, meaning no job is assigned in this slot. The number in a rectangle of the fifth segment represents completed jobs. If the number is greater than 0, it presents the job's competition time; if it is 0, it means the job is not yet completed.

5.1.2 Rewards

The rewards function is a crucial component in reinforcement learning. Therefore, properly defining the rewards function affects the capabilities of DRL to learn. The importance of the rewards function is demonstrated in Section 5.1.3. The reward is calculated after each Agent action based on a new Environment state. The following three objectives are rewarded or penalized when the objective is met:

1. jobs completed
2. resource utilization
3. job completion prediction

Rewards functions for each objective is presented in the following sections.

5.1.2.1 Reward 1: Job Completed

The job completed reward $job_{compl-rew}$ is given by Equation 5.1, where the p_{comp-d} presents the percentage of all PP_MS jobs that are completed before the deadline, the $notif_{comp-d}$ presents the percentage of all $NOTIF\_MS$ jobs completed before $deadlineD$, the $pp_{notcomp-d}$ presents the percentage of all PP_MS jobs that did not get completed before $deadlineD$. The $notif_{notcomp_d}$ presents the percentage of all $NOTIF\_MS$ jobs that were not completed before $deadline\ D$.

$$job_{compl-rew} = \alpha_1 * pp_{comp-d} \ + \alpha_2 * notif_{comp-d} - \alpha_3 * pp_{notcomp-d} - \alpha_4 * notif_{notcomp-d} \quad (5.1)$$

5.1.2.2 Reward 2: Resource Utilization

The resource utilization reward $util\_rew$ is given by Equation 5.2 as a sum of $perc_{pp-util}$, which presents the utilization percentage of PP_MS containers, and $perc_{notif-util}$, which presents the utilization percentage of $NOTIF\_MS$ containers. Equation 5.3 presents the $perc_{pp-util}$, and Equation 5.4 presents $perc_{notif-util}$. β_1, β_2 are coefficients, where $\beta_1, \beta_2 \geq 0$ and $\beta_1 + \beta_2 = 1$.

$$util\_rew = \beta_1 * perc_{pp-util} + \beta_2 perc_{notif-util} \quad (5.2)$$

$$perc_{pp-util} = \Sigma_{i=1}^{pp_{count}} \frac{pp_{usedi}}{pp_{count}} \quad (5.3)$$

$$perc_{notif-util} = \Sigma_{i=1}^{notif_{count}} \frac{notif_{usedi}}{notif_{count}} \quad (5.4)$$

5.1.2.3 Reward 3: Job Completion Prediction

This section presents job completion prediction reward function. In this reward, we observe the state of the three Environment components: cluster, jobslots and backlog.

In the cluster component, we observe the status of ms containers. We try to predict the following: "What is the probability that currently running jobs in the cluster's containers will be completed by the deadline?" For each job that has a chance to complete before *deadline* D, the counter $complete\_count$ is increased by one. For the job failing to complete by the deadline, the $not\_complete\_count$ counter is increased by one.

In the *Jobslots* component: We try to predict the following: "What is the probability that jobs in jobslots will be completed by the deadline?" For each job that can be completed before the deadline, the counter $complete\_count$ is increased by one. For the job that fails to be completed before the deadline, the $not\_complete\_count$ counter is increased by one.

In the *Backlog* component: We try to predict the following: "What is the probability that jobs in backlog will be completed by the deadline?" For each job that can be completed before the deadline, the counter $complete\_count$ is increased by one. For the job that fails to be completed before the deadline, the $not\_complete\_count$ counter is increased by one.

The job competition reward is expressed by the Equation 5.5, where γ_1, γ_2 are reward optimization coefficients:

$$job\_pred\_rew = \gamma_1 * complete\_count - \gamma_2 * not\_complete\_count \tag{5.5}$$

5.1.2.4 Total Reward

The total reward is expressed in Equation 5.6, and α, β, and γ, are reward optimization coefficients:

$$total\_rew = \alpha * job\_compl\_rew + \beta * job\_util\_rew + \gamma * job\_pred\_rew \tag{5.6}$$

There are 11 ($\alpha, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \beta, \beta_1, \beta_2, \gamma, \gamma_1, \gamma_2$) reward coefficients. Further in the text, we refer to them as Rewards parameters. Rewards parameters are used to tune the total reward function. Section 5.1.4 shows how rewards parameters affect policy performance.

5.1.3 Other Agents

The $RSC-DRL$ algorithm has undergone evaluation and comparison against well-established CPU scheduling algorithms (Section 2.8), including FCFS, SJF, and LJF, which have

served as benchmarks for other researchers [65], [66], [69], [71], [70]. Additionally, we assessed RCS-$RSC - DRL$ alongside a Random algorithm, which randomly selected one of 10 actions and was also utilized as a benchmark by [65] [66].

In addition, we devised two heuristic algorithms:

1. $First\_VM$ presented in the Algorithm 5.1
2. $Same\_VM$ presented in the Algorithm 5.2.

5.1.3.1 $First\_VM$ and $Same\_VM$ Heuristics Algorithms

Both $First\_VM$ and $Same\_VM$ algorithms monitor cluster containers' usage, including job_slot and backlog. These algorithms make decision based on the observed monitored values. The fundamental difference between these two algorithms lies in their scheduling approach:

- $First\_VM$ schedules a notification job on the first available notification container.
- $Same\_VM$ waits until the same VM, where the pp job was running, is freed to schedule the notification job. This distinction is important because if the pp job and the notif job are scheduled to be executed on two different VMs, the network latency time needs to be added to the processing time.

Here is a list of the *cluster containers' usage* parameters used in algorithms:

- $pp.contaniers$: The number of pp containers available in the cluster.
- $notif.contaniers$: THe number of $notif$ containers available in the cluster.
- $c.pp.running$: The number of pp containers currently processing a pp job.
- $c.notif.running$: The number of $notif$ containers currently processing a $notif$ job.
- $c.per.pp.allocated$: The percentage of how many pp containers is currently running a pp job.
- $c.per.notif.allocated$: The percentage of how many $notif$ containers is currently running a $notif$ job.

Algorithm 5.1 : $First\_VM$ Heuristic Algorithm

Input: *cluster*, *jobslot*, *backolog*
Output: *action a*, *where* $a \in \{0, ..9\}$

```
1: init a = 0
2:
3: pp.contaniers → cluster.get.pp.containers()
4: notif.containers → cluster.get.notif.containers()
5: c.pp.running → cluster.get.pp.running.jobs()
6: c.notif.running → cluster.get.notif.running.jobs()
7: c.per.pp.allocated → len(c.pp.running)/len(pp.containers)
8: c.per.notif.allocated → len(c.notif.running)/len(notif.containers)
9:
10: slots → get(jobslot.get.available.slots())
11: per.pp.slots = 0, per.notif.slots = 0, per.slots = 0
12: if slots is not 0 then
13:     per.pp.slots = jobslot.get.pp.slots()/slots
14:     per.notif.slots = jobslot.get.notif.slots()/slots
15:     per.slots = slots/len(jobslot.get.job.slots())
16: end if
17:
18: current.size, pp.size, notif.size = backlog.get.backlog.job.count()
19: backlog.holds → len(backog.get.backlog()
20: per.backlog.size → current.size/backlog.holds
21: per.pp.b.size → pp.size/backlog.holds
22: per.notif.b.size → notif.size/backlog.holds
23:
24: if c.per.pp.allocated is 100 then
25:     if per.notif.b.size is 0 then
26:         a=0
27:     end if
28: else if c.per.pp.allocated < 80 then
29:     if per.pp.b.size > 10 then
30:         a=1
31:     else if per.notif.b.size > 1 then
32:         a=7
33:     else
34:         a=0
35:     end if
36: else if c.per.notif.allocated < 50 then
37:     if per.notif.b.size > 1 then
38:         a=1
39:     else if per.notif.b.size > 0 then
40:         a=0
41:     else
42:         a=1
43:     end if
44: else
45:     if backlog.holds is not 0 then
46:         a=1
47:     end if
48: end if
49:
50: return a
```

Algorithm 5.2 : $Same\_VM$ Heuristic Algorithm

Input: *cluster*, *jobslot*
Output: *action a*, *where* $a \in \{0, ..9\}$

```
init a = 0

pp.contaniers → cluster.get.pp.containers()
notif.containers → cluster.get.notif.containers()
c.pp.running → cluster.get.pp.running.jobs()
c.notif.running → cluster.get.notif.running.jobs()
c.per.pp.allocated → len(c.pp.running)/len(pp.containers)
c.per.notif.allocated → len(c.notif.running)/len(notif.containers)
c.notif.detected → cluster.notif_detected

slots → get(jobslot.get.available.slots())
per.pp.slots = 0, per.notif.slots = 0, per.slots = 0
if slots is not 0 then
    per.pp.slots = jobslot.get.pp.slots()/slots
    per.notif.slots = jobslot.get.notif.slots()/slots
    per.slots = slots/len(jobslot.get.job.slots())
end if

current.size, pp.size, notif.size = backlog.get.backlog.job.count()
backlog.holds → len(backog.get.backlog()
per.backlog.size → current.size/backlog.holds
per.pp.b.size → pp.size/backlog.holds
per.notif.b.size → notif.size/backlog.holds

if c.per.pp.allocated is 100 then
    if c.per.notif.allocated is 100 then
        if per.slot is 100 then
            a=0
        else if per.slot > 50 then
            if c.notif.detected > 0 then
                a=5
            else
                a=0
            end if
        else
            if c.notif.detected > 0 then
                a=5
            else
                a=2
            end if
        end if
    else if c.per.notif.allocated < 50 then
        if c.per.notif.allocated > 80 then
            a=1
        else
            a=2
        end if
    else
        if c.per.notif.allocated < 50 then
            a=5
        else
            if per.slot is 1 then
                a=0
            else
                a=2
            end if
        end if
    end if
else if c.per.pp.allocated < 80 then
    if per.pp.b.size > 10 then
        a=1
    else if per.notif.b.size > 1 then
        a=7
    else
        a=0
    end if
end if
```

- *c.notif.detected*: This parameter informs us if a scheduled job is a *notif* or *pp* job (used only in $Same\_VM$, and linked with Action 5 (for more info see Table 5.2), which is also only used only $Same\_VM$).

Here is the list of the following *job_slot* parameters used in both algorithms:

- *slots*: The number of available ("free") slots.
- *per.pp.slots*: The percentage of slots that hold *pp* jobs.
- *per.notif.slot*: The percentage of slots that hold *notif* jobs.
- *per.slots*: The percentage of slots that are still "free".

Here is the list of the following *backlog* parameters used in algorithms:

- *backolog.holds*: The backlog capacity.
- *per.pp.b.size*: The percentage of *pp* jobs in the backlog.
- *per.notif.b.size*: The percentage of *notif* jobs in the backlog.

