UNIVERSIDADE D
COIMBRA

Bruno Ricardo Leitão Faria

# Self-organising engine for the Cloud-to-Edge continuum

**Dissertation in the context of the Master in Informatics Engineering, specialisation in Intelligent Systems, advised by Prof. Dr. Karima Velasquez and Prof. Dr. David Abreu and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.**

September 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Bruno Ricardo Leitão Faria

# SELF-ORGANISING ENGINE FOR THE CLOUD-TO-EDGE CONTINUUM

**Dissertation in the context of the Master in Informatics Engineering, specialisation in Intelligent Systems, advised by Prof. Dr. Karima Velasquez and Prof. Dr. David Abreu and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.**

September 2023

BibTeX:

```
@mastersthesis{faria_msc_thesis,
    author       = {Bruno Ricardo Leitão Faria},
    title        = {Self-organising engine for the Cloud-to-Edge continuum},
    school       = {University of Coimbra},
    year         = {2023},
    month        = sep,
    keywords     = {Service Function Chains, Self-Healing, Machine Learning},
}
```

This work is funded by the project OREOS (POCI-01-0247-FEDER-049029), co-financed by the European Regional Development Fund (FEDER), through Portugal 2020 (PT2020), and by the Competitiveness and Internationalization Operational Programme (COMPETE 2020).

# Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus orientadores, Professora Karima Daniela Velasquez Castro e Professor David Alejandro Perez Abreu pela oportunidade de realizar este trabalho, pela sua orientação, disponibilidade e apoio ao longo de todo este processo.

Um especial obrigado também à Professora Marília Pascoal Curado por todo o apio e confiança depositada em mim ao longo deste período, despertando em mim o interesse pela área da investigação.

Aos meus amigos, dos mais antigo aos mais recentes, da primária à universidade, que me acompanharam ao longo da minha vida, que me apoiaram e que me ajudaram a criar memórias que levarei para sempre comigo. O meu obrigado por tudo o que fizeram e fazem por mim.

À Quantunna, bem como a todos os amigos que fiz lá. O meu obrigado por me terem acolhido e por me terem proporcionado momentos de diversão e folia que me ajudaram desconectar do projetos e das aulas.

À Ana, quem mais me acompanhou ao longo de todo o desenvolvimento deste trabalho, pelo teu apoio, carinho, compreensão e reconforto, bem como toda a felicidade e novidades experienciadas, o meu obrigado do fundo do coração por tudo o que fizeste e fazes por mim.

Por fim, e o mais importante, queria agradecer a toda a minha família, pais, padrinhos, tios, primos e avós por sempre acreditarem e apoiarem durante todo o meu precurso académico. Em especial à minha irmã pelos momentos de felicidade, descontração e de descompressão que me proporcionou ao longo destes anos.

Por tudo isto e muito mais, o meu profundo Obrigado!

Bruno Ricardo Leitão Faria

# Abstract

In the Cloud-to-Edge continuum, heterogeneous devices are distributed in a large area, which makes it challenging to manage. Furthermore, those devices are prone to performance degradation or even failures, which can cause the services to be unavailable or unreliable. Due to the distributed nature of the devices, it is not attainable to manually detect and recover the failures of the devices. Therefore, zero-touch techniques are required to manage the devices' failures to improve the services' availability and reliability by speeding up the recovery process [1].

With this in mind, this work presents a self-organising engine that can be used to manage the failures of the devices, focusing on Central Processing Unit (CPU) failures, in a Cloud-to-Edge environment, aiming to improve the service's availability and reliability. The proposed engine comprises three main components: fault detection, prediction, and mitigation, by migrating the heaviest work to a replica. Besides the engine, a fault injector was also implemented, simulating various levels of stress to the CPU and Random Access Memory (RAM). The engine was tested in a simulated environment, using the COupled Simulation and Container Orchestration framework (COSCO) simulator [2].

The results show that the fault injection component is able to simulate stress on the devices, which can lead to failures. Additionally, the fault detection component can detect the failures of the devices after they occur. Moreover, the fault mitigation component can alleviate the failures of the devices using replicas and thus allow the service to continue to operate. Finally, the fault prediction component can predict CPU failures with an f1 score of around 87% and 73% for binary and multi-class classification problems, respectively.

# Keywords

Service Function Chains, Cloud-to-Edge Continuum, Self-Healing, Machine Learning, Time-Series Classification.

# Resumo

Num continuum de Nuvem-a-Ponta, os dispositivos estão distribuídos por uma vasta área, o que torna desafiante a sua gestão. Além disso, esses dispositivos estão sujeitos a degradação de desempenho ou até mesmo a falhas, o que pode levar a que os serviços fiquem indisponíveis ou não fiáveis. Devido à natureza distribuída dos dispositivos, é impossível detetar e recuperar manualmente as falhas dos dispositivos. Portanto, são necessárias técnicas de toque-zero para gerir as falhas dos dispositivos e, assim, melhorar a disponibilidade e fiabilidade dos serviços, acelerando o processo de recuperação [1].

Tendo isto em conta, este trabalho apresenta um mecanismo auto-organizável que pode ser utilizado para gerir as falhas dos dispositivos, com foco em falhas na CPU, num ambiente Nuvem-a-Ponta, visando melhorar a disponibilidade e fiabilidade dos serviços. O mecanismo proposto é composto por três componentes principais: deteção, previsão e mitigação de falhas, através da migração das tarefas mais pesadas para uma réplica. Além do mecanismo, também foi implementado um injetor de falhas que simula vários níveis de stress na CPU e RAM. O mecanismo foi testado num ambiente simulado, utilizando o simulador COSCO [2].

Os resultados preliminares mostram que a componente de deteção de falhas consegue detetar as falhas dos dispositivos após a sua ocorrência. Além disso, a componente de mitigação de falhas consegue aliviar as falhas dos dispositivos utilizando réplicas e, assim, permitir que o serviço continue a funcionar. Adicionalmente, a componente de injeção de falhas é capaz de simular stress nos dispositivos, podendo eventualmente levar a falhas. Por fim, a componente de previsão de falhas consegue prever falhas na CPU com um f1 score de cerca de 87% para problemas de classificação binária e 73% para problemas de classificação multi-classe.

# Palavras-Chave

Cadeias de Função de Serviço, Continuum de Nuvem-a-Ponta, Auto-Recuperação, Aprendizagem Computacional, Classificação de Séries Temporais.

# Contents

# Acronyms

**AI** Artificial Intelligence.

**ANOVA** Analysis of Variance.

**AVA-SFC** AVailability Aware Service Function Chaining.

**BN** Bayesian Network.

**CNN** Convolutional Neural Network.

**COSCO** COupled Simulation and Container Orchestration framework.

**CPU** Central Processing Unit.

**DNN** Deep Neural Network.

**DT** Decision Tree.

**DTC** Decision Tree Classifier.

**EM** Expectation Maximization.

**ETSI** European Telecommunications Standards Institute.

**FCM** Fuzzy C Means.

**FMS** Fault Management System.

**FN** False Negatives.

**FP** False Positives.

**GPU** Graphics Processing Unit.

**ICT** Information and Communication Technology.

**IPC** Instructions Per Cycle.

**IPS** Instructions Per Second.

**k-NN** k-Nearest Neighbours.

**LAN** Local Area Network.

**LASSO** Least Absolute Selection and Shrinkage Operator.

**LOF** Local Outlier Factor.

**LOP** Local Outlier Probability.

**LSTM** Long Short-Term Memory.

**MIPS** Million Instructions Per Second.

**ML** Machine Learning.

**MSE** Mean Squared Error.

**MTTF** Mean Time To Failure.

**NBC** Naive Bayes Classifier.

**NFV** Network Function Virtualisation.

**NN** Neural Network.

**PCA** Principal Component Analysis.

**RAM** Random Access Memory.

**REPTree** Reduced Error Pruning Tree.

**RFC** Random Forest Classifier.

**RFR** Random Forest Regressor.

**RNN** Recurrent Neural Network.

**RTTF** Remaining Time To Failure.

**SDN** Software Defined Networking.

**SF** Service Function.

**SFC** Service Function Chain.

**SFP** Service Function Path.

**SHLLE** Supervised Hessian Locally Linear Embedding.

**SLA** Service Level Agreement.

**SMAE** Soft Mean Absolute Error.

**SMOTE** Synthetic Minority Oversampling Technique.

**SOM** Self-Organising Map.

**SVM** Support Vector Machine.

**TN** True Negatives.

**TP** True Positives.

**TSF** Time Series Forecasting.

**t-SNE** t-Distributed Stochastic Neighbor Embedding.

**vCPU** Virtual Central Processing Unit.

**VM** Virtual Machine.

**VNF** Virtual Network Function.

**YAFS** Yet Another Fault Simulator.

# List of Figures

# List of Tables

# List of Listings

# Chapter 1

# Introduction

Smart Cities are a paradigm focused on creating efficient urban environments by making use of information and communication technology, such as sensors and actuators, to sense, analyse and react to the city's needs [3]. One widely adopted approach is the Cloud-to-Edge continuum, where devices, besides having a wide variation in resources, are distributed across a large area. Figure 1.1 illustrates the Cloud-to-Edge continuum, where the Cloud is located at the top, having more resources, such as computational power and storage, at a cost of higher latency, and the Edge is located at the bottom, having less computational power and storage but with lower latency.

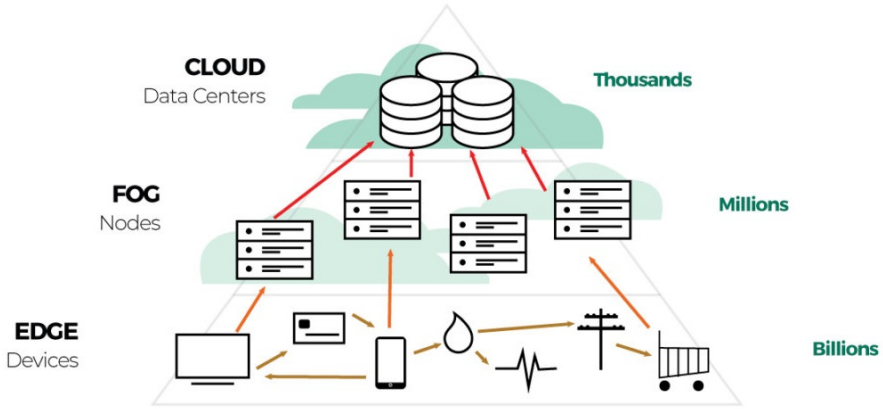



Figure 1.1: Cloud-to-Edge continuum [4].

In that environment, there are a plethora of devices that need to be monitored and managed in order to provide a good service to the citizens. Some of them are critical, such as the traffic lights, and some of them are not critical, such as the sensors. In critical services, the failure of a device can cause a big impact on the city, such as the malfunction of a traffic light, which can cause a traffic jam. Therefore, availability and reliability are very important factors in this scenario.

Consequently, it is important to have a fault management system that can detect, predict and mitigate the failures of the devices to provide a good service to the citizens and to avoid the impact of the failures in the city.

With this in mind, this work presents a self-organising engine that can be used to manage the failures of the devices in a Cloud-to-Edge environment, aiming to improve the availability and reliability of the services.

The remainder of this chapter presents the motivation for this work, followed by the identification of the main and specific objectives to be attained. Then the contributions of the work during the development of this thesis are presented. Finally, the last section presents the structure of the thesis, indicating what will be addressed in the next chapters.

## 1.1 Motivation

Technology's rapid advancement and connection with the physical environment hold an enormous promise for transforming today's large cities into smart cities. In these intelligent environments, Cloud-to-Edge computing is a widely adopted approach, where there is a massive number of different connected devices that are prone to performance degradation or even failure. In order to cope with the diversity of these services and, avoid the dependency on dedicated hardware, the use of virtualisation techniques, such as Network Function Virtualisation (NFV) and Service Function Chain (SFC), has been widely adopted in the last years [5]. However, when a node fails, the whole service is affected or even interrupted. In a real scenario, this can have a big impact on the city, such as the malfunction of a traffic light, which can cause a traffic jam, or even problems in the function of hospitals, banks and other critical services.

Research has been carried out in the area of fault management to combat this hardware degradation and failure problem. Given the complexity of this environment, due to the high number of devices and its heterogeneity, it is necessary to automate the fault management process. That is, zero-touch self-organising techniques must prevail over manual intervention. Here, self-healing attempts to lessen or recover from flaws by automatically activating measures to correct them [1].

When implemented to automate the fault management process, Artificial Intelligence (AI) approaches are proven to be a potent tool for making intelligent decisions by learning patterns from data and using them to try to predict future behaviour. To be able to do so, it is pivotal to monitor all the assets in the environment, physical and virtual devices, services and applications, to try to detect anomalies before failures even occur and thus plan and speed up the recovery of those services.

In this context, this is primary the motivation for the development of this work: to develop a self-organising engine for the Cloud-to-Edge continuum, which can applied to smart cities, that can automate the fault management process. This framework will use a zero-touch approach to detect and, via Machine Learning (ML) techniques, predict the occurrence of hardware faults, as well as mitigate them by introducing replicas. It will be composed of a central controller, which has a global view of the system, monitoring all the devices, gathering data, and making decisions based on the information collected.

## 1.2 Objectives

The main goal of this work is to develop a self-organising engine for the Cloud-to-Edge continuum that can automate the fault management process. To attain this zero-touch self-healing capability in SFC environments, the following objectives were defined:

- Design and implement a failure detection system for the Cloud-to-Edge continuum
- Measure the performance of the use of replicas to mitigate failures in the Information and Communication Technology (ICT) infrastructure.
- Design and implement a fault prediction system.
- Validate and evaluate the proposed framework in a controlled testbed.

These objectives were designed in chronological order, thus providing a guide for the work to follow.

## 1.3 Contributions

The main contributions of this thesis are the following:

- A survey of the state-of-the-art in the area of fault management.
- Extensions to the COupled Simulation and Container Orchestration framework (COSCO) simulator to support:
  - SFC.
  - Fault injection.
  - Failure detection and mitigation.
  - Capability to generate datasets for fault prediction.

  All the developed material is available on a public repository [1].
- Three different datasets that can be used for fault prediction in Cloud-to-Edge computing environments.
- Mechanisms for fault prediction based on various ML models and approaches.

Partial results of this work were presented at:

- Bruno Faria, David Abreu, Karima Velasquez and Marília Curado. *"Self-organising Engine for the Cloud-to-edge Continuum"*. RTCM 2023 - 34th Seminar on Mobile Communications Thematic Network.

  ***Presented on 7th of July 2023***.

There is also one article in preparation to be submitted at:

- Bruno Faria, David Abreu, Karima Velasquez and Marília Curado. *"Self-organising Engine for the Cloud-to-edge Continuum"*. DML-ICC 2023 - 3rd Workshop on Distributed Machine Learning for the Intelligent Computing Continuum in conjunction with IEEE/ACM UCC 2023.