5.1.4 Tuning of Rewards

We executed the following NN training sessions to find the best-performing Rewards parameters. We used NN1, the simplest of the three neural networks, for this set of tests. Six NN1 neural network training were performed with 200 training iterations, p_{notif} =0.2, and rewards parameters are shown in Table 5.3. Once the six NNs had been trained, we performed six tests for each of the trained NNs, where we changed the notification load.

This training session was performed to find the best performing Rewards parameters. Six NN trainings were executed with: 200 training iterations, p_{notif} =0.2, and rewards parameters are shown in Table 5.3. Once the NNs were trained, we performed six tests for each trained NNs, where we were changing the notification load. The test values for p_{notif} are: 0.1, 0.2, 0.3, 0.4, 0.5 and 0.6.

Figure 5.3 shows the results of the tests. In Figure 5.3, we omitted the results of TEST 3 and TEST 4 since training NN with those parameters produced significantly worse results than those achieved with the other tests. Furthermore, Figure 5.3 shows that TEST1 and TEST6 have jobs and notification jobs that did not complete (TEST1 -30% of jobs

Table 5.3: Rewards Parameters for Tests

Id	β	γ	α	α_1	α_2	α_3	α_4	γ_1	γ_2
TEST1	0.5	0.5	0.5	3	0.7	1	0.5	0.0003	0.5
TEST2	0.1	0.45	0.45	3	0.7	1	0.5	0.0003	0.5
TEST3	0.1	0.45	0.45	5	2	1	0.5	0.005	0.5
TEST4	0.1	0.45	0.45	5	2	1	0.5	0.0003	0.5
TEST5	0.1	0.45	0.45	3.3	1.1	1	0.5	0.0003	0.5
TEST6	0.1	0.45	0.45	3.2	0.9	1	0.5	0.0003	0.5

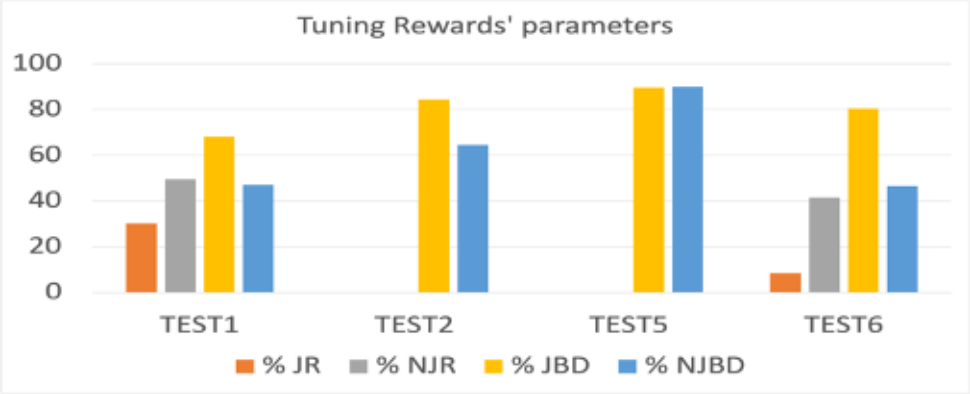


Figure 5.3: Performance evaluation for different test settings, where JR - jobs remained, NJR - notification jobs remained, JBD – jobs completed before the deadline, NJBD – notification jobs completed before the deadline.

and 49.5% of notification jobs, and TEST6 - 8.3% jobs, and 41.5% of notification jobs). Furthermore, TEST 5 performs the best: 89.7% of jobs completed before the deadline (JBD) and 90.0% of notification jobs completed before the deadline (NJBD). The second best is TEST2, with 84.5% JBD and 64.5% NJBD.

Figure 5.4 shows the test results after running TEST2, TEST5, and TEST 6 with p_notif of 0.1, 0.2, 0.3, 0.4, 0.5, and 0.6. TEST 5 has the best performance results, where JBD ranges from 88.2 to 89.7%, and NJBD ranges from 86.36 to 90.0%.

Surprisingly, in the case of TEST5, changing the $p\{notif$ notification percentage does not significantly affect the performance of the TEST5 NN Agent; JBD and NJBD in TEST 5 stay almost the same even if we increase the p_{notif}.

The tests show that selecting different values for the Reward Parameters significantly

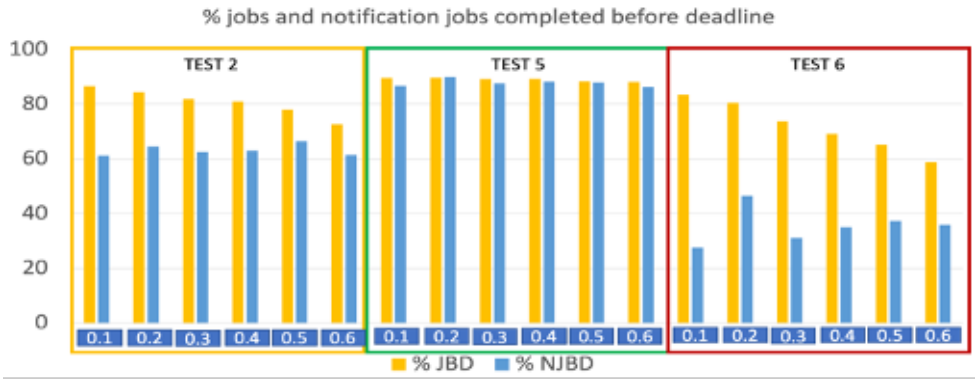


Figure 5.4: Test results of running NNs with p_{notif} of 0.1, 0.2, 0.3, 0.4, 0.5 and 0.6.

affects the NN's performance. For example, using NNs trained with TEST3 and TEST4 produced significantly worse results than the other tests. Furthermore, event TEST1 and TEST6 perform better than TEST3 and TEST4, but still, there are jobs and notification jobs that are never completed. TEST5 performs the best, with 89.7% of jobs completed before the deadline (JBD) and 90.0% of notification jobs completed before the deadline (NJBD). The second-best result was TEST2, with 84.5% JBD and 64.5% NJBD.

To our surprise, in the case of TEST5, changing the p_{notif} notification percentage does not significantly affect the performance of the TEST5 NN Agent; JBD and NJBD in TEST5 stay almost the same even if we increase the p_{notif}. For all future tests, the Reward parameters in TEST5 are used during the NNs training.

5.1.5 Other Agents Used for Comparative with Neural Network NN1

An important question is how the NN agent in TEST 5 compares with well-known CPU scheduling algorithms FCFS, SJP, LJF, and heuristic algorithms $First\_VM$, $Same\_VM$ and $Random$ described in Section 5.1.3. Figure 5.5 shows test results using the 0.2 notification workload. We can notice that classical scheduling algorithms FCFS, SJF, and LJF perform very poorly (jobs completed before the deadline are 25.2% for FCFS, 23.3% for SJF, 33.4% for LJF, and the notification jobs completed before the deadline are 0% for all algorithms). Thus, further in our research, we only compared $RCN - DRL$ with $First\_VM$, $Same\_VM$, and $Random$.

Figure 5.7 presents test results of JBD for NN1, $First\_VM$, $Same\_VM$ and $Random$

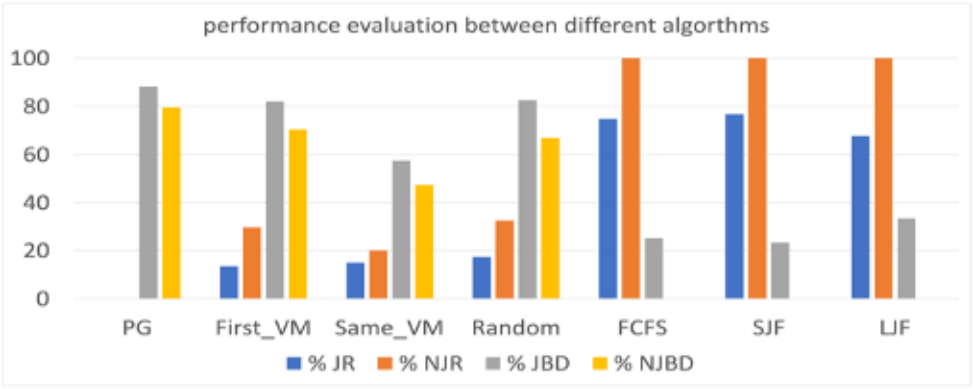


Figure 5.5: Performance evaluation between different algorithms (x-axis). The algorithms are evaluated for % JR, % NJR, % JBD, % NJBD (y-axis). Where % JR presents the percentage of jobs that remained uncompleted, % NJR presents the percentage of notification jobs remained uncompleted, % JBD presents the percentage of jobs completed before deadline, % NJBD present the percentage of notification jobs that completed before deadline.

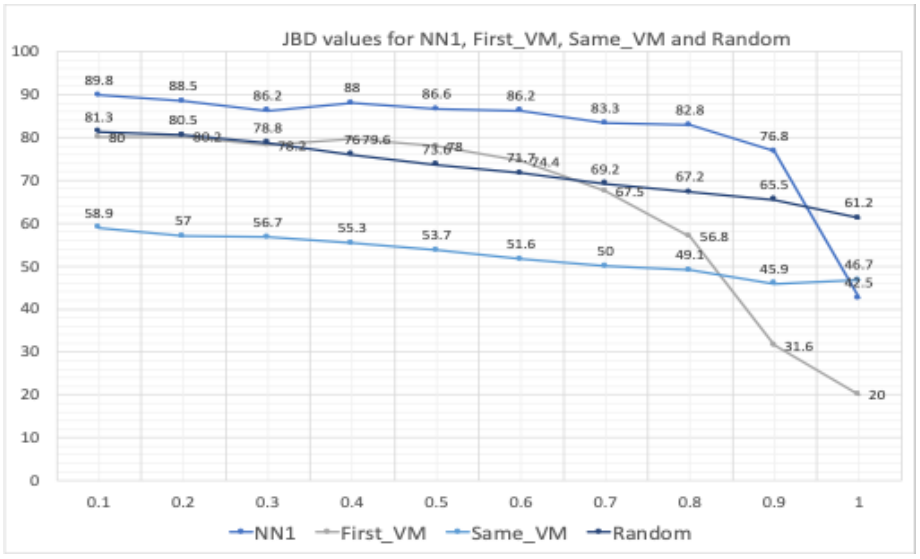


Figure 5.6: JBD - jobs completed before the deadline. The x-axis presents p_{notif} and the y-axis presents the percentage of jobs completed before the deadline.

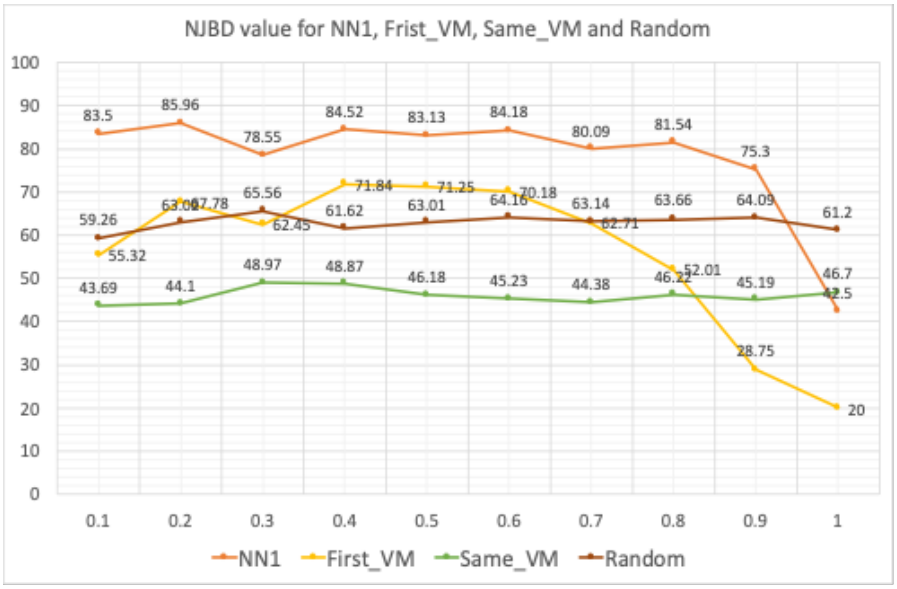


Figure 5.7: NJBD - notification jobs completed before the deadline. The x-axis presents p_{notif} and the y-axis presents the percentage of notification jobs before the deadline.

agents for $notification\_workload$ p_notif range from 0.1 to 1. To our surprise, the PG stays consistent with the increase of $notification\_workload$ until 0.9 p_{notif}, where, after, that has a significant drop from 76.8 to 42.5%. It is essential to mention that this differed from the other Reward parameter selections.

Figure 5.8 presents test results of NJBD for NN1, $First_{VM}$, $Same_{MV}$ and $Random$ agents for notification $total\_system\_workload$ p_{notif} range from 0.1 to 1. Again, to our surprise, the NN1 stayed consistent with the increase of $notification\_workload$ until 0.9 p_{notif}.

5.2 Training Neural Networks

Deep Reinforcement Learning (DRL) uses a Neural Network (NN) as an RL Agent to make decisions. In this work, we developed three NNs to compare their effectiveness, training and execution time. There are the following three NNs: a) Small Compact NN (NN1), b) Compact NN (NN2), and c) Small Convolutional NN (NN3). More details for all three NNs will be presented further in the text.

SCN-DRL uses one of the three NNs mentioned above as a policy. In DRL, the input of the NN is the state space; a district image is input; and the output is possible distributions of all actions in the action space. The SCN-DRL uses the Policy Gradient (PG) RL technique for training the neural network [29]. The training is done by repeating the same execution process E times, known as an episode (or epoch in RL terminology), with a different data set. Number E presents the number of episodes the system will take during the policy training. The episode terminates when all jobs are executed, or the maximum execution time is reached. The datasets (batches) used for training and testing are synthetically generated. The batch size is n, the number of requests in the system. The p_{notif} is the percent of total jobs that require notification and will be processed on the "notification" microservice. However, exactly which job will require the notification processing is known only after the probabilistic interference is completed [3] [2].

To speed up the training process of reinforcement learning, we first used the imitation learning algorithm for pre-training, also known as BC [27], which is similar to supervised learning. Then, we used the $First\_VM$ heuristic algorithm as a mock object for behaviour cloning, previously covered in Section 5.1.3. Next, we used generated datasets as data samples in supervised learning. Then, using the $First\_VM$ algorithm, the corresponding scheduling order of the samples is determined, and these sequences are considered the labels of the samples. Finally, the results were obtained and divided into two groups: the training and the test sets. Then, we train the NN by using the training set. Pre-training of the NN is terminated when the accuracy of the test set stops increasing, and the model parameters are saved. Finally, we used the pre-trained NN model parameters as a starting point for deep reinforcement learning, which saves learning time.

To implement our NNs, we used the following open-source libraries: Theano [77] and Lasagne [78]. Theano is an open-source project, a Python library that effectively allows users to evaluate mathematical operations, including multi-dimensional arrays. It is primarily used in building Deep Learning Projects. It works much faster on the Graphics Processing Unit (GPU) than on the CPU. Theano attains high speeds that give tough competition to C implementations for problems involving large amounts of data.

Furthermore, it can take advantage of GPUs, which makes it perform better than C on a CPU by large orders of magnitude under certain circumstances [77]. Theano is a prevalent library in the field of Deep Learning. It is mainly designed to handle the types of computation required for large neural network algorithms in Deep Learning. Theano is designed to know how to take structures and convert them into very efficient code that uses NumPy and some native libraries [77]. Lasagne [18] is a lightweight library to build and train neural networks in Theano.

For our experiment, we built the following three NNs using the Theano and Lasagne libraries:

1. Small compact neural network (NN1). The Small Compact Neural Network (NN1) consists of one hidden layer with 20 neurons and 89,451 parameters.

2. Compact neural network (NN2). The Compact Neural Network (NN2) consists of three hidden layers. The first hidden layer is a DenseLayer of 520 units. The second hidden layer is also a DenseLayer of 20 units, and the outer layer is a DenseLayer of 20 units.

3. Small convolutional neural network (NN3). The Small Convolutional Neural Network (NN3) consists of one hidden layer. The hidden layer is Conv2D of 20 units. The outer Layer is a DenseLayer of 20 units.

The SCN-DRL model synthetically generated data has been used for training and testing. If not otherwise specified, the batch size for all performance tests was 100, presenting 100 complex events in the system. In addition, it is known that p_{notif} percent of total jobs in the batch will be required to be processed in the notification microservice, and which one will require the notification processing is known after the probabilistic programming process is completed [2] [3]. Jobs that required notification were selected randomly. A uniform distribution (from 0.9 p_{notif} to 1.0 p_{notif}) was used for the job selection. Then, according to the algorithm of $First\_VM$, the corresponding scheduling order of the samples is calculated. These sequences are considered as the labels of the samples. By doing that, we are shortening the neural network training time, and this technique is known as Imitation learning (Section 2.3.2). Finally, the results obtained in the previous step are divided into the training set (80% of results) and the test set (20% of results). The training is used to pre-train the neural network, and the test set is used to validate the neural network model's accuracy.

We conducted a preliminary evaluation of SCN-DRL for all three NNs, to answer the following research questions:

1. How does the length of $arrival\_time\_period$ affect the performance of neural networks that are previously trained?

2. How does a change in the $notification\_workload$ affect performance?

3. How are NNs resilient when resource containers are dropped?

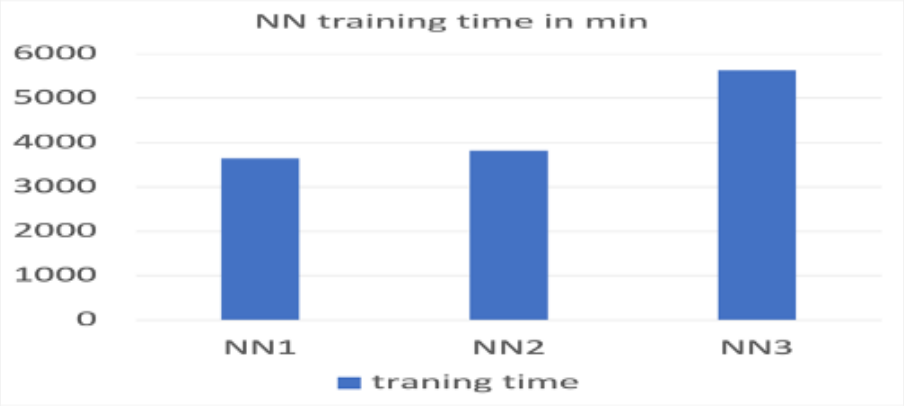


Figure 5.8: NJBD - notification jobs completed before the deadline. The x-axis presents p_{notif} and the y-axis presents the percentage of notification jobs before the deadline.

5.3 Workload

We mimic the setup described in Section 4.1. Specifically, complex event jobs arrive, and a p_{notif} percentage of total jobs must be processed in the notification microservice. The cluster is deployed based on the deployment schema presented in TABLE 5.1. The deployment schema specifies the required number of VMs and microservices containers per VM. The PP_MS job's length is uniformly distributed between 45 and 55 milliseconds. The $NOTIF\_MS$ job's length is chosen to be distributed uniformly between 18 and 22 milliseconds. The notification latency between two VMs is distributed uniformly between 18 and 22 milliseconds. Finally, if not specified otherwise, the $arrival\_time\_period$ is set to 20 milliseconds.

5.3.1 Policy Gradient Agents - Neural Networks

We built the three SCN-DRLs prototypes using the three NNs covered in Section 5.2. An example of State space, aka "images", used by the PG agent is presented in Figure 5.9. The SNC-DRL agent allocates from a subset of 10 jobs but can also observe the number of other jobs ("backlog", which we set to 100 job slots). We used 100 different jobsets during training. We run N = 20 Monte Carlo simulations in parallel per jobset in each training iteration. We updated the policy network parameters using the rmsprop [28] algorithm; the learning rate is set to 0.001. Unless otherwise specified, SNC-DRL is trained with 200 training iterations.

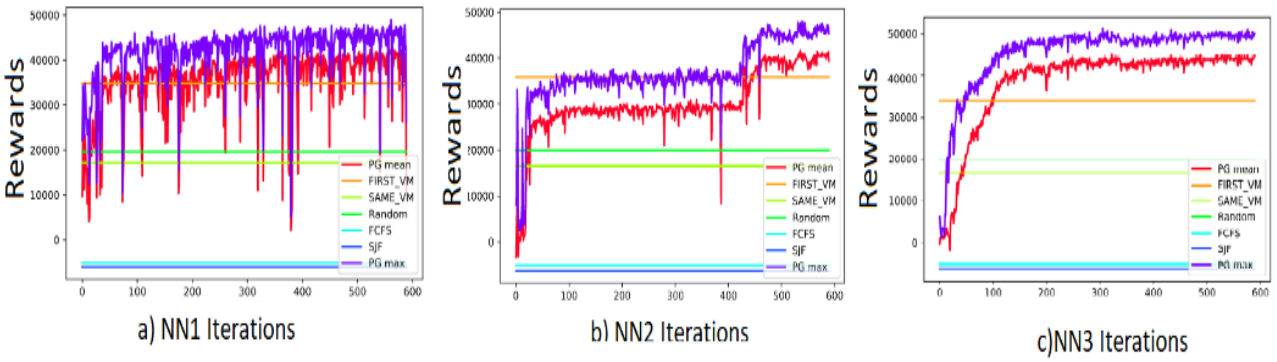


Figure 5.9: Observes NNs and plots rewards a) NN1, b) NN2, and c) NN3. The x-axis shows the number of iterations and the y-axis shows the total rewards

5.4 Performance Evaluation

The objective of the following tests was to compare three neural networks: NN1, NN2, and NN3. All three are present with their characteristics in Section 5.2. The NN1 is the simplest NN, a Fully Connected Neural Network, with only one hidden layer. The second NN2, a Fully Connected Neural Network, has three hidden layers. Finally, the third NN3, a Convolutional 2D Neural Network, has one hidden layer. All three neural networks are trained with the same settings and parameters. Neural networks are trained by execution of 600 training iterations (epoch) with p_notif =0.2 $notification\_workload$. The rewards parameters used in the setting are from TEST5, which had the best-observed performance, shown in Table III. Training and evaluation of the NNs were done on a Mac Mini M1 (8-Core, 8-Core GPU), with 8 GB memory and 512 GB of storage.