  ***To be submitted on 21st of September 2023***.

[1] `https://github.com/brunofaria1322/COSCO`

## 1.4 Document Structure

The remainder of this thesis is organised as follows:

Chapter 2 presents the necessary background knowledge, presenting the main concepts of SFC, fault management, self-healing and ML.

Chapter 3 offers the state-of-the-art revision in the area of fault management, taking into account ML algorithms and techniques for fault prediction, fault detection and self-healing, as well as some related work in the area of SFC and NFV, mainly regarding the use of replicas.

Chapter 4 presents the proposed solution. It starts with a proposed framework for the self-organising engine. Then, some of the failures and metrics to be explored are mentioned, as well as a brief description of the proposed experimentations.

Chapter 5 introduces the simulation methodology. It starts by presenting the simulator used followed by all the configurations and parameters used in the simulations. Finally, it presents all the additions made to the simulator to support the proposed solution.

Chapter 6 presents the scenarios that were implemented in the simulator. For each scenario, was generated a dataset that is also presented in this chapter, as well as the exploratory data analysis performed on the datasets.

Chapter 7 presents the ML models used to predict the occurrence of faults in the generated datasets as well as all the preprocessing steps performed.

Chapter 8 presents an discusses the results obtained when training the ML models over the datasets.

Finally, Chapter 9 concludes this work and presents some future work directions.

# Chapter 2

# Background

The background information needed to comprehend the rest of the thesis is supplied in this chapter. It is divided into three major sections: Section 2.1 describes the SFC concept, Section 2.2 defines fault management, Section 2.3 describes a brief explanation of what ML consists of as well as some of the most common algorithms used in this thesis and finally, Section 2.4 presents some feature reduction, selection and importance techniques.

## 2.1 Service Function Chains

The European Telecommunications Standards Institute (ETSI) proposed the NFV, a network architecture concept to change how network operators construct and maintain networks by adopting virtualisation technology [5]. Virtual Network Functions (VNFs) are software programs that provide network services, including file sharing, directory services, IP configuration, etc., without requiring dedicated hardware. They are packaged as Virtual Machines (VMs) or containers on commodity servers to replace specialised network appliances. An ordered set of VNFs and subsequent "steering" of traffic via them, known as the Service Function Chain, are developed to suit various service requirements. Figure 2.1 shows the ETSI's proposed NFV architecture framework [5].

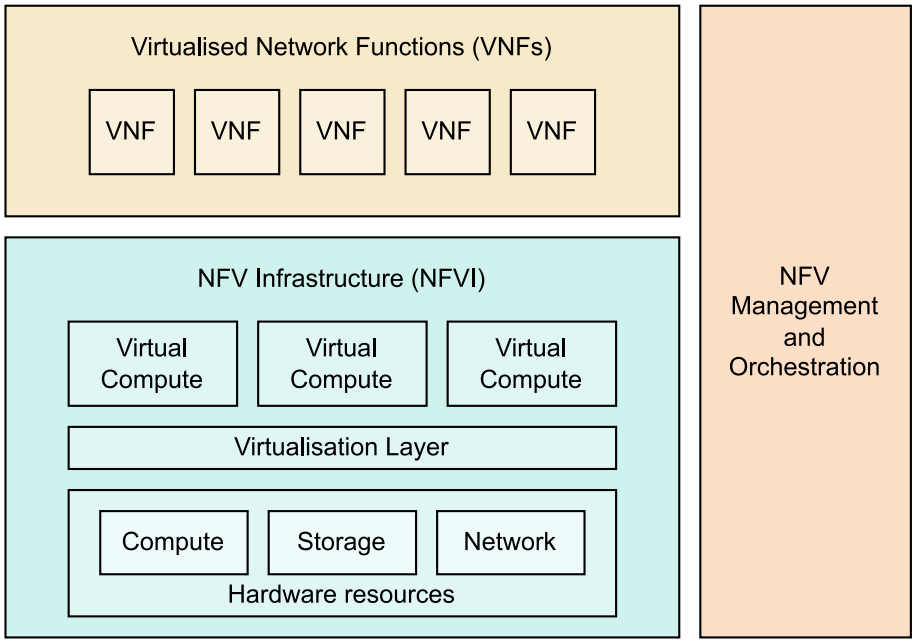



Figure 2.1: High-level NFV architecture framework [5].

According to ETSI [5], the NFV architecture framework is composed of three working domains:

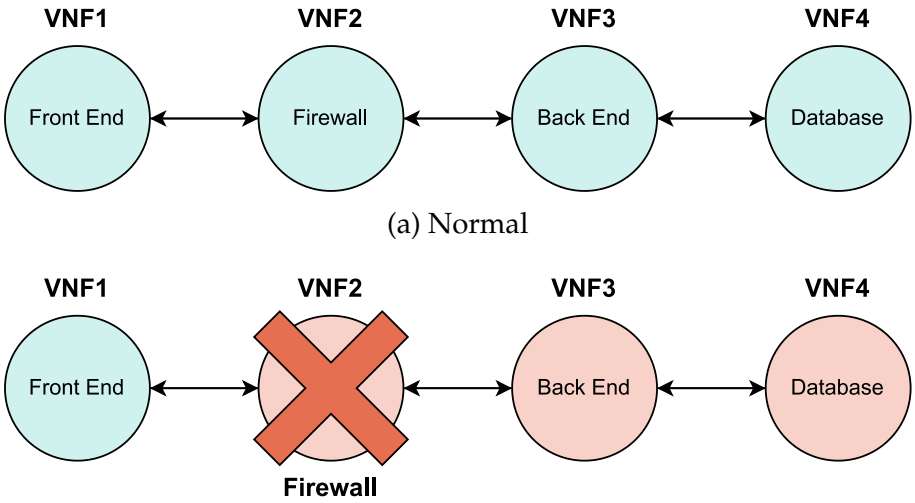
- **Virtualised Network Function**: software implementation of a network function that runs on a VM or container over the NFV Infrastructure.
- **NFV Infrastructure**: a diversity of components that provide the physical resources supporting the execution of VNFs.
- **NFV Management and Orchestration**: a set of components that covers the management and orchestration of physical and virtual resources, including the VNFs and the NFV Infrastructure. It focuses on all the virtualisation-specific aspects of the NFV architecture.

In short, an SFC is a sequence of functions running in different devices, such as VMs or containers, that when executed in a specific order form a chain, providing a service.

Although NFV has changed the telecommunication sector thanks to the various benefits of virtualisation technology, several difficulties must be considered, particularly concerning fault recovery. The current NFV implementation only provides a few self-healing functionalities, like ping for a health check, and the recovery process, which entails tearing down the entire virtual machine and starting over with a new one, is a very expensive process.

## 2.2 Fault Management

Faults, or anomalies, may have various causes, such as hardware failures or degradation, software bugs, human errors, and connectivity loss. These faults, for example, in an SFC can compromise the performance of the full service, as shown in Figure 2.2 where a failure in one VNF can compromise the whole service.




(a) Normal

(b) Faulty

Figure 2.2: Example of a normal and faulty SFC.

In the example in Figure 2.2, we have an SFC with four VNFs simulating a simple application. Here we can see that, in the first case, Figure 2.2a, the whole service is working as expected as a user can make use of the application and access the database without any problem. However, in the second case, Figure 2.2b, the VNF2, which runs the firewall for the application, had a failure and the next VNF in the SFC, the back end, will never receive the data from the failed VNF, resulting in a failure in the whole service, turning the application unusable.

Fault Management is a process that aims to detect, isolate, and correct abnormal conditions in a network [6]. In the context of a Cloud-to-Edge continuum, fault management is a crucial step to ensure the quality, reliability and availability of the services. For the scope of this thesis, fault management will be divided into three main categories: fault prediction, a proactive approach explained in Section 2.2.1, fault detection, a reactive approach described in Section 2.2.2, and self-healing, a strategy that aims to recover from faults automatically without any human intervention, introduced in Section 2.2.3.

### 2.2.1 Fault Prediction

Fault prediction is a fundamental challenge in fault management [6]. It is a proactive approach to fault management that aims to predict upcoming failures or performance degradation. This technique allows for preventative measures to be taken, to minimise the impact of the fault on the network, even before the failure occurs, giving a bigger window of time to take action.

Due to the complexity of modern networks, fault prediction is a challenging task, and ML techniques have been proposed to improve fault prediction. For that, there is a need for a large amount of data to train the models, reinforcing the need for a monitoring system.

### 2.2.2 Fault Detection

Unlike fault prediction, fault detection is a reactive approach to fault management. It is used to identify or classify, faults as soon as possible after a failure occurs [6]. This approach is more common in the industry, as it is easier to implement and requires fewer resources. Fault detection is a crucial step in fault management, as it allows for the localisation and mitigation of faults by triggering actions.

### 2.2.3 Self-Healing

In traditional systems or networks, operators typically only discover service failures after receiving several customer complaints. Besides that, the technician's knowledge is crucial for troubleshooting and failure recovery. On the other hand, the objective of self-healing is to carry out these actions automatically and proactively.

According to Salehie and Tahvildari [7], self-healing is a process that is linked to self-diagnosing,[8], and self-repairing [9]. Self-diagnosing refers to diagnosing errors, faults and failures while self-repairing focuses on recovery from them. With that in mind, self-healing can be defined as the combining capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and take proper actions accordingly to prevent that failure. All of these actions are performed without human involvement so that they remain hidden from the average user.

Both works by Salehie and Tahvildari [7] and Psaier and Dustdar [10] present the self-healing process as a cycle, as shown in Figure 2.3.

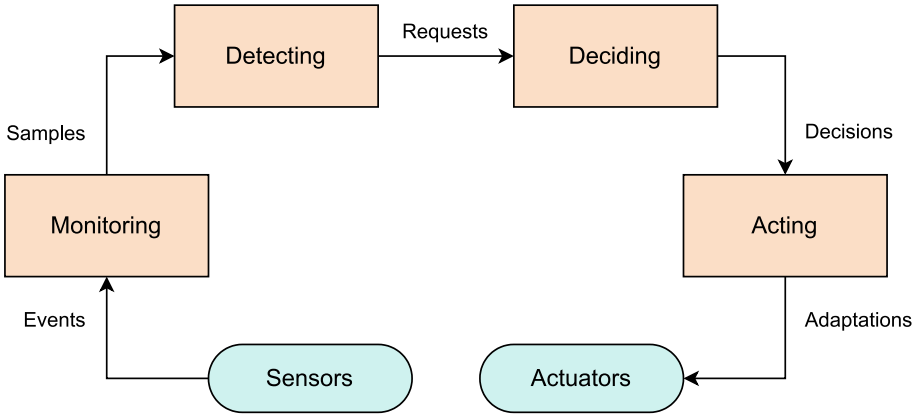



Figure 2.3: Staged loop of self-healing.

The cycle as shown in Figure 2.3, can be summarised as follows:

- **Monitoring Process**: responsible for collecting and correlating data from the sensors of the system.
- **Detecting Process**: responsible for analysing all the samples provided by the monitoring process and detecting, or even predicting, faults or other anomalies.
- **Deciding Process**: responsible for deciding what action should be taken based on the detected faults.
- **Acting Process**: responsible for applying the adaptation determined by the decision process.

Regarding the acting process, in order to recover from faults, Ghosh et al. [11] present the introduction of redundancy techniques for the healing process. Redundancy techniques are used to increase the availability of the system by providing a backup system that can take over in case of failure. This backup system can be a spare component, a backup server, or even a backup network, and is present in the objectives of this thesis.

## 2.3 Machine Learning

Machine Learning is a branch of AI and computer science that uses statistical techniques to give computer systems the ability to learn with data, without being explicitly programmed, by exploiting hidden patterns in this data [12].

ML and AI in general are heavily utilised to achieve this zero-touch approach due to the arising complexity of current networks and systems. The main goal is to train a model that can detect or even predict system failures and take the appropriate action to prevent them, using the data gathered from the system.

Supervised learning is a subcategory of ML algorithm that learns from labelled training data. The training data consists of a set of inputs and the corresponding outputs. The goal of the algorithm is to learn a function that can map the inputs to the outputs. This function can then be used to predict the output of new inputs [13].

Random Forest Classifier (RFC) is a supervised learning algorithm that fits several Decision Tree Classifiers (DTCs) on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. RFC is one of the most used algorithms in ML due to its simplicity and good performance.

Support Vector Machine (SVM) is a supervised learning algorithm that can be used for binary classification problems [14]. It is a binary classifier that separates the data into classes by finding the hyperplane that maximises the margin between the two classes. Although it is a binary classifier, it can be used for multiclass classification problems by using a one-vs-one or one-vs-all approach. This means that the multiclass problem is divided into several binary classification problems, and the results are combined to obtain the final result.

Neural Network (NN) is an algorithm that is inspired by the biological neural networks that constitute animal brains. It consists of layers of neurons, which are connected to each other. Each neuron receives an input, performs a computation, and passes the output to the next neuron. The output of the last layer is the output of the NN. The connections between the neurons have weights that are adjusted during the training process. NNs are used for both classification and regression problems.

Convolutional Neural Networks (CNNs) are a type of NN that is commonly used for image classification. The main difference between a NN and a CNN is that the CNN has one or more convolutional layers, which are used to extract features from the input data. The convolutional layers are composed of filters that, when applied to the input data result in a feature map.

Recurrent Neural Network (RNN) is a type of NN that uses sequential data or time series data. They are distinguished by their memory as they take information from prior inputs to influence the current input and output. This means that the output of the current input depends not only on the current input like the previous models, but also on the previous inputs.

Long Short-Term Memory (LSTM) is a type of NN that is also commonly used for time series prediction. It is a RNN that has a more complex structure using gates to control the information that is allowed to enter, leave, or be forgotten. This allows the LSTM to learn long-term dependencies.

Random Forest Regressor (RFR) works in the same way as RFC. The main difference is that RFR is used for regression problems, while RFC is used for classification problems.

## 2.4 Feature Reduction, Selection and Importance

Feature reduction, selection and importance are techniques used to reduce the number of features in a dataset. These techniques are used to improve the performance of the models, as they reduce the complexity of the dataset, and also to improve the interpretability of the models, as they reduce the number of features to analyse as well as speed up the training process.

Feature Importance is a technique that assigns a score to each feature of the dataset based on how important it is to the model's output. The higher the score, the more important the feature is. Two of the most common methods to calculate feature importance are Analysis of Variance (ANOVA) and Chi-Square. ANOVA is commonly used for continuous numerical features, while Chi-Square is used for categorical features [15].