Figure 5.9 shows the training time for all three NNs. As expected, the shortest time was 3645 min (60 hours and 45 min) achieved with NN1. The NN2 had the second shortest training time of 3818 min (63 hours and 38 minutes), and the longest training time 5626 min (93 hours and 46 minutes) achieved with NN3. The NN2 took 4.7% more time than NN1, and NN3 took 53% more time than NN1 to train. Performance reporting and the neural network parameters have been stored in 50 iteration intervals during the neural network training.

The following text will discuss five collected performance objectives for all three NNs. The following performance objectives are:

1. the total reward for the average number of total jobs completed by the *deadline D*

2. the average number of notification jobs completed by the *deadline* D

Figure 5.9 shows NNs (NN1, NN2, and NN3, respectively) and maximum plots rewards (PG max curve on the diagram).

5.4.1 Total Reward During the NNs Training

This section shows how the total reward for all three NNs changed during the training period. Figure 5.9 shows total rewards results for all three NNs (a presents NN1, b presents NN2, and c presents NN3) across all of the Monte Carlo Runs at each iteration and the average rewards (PG mean curve on the diagram). The NNs are also compared with some heuristic algorithms ($First\_VM$, $Same\_VM$, $Random$, $FCFS$ and SJF). Total rewards have been shown as constant values (horizontal lines) since they do not change over time.

The NN1 (Figure 5.9 a) starts to perform better than $Fist\_VM$, after the short training period (approximately 50 intervals). However, it continues to have substantial performance oscillations until the end of training.

Interestingly, the NN2 (Figure 5.9 b) is not rewarded more than $First\_VM$ for a long training period. The tipping point is approximately 400 intervals, when NN2 has a noticeable reward jump. After that point, the learning continues without huge oscillations. NN2 takes the longest training time to converge to the same reward as NN1 and NN3. NN3 (Figure 5.9 c), on the other hand, has steady convergent, and oscillation is minimal.

5.4.2 The Average Deadline

In this section, we discuss the average number of jobs completed by *deadline* D during the training period. All three NNs are considered.

Similar results observed for the total reward can be observed in this test. NN1(Figure 5.10 a) again has the highest oscillation in the near time than the other two NNs. NN2 (Figure 5.10 b) it takes the longest time to perform better than the heuristic $First\_VM$ algorithm (approximately 400 intervals), and NN3 has steady and minimal oscillations.

5.4.3 Performance of Notification jobs Before the Deadline

In this section, we discuss the average number of notification jobs completed by the *deadline* D during the training period. All three NNs are considered.

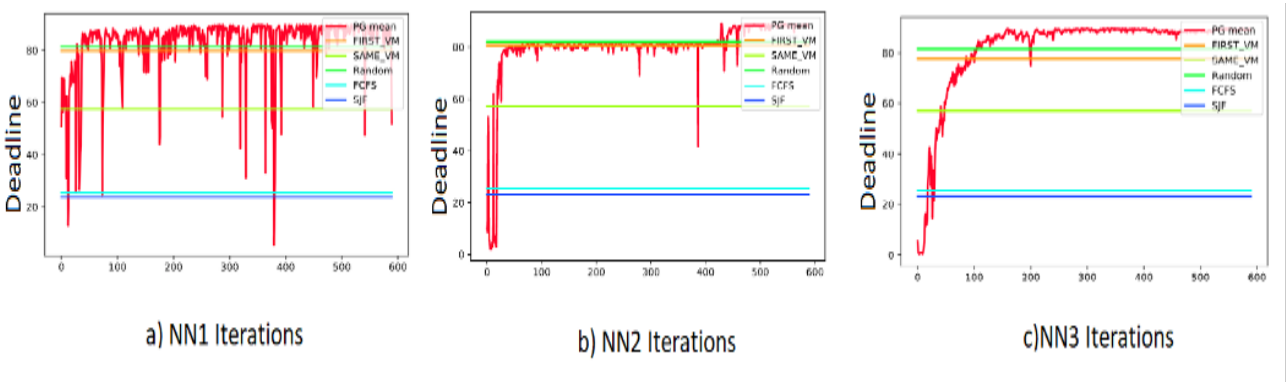


Figure 5.10: The percentage of jobs completed by deadline D for all three NNs a) NN1, b) NN2 and c) NN3). The x-axis shows the number of iterations and the y-axis shows the total rewards

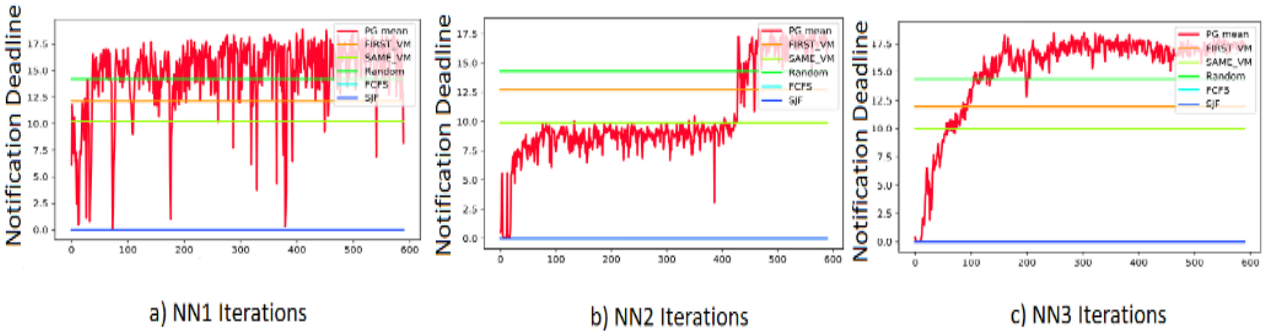


Figure 5.11: The percentage of notification jobs completed by deadline D for all three NNs (a) NN1, b) NN2 and c) NN3). The x-axis shows the number of iterations and the y-axis shows total rewards.

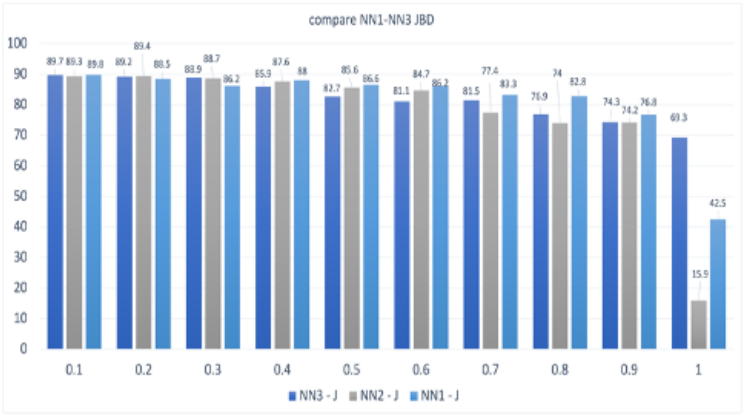


Figure 5.12: Compare the performance of all three NNs regarding the percentage of all jobs completed before the deadline.

Figure 5.10 shows the percentage of jobs completed by deadline D for all three NNs.

Again, we see a similar pattern to the one we saw in previous observations. NN1 (Figure 5.10 a) again has the highest oscillation in the near time values than the other two NNs. NN2 (Figure 5.10.12 b) takes longer to improve performance compared to the heuristic $First\_VM$ algorithm, and NN3 (Figure 5.10 c) has steady rewards growth over training time with minimal oscillations in performance values.

In conclusion, regarding the training time, the NN1 takes the shortest time to train, followed by the NN2 where the time increases by 4.7%, over the NN1. The NN3 requires the longest time, which is expected, to train, since it is built by a convolutional 2D layer. The NN3 requires 53% more to train than NN1.

Figure 5.9 shows the percentage of jobs completed by *deadline D* for all three NNs.

Let us compare the NNs regarding the recorded performances. First, we can notice that the NN3 steadily converges with minimal oscillation in recorded values in all recorded tests, opposite to the NN1, where values oscillate a lot. In the following text, we will evaluate NNs' performance after completing NNs training.

5.4.4 Performance Testing of Neural Networks

Once the NNs were trained, we performed additional performance tests for each trained NN. The following four performance tests try to answer the following questions:

1. How does changing p_notif affect performance?
2. How does changing $arrival\_time\_period$ affect performance?
3. How does changing $total\_system\_workload$ affect performance?
4. How does the reduction of container resources affect performance?

These questions are discussed in the following sections.

5.4.5 How Does Changing p_notif Affect Performance?

In this set of tests, we used previously trained NNs (Section 5.2). The tests compare how three NNs perform when the $notification\_workload$ p_notif changes from 0.1 to 1; the results are presented in Figure 5.11 for JBD and Figure 5.12 for NJBD.

Please note that in Figure 5.11 NN1 still holds the JBD value the best with an increase of the p_notif up p_notif = 0.9, then at p_notif = 1 JBD significantly drops to 42.5%. The second best is NN3, which surprisingly has the smallest drop at p_notif =1 where the $notification\_workload$ is 69.3%. Finally, the NN2 has the steepest drop at p_notif =1 where the $notification\_workload$ to 15.9%.

Figure 5.12 shows NJBD test results, where the NN2 performs better than the NN1 and the NN3 for p_notif from 0.1 to 0.4. then, the NN1 continues to hold the NJBD steady until p_notif =0.9, then drops to 42.5% for p_notif=1. The NN3 again has the smallest drop in performance for the p_notif =1, where the performance value is 69.3%. In conclusion, all three NNs' performance results are comparable for JBD and NJBD for 10 to 90% of the $notification\_workload$. However, when the $notification\_workload$ is 100% NNs performed significantly better at 69.3%, compared with NN1 at 42.5% and NN2 at only 15.9%.