Feature Selection is a technique that selects a subset of features from the original dataset whilst discarding the less informative ones. This technique is similar to Feature Importance. Extra Tree Classifier is one of the most common methods used for feature selection. It is a tree-based ensemble method that ranks the features based on their importance by training several different decision trees and averaging the results.

Finally, feature reduction is a technique that reduces the number of features in the dataset by combining them into a smaller set of features. This technique is different from the previous ones as it creates new features from the original ones. Principal Component Analysis (PCA) and t-Distributed Stochastic Neighbor Embedding (t-SNE) are two of the most common methods used for feature reduction.

## 2.5 Summary

In this chapter, the basic knowledge needed to comprehend the rest of the thesis was presented. Here some of the main concepts of the thesis were introduced, such as SFC, VNF, fault management, fault prediction, fault detection and self-healing. Finally, a brief explanation of what ML consists of as well as some of the most common algorithms was presented.

# Chapter 3

# State of the Art

In this chapter, the state of the art is presented. Section 3.1 presents a comprehensive survey on fault management for networking, which discusses the application of ML in various networking domains, including fault prediction, fault detection, and self-healing, which are relevant to the work presented in this thesis. Section 3.2 discusses some works on Fault Management and self-healing for SFC as well as the use of replicas. Section 3.3 presents a brief analysis of the state of the art. Finally, Section 3.4 summarises the whole chapter.

## 3.1 Fault Management

Boutaba et al. [6] present a comprehensive survey on ML for networking. They discuss the application of ML in various networking domains, including fault management, and provide a detailed overview of the state of the art in this field. They also discuss the challenges and opportunities of ML in networking. Regarding fault management, they divide the state-of-the-art into four categories: fault prediction, fault detection, fault localisation, and automated mitigation. This survey's fault prediction and detection categories are relevant to the work presented in this thesis. Some of the works cited in this survey are discussed in more detail in Section 3.1.1 and Section 3.1.2, respectively.

Zhong et al. [16] explore the automated management of containerised applications using machine learning techniques in container orchestration systems. They also review current ML-based container orchestration solutions, discussing their development from 2016 to 2021 and grouping them according to shared characteristics. One of those solutions, regarding anomaly detection, is the work by Du, Xie and He [17].

Du, Xie and He [17] propose an anomaly detection system (ADS) that addresses the challenges of monitoring, detecting and diagnosing the root cause of any anomalies found in microservice architectures. The ADS comprises a monitoring module, a detection module, and a fault injection module. They monitor some critical metrics of the system concerning the CPU, memory and network. For the detection module, they experiment with SVMs, RFCs, Naive Bayes Classifiers (NBCs) and k-Nearest Neighbourss (k-NNs). In their experimental set, they present the k-NN as the best-performing algorithm, followed by Bayesian Network (BN). They also mention that SVM is unsuitable for their specific case study due to the high number of features.

Zhang, Zhu and Hossain [18] present challenges and solutions for Data-Driven Machine Learning Techniques in the context of self-healing. They classify those challenges into five categories: (i) data imbalance, (ii) data insufficiency, (iii) cost insensitivity, (iv) non-real-time response, and (v) multisource data fusion. The proposed solutions for each of these challenges are: (i) data preprocessing (oversampling, undersampling, hybrid sampling and Synthetic Minority Oversampling Technique (SMOTE)), or algorithmic (one-class classifier and cost-sensitive learning), (ii) data preprocessing (oversampling and SMOTE), using unlabelled data (active learning, unsupervised learning and semi-supervised learning), or using transfer learning, (iii) cost-sensitive learning and introduction of new evaluation metrics (F1 score, precision, recall, G-mean, ROC curve, AUC, among others), (iv) proactive response, and (v) data fusion. Finally, they present a case study where they test some of their solutions and present the results accompanied by a final discussion.

To summarise, ML has been widely used in the context of fault management. The main challenges are related to data imbalance, data insufficiency, cost insensitivity, non-real-time response, and multisource data fusion. Besides that, the high number of features also poses a challenge to some ML algorithms.

### 3.1.1 Fault Prediction

BNs are common models used in fault prediction, as can be seen in the works of Hood and Ji [19], Kogeda, Agbinya and Omlin [20] and Kogeda and Agbinya [21]. BNs are probabilistic models that combine the expected behaviour of a network with deviations from the normal to predict future faults. However, the authors identify a common drawback of BNs: they are not sensitive to temporal factors and cannot model networks that dynamically evolve. To address this issue, Ding et al. [22] introduces a dynamic BN model that is robust in fault prediction, localisation, and cause and effect analysis.

Wang, Martonosi and Peh [23] uses supervised learning methods, such as Decision Tree (DT), rule learner, SVM, BN, and ensemble methods, to classify the quality of connections in wireless sensor networks. The findings show that rule learners and DTs achieve the highest accuracy.

Due to the enormous volume of logs and metrics on a large-scale distributed network system the work of Lu et al. [24] highlights the importance of feature extraction, selection, and dimensionality reduction as an essential but non-trivial prerequisite. The authors present a technique called Supervised Hessian Locally Linear Embedding (SHLLE) to extract features automatically and generate failure predictions. Empirical experiments show that SHLLE outperforms the other feature extraction methods, such as PCA, and classification methods, such as SVM and k-NN.

Pellegrini, Sanzo and Avresky [25] make use of different ML techniques, such as linear regression, MP5, Reduced Error Pruning Tree (REPTree), SVM, Least Absolute Selection and Shrinkage Operator (LASSO), and Least-Square SVM, to predict Remaining Time To Failure (RTTF) of applications. The authors compare the Soft Mean Absolute Errors (SMAEs) of these techniques and find that, for their testbed, REPTree and M5P outperform the other methods. However, the model has a high prediction error when the system is temporally far from the failure time.

Wang et al. [26] present the use of Time Series Forecasting (TSF) combined with ML techniques as a promising approach to improve the accuracy of equipment failure prediction in an optical network. It uses a kernel function and penalty factor in a SVM to model non-linear relationships and reduce misclassification, respectively, achieving 95% accuracy in fault prediction.

Deep Neural Network (DNN) with autoencoders are also used to predict the inter-arrival time of faults in a network as shown in the work by Kumar, Farooq and Imran [27]. They compare the performance of this technique with other ML methods, such as autoregressive NN, linear and nonlinear SVM, and exponential and linear regression, and find that DNN with autoencoders outperform the others.

To summarise, ML is a good technique to use for fault prediction due to its ability to learn from data and make predictions. However, the high number of features and the lack of data are the main challenges to overcome. Besides that, there are also problems with the simulation and experimentation of the proposed solutions due to the artificiality of the data, since immediately injected faults can not be predicted.

### 3.1.2 Fault Detection

One of the first works to use ML for fault detection is the one by Maxion [28].

Rao [29] uses statistical hypothesis testing methods to detect faults in a network but does not use any ML or any AI in general.

In the work by Baras et al. [30], a reactive strategy is implemented to detect and localise the root cause of faults. The system uses an NN classifier to output a code representing various fault scenarios, and an expert system is only activated when a specific confidence value is achieved.

Clustering algorithms are also used for real-time fault detection and classification. Qader, Adda and Al-Kasassbeh [31] uses techniques like *k*-Means, Fuzzy C Means (FCM) and Expectation Maximization (EM) to classify faults in a network. The evaluation shows that, although *k*-Means and EM are faster, FCM is more accurate.

Moustapha and Selmic [32] meke use of RNNs to detect faulty nodes in wireless sensor networks. They use an RNN to successfully detect faults without early false alarms in a small network with 15 sensors and synthetically introduced faults.

Hajji [33] present an unsupervised mechanism for fast anomalies detection in Local Area Network (LAN) thorough traffic analysis. Experimental evaluation shows that the proposed tool detects faults in real time on a real network with high detection accuracy.

The work by Hashmi, Darbandi and Imran [34] uses different unsupervised algorithms, such as *k*-Means, FCM, Kohonen's Self-Organising Map (SOM), Local Outlier Factor (LOF) and Local Outlier Probability (LOP), to detect anomalies in a network. They analyse a real network failure log dataset that contains records over 12 months. The results show that SOM outperforms *k*-means and FCM in terms of error metric. Furthermore, LOP applied to the SOM is more reliable.

To summarise, there is a lot of research in the field of fault detection using ML and AI, sometimes combined with fault localisation and classification. However, the main challenge is to find a way to use these techniques in real-time, since the time between the fault and the detection is crucial for rapid network or service recovery.

## 3.2 Service Function Chain

Kaur, Mangat and Kumar [35] present a survey in the field of SFC provisioning using Software Defined Networking (SDN) and NFV. The takeaways from this survey are mainly regarding the AVailability Aware Service Function Chaining (AVA-SFC), the challenges and the research gaps in the field. Regarding AVA-SFC the authors announce that, for the 21 articles reviewed in this category, most of the researchers worked on either node or link failure, whereas in a real scenario, the whole Service Function Path (SFP) has to recover from the failure. Besides that, the authors also point out that the recovery of nodes and links is always done after a failure occurs, by assigning the workload of the failed node to another node, with no redeployment of the failed node. The authors also point out that, to improve the availability of network services, re-composition, re-mapping, and re-scheduling of the failed SFC should be automated. This mechanism should not impact the other service chains to maintain service continuity. Those are the main challenges and research gaps presented by the authors that are most relevant to this thesis.

Herker et al. [36] discusses the challenges of implementing NFV in data centres, with a focus on ensuring high availability. The paper presents algorithms for resiliently embedding VNF service chains in a data centre and explores different data centre topologies to determine the best cost-per-throughput relation for a given level of resilience and availability. The authors also present two models for the high availability of SFCs by creating different backup strategies and calculating the number of backups VNFs required to achieve a given availability.

Medhat et al. [37] discusses the concept of SFC and its use in creating a resilient network service deployment model. SFC relies on technologies such as SDN and NFV to create, modify, and delete SFCs in a cost-efficient and rapid way. However, during the runtime phase, Service Functions (SFs) can be subject to failures, which can result in an end-to-end failure at the application level. To address this issue, the paper proposes the use of a resilient SFC Orchestrator, which is capable of deploying SFCs following the ETSI NFV architectural model, as well as controlling the runtime phase and rerouting traffic in case of faults. Their Fault Management System (FMS) provides a 1:N redundancy creating only one standby VNF component for each VNF in the SFC. In case of failure, the FMS reroutes the traffic to the standby VNF component and removes the failed VNF component from the SFC. After that, a Heal event is sent to the Orchestrator, which starts searching for the SFPs involved in order to update them, rerouting the traffic. The concept is demonstrated using the Fraunhofer FOKUS Open Baton toolkit in an OpenStack and OpenDayLight-based environment.

Karra and Sivalingam [38] discusses the use of SDN and NFV to provide resilient services with minimal redundancy. The paper proposes two algorithms for the migration of functions from failing to functioning servers and for improving the robustness of failed parts of a SFC. Failure is detected using a heartbeat mechanism but, for critical SFCs, the Mean Time To Failure (MTTF) is calculated and, when it is below a threshold, the migration of functions is initialised. Whenever a failure occurs, the predecessor of the failed node stores the packets until the SFC is up and running. Then, it invokes the K-shortest path algorithm for finding the node(s) for deploying the functions that were previously on the failed node(s). This acts as an initial solution and makes sure that the whole SFC is up and running in the minimum possible time. Thereafter, a TabuSearch procedure attempts to make the SFC robust concerning the metric MTTF, aiming for a better set of nodes for the failed part of SFC, such that there is an improvement in the minimum MTTF compared to the initial solution. The algorithms are studied using a simulation model, and the results show that reactive handling of failure is feasible while honouring Service Level Agreements (SLAs), and an SFC can be made robust concerning the considered parameter.

To summarise, regarding SFCs some works focus on fault management and self-healing. Besides fault detection and prediction, the works also focus on the migration of VNFs in case of failure. The use of replicas is also a common technique to ensure high availability.

## 3.3 Analysis

Table 3.1 summarises some algorithms and techniques for current solutions in the area of fault detection and prediction. It is composed of 6 columns: the reference, the type of mechanism, reactive if triggered after the fault occurs or reactive if triggered before the fault occurs, the features collected and used to train the models, the output to predict, the ML algorithms explored and finally the evaluation metrics for those algorithms.

It can be seen that ML is a common technique used in fault management, especially for fault prediction and detection. However, the lack of data is one of the main challenges to overcome.

The works here presented, as well as the ones in the previous sections, are relevant to the work presented in this thesis since they inspired the development of the simulation framework and the models used in the fault prediction experiments.

Table 3.1: Summary of the most relevant algorithms in the state-of-the-art.

| Reference | Type | Features | Output | ML Technique | Evaluation |
|---|---|---|---|---|---|
| [17] | Reactive | System metrics regarding:<br>CPU<br>Memory<br>Network | 14 classes:<br>(3 services x<br>4 fault types) +<br>normal + overload | SVM<br>RFC<br>NBC<br>k-NN | Precision, recall and<br>accuracy round 90% |
| [18] | Reactive | retainability; handover success rate;<br>reference signal received power;<br>reference signal received quality;<br>signal-to-interference-plus-noise ratio;<br>throughput ·distance | 2 classes:<br>Faulty or not | SVM<br>SVM + SMOTE<br>SVM + oversampling | AUC = 0.624<br>AUC = 0.945<br>AUC = 0.806 |
| [25] | Proactive | System metrics regarding:<br>Threads<br>Memory<br>Swap space<br>CPU | RTTF | Linear Regression<br>M5P<br>REP-Tree<br>LASSO<br>SVM<br>Least-Square SVM | SMAE: 137.600<br>SMAE: 79.182<br>SMAE: 69.832<br>SMAE: 405.187<br>SMAE: 132.668<br>SMAE: 132.675 |
| [27] | Proactive | Historical data of fault occurrence<br>and their inter-arrival times | Inter-arrival time<br>of faults | DNN<br>with Autoencoders | NRMSE: 0.122092<br>RMSE: 0.504425 |
| [32] | Reactive | Previous outputs of sensor nodes and<br>Current and previous output samples<br>of neighboring sensor nodes | Approximation<br>of the output of<br>the sensor node | RNN | Constant error smaller<br>than the shown in<br>their state-of-the-art |

## 3.4 Summary

This chapter presented a survey of the state of the art in the field of fault management, including fault prediction, fault detection and self-healing, as well as the combination with SFC. The use of replicas for recovery of anomalies for SFC was also explored. Finally, a brief analysis of the state of the art was presented.

# Chapter 4

# Proposed Solution

In this chapter, the proposed solution is presented. In Section 4.1 a framework for the self-organising engine is proposed. Section 4.2 defines the experiments that will be performed. Finally, Section 4.3 presents the summary of this chapter.