5.4.6 How Does Changing $arrival\_time\_period$ Length Affect Performance?

Figure 5.14 shows JBD's performance results for all three NNs where $arrival\_time\_period$ was decreased/increased. The y-axis shows the percentage of jobs completed before the deadline (JBD), and the x-axis shows NNs test results. NN1 reg, NN2 reg, and NN3 reg correspond to nominal test results. NN1 -10%, NN2 -10%, and NN3 -10% corresponds

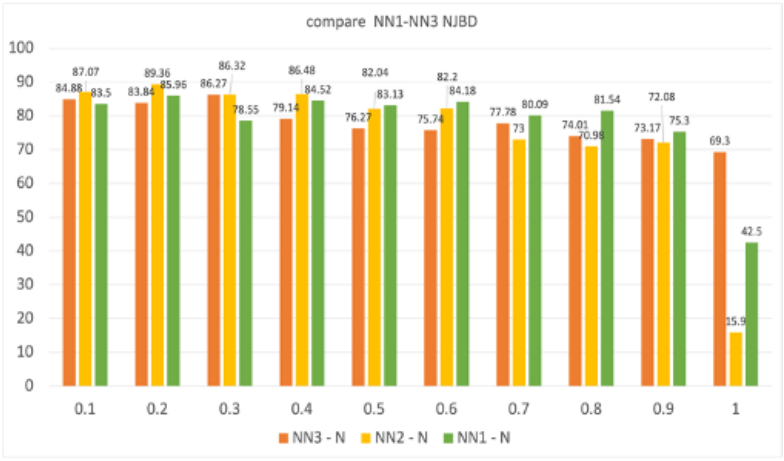


Figure 5.13: Performance tests on all three NNs regarding the percentage of all jobs completed before the deadline. The x-axis shows p_notif from 0.1 to 1, the y-axis shows the percentage of notification jobs completed before the deadline (NJBD), where the red bars present NN3, yellow bars present NN2 and green bars present NN1

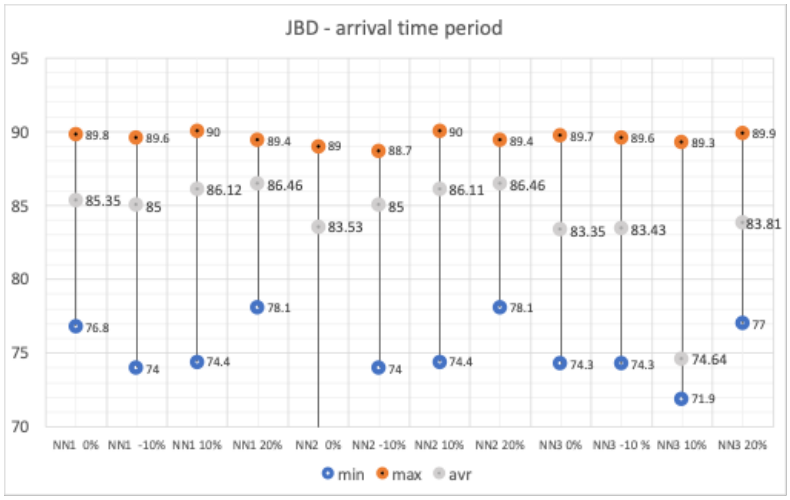


Figure 5.14: *arrival_time_period* performance tests for all three NNs regarding the percentage of all jobs completed before the deadline (JBD). The x-axis shows for all three NNs performance test results when *arrival_time_period* is 20ms (0% change), 18 ms (10% decrease), 22 ms (10% increase) and 24 ms (20% increase). The y-axis shows minimal, maximum and average results values.

to a decrease of 10% in the *arrival_time_period*. NN1 10%, NN2 10%, and NN3 10% correspond to increase by 10% of *arrival_time_period*, and NN1, 20%, NN2 20% and NN3 20% increase of *arrival_time_period*.

For this set of tests, we used previously trained NNs (Section 5.2). During NNs training *arrival_time_period* was set to 20 ms. The following three tests were executed where:

1. *arrival_time_period* is increased by 20% (24 ms)
2. *arrival_time_period* is increased by 10% (22 ms)
3. *arrival_time_period* is decreased by 10% (18 ms)

All test results are compared with nominal ones when the *arrival_time_period* is 20 ms.

Fig. 15 shows NJBD's performance results for all three NNs tests (NN1, NN2, NN3) where *arrival_time_period* was decreased/increased. Again, the image shows minimum (min), maximum (max), and average values. We expected that increasing the *arrival_time_period* would lead to better performance. Instead, the results show non-significant performance improvements for all three NNs.

Figure 5.15 shows NJBD's performance results for all three *arrival_time_period* decreased/increased tests. Again, we expected that increasing the *arrival_time_period* would lead to better performance, but the results show non-significant performance improvements. In conclusion, all three NNs showed no significant performance change when the *arival_time_period* was changed during the test for pre-developed neural networks. However, the question is still open and will be part of future research to compare NNs trained with different *arrival_time_periods*.

5.4.7 How Does Changing *total_system_workload* Affect Performance?

For this set of tests, we used previously trained NNs (Section 5.2). During NNs training, the *total_system_workload* was set to 100 DTs, the system capacity, which is also the maximum number in our environmental setup. The three different case are considered:

1. The *total_system_workload* is reduced by 5% of the system capacity (95 DTs)
2. The *total_system_workload* is reduced by 10% of the system capacity (90 DTs),

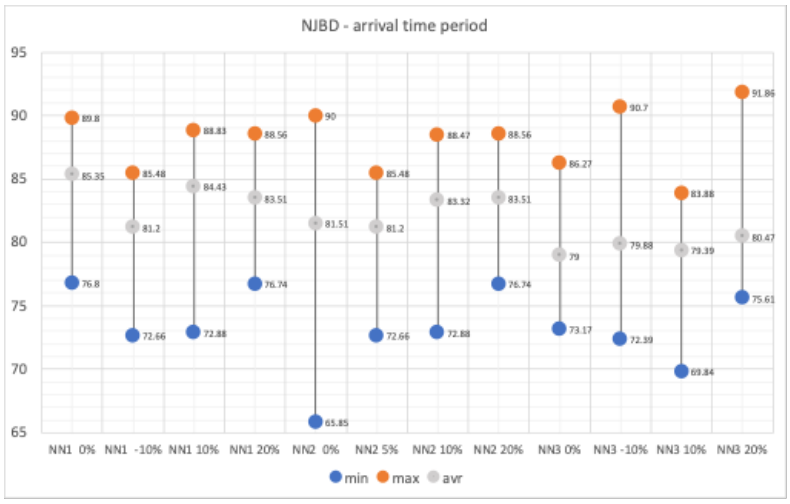


Figure 5.15: NJBD's performance results for all three NNs tests where *arrival_time_period* was decreased/increased. The y-axis shows the percentage of notification jobs completed before the deadline (NJBD), and the x-axis shows NNs test results.NN1 reg, NN2 reg and NN3 reg present nominal test results. NN1 -10%, NN2 -10%, and NN3 -10% present a decrease of 10% in the *arrival_time_period*. NN1 10%, NN2 10%, and NN3 10% present increase by 10% of *arrival_time_period*, and NN1, 20%, NN2 20% and NN3 20% increase of *arrival_time_period*.

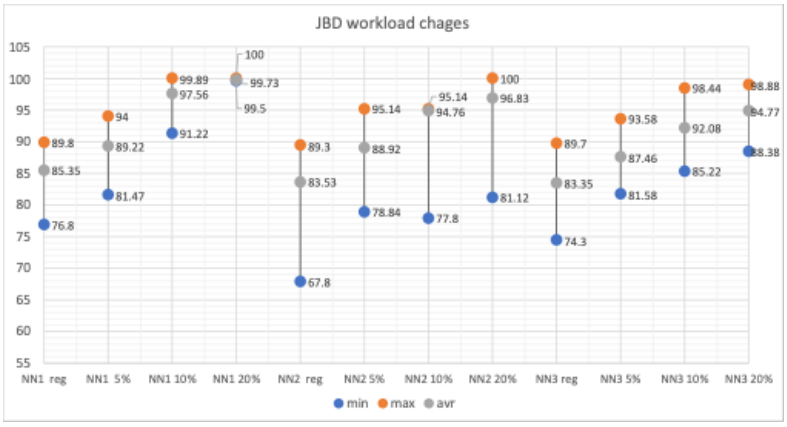


Figure 5.16: shows JBD's performance results for decreased workload test for all three NNs. The y-axis shows the percentage of jobs completed before the deadline (JBD), and the x-axis show NNs test results in NN1 reg, NN2 reg and NN3 reg, presenting the system capacity. NN1 5%, NN2 5%, and NN3 5% present a decrease of 5% in the system capacity. NN1 10%, NN2 10%, and NN3 10% present a decrease of 10% in the system capacity, and NN1 20%, NN2 20% and NN3 20% decrease the system capacity.

3. The *total_system_workload* is reduced by 20% of the system capacity (80 DTs)

All tests are executed separately for each of the NNs. Finally, the results are compared with test results when the *total_system_workload* is 100 DTs (considered system capacity).

Figure 5.16 shows JBD's performance results for all three NNs. Even a 5% decrease in the *total_system_workload* significantly improves the performance results: on average, a 4.53% for NN1, a 6.31% for NN2, a 4.93% for NN3, increase in performance. In addition, a 10% and 20% of *total_system_workload* decrease improved performance even more. With the 10% decrease in *total_system_workload*, the performance increased by an average of 14.3% for NN1, 13.44% for NN2, and 15.9% for NN3. With the 20% decrease in the *total_system_workload*, the performance increased by an average of 16.84% for NN1, 15.92% for NN2, and 13.7% for NN3.

In conclusion, all three NNs showed performance improvements when the test value *total_system_workload* was decreased than the *total_system_workload* used for training NNs. In addition, we can observe that NN1 has a higher average and minimum performance value than the other two NNs.

Figure 5.17 shows NJBD's performance results for the decreased *total_system_workload*

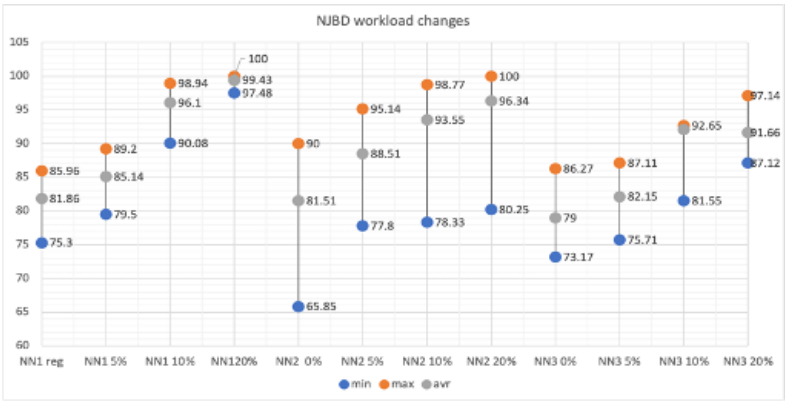


Figure 5.17: shows NJBD's performance results for decreased *total_system_workload* tests for all three NNs. The result shows significant performance improvement with t a 5% decrease in the *total_system_workload*. On average, 4% for NN1, 8.58% for NN2, and 3.98% for NN3 increase performance results. Furthermore, a 10% and 20% decrease of *total_system_workload* improved performance results even more. With the 10% decrease in *total_system_workload*, the performance results increased by an average of 17.39 for NN1, 14.77% for NN2, and 17.27% for NN3. With the 20% decrease in the *total_system_workload*, the performance results increased by an average of 21.14% for NN1, 18.19% for NN2, and 16.02% for NN3.

test for all three NNs. The y-axis shows the percentage of notification jobs completed before the deadline (NJBD), and the x-axis shows NNs test results. NN1 reg, NN2 reg and NN3 reg present the system capacity. NN1 5%, NN2 5%, and NN3 5% present a decrease of 5% in the system capacity. NN1 10%, NN2 10%, and NN3 10% present a decrease of 10% in the system capacity, and NN1 20%, NN2 20% and NN3 20% decrease the system capacity.