## 4.1 Proposed Framework

The proposed framework is composed of three main components: fault detection, fault prediction and fault mitigation with the use of replicas. The proposed framework architecture is presented in Figure 4.1.

Figure 4.1 is composed of different nodes with different characteristics, in order to simulate a Cloud-to-Edge continuum. That means, the cloud nodes will have more resources and more latency, while the edge nodes will have fewer resources and less latency. There are already some frameworks that simulate this Cloud-to-Edge continuum in mind, such as Yet Another Fault Simulator (YAFS) [39] and COSCO [2]. In the proposed framework, there are a plethora of nodes. For each node, there is a monitoring tool that sends the information to the fault detection and prediction system.

After testing with this first framework, and having a working environment, the next step is to add the fault mitigation system, presented in Figure 4.2. The figure shows that the fault mitigation system will be composed of replicas of the nodes. The replicas will be used to mitigate the failures of the whole system. That way, expanding the previous example of fault in Figure 2.2, Figure 4.3 is introduced, aiming to show the full self-healing system.

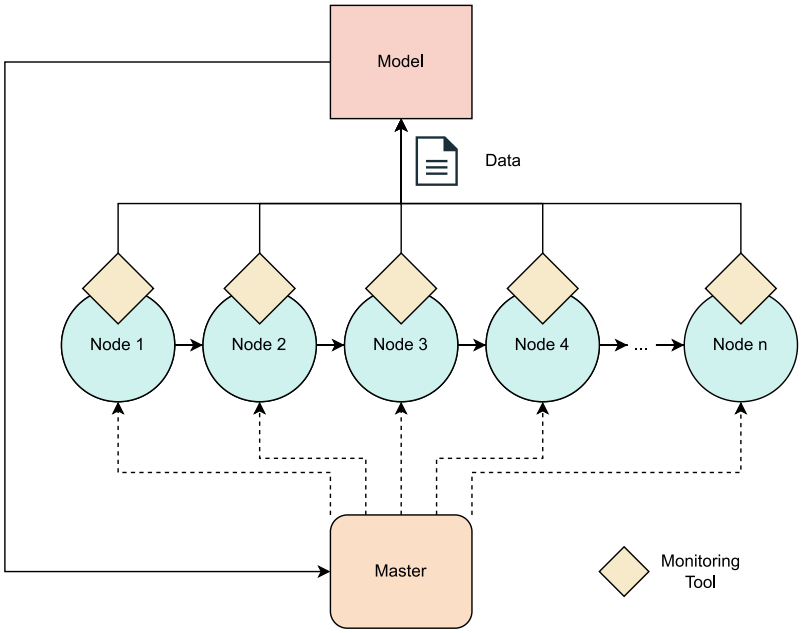



Figure 4.1: High-Level Proposed Architecture without replicas.

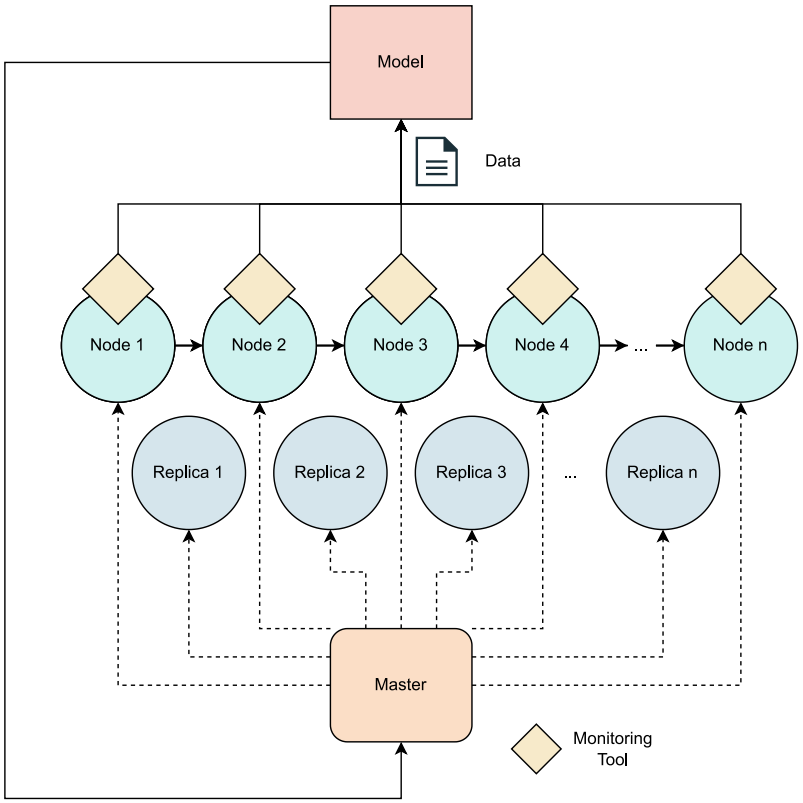



Figure 4.2: High-Level Proposed Architecture with replicas.

In Figure 4.3, a simple SFC is presented, with four nodes, with their normal behaviour being represented in Figure 4.3a. Whenever a failure occurs or is predicted, Figure 4.3b, the replicas will be activated and used to mitigate the failure, as shown in Figure 4.3c, in order to keep the SFC running.

## 4.2 Experiments

This work will be divided into seven different experiments or steps in order to develop the proposed framework.

- **Step 1:** Linear SFC.

  A baseline SFC that does not use any fault management technique will be deployed. It will consist of three nodes, one for each layer. The SFC will flow in an unidirectional way, from the edge to the cloud

- **Step 2:** Failure detection system.

  Here simple Central Processing Unit (CPU) and Random Access Memory (RAM) thresholds will be defined in order to detect the failure of nodes.

- **Step 3:** Failure migration system.

  In this step, the replicas will be deployed and, whenever a failure is detected, the heaviest virtual function will be migrated to the host's replica. Each host will have its unique replica.

- **Step 4:** Fault injection system.

  In this experiment, a fault injection system will be designed and implemented in order to stress the CPU and RAM of the nodes leading to the failure of nodes. This will also be used to test the fault detection and mitigation systems and to generate datasets for the fault prediction system.

- **Step 5:** Fault prediction.

  In this step, the fault prediction system will be developed in order to predict the faults in nodes. For that, some classification models will be explored.

- **Step 6:** The second scenario.

  Here a more complex scenario will be deployed, with more nodes and more SFCs. This scenario will be in a tree topology, with the SFCs flowing from the edge to the cloud, which means, from the leaves to the root.

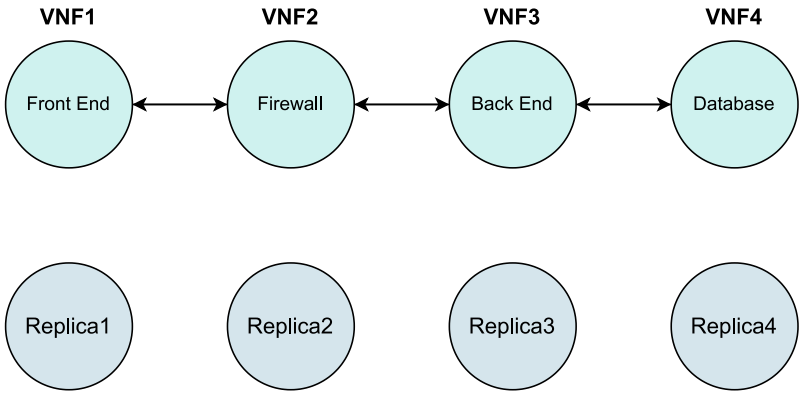
- **Step 7:** The final scenario.

  Finally, the last scenario will be deployed. This scenario will be a more complex scenario, with more nodes and more SFCs. This scenario will also be in a tree topology, but with much more nodes in the edge layer than the fog or cloud. This scenario is the closest to the real-world scenario, but more complex ones could be deployed.

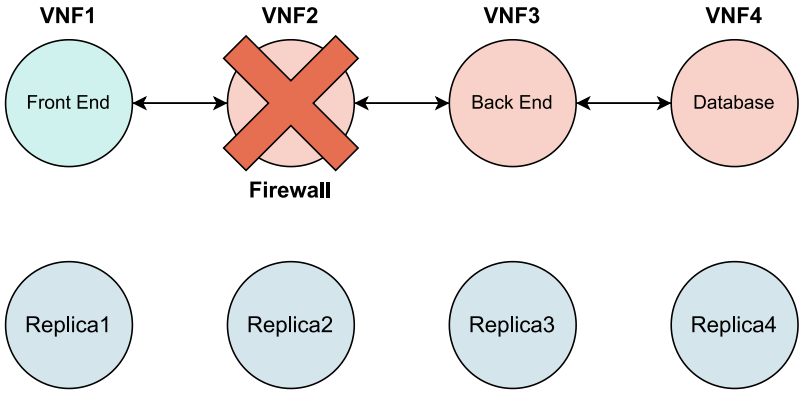
## 4.3 Summary

In this chapter, the proposed solution for the problem statement was presented. The proposed solution is composed of a self-healing framework that will be deployed in a Cloud-to-Edge continuum. This framework will be composed of a fault detection system, a fault prediction system and a fault mitigation system. The fault detection system will be used to detect the failure of nodes. The fault prediction system will be used to predict the failure and anomalies of nodes. Finally, the fault mitigation system will be used to mitigate the failure of nodes. The latter will be composed of replicas of the virtual functions that will be deployed in a different node than the original virtual function. Faults will be detected or predicted and then mitigated by the replicas.

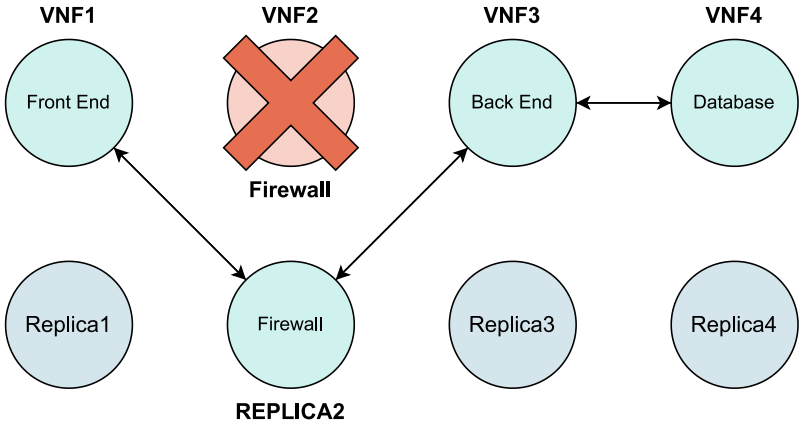
The proposed experimentation was also presented. There are seven different experiments or steps.




(a) Normal




(b) Faulty




(c) Recovered

Figure 4.3: Proposed SFC for the self-healing scenario.

# Chapter 5

# Simulation Methodology

The simulation methodology is described in this chapter. First, the used simulator is presented. Then, the hosts' configuration is presented followed by the workload description. After that, the main modifications made to the simulator are described, namely, the implementation of chaining, a fault injection mechanism and a failure detection and mitigation system. Finally, some other important modifications are shown.

## 5.1 Simulator

In order to generate the dataset needed to train the fault prediction system, either a simulator or a real environment is required. A simulator was preferred because it is easier to control and reproduce the experiments as well as faster and cheaper to implement, run and maintain. The chosen simulator was COSCO [2].

COSCO is an AI-based container orchestrator for edge, fog and cloud computing environments. Its main goal is to optimise resource allocation to reduce the cost of the infrastructure and energy consumption, being used to test, train and validate various task-scheduling algorithms. As Figure 5.1 illustrates, it is divided into two main components: the simulator and the framework, corresponding to simulated and physical environments, respectively.

The simulator emulates the hosts, being able to define the hosts' characteristics, and workloads, which can use Azure and BitBrain datasets built with real-world data. The framework uses physical computational nodes, in the same Virtual Local Area Network (VLAN), as hosts and instantiates the tasks as Docker containers. The framework can communicate with the simulator via HTTP REST APIs. For the purpose of this work, only the simulator was used.

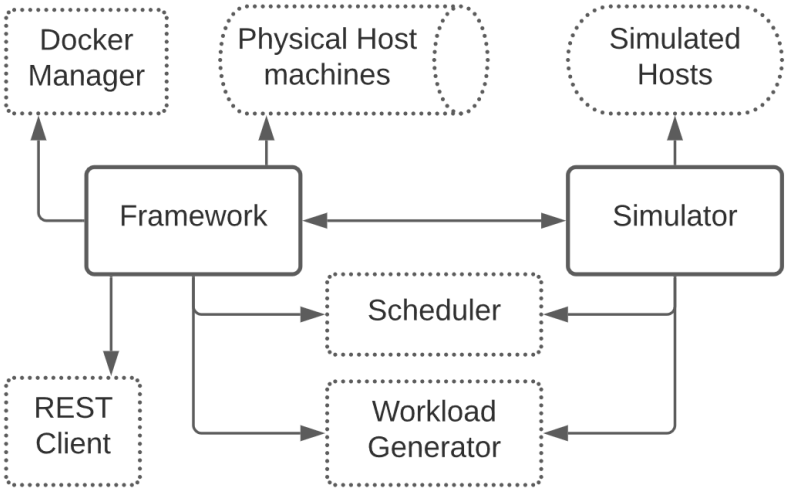



Figure 5.1: COSCO high level architecture [2].

Although COSCO is a great tool and reduced the proposed environment implementation time, it was not enough to simulate it in its entirety, since it was designed for task placement problems. Therefore, some pruning, configurations and extensions were made to the simulator in order to create and simulate the proposed scenarios. These modifications are described in the following sections.

## 5.2 Hosts Configuration

The hosts' CPU, RAM and Disk capacities are based on Azure's B-series burstable VMs [40], presented in Table 5.1.

Table 5.1: Azure's B-series burstable VM sizes.

| Name | vCPU | MIPS | Memory (GB) | Disk (GB) |
|---|---|---|---|---|
| Standard_B1ls | 1 | 2054 | 0.5 | 4 |
| Standard_B1s | 1 | 2054 | 1 | 4 |
| Standard_B1ms | 1 | 2054 | 2 | 4 |
| Standard_B2s | 2 | 4096 | 4 | 8 |
| Standard_B2ms | 2 | 4096 | 8 | 16 |
| Standard_B4ms | 4 | 8192 | 16 | 32 |
| Standard_B8ms | 8 | 16384 | 32 | 64 |
| Standard_B12ms | 12 | 24576 | 48 | 96 |
| Standard_B16ms | 16 | 32768 | 64 | 128 |
| Standard_B20ms | 20 | 40960 | 80 | 160 |

The simulator required a Million Instructions Per Second (MIPS) value to simulate the CPU capacity, which is not provided by Azure's website [40, 41]. According to this documentation, a list of processors is used in the B-series. Still, their MIPS values are not provided since they are no longer relevant for the current generation of processors, and no benchmark evaluation was found, for these processors.

> *"B-series run on the 3rd Generation Intel Xeon Platinum 8370C (Ice Lake), the Intel Xeon Platinum 8272CL (Cascade Lake), the Intel Xeon 8171M 2.1 GHz (Skylake), the Intel Xeon E5-2673 v4 2.3 GHz (Broadwell), or the Intel Xeon E5-2673 v3 2.4 GHz (Haswell) processors."* [40]

With this in mind, there was still a need to calculate the MIPS values for each host. To do so, and with the workloads presented in Section 5.3 in mind, it was decided that each Virtual Central Processing Unit (vCPU) would have a capacity of 2054 MIPS, which is the value of an Intel Pentium III processor [42]. This resulted in the estimates presented in Table 5.1, similar to the ones shown in the COSCO paper [2]. The MIPS values were calculated using the Equation (5.1)

$$MIPS = vCPU \times 2054 \tag{5.1}$$

In addition to the characteristics introduced in Table 5.1, the simulator needed more configurations defined based on the COSCO paper [2] and GitHub [43]. With this in mind, Table 5.2 represents the other configurations used in the simulator, according to each host's layer.

Table 5.2: Additional Host Characteristics.

| **Layer** | **RAM read** (MB/s) | **RAM write** (MB/s) | **Disk read** (MB/s) | **Disk write** (MB/s) | **Latency** (ms) | **Bandwidth** (MB/s) |
|---|---|---|---|---|---|---|
| Cloud | 376.54 | 200.00 | 11.64 | 1.164 | 76 | 2500 |
| Fog | 360.00 | 305.00 | 10.38 | 0.619 | 20 | 2000 |
| Edge | 372.00 | 266.75 | 13.42 | 1.011 | 3 | 1000 |

These values do not significantly impact the results since the main focus is on the CPU, but they must be defined because they are mandatory for the simulator to work.

## 5.3 Workload Configuration

To simulate the proposed environment, some workload had to be defined. The used workload is based on the datasets created by Azure and BitBrains, which are built with real-world data.

The BitBrains dataset [44] was created using metrics from 1750 VMs from a distributed datacenter from Bitbrains [45]. It is organised in 2 traces: fastStorage and Rnd. Because fastStorage is focused on fast storage machines, the more general Rnd trace was chosen, which contains the performance metrics of 500 VMs. The Rnd trace contains 11 properties:

1. **Timestamp**: in UNIX, corresponding to the number of milliseconds since 1970-01-01,
2. **CPU cores**: number of provisioned virtual CPU cores,
3. **Provisioned CPU capacity (requested CPU)**: the capacity of the CPUs in terms of MHz, it equals the $numberofcoresxspeedpercore$,
4. **CPU usage**: in terms of MHz,
5. **CPU usage**: in terms of percentage,
6. **Provisioned memory capacity (requested memory)**: the capacity of the memory of the VM in terms of KB,
7. **Memory usage**: the memory that is actively used in terms of KB,
8. **Disk read throughput**: in terms of KB/s,
9. **Disk write throughput**: in terms of KB/s,
10. **Network received throughput**: in terms of KB/s,
11. **Network transmitted throughput**: in terms of KB/s

Although each file had 11 columns, only 6 were used: provisioned CPU capacity, memory usage, Disk read throughput, Disk write throughput, Network received throughput and Network transmitted throughput. Table 5.3 shows the first interval of the first VM of the Rnd trace.

The Azure dataset [46, 47] is divided into 2 traces, corresponding to data from 2017 and 2019. In this work, only the 2019 trace was used which contains information about 2 million VMs and 1.9 billion readings. From this dataset only one column was used: CPU usage.

Table 5.3: BitBrains First Metrics of the First VM.

| **CPU capacity** [MHZ] | **Memory usage** [KB] | **Disk read** [KB/s] | **Disk write** [KB/s] | **Network received** [KB/s] | **Network transmitted** [KB/s] |
|---|---|---|---|---|---|
| 5851.9989 | 520092.8 | 0.0 | 0.2667 | 15.9333 | 22.0667 |
| 5851.9989 | 536869.6 | 0.0 | 0.4 | 12.6667 | 17.5333 |
| 5851.9989 | 782934.7 | 8.4 | 3.6667 | 13.6 | 18.6667 |
| 5851.9989 | 911559.2 | 0.0 | 0.8667 | 15.0667 | 20.9333 |
| 5851.9989 | 531276.8 | 0.0 | 0.4667 | 12.7333 | 17.6 |

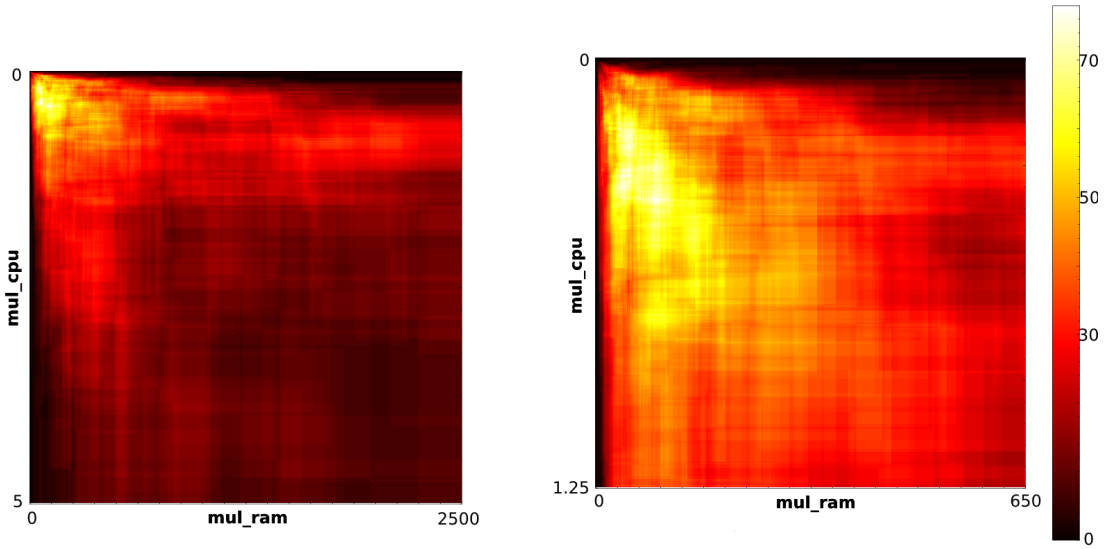
With these two datasets in mind, the workloads could be created. They were created using the BitBrains dataset as a base and then adding the Azure dataset, by attributing a percentage of the CPU usage of the Azure dataset to each VM of the BitBrains dataset for each interval.

Since the BitBrains dataset gives the CPU usage in terms of MHz, and no Instructions Per Cycle (IPC) is provided, and once the simulator requires MIPS values, the CPU usage was converted to MIPS using the following formula, and considering that the IPC is 1:

$$MIPS = Frequency * IPC \tag{5.2}$$

This means that 500 different containers that simulate work can be created, one for each VM of the BitBrains dataset. It is intended to simulate a cloud-to-edge environment, thus, it is necessary to distribute these workloads through the three layers of the environment, where cloud nodes will have the heaviest workloads and edge nodes the lightest. To do so, in the first approach, and for the first scenario presented in Section 6.1, it was attempted to find a `cpu_multiplier` and a `ram_multiplier` in order to fit most of the workloads in the three layers. The `cpu_multiplier` and the `ram_multiplier` are used to multiply the CPU and RAM usage of each container. The best `cpu_multiplier` and `ram_multiplier` combination was approximated by simulating the workloads and verifying where the containers would fit. The idea was to use most of the 500 workloads and distribute them as evenly as possible through the three layers. The results of this simulation are presented in Figure 5.2.

After some more experimentation, this approach was abandoned because the number of containers per host per interval was increased and so, these multipliers were reset to 1.




(a) General (b) Closer look

Figure 5.2: Heatmap of the number of possible workloads for each combination of CPU and RAM multipliers.

Besides the workloads, it is also necessary to define the containers' duration and frequency of arrival. The duration of the containers' execution was defined by following a normal distribution. The mean and standard deviation of the normal distribution varied along the experimentations and ended up in the values of 4 and 1, respectively. These values result in the distribution presented in Figure 5.3.

New containers arrive every interval. The number of containers to arrive in each interval is the value that is being defined. In the first experimentations, for every interval one new container arrived, for each host in the edge layer. With the development of the scenarios, this was changed to a Poisson distribution with a mean of 2. This resulted in the distribution presented in Figure 5.4.

It is important to note that Figure 5.4a represents the true Poisson distribution, but, since we want to add at least one container per interval, the distribution presented in Figure 5.4b was used instead. It is the same as the Poisson distribution but with the probability of 0 added to the probability of 1.

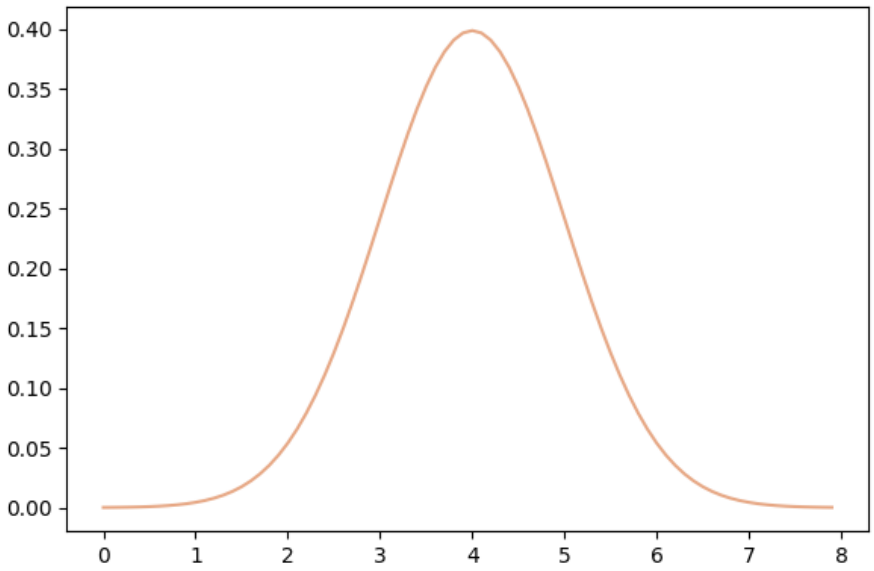
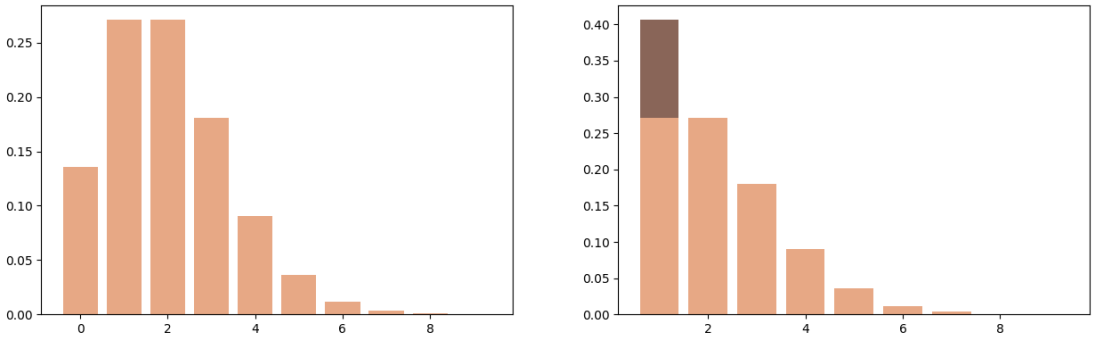



Figure 5.3: Distribution of the containers' duration.




(a) Poisson

(b) Poisson Arrival

Figure 5.4: Distribution of number of containers to arrive.

## 5.4 Adding SFC support

As mentioned before, the COSCO simulator was designed to simulate task placement problems and was therefore not fit to simulate the proposed environment without some modifications. For each time interval, new containers arrived and a scheduler decided where to place them. This was not the desired behaviour since the containers should flow through the environment, and so, one of the main modifications was the addition of SFC support.

SFC is a technique that allows the creation of a chain of services that must be executed in a specific order. In this work, it is used to simulate the flow of containers through the three layers of the environment. The flow of containers is presented in Figure 5.5.

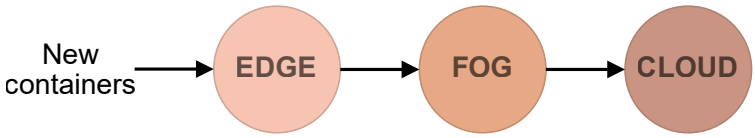



Figure 5.5: Flow of containers in the SFC.

The flux of containers starts in the edge and ends in the cloud in a unidirectional way. The containers arrive at the edge layer and are placed in the edge hosts. When a container finishes its execution in an edge host, a new container is created on the fog layer and placed in a fog host that is connected to the edge host. Similarly, whenever a container finishes its execution in a fog host, a new container is created on the cloud layer and placed in a cloud host that is connected to the fog host.

## 5.5 Fault Injection

The fault injector is one of the main contributions of this work. It is used to simulate the faults that can happen in the environment. The fault injection is based on the *stress-ng* tool [48] and is used to simulate recurrent and cumulative faults.

*Stress-ng* is widely used in the industry to test the reliability of systems as can be seen in many works [49–52]. It is a tool used to stress-test a computer system. It has a wide range of CPU-specific stress tests that exercise floating point, integer, bit manipulation and control flow. There are a plethora of stress-inducing methods, for example, the computation of Hamming codes, Fibonacci sequences, recursive calls, a lot of iterations, etc.

From the *stress-ng* tool, the idea that faults are some real work that is done in the background was taken. Thus, to simulate stress, the fault injector creates phantom containers that the host can't see but consume its resources, using the same workload as the normal work containers.

The work by Soualhia, Fu and Khomh [52] also inspired the fault injector. In this work, two types of faults are simulated:

- **Recurrent faults**: They happen recurrently. For example, a recurrent fault can be a fault that happens every $x$ time interval and lasts $y$ time intervals. Each time the fault happens, it can have a different intensity.
- **Cumulative faults**: They can get worse over time. For example, a cumulative fault can be a fault that happens and then, from time to time, the fault increases in intensity, getting progressively worse.

From the same work [52], the idea of a fault interval and cooldown interval was taken. The fault interval is the interval when faults can be injected and the cooldown interval is the time window between each fault, when no faults are injected and older faults are cleared. Figure 5.6 shows the fault injection process for recurrent and cumulative faults.