5.4.8 How does the Reduction of Container Resources Affect Performance?

In this set of tests, we would like to determine how trained NNs are resilient to resource availability. To be more precise, we would like to learn how performance is affected when the number of containers is reduced, for example, due to container failures.

For this set of tests, we used previously trained NN (Section 5.2). During the NNs training, the available resources have been set to 30 PP_MS containers and 20 $NOTIF\_MS$ containers. In the following three tests, we decreased the number of available containers in the system:

1. The number of available PP_MS and $NOTIF\_MS$ containers dropped by 5% (which was 28 pp_ms and 19 $notif\_ms$ containers)
2. the number of available PP_MS and $NOTIF\_MS$ containers dropped by 10% (which was 27 pp_ms and 18 $notif\_ms$ containers)
3. the number of available PP_MS and $NOTIF\_MS$ containers dropped by 20% (which was 24 PP_MS and 16 $NOTIF\_MS$ containers)

All tests are executed separately for each of the NNs. Finally, the results are compared with test results with the maximum number of containers.

Figure 5.18 The JBD performance result Figure 5.19 The NJBD performance results during the container reduction tests for all three NNs. The y-axis shows the percentage of notification jobs completed before the deadline (NJBD), and the x-axis shows NNs test results. NN1 reg, NN2 reg, and NN3 reg present the system capacity. NN1 5%, NN2 5%, and NN3 5% present a decrease of 5% in the system capacity. NN1 10%, NN2 10%, and NN3 10% present a decrease of 10% of the system capacity, and NN1 20%, NN2 20% and NN3 20% decrease the system capacity.

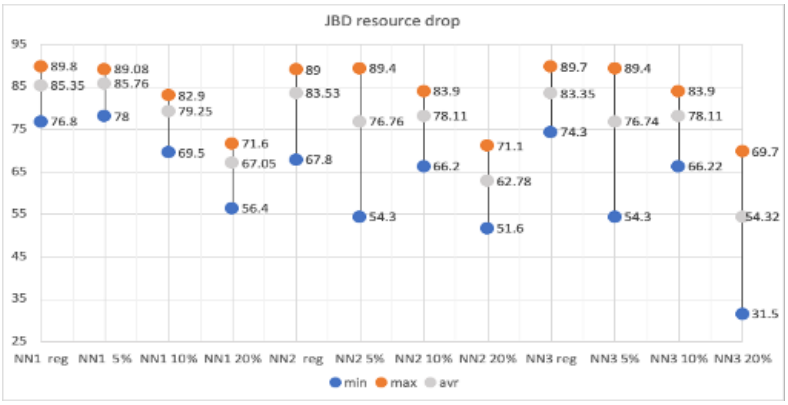


Figure 5.18: shows the JBD performance results for all three NNS when the total number of resource containers is reduced. The NN1 test results show that 5% of container reduction in the case of the NN1 event slightly improved. NN2 and NN3 5% test results show that the maximum value is comparable with NN2/NN3 reg value, but the average and minimum values have a drop. With the reduction of 10% of containers, on average, the performance dropped by 7.14, 6.48, and 6.48%. Furthermore, with a reduction of 20% in containers, the performance dropped by 21.44, 24.84, and 34.82% on average.

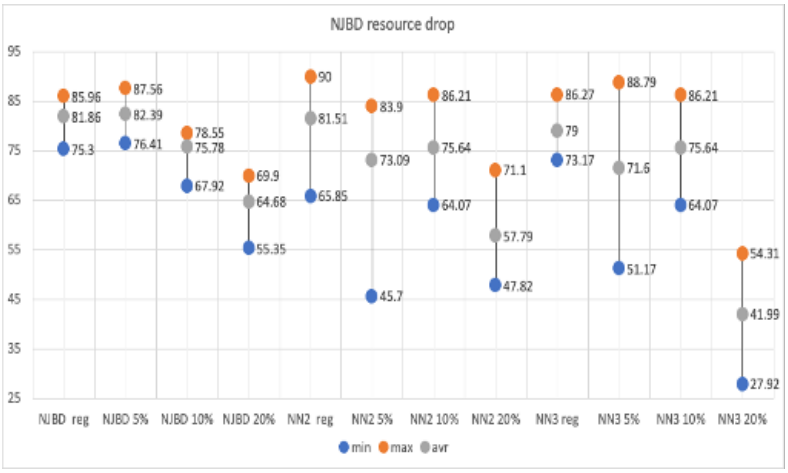


Figure 5.19: shows the NJBD performance results for all three NNS when the total number of resource containers is reduced.

The NN1 test results show that 5% of container reduction improved in the case of NN1. NN2 test results show a drop for maximum, average, and minimum values. NN3 test results show a small increase in maximum value and a drop in average and minimum value. With the reduction of 10% of resource containers, on average, the performance dropped by 7.14, 6.48, and 6.48%. Furthermore, with a reduction of 20% in containers, the performance dropped by 21.44, 24.84, and 34.82% on average.

In conclusion, all three NNs showed resilience in the 5% reduction of containers, which means that the system will have time to recover dropped containers. However, all three NNs showed a drop in JBD and NJBD average performances from 6.48 to 7.14% after the 10% reduction of containers. Furthermore, a 20% reduction in containers shows a drop of over 20% in performance for both JBD and NJBD.

5.5 Discussion

This chapter presents a multi-objective optimization scheduling approach for large-scale micro-service critical notification system applications, SCN_DRL, based on Reinforcement Learning. SCN_DRL is an example solution that translates the containerized micro-service cloud cluster scheduling problem into a learning problem.

Our initial results show that SCN_DRL performs better than well-known heuristic algorithms (FCFS, SJF, LJF). In addition, it performs better than two heuristics algorithms introduced in this chapter: "$First\_VM$" and "$Same\_VM$".

In this chapter, we present our experiments and results with three types of NNs: 1) NN1 is built using the fully connected NN with one hidden layer, 2) NN2 is built using the fully connected NN with three hidden layers and 3) NN3 is built using 2D Convolutional NN. Lead by the literature review [66] where the Convolutional 2D NN significantly improved performance over the connected NN, we decided to research and compare the performance of the "$First\_VM$" with not only connected NNs (NN1 and NN2) but also with Convolutional 2D NN (NN3).

Our research results did not show the advantage of choosing the Convolutional NNs (NN3) over connected NNs (NN1 and NN2) to perform the scheduling jobs in the system observed in this chapter. It is important to note that while the current findings did not showcase a performance advantage for Convolutional NNs, more research is needed. The future research direction will be to build the Convolutional NN with more hidden layers and/or consider a large workload.

In addition, the research further indicates that the "SCN_DRL" approach exhibits consistent performance characteristics in response to changes in notification workload p_{notif}. Specifically, as the notification workload increases from 10% to 90%, the performance of the "SCN_DRL" solution remains steady. This suggests that for the system and workload parameters experimented with, the system's ability to manage notifications remains resilient and unaffected by variations in the notification workload within this range.

Furthermore, the "SCN_DRL" approach demonstrates an interesting behavior when it comes to the total system workload. The research indicates that this approach improves its performance when the total system workload is reduced by 5% compared to the initial system workload. In other words, when the system's overall workload experiences a slight reduction, the "SCN_DRL" solution exhibits enhanced performance characteristics.

Finally, our research findings reveal that the "SCN_DRL" approach, when utilizing all three neural networks (NNs), exhibits remarkable resilience to sudden drops in container resources. Specifically, the solution demonstrates its ability to maintain stable performance levels even when faced with a significant reduction of 10% in container resources. Intriguingly, the solution's performance shows further improvement when container resources experience a 5% drop. This heightened resilience to resource container failures is a crucial attribute for distributed systems, demonstrating the system's capacity to adapt and deliver consistent results even under challenging circumstances. In our future research, we will investigate extending SCN_DRL to consider predictive affinity-based scheduling, where previous DTs states will be considered.

Chapter 6

Conclusions and Future Research

This Ph.D. thesis focused on system scalability, multi-objectivity and scheduling of CE processes using Reinforcement Learning (RL) in Large-Scale Digital Twins Notification Systems (LSDTNS). The dissertation presents a complex methodology that leads to three contributions, IoT Notification System Architecture, MOOP, and SCN-DRL, that addresses the system scalability, multi-objectivity, and scheduling of CE processes using Reinforcement Learning (RL). This chapter summarizes the scientific and business value of all three contributions in Section 6.1. Section 6.2 suggests various directions for future research.

6.1 Summary of Contributions

There are three primary contributions to this dissertation. For each contribution, we will provide a conclusion in the form of a short problem statement, the solution, and the resulting research insights.

6.1.1 IoT Notification System Architecture

During the literary survey, we learned that Complex Event Reasoning (CER), rule-based or probabilistic programming solutions, lack scalability. We also learned that although the existing rule-based CER can address the system's uncertainty, it is not the most feasible solution.

To address the issues, we presented the IoT Notification System Architecture, a microservice-based notification methodology that uses CE recognition to handle the uncertainty

of IoT systems. The IoT Notification System architecture consists of modules (components) and defines the interconnection between those modules through well-defined APIs. The interconnection between modules can be presented as a directed acyclic graph (DAG), which is used in two contributions, MOOP and DTRAS, presented in this thesis. Each module is presented as a micro-service, which is then containerized, making an easily deployable micro-service deployment image.

The proposed IoT Notification Architecture solves the scalability issue that CER systems face and provides few business values. Choosing modularity and well-defined APIs in the architecture allows us to isolate the CER domain knowledge, which allows the system to be easily and quickly maintained, thus lowering the maintenance cost of the system. In addition, choosing the containerized deployment of the micro-services allows deployment in the cloud computing environment and leveraging containerized container scaling capabilities, allowing us to use a more flexible "pay as you go" service provider payment option.

6.1.2 MOOP

Micro-service architecture provides many benefits but also brings some deployment complexity. For example, LSDTNS deployed in the cloud computing environment faces the following challenges: CE has to be processed, and if CEC is reached, the system sends a notification to a predefined recipient to meet the system notification deadline. In addition, the deployment cost should be as low as possible, and the cloud resource utilization should be high. The presented research on MOOP is one of the first works dealing with multi-objective notification applications where the notification load is variable and depends on other priorly executed micro-services in the request process.