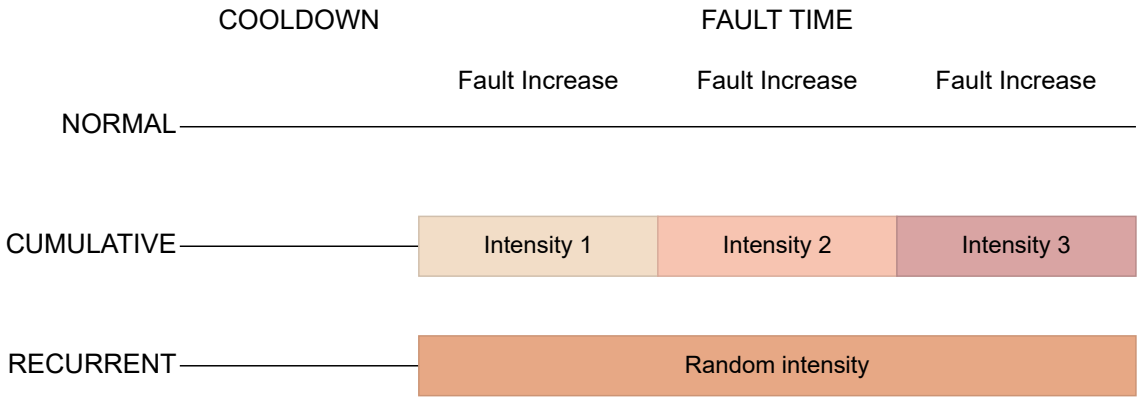



Figure 5.6: Types of faults that can be injected.

Despite both types of faults having been implemented and tested, this work only focuses on cumulative faults. In this sense, the fault injection process pseudocode is presented in Listing 5.1.

```
1  CYCLE_TIME = COOLDOWN_TIME + FAULT_TIME
2  cycle_stage = current_interval % CYCLE_TIME

3  if cycle_stage == 0:
4      # First interval of cooldown
5      clearFaults()

6  elif cycle_stage >= COOLDOWN_TIME:
7      # Fault time
8      if (cycle_stage - COOLDOWN_TIME) % FAULT_INCREASE_TIME == 0:
9          # Increase fault intensity time
10         if random() < FAULT_PROBABILITY:
11             injectFaults()
```

Listing 5.1: Fault injection process.

Therefore, for a process where there can be several different intervals for fault intensity increase, as Figure 5.6 shows, the fault injection process works as follows:

1. The fault injector receives the cooldown interval, the fault interval, the increase interval and the probabilities of fault.
2. The main loop starts:
   2.1. Waits for the cooldown interval to end.
   2.2. Enters the fault interval. The second loop starts *(line 6)*:
      2.2.1. Injects faults with the given probability *(lines 10 and 11)*.
      2.2.2. Waits for the increase interval *(line 8)*.
   2.3. Enters the cooldown interval. Clears all faults *(lines 3 and 5)*.

In the initial stage, there was only 1 phantom container per injection. This was changed to 2 phantom containers per injection, in order to increase the impact of the faults once the size of the container was reduced and the number of containers per host per interval was increased.

Exploring fault injector parameter combinations, and only considering cumulative faults, the simulator is able to generate different types of datasets. As a result, 3 different datasets could be generated. The following list describes the parameters used to generate each dataset as well as the type of dataset generated, along with a brief description.

- **Normal dataset**: No faults are injected.

  `FAULT_PROBABILITY = 0`

- **Imbalanced dataset**: Faults are injected with a given probability. The vast majority of the dataset will have no faults. There are more faults with lower intensity and less faults with higher intensity. There are 3 different levels of intensity of faults.

  `FAULT_PROBABILITY = 0.3`

  `FAULT_TIME = 6`

  `FAULT_INCREASE_TIME = 2`

  `COOLDOWN_TIME = 14`

- **Balanced dataset**: Faults are always injected. `FAULT_INCREASE_TIME` must be equal to `COOLDOWN_TIME`. The dataset will have the same amount of data with no faults as with each level of intensity of faults.

  `FAULT_PROBABILITY = 1`

  `FAULT_TIME = 15`

  `FAULT_INCREASE_TIME = 5`

  `COOLDOWN_TIME = 5`

## 5.6 Failure Detection and Mitigation

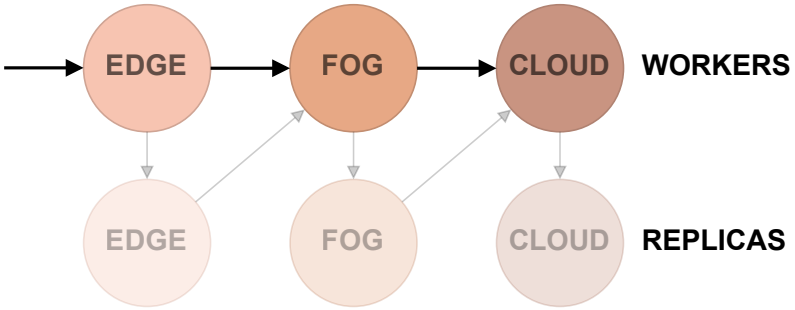
Once faults could be injected, the next step was to detect the failure of a host. It was decided that when the CPU or RAM usage of a host is above 90% it is considered a failure.

Now that failures can be detected, the next step is to mitigate them. The mitigation process makes use of the replicas that are created in the chaining process. These replicas do not receive containers, they only treat migrated containers from the associated host. Besides the associated host, each replica also has a parent in the layer above, which is the same parent as the associated host.

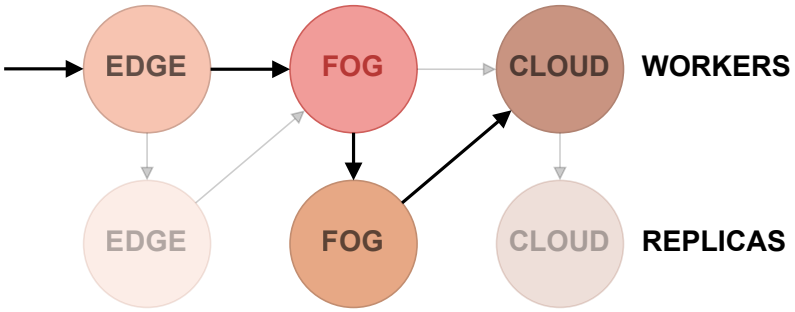
When a failure of a host is detected, meaning it is overloaded, the heaviest container that is running on that host is migrated to the replica. The heaviest container is the one that uses the most CPU or RAM resources, according to where the host is overloaded.

The migration process is done by moving the container from the host to the replica, keeping the container's state. This means that the container does not have to start from the beginning, it resumes its execution, from where it was, in the replica until it finishes.

When the container ends its execution in the replica, a new container is created in the parent of the replica, similar to what happens when a container finishes its execution in a host. This new container will be placed in the parent of the replica, which is the same parent as the host, simulating the flow of containers through the environment. Figure 5.7 shows the flow of containers in a scenario with replicas, both in a normal situation and when a failure is detected and mitigated.




(a) Normal




(b) Fog Overload

Figure 5.7: Flow of containers in the SFC.

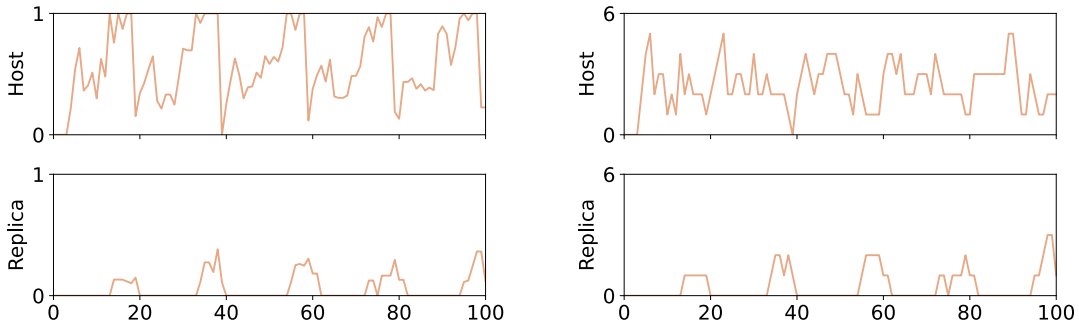
In Figure 5.7a, the containers arrive at the edge layer and follow the regular flow already described in Section 5.4, Figure 5.5. In Figure 5.7b, the containers arrive at the edge layer and follow the normal flow until the fog node. In the fog node, as the host is overloaded, the failure detection system detects the failure and migrates the heaviest container to the replica. The container continues its execution in the replica until it finishes. When the container finishes its execution in the replica, a new container is created in the cloud node, which is the same parent as the host, continuing the normal flow of containers through the environment.

In Listing 5.2 the failure detection and mitigation process pseudocode is shown.

Figure 5.8 shows the behaviour of the fog host and replica during the fault injection test. The test was done with an `COOLDOWN_TIME` of 10 and a `FAULT_TIME` of 10 as well. The `FAULT_INCREASE_TIME` was set to 1 as well as the `FAULT_PROBABILITY`. These would result in 10 different intensities of faults but result in a better visualisation of the failure detection and mitigation system working.

```
for host in host_list:

    # CPU usage above 90% - CPU Failure
    if host.getCPU() > 90:

        # All the containers in the host
        host_containers = getContainersOfHost(host)

        # All the containers in the host (Instructions Per Second)
        host_containers_IPS = [
            container.getBaseIPS() for container in host_containers
        ]

        # Container that consumes more CPU resources
        # -> has more instruction per second (IPs)
        heaviest_id = np.argmax(host_containers_IPS)
        heaviest_container = host_containers[heaviest_id]

        migrateContainer(heaviest_container, host)

    # RAM usage above 90% - RAM Failure
    if host.getRAM() > 90:

        # All the containers in the host
        host_containers = getContainersOfHost(host)

        # All the containers in the host (RAM size)
        host_containers_RAM = [
            container.getRAMSize() for container in host_containers
        ]

        # Container that consumes more RAM resources
        heaviest_id = np.argmax(host_containers_RAM)
        heaviest_container = host_containers[heaviest_id]

        migrateContainer(heaviest_container, host)
```

Listing 5.2: Failure detection and mitigation process.




(a) CPU Usage.

(b) Number of containers.

Figure 5.8: Metrics of the fog host and replica during the fault injection test.

As it can be seen in Figure 5.8, the fog host CPU usage starts increasing at the 10th interval for 10 intervals. In this time interval, the CPU usage surpasses the 90% threshold, leading to the failure detection and migration of the heaviest container to the replica. The CPU usage of the fog host decreases and the CPU usage of the replica starts to increase.

## 5.7 Additional Modifications

Aside from the main modifications presented in the previous sections, some other modifications were made to the simulator. The most important one was the improvement of the simulator's performance by reducing the time complexity. The simulator was very slow, taking more than 15 hours to simulate 30 runs with 1000 intervals each, in the first scenario. This was a problem because, if the simulator took that long to simulate the first scenario, which has only 3 hosts, it would take a lot more time to simulate the second and third scenarios, which have 7 and 25 hosts, respectively. This would be unfeasible due to the time constraints of this work.

After an exploration of the COSCO source code, it was found that the simulator had a time complexity of $O(n * m)$, where $n$ is the number of intervals in the simulation and $m$ is the number of containers created. This happens because, for each interval, the simulator iterates over all the created containers until that interval. It is important to note that, in the first scenario, there is only one edge node, resulting in only one entry point for the SFC. Apart from that, there are containers being created in the fog and cloud layers, whenever a container finishes its execution in the edge and fog layers, respectively. This means that, over the iterations, the number of containers increases, resulting in a slower and slower simulation.

If for this first scenario the simulator took more than 15 hours, for the second where there are 4 edge hosts, it would take a lot more than 4 days, and more than 2 weeks for the third scenario.

This problem was solved by changing the simulator's source code to have a time complexity of $O(n)$, where $n$ is the number of intervals in the simulation. This was achieved by removing the loop that iterated over all the created containers. This way, for each interval, the simulator only iterates over the containers actively working. This resulted in a significant improvement in the simulator's performance, reducing the simulation to less than 1 hour, for the first scenario. This improvement was also reflected in the second and third scenarios, making it possible to simulate many more times and allowing the tuning of the parameters of the fault injector and the simulator.

The number of hosts also plays a relevant role in the simulation time, but, since the number of hosts does not vary for the same scenario, it was not the main factor that impacted the simulation time, it is the number of containers created, which is proportional to the number of intervals.

It is important to remark that, although some pruning was made in order to improve the simulator's performance, it did not have any impact on the results since it was only removed iterations over containers that were not actively working. This means that the results are the same, but the simulation time is reduced.

## 5.8 Summary

In this chapter, the simulation methodology was presented. The used simulator was COSCO, which was modified to effectively simulate the proposed environment.

The hosts' configuration was presented, as well as the workload used to simulate the environment, which was based on the datasets created by BitBrains [44] and Azure [46].

The main modifications made to the simulator were described:

- A chaining mechanism was implemented to simulate the flow of containers through the environment, from the edge to the cloud unidirectionally.
- A fault injection mechanism was implemented to simulate recurrent and cumulative faults.
- A failure detection and mitigation system was implemented to detect and mitigate the failure of a host. Whenever a host was overloaded, the heaviest container was migrated to a replica of the host.

Finally, some other important modifications were described, like the improvement of the simulator's performance by reducing the time complexity.

# Chapter 6

# Simulation Scenarios and Dataset Generation

The implemented simulation scenarios are detailed in this chapter. The scenarios are divided into three types: linear chaining, balanced fixed tree and imbalanced dynamic tree. Each scenario is described in detail, including the number of hosts, the resources of each host and, most importantly, the architecture used to connect the hosts. The scenarios are used to create the datasets, making use of the fault injection mechanism, so the datasets and corresponding exploratory data analysis are also presented in this chapter.

## 6.1 Scenario 1: Linear Chaining

The first scenario is the simplest one but it is also the most important one because, with it, it is possible to test the fault injection mechanism and the simulator itself, as well as to tune the parameters of the fault injection mechanism. Moreover, it is the one that is used to generate the first dataset, which is used to train the first models.

It is composed of a linear chaining of hosts, where each host is connected to the next one. The scenario is the simplest one because it is made up of only three hosts. One edge host, one fog host and one cloud host, each having a corresponding replica. The architecture of the scenario is shown in Figure 6.1.

For this scenario, the hosts have the same resources as Azure's B2s, B4ms and B8ms virtual machines, for the edge, fog and cloud hosts, respectively. These resource specifications are shown in Table 6.1.