MOOP provides a multi-objective mathematical cloud resource deployment model, and its effectiveness is demonstrated through a case study (see Section 4.5). First, MOOP provides the multi-objective mathematical deployment model. Then, the presented work demonstrates the methodology through the case study in three steps. In step one, the required number of microservice containers for handling the expected workload is estimated. Step two focuses on selecting the Pareto optimal front for leading VM-type candidates based on multi-objective optimization using the results from step one. Finally, using the results of step two, step three delivers the deployment model for the preselected VM types.

The case study presented in Section 4.5 demonstrates MOOP methodology that significantly reduces deployment cost by selecting a set of VM-type candidates offered by a commercial Service Cloud provider.

6.1.3 SCN-DRL

CPU Scheduling is a well-researched area; however, applying the well-known CPU scheduling heuristics algorithms in the case of critical micro-service notification architecture where notification load is variable and depends on the results of the other previously processed microservices. In addition, the scheduling has multiple objectives that include controlling the cloud service cost, maximizing cloud resource utilization, and meeting system notification deadlines.

To the best of our knowledge, SCN-DRL is the first solution to the multi-optimization objective for critical notification systems, where the notification load is variable and depends on the results of the other prior processed microservices. SCN-DRL uses DRL that translates the containerized microservice cloud cluster scheduling problem into a learning problem. The results show that SCN-DRL performs better than state-of-the-art heuristics scheduling policies and adapt to different workload conditions without a noticeable drop in performance. During further performance evaluation, SCN-DRL showed a steady performance when the notification workload increased from 10% to 90%. In addition, SCN-DRL, tested with three neural networks, shows that it is resilient to sudden drops in the number of container resources by 10%. Such resilience to resource container failures is a desirable characteristic of a distributed system.

6.2 Future Research

6.2.1 Digital Twin Resource Allocation Scheduler Future work

In Chapter 5, we presented DTRAS. DTRAS processes every incoming DT using complex event criteria to determine if the observed DT requires notification. In our future research, we will investigate extending SCN-DRL to consider predictive affinity-based scheduling, where previous DTs states will be considered.

6.2.2 Digital Twin Priority Controller Algorithm

In Chapter 3, Section 3.2.3, we covered the purpose of the IoT controller module, which is to manage and orchestrate other modules in the system. During the system development phase, a list of k predefined DT resource types for the system is created. Each DT resource type is mapped with a corresponding containerized module type, encapsulating a complex

event recognition technique and/or algorithm. As previously mentioned, selecting the complex event recognition algorithm is a domain-specific problem, and a domain expert will perform the algorithm selection task. The domain expert will select the appropriate algorithm for the corresponding DT resource type based on the domain requirements. In addition, we mentioned that the Digital Twin Priority Controller Algorithm (DTPCA) and the Digital Twin Resource Allocation Scheduler (DTRAS) are subcomponents of the IoT controller module. DTRAS has been covered in Chapter 5, and DTPCA that has been left for future work will be discussed more in the following text.

DTPCA can calculate the DTs priority using the following data: a) the current DT CE data, b) a DT profile, and c) a historical DT priority data.

The solution proposed in Chapter 5 uses the current DT CE data to determine if the DT is in the critical stage. In future work, we propose additionally to use a DT profile. The DT profile is a set of additional information on the DT. For example, a DT profile can contain additional information on a monitored prisoner: the length of the prisoner's sentence, medical condition, psychological state, and similar personal data. Then, the DT profile should be combined with the current DT CE data to define the priority.

Additionally, to determine the priority and critical state of the DT we can use the historical data. The historical data is the data collected in the system by the system over time. Further research on Unsupervised learning for anomaly detection is suggested.

Once DTPCA can group DT based on priority, then this knowledge can be used to design a system that is capable of selecting different algorithms and different types of learning. That will allow us to optimize the cost of the system furter without compromising other system objectives.

References

[1] M. Forouzanfar, M. Mabrouk, S. Rajan, M. Bolic, and H. R. Dajani, "Event Recognition for Contactless Activity Monitoring Using Phase-Modulated Continuous Wave Radar," IEEE Transactions on Biomedical Engineering, Vol: 64, Issue: 2, pp. 479 – 491, Feb. 2017.

[2] M. Vrbaski, M. Bolic, and S. Majumdar, "Complex Event Recognition Notification Methodology for Uncertain IoT Systems Based on Micro-Service Architecture," The IEEE 6th International Conference on Future Internet of Things and Cloud, FiCloud, 2018.

[3] M. Vrbaski, M. Bolic, and S. Majumdar," A Performance-Driven Micro Services-Based Architecture/System for Analyzing Noisy IoT Data," 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid, 2019.

[4] M. Vrbaski, M.Bolic, and S. Majumdar, "Multi-objective optimization for cloud provisioning: A case study in large-scale microservice notification applications," The IEEE 10th International Conference on Future Internet of Things and Cloud FiCloud 2022, Rome, Italy.

[5] M. Vrbaski, M.Bolic, and S. Majumdar, " SCN-DRL: Scheduler for large-scale Critical Notification applications based on Deep Reinforcement Learning," The IEEE 10th International Conference on Future Internet of Things and Cloud FiCloud 2022, Rome, Italy.

[6] M. Vrbaski, M. Bolic, and S. Majumdar, "Deep Reinforcement Learning solution for Scheduling critical notifications in a Digital Twin cluster," ready for submission.

[7] S. Greengard, "The Internet of Things," The MIT press, Massachusetts Institute of Technology, 2015.

[8] E. Alevizos, A. Skarlatidis, A. Artikis, and G. Paliouras, “Complex Event Recognition under Uncertainty: A Short Servey,” Published in Workshop Proceedings of the EDBT/ICDT 2015.

[9] A. Pfeffer, “Practical probabilistic Programming," Manning publication, April 10, 2016.

[10] G. Cugola, and A. Margara, “TESLA: a formally defined event specification language,” In Proceedings of Conference on Distributed-Event Based Systems (DEBS), pages 50-61, 2010.

[11] G. Cugola, A. Margara, M. Matteucci, and G. Tamburrelli. “Introducing uncertainty in complex event processing: model, implementation, and validation,” Computing, 2014, pages 1-42.

[12] R. Hariri, E. Fredericks, and K. Bowers,” Uncertainty in big data analytics: survey, opportunities, and challenges,” Journal of Big Data, Springer Open, 2019.

[13] A. Pfeffer, “Practical probabilistic Programming" Manning publication, April 10, 2016.

[14] D. C. Luckham, ”The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems,” (eds) Rule Representation, Interchange and Reasoning on the Web. RuleML 2008. Lecture Notes in Computer Science, vol 5321. Springer, Berlin, Heidelberg, 2008.

[15] M.Rouse, “Internet of Things (IoT)”, Accessed: June 1st, 2023, Available: http://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

[16] S. Wasserkrug, A. Gal, O. Etzion, and Y. Turchin. “Complex event processing over uncertain data,” In Proceedings of the second international conference on Distributed event-based systems, pages 253-264. ACM,2008.

[17] S. Wasserkrug, A. Gal, and O. Etzion, "A taxonomy and representation of sources of uncertainty in active systems," Next Generation Information Technologies and Systems, number 4032 in Lecture Notes in Computer Science, pages 174-185. Springer Berlin Heidelberg, Jan. 2006.

[18] IBM, "What is a digital twin?", Accessed: June 1st, 2023, Available: https://www.ibm.com/topics/what-is-a-digital-twin.

[19] F. Tao, M. Shang, and A.Y.C. Nee, "Digital Twin Driven Smart Manufacturing," Science Direct, Journals and Books 2019, Background and Concept of Digital Twin - ScienceDirect, 2019.

[20] M. Grieves, "Digital twin: manufacturing excellence through virtual factory replication," Whitepaper, 2014, Accessed: June 1st, 2023, Available: https://www.3ds.com/fileadmin/PRODUCTS-SERVICES/DELMIA/PDF/Whitepaper/DELMIA-APRISO-Digital-Twin-Whitepaper.pdf.

[21] F. Tao, M. Zhang, Y. Liu, A.Y.C. Nee, "Digital twin driven prognostics and health management for complex equipment," CIRP Annanals, volume 67, issue 1, page 169-172, 2018.

[22] J. Chen, "A Cloud Resource Allocation Method Supporting Sudden and Urgent Demands", 2108 6th International Conference on Advance Cloud and Big Data.

[23] G. Rjoub, J. Bentahar, and O. A. Wahab, "BigTrustScheduling: Trust-aware big data task scheduling approach in cloud computing environments", Future Generation Computer Systems, 110, p1079-1097, 2020.

[24] G. Rjoub, J. Bentahar, A. O. Wahab, and S. A. Bataineh, "Deep and reinforcement learning for automated task scheduling in large-scale cloud computing systems. Concurrency and Computation: Practice and Experience," 33(23), e5919-e5919. https://doi.org/10.1002/cpe.5919, 2021.

[25] G. Rjoub, J. Benetahar, O. A. Wahab, and S. A. Bataineh, "Deep Smart Scheduling: A Deep Learning Approach for Automated Big Data Scheduling over the Cloud", 2019, 7th International Conference on Future Internet of Things and Cloud (FiCloud).

[26] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," IEEE Trans. Neural Networks, vol. 9, no. 5, pp. 1054–1054, 1998.

[27] C. Sammut, "Behavioral Cloning," Encyclopedia of Machine Learning. Springer, Boston, MA, 2021. Available: https://doi.org/10.1007/978-0-387-30164-8_69.

[28] Continuous programming, Accessed: June 1st, 2023, Available: https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3.html.

[29] G. Hinton, Lecture 6.1, “Overview of mini-batch gradient descent,” Neural Networks for Machine Learning, Accessed: June 1th, 2023, Available: https://www.youtube.com/watch?v=EANf.

[30] D.Poole, and A.Mackworth, ”Variable Elimination for Beliefs Network, Artificial Intelligence foundations of computation agents,” Accessed: June 1st, 2023, Available: http://artint.info/html/ArtInt_151.html".

[31] Iswarrapa, and J. Anuradha, “A brief introduction on Big Data 5Vs Characteristics and Hadoop Technology,” International Conference on Computing, Communication and Coverages, ICCC-2015, Elsevir, Procedia Computer Science, volume 48, page 319–324, 2015.

[32] J. Changqing, L. Yu, Q. Wenming, A. Uchechukwu, and L. Keqiu, ”Big data processing in cloud computing environments. In Pervasive Systems, Algorithms and Networks (ISPAN),” 2012 12th International Symposium on, page 17–23, 2012.

[33] D. Haywood, "In defence of Monolith", "Microservices vs. monoliths the reality beyond the hype”, InfoQ eMagazine, Issue 51, May 2017.