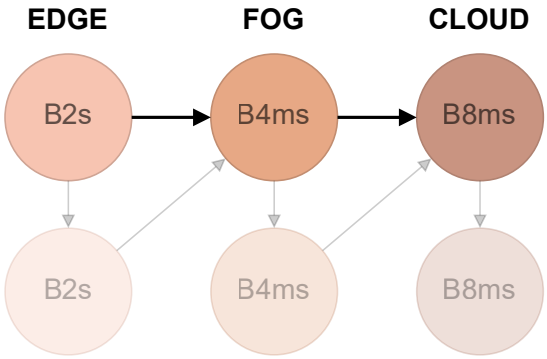
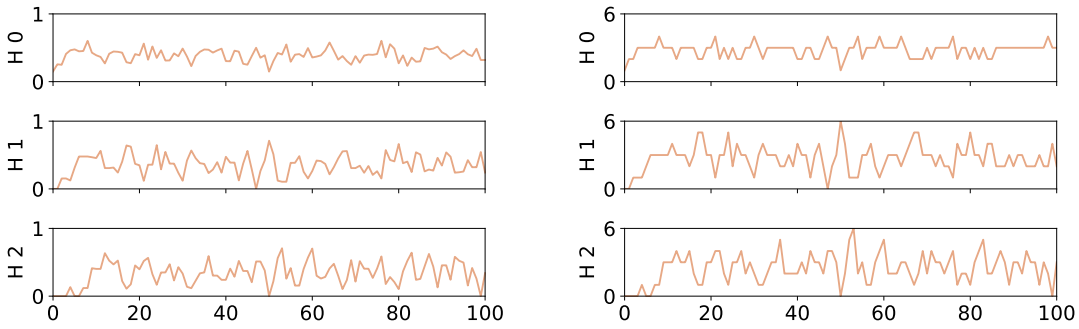



Figure 6.1: Architecture of the first scenario.

Table 6.1: Scenario 1 Hosts Configuration.

| Layer | Name | MIPS | Memory (GB) | Disk (GB) |
|---|---|---|---|---|
| EDGE | B2s | 4096 | 4 | 8 |
| FOG | B4ms | 8192 | 16 | 32 |
| CLOUD | B8ms | 16384 | 32 | 64 |

Initially, one new container arrived at the edge host every interval, with a duration following a Normal distribution with a mean of 3 and a standard deviation of 0.5, $\mathcal{N}(3, 0.5^2)$. This resulted in the metrics presented in Figure 6.2, where `H0` is the edge host, `H1` the fog host and `H2` the cloud host. The simulation was run for 100 intervals, and no faults were injected, so both metrics were stable and the replicas were not used.

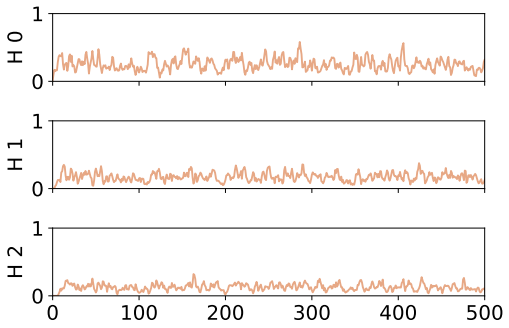



(a) CPU Usage.

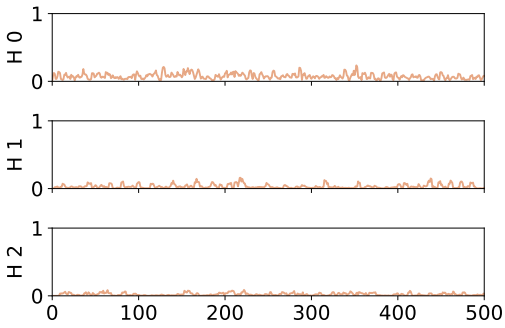
(b) Number of containers.

Figure 6.2: Scenario 1 initial metrics.

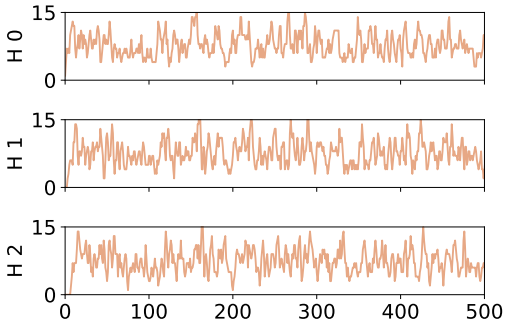
It can be seen in Figure 6.2 that, in this stable version of the scenario, the hosts only have, on average, 3 to 4 containers running at the same time, which is a very small number of containers. This is due to the fact that the containers have a large need for resources, a minimum of 10% and a maximum of 20% of the CPU capacity of the host, and a small duration combined with a small number of containers that arrive at the system every interval. This results in a small number of containers running at the same time, which is not realistic. This problem was solved by increasing the number of containers that arrive at the system every interval, now following a Poisson distribution with a mean of 2, $\mathcal{P}(2)$, increasing the container's duration, now following a Normal distribution with a mean of 4 and a standard deviation of 1, $\mathcal{N}(4, 1^2)$, and, finally, decreasing the CPU resources consumed by each container. These final values are discussed in Section 5.3 and, when combined with the injection of faults, result in the metrics presented in Figure 6.3. They were chosen because their combination resulted in a stable scenario, without the CPU usage growing infinitely and always having some containers running at a given interval.




(a) CPU Usage.




(b) RAM Usage.




(c) Number of containers.

Figure 6.3: Scenario 1 metrics.

This version of the scenario is the one that was used to generate the first dataset. Although there is a reduction of the overall CPU usage per host, in comparison with the previous version, this was not considered a problem at all because, in the next scenarios, the fog and cloud hosts will have more children, resulting in a higher number of containers running at the same time, which will lead to a higher CPU usage.

That being said, the first dataset was generated and the exploratory data analysis was performed. For this analysis, all the data from all the hosts was merged into one unique dataset. This dataset, like the other ones, is composed of 30 simulations, each with 1001 intervals, resulting in a total of $30030 \times number\_of\_hosts$ rows. This was done in order to minimise statistical errors.

For each scenario, 2 different datasets were created, one for the CPU faults and another for the RAM faults. Each dataset has 4 different labels, one for each fault intensity, and one for the normal behaviour. Table 6.2 and Table 6.3 show a sample of the datasets for the CPU and RAM faults, respectively.

Table 6.2: Sample of the dataset for CPU faults in scenario 1.

| CPU usage | Number of containers | Base IPS | Available IPS | IPS cap | Apparent IPS | Fault intensity |
|---|---|---|---|---|---|---|
| 6.6 | 4 | 467.1 | 15643.9 | 16111.0 | 1060.0 | 0 |
| 31.7 | 10 | 551.3 | 3477.7 | 4029.0 | 1278.0 | 1 |
| 16.2 | 8 | 748.0 | 7354.0 | 8102.0 | 1313.0 | 0 |
| 23.8 | 7 | 257.7 | 7844.3 | 8102.0 | 1927.0 | 2 |
| 15.3 | 7 | 399.8 | 7702.2 | 8102.0 | 1242.0 | 0 |

Table 6.3: Sample of the dataset for RAM faults in scenario 1.

| RAM | Number of | RAM | | | Available RAM | | | Fault |
|---|---|---|---|---|---|---|---|---|
| usage | containers | size | read | write | size | read | write | intensity |
| 0.6 | 5 | 95.1 | 24.23 | 24.28 | 17084.9 | 335.3 | 280.7 | 0 |
| 6.7 | 8 | 289.5 | 0.07 | 0.06 | 4005.5 | 371.9 | 199.9 | 0 |
| 1.4 | 8 | 239.3 | 0.04 | 0.03 | 16940.7 | 360.0 | 305.0 | 1 |
| 12.7 | 9 | 546.6 | 0.01 | 0.01 | 3748.4 | 372.0 | 200.0 | 2 |
| 7.8 | 7 | 333.2 | 0.04 | 0.04 | 3961.8 | 371.9 | 199.9 | 0 |

With this modified version of the simulator, metrics related to CPU, RAM, disk and containers could be gathered, but only the metrics presented in Table 6.2 and Table 6.3 were used.

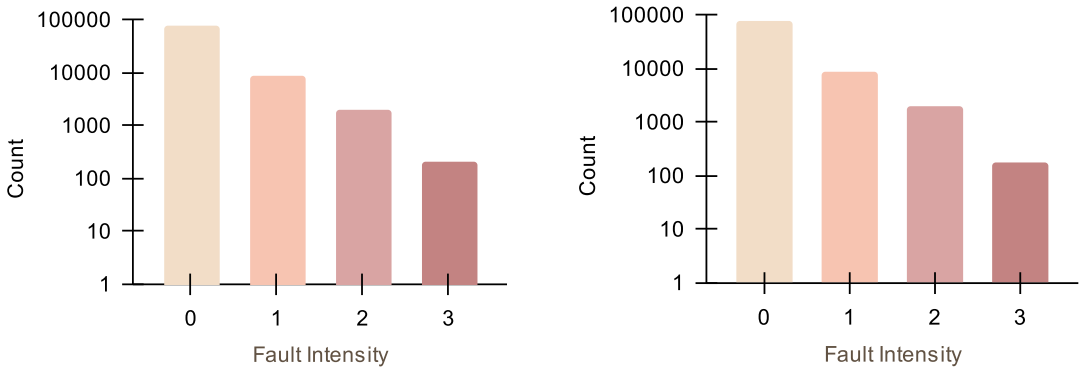
Those metrics are the following:

1. **Number of containers**: the number of containers running,
2. **CPU usage**: in terms of percentage,
3. **Base Instructions Per Second (IPS)**: the base MIPS of the host, it is the sum of mandatory MIPS of all the containers running on the host,
4. **Available IPS**: in terms of MIPS,
5. **IPS cap**: capacity of the host in terms of MIPS,
6. **Apparent IPS**: the actual MIPS of the host, it is the sum of the MIPS of all the containers running on the host,
7. **RAM usage**: in terms of percentage,
8. **RAM read, write**: in terms of KB/s,
9. **RAM size**: in terms of KB,
10. **Available RAM read, write**: in terms of KB/s,
11. **Available RAM size**: in terms of KB,

After the generation of the dataset, the exploratory data analysis was performed. The rest of the exploratory data analysis of the dataset is presented in Section A.1.

As Figure 6.4 shows, the dataset is very imbalanced, with the normal behaviour being the most common one, 88.2% for the CPU faults datasets, followed by the level 1 fault intensity, 9.4%. This is due to the fact that, besides the time to cool down, the fault injection mechanism injects faults randomly, so it is expected that the normal behaviour is the most common one. Fault intensity 1 is the second most common one because it is the first level of fault intensity, so it is expected. The fault intensity 3 is, similarly, the rarest one because, for it to happen, the fault injection mechanism has to inject 3 faults in the same fault interval, which is very unlikely to happen corresponding to only 0.2% of the dataset.

By analysing the pairplot of Figure 6.6 it can be seen that, for some metrics, there is a gap in the data. This happens because of the lower number of hosts and the lack of variability in terms of hosts' resources, resulting in the data being divided into 3 groups. With this pairplot, it can also be seen that the faults are not very well separated in some cases. The faults can be somewhat separated from the normal behaviour, but the fault intensities are not very well separated from each other, which could be a problem for the models to learn the differences between the fault intensities.




(a) CPU faults.

(b) RAM faults.

Figure 6.4: Scenario 1 - Fault distribution.

Figure 6.7 shows the correlation between the metrics for the CPU faults dataset. It can be seen that the features are not very correlated with the faults. The only values not close to zero are for the `cpu` and `apparentips` metrics.

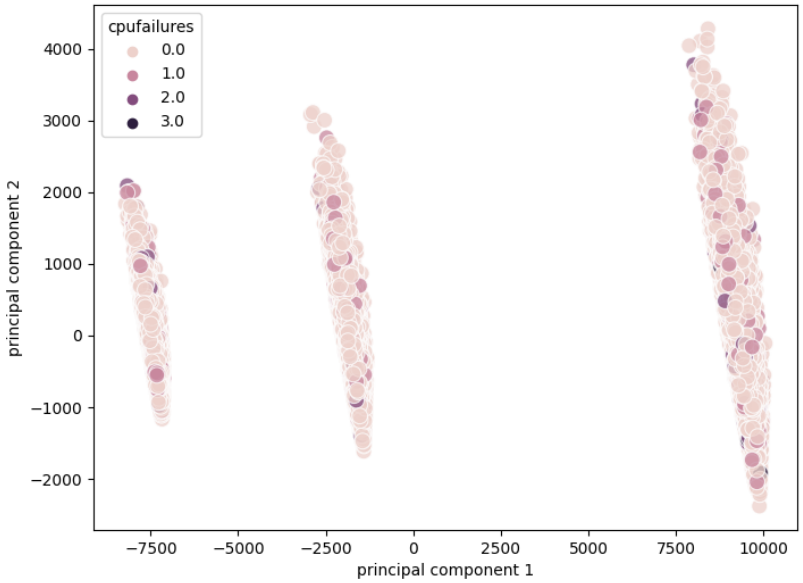
Feature importance was also calculated for the CPU faults dataset. The results are shown in Table 6.4. It can be seen that, once again, the `cpu` and the `apparentips` metrics appear to be the most important ones. This is due to the direct relation between the `cpu` and `apparentips` metrics. The `cpu` is the percentage of CPU used, which is calculated as follows:

$$cpu = apparentips/ipscap \tag{6.1}$$

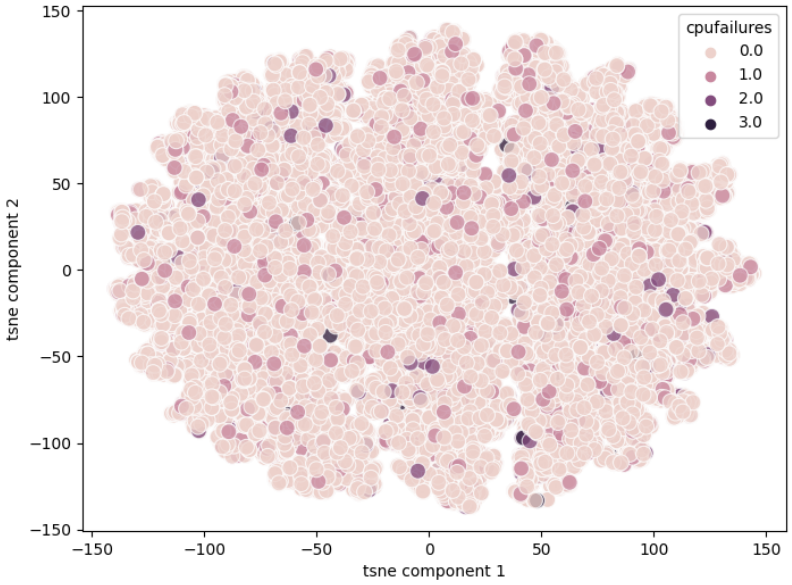
Table 6.4: Feature importance for CPU faults in scenario 1.

| | **Select K Best** | | **Feature Importance** |
|---|---|---|---|
| | **ANOVA** | **CHI2** | **Extra Tree Classifier** |
| cpu | **1657.57** | **1.74e+04** | **0.210** |
| apparentips | **1639.30** | **1.52e+06** | **0.208** |
| numcontainers | 4.11 | 1.02e+01 | **0.296** |
| baseips | 2.21 | 1.43e+03 | 0.137 |
| ipsavailable | 0.71 | 5.74e+03 | 0.139 |
| ipscap | 0.54 | 4.35e+03 | 0.009 |

Because feature importance and selection did not have a big impact, it was decided to use feature reduction in an attempt to better visualise the data. The results are shown in Figure 6.5. This reduction did not give a good visualisation of the data and led to worse results in the models, so it was not used.