[34] S. Subhajit, G. Wojciech, and M. Supratik, “OptEx: A Deadline-Aware Cost Optimization Model for Spark,” 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing.

[35] A. Akbar, G. Kousiourus, H. Pervaiz, J. Santosh, P. Tasgma, F. Carrezi, and K. Moessner, "Real-Time Probabilistic Data Fusion for Large-Scale IoT Applications," in IEEE Access, vol. 6, pp. 10015-10027, 2018, doi: 10.1109/ACCESS.2018.2804623.

[36] S. Song, L. Deng, J. Gong, and H. Luo, "Gaia Scheduler: A Kubernetes-Based Scheduler Framework," 2018 IEEE Intl Conf on Parallel and Distributed Processing with Applications, Ubiquitous Computing and Communications, Big Data and Cloud Computing, Social Computing and Networking, Sustainable Computing and Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom), 2018, pp. 252-259.

[37] J. Huang, C. Xiao, W. Wu, "RLSK: A Job Scheduler for Federated Kubernetes Clusters based on Reinforcement Learning," 2020 IEEE International Conference on Cloud Engineering (IC2E), 2020, pp. 116-123.

[38] D.Poole, A.Mackworth,"Variable Elimination for Beliefs Network, Artificial Intelligence foundations of computation agents," $http//artint.infohtmlArtInt_151.html$,l ast accessed on June 1st, 2023

[39] Dhillon, A.S, Majumdar, S., St-Hilaire, M., El-Haraki, A., "MCEP: a Mobile Device Based Complex Event Processing System for Remote Healthcare," Proc. The 11th IEEE International Conference on Internet of Things (IThings), Halifax, Canada, August 2018.

[40] J. Wu, and T. Yang, "Dynamic CPU Allocation for Docker Containerized Mixed-Criticality Real-Time Systems" IEEE 2018

[41] M. T. Islam, S. Karunasekera and R. Buyya, "dSpark: Deadline-Based Resource Allocation for Big Data Applications in Apache Spark," 2017 IEEE 13th International Conference on e-Science (e-Science), Auckland, New Zealand, 2017, pp. 89-98, doi: 10.1109/eScience.2017.21.

[42] Rajkumar Buyya, James Broberg, Andrzej Goscinski, "Cloud Computing: Principles and Paradigms," Wiley book 2011.

[43] Bhumi Chauhan, Chintan Bhatt "Bigdata Analytics in Industrial IoT", Springer International Publishing AG 2018.

[44] Microsoft Azure Fundamental -Explore Azure App services https://docs.microsoft.com/en-us/learn/modules/intro-to-azure-compute/5-appservice last accessed on June 1st, 2023.

[45] Rajkumar Buyya, James Broberg, Andrzej Goscinski, "Cloud Computing: Principles and Paradigms," Wiley book 2011.

[46] Continuous programming https://databricks.com/blog/2018/03/20/low-latency-continuous-processing-mode-in-structured-streaming-in-apache-spark-2-3-0.html last accessed on June 1st, 2023

[47] S. Sidhanta, W. Golab, S. Mukhopadhyay "OptEx: A Deadline-Aware Cost Optimization Model for Spark," 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

[48] C. Richardson, "Developing Transnational Microservices Using Aggregates, Event Sourcing and CQRS", "Microservices vs. monoliths the reality beyond the hype", InfoQ eMagazine Issue 51, May 2017.

[49] S. Melnik, "Dremel: Interactive analysis of Webscale datasets," Proceedings of the VLDB Endowment 3 (Sept. 2010), 330–339.

[50] G. Malewicz, "Pregel: A system for large-scale graph processing," In Proceedings of the ACM SIGMOD/PODS Conference (Indianapolis, IN, June 6–11). ACM Press, New York, 2010. Apache Storm.

[51] M. Zaharia, R.S. Xin, P. Wendell, T.Das, M. Armbrust, A. Dave. X.Meng, J. Rosen, S. Venkataraman, M.J. Frankilin, A.Ghodsi, J. Gonzalez, S.Shenker, and I. Stoica, "Apache spark: Aunifide engine for big data processing", Communications of the ACM, 2016.

[52] J. Xu, Y. Tian, P. Ma, D. Rus, S. Sueda, and W. Matusik, "Prediction-Guided Multi-Objective Reinforcement Learning for Continuous Robot Control," Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020.

[53] M. E. Frincu, and C. Craciun, "Multi-objective Meta-heuristics for Scheduling Applications with High Availability Requirements and Cost Constraints in Multi-Cloud Environments," 2011 Fourth IEEE International Conference on Utility and Cloud Computing, 2011, pp. 267-274, doi: 10.1109/UCC.2011.43.

[54] F. Legillon, N. Melab, D. Renard, and El-G. Talbi, "Cost minimization of service deployment in a multi-cloud environment," CEC 2013 - IEEE Conference on Evolutionary computation, Jun 2013, Cancun, Mexico. pp.2580-2587.

[55] C. Guerrero, I. Lera, and C. Juiz,"Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture," J Grid Computing 16, 113–135 (2018).https://doi.org/10.1007/s10723-017-9419-x.

[56] C. Guerrero, I. Lera, and C. Juiz, "Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications," J Supercomput 74, 2956–2983 (2018).https:https://doi.org/10.1007/s11227-018-2345-2.

[57] G. Fan, L. Chen, H. Yu,and W. Qi, "Multi-Objective Optimization of Container-Based Microservice Scheduling in Edge Computing," Computer Science and Information Systems, Vol. 18, No. 1, 23–42. 2021.

[58] D. Chen, Y. Wang, W. Gao, "A Two-stages Multi-Objective Deep Reinforcement Learning Framework," 24th European Conference on Artificial Intelligence ECAI 2020, Santiago de Compostela, Spain.

[59] J.Xu, Y. Tan, D. Rus, and W. Matusik, "Prediction-Guided Multi-objective Reinforcement Learning for Continuous Robot Control," Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020.

[60] R. Yang, X. Sun, and K. Narasimhan, "A Generalized Algorithm for Multi-Objective Reinforcement Learning and Policy Adaptation," 33rd Conference on Neural Information Processing Systems (NeurIPS 2019), Vancouver, Canada.

[61] G.Zhou, R. Wen, W. Tian, and R. Buyya, "Deep reinforcement learning-based algorithms selectors for the resource scheduling in hierarchical Cloud computing," Journal of Network and Computer Applications, Volume 208, 2022.

[62] J. Chen, "A Cloud Resource Allocation Method Supporting Sudden and Urgent Demands", 2018 Sixth International Conference on Advance Cloud and Big Data. *

[63] R. Mahapatra, B.H. Ahn, S. Wang, H. Xu, and H. Esmaeilzadeh, " Exploring Efficient ML-based Scheduler for Microservices in Heterogeneous Clusters," 2022.

[64] Ahmad, M. G. AlFailakawi, A. AlMutawa, and L. Alsalman, "Container scheduling techniques: A Survey and assessment," Journal of King Saudi University - Computer and Information Sciences. doi:10.1016/j.jksuci.2021.03.002 , 2021.

[65] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource Management with Deep Reinforcement Learning," ACM Workshop on Hot Topics in Networks (pp.50-56), 2016.

[66] Ye Y., Ren X., Wang J., Xu L., Guo W., Huang W. and Tian W., "A New Approach for Resource Scheduling with Deep Reinforcement Learning," Cornell University archive submission [1806.08122].

[67] R. Zhou, Z. Li, and C. Wu, "Scheduling Frameworks for Cloud Container Services," in IEEE/ACM Transactions on Networking, vol. 26, no. 1, pp. 436-450, Feb. 2018.

[68] J. Tordsson, R. S. Montero, R. Moreno Vozmediano, and I. M. Lorente, "Cloud brokering mechanisms for optimized placement of virtual machines across multiple providers," Future Generation Computer Systems, volume 28, issue 2, 2012, pages 358-367,ISSN 0167-739X.

[69] G. Rjoub, J. Bentahar, and O. A. Wahab, "BigTrustScheduling: Trust-aware big data task scheduling approach in cloud computing environments", Future Generation Computer Systems, 110, p1079-1097, 2020

[70] G. Rjoub, J. Benetahar, O. A. Wahab, and A. S. Bataineh, "Deep and reinforcement learning for automated task scheduling in large-scale cloud computing systems", Concurrency and Computation: Practice and Experience 33 (2020).

[71] G. Rjoub, J. Benetahar, O. A. Wahab, and A. Bataineh, "Deep Smart Scheduling: A Deep Learning Approach for Automated Big Data Scheduling over the Cloud", 2019, 7th International Conference on Future Internet of Things and Cloud (FiCloud).

[72] L. Xu, Z. Zeng, and X. Ye, "Multi-objective Optimization Based Virtual Resource Allocation Strategy for Cloud Computing", 2012 IEEE/ACIS 11th International Conference on Computer and Information Science.

[73] Kubernetes, "Resource Management for Pods and Containers," Accessed: June 1st,2023, Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

[74] Azure pricing calculator, Accessed: June 1st, 2023, Available: https://azure.microsoft.com/en-ca/pricing/calculator/.

[75] Azure VM selector: Accessed: June 1st, 2023, Available: https://azure.microsoft.com/en-us/pricing/vm-selector/.

[76] GeeksForGeeks, "CPU scheduling in Operating System", Accessed: June 1st, 2023, Available: $https://www.geeksforgeeks.org/cpu-scheduling-in-operating-systems/.

[77] PyPl organization, Theano project, Accessed: June 1st, 2023, Available: https://pypi.org/project/Theano/.

[78] PyPl organization, Lasagne project, Accessed: June 1st, 2023, Avaialable: https://pypi.org/project/Lasagne/.

[79] Lasagne layers , Accessed: June 1st, 2023, Available: https://lasagne.readthedocs.io/en/latest/modules/layers.html.

[80] Tensor Flow Blog: Industrial AI: BHGE's Physics-based, Probabilistic Deep Learning Using TensorFlow Probability — Part 1 — The TensorFlow Blog

[81] Pivotal Software Inc., "Spring boot framework", Accessed: June 1st, 2023, Available: https://projects.spring.io/spring-boot/.

[82] Docker Inc, “Docker container”, Accessed: June 1st, 203, Availalble: https://www.docker.com/what-docker.

[83] JBoss Community, “Drools Fusion”, Accessed: June 1st, 2023, Available: http://drools.jboss.org/drools-fusion.html.

[84] Docker, Accessed: June 1st, 2023, Available: https://docs.docker.com/engine/release-notes/17.09/.

[85] Charles River Analytics, “Figaro probability programming library:”, Accessed: June 1st, 2023, Available: https://www.cra.com/work/case-studies/figaro.

[86] A. Geron, "Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems," (1st. ed.), O’Reilly Media, Inc.,2017.

[87] Windows containers, Microsoft, Accessed: June 1st, 2023, Available: https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/

[88] Apache, Apache Impala, Accessed: June 1st, 2023 Available: https://impala.apache.org.