(a) PCA




(b) TSNE

Figure 6.5: Feature reduction for CPU faults in scenario 1.

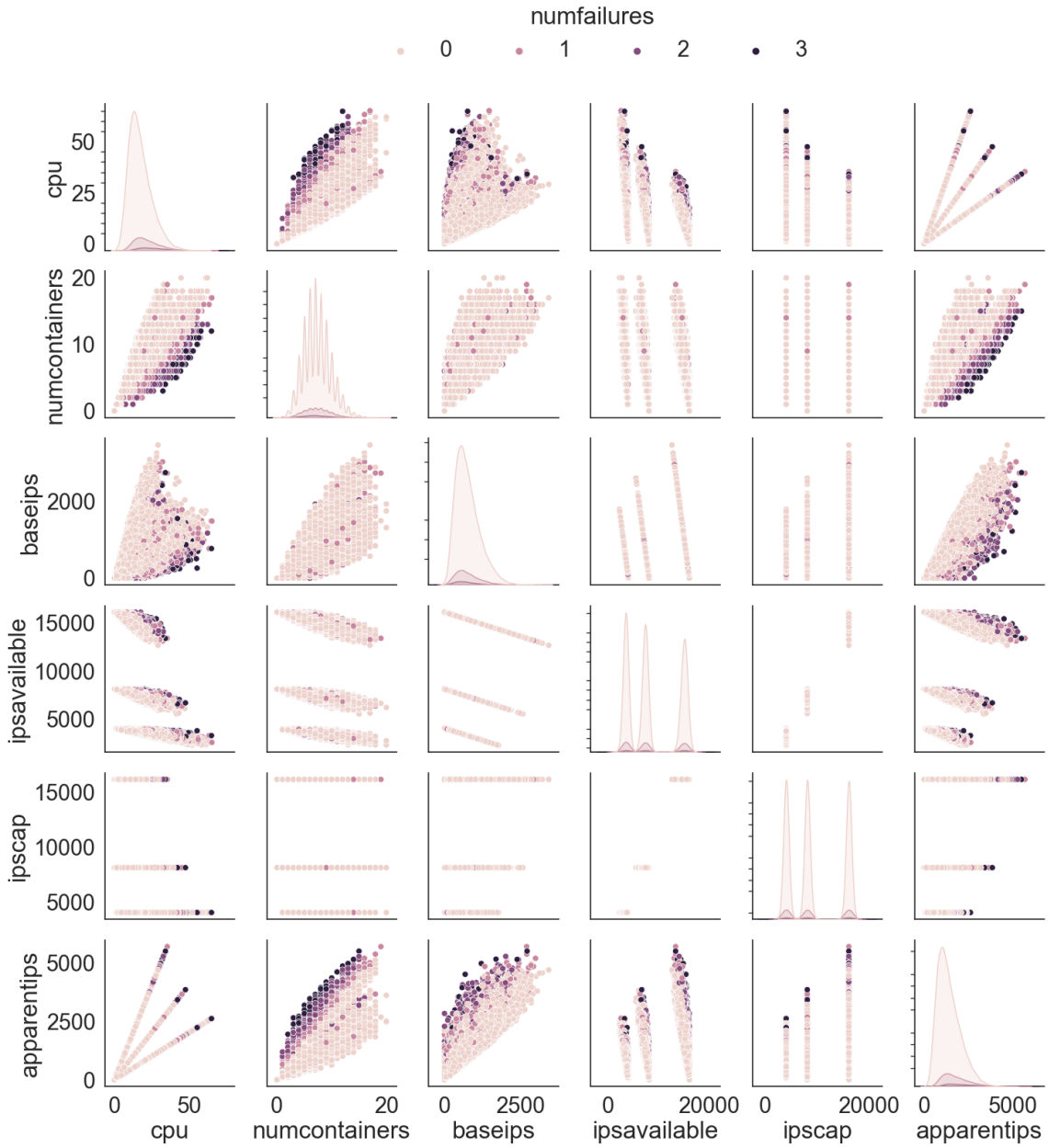



Figure 6.6: Pairplot of the CPU metrics collected from the hosts in scenario 1.


Figure 6.7: Correlation matrix of the CPU metrics collected from the hosts in scenario 1.

## 6.2 Scenario 2: Balanced Fixed Tree

The second scenario is a more complex one, with a balanced fixed tree architecture. It is composed of 7 hosts: 4 edge hosts, 2 fog hosts and 1 cloud host, each having a corresponding replica. The architecture of this scenario is shown in Figure 6.8. It is more complex because it has more hosts and because they also have more children, but it is also closer to real-world scenarios where we have more hosts in the edge and less in the cloud, and more resources in the cloud and less in the edge. Fog and cloud hosts also receive work from more hosts, which is more realistic.

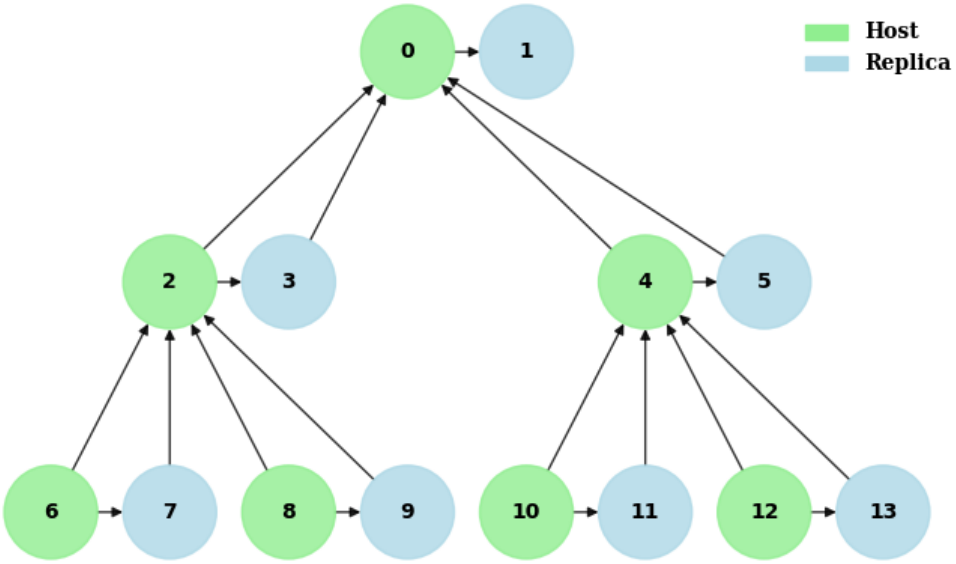



Figure 6.8: Architecture of the second scenario.

In the first version of this scenario, the hosts had the same resources as the last scenario, Table 6.1, but this led to the same problem mentioned in the previous scenario, the low variability in terms of hosts' resources. This problem was solved by assigning a range of resources to each layer instead of one fixed value, as shown in Table 6.5.

Table 6.5: Scenario 2 Hosts Configuration.

| Layer | Name | MIPS | Memory (GB) | Disk (GB) |
|---|---|---|---|---|
| EDGE | B1ms - B2s | 2054 - 4096 | 2 - 4 | 4 - 8 |
| FOG | B8ms - B12ms | 16384 - 24576 | 32 - 48 | 64 - 96 |
| CLOUD | B20ms | 40960 | 80 | 160 |

For this scenario, only a balanced dataset was generated.

Looking at the pairplot of Figure 6.9 it can be seen that the data is more spread out, which is due to the higher number of hosts and the higher variability in terms of hosts' resources. Despite this improvement, there are still some gaps in the data, which is due to the difference in resources of hosts from different layers, so the data is divided into 3 groups, one for each layer.

The rest of the exploratory data analysis of the dataset is presented in Section A.2.

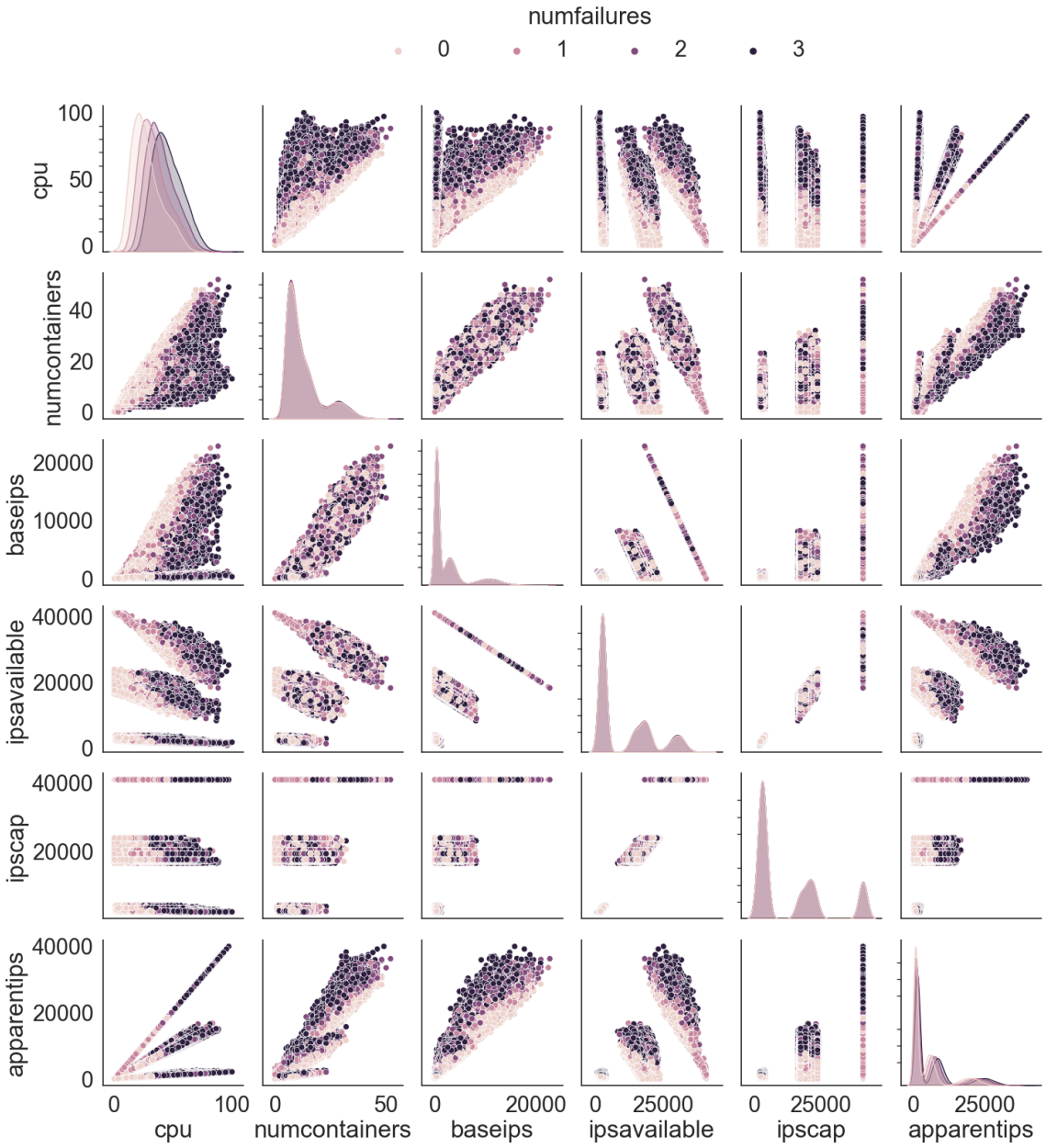
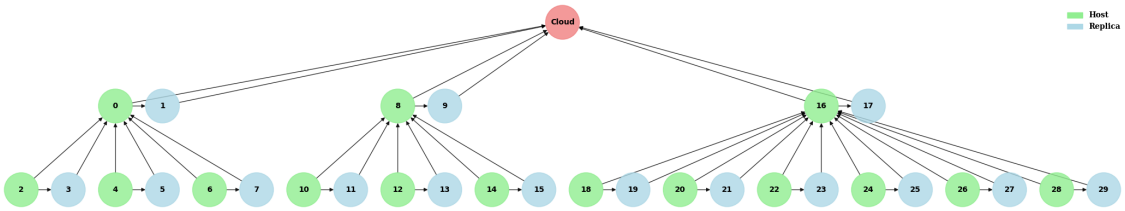



Figure 6.9: Pairplot of the CPU metrics collected from the hosts in scenario 2.

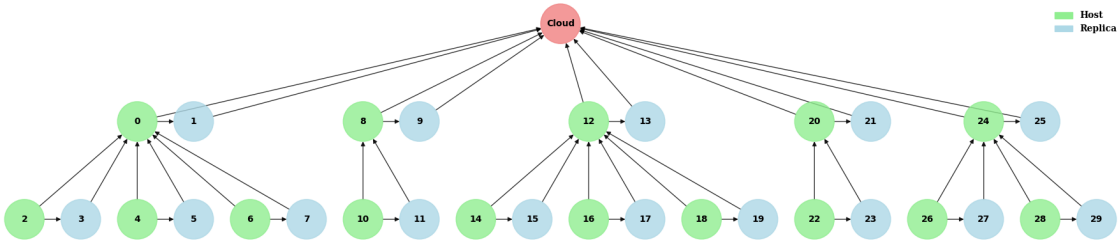
## 6.3 Scenario 3: Imbalanced Dynamic Tree

The third and last scenario is the most complex one. It is composed of 50 nodes: 25 hosts and 25 replicas, which are distributed across the fog and edge layers. For this, there was a need to create an algorithm to distribute these nodes depicted in Listing 6.1. The algorithm is very simple and starts by creating a fog host and respective replica *(line 7)*. Then, according to this node's capacity, it calculates the maximum number of children that this node can easily support *(line 12)*. Then, it creates a random number of children, between 1 and the maximum number of children, and respective replicas *(lines 18-22)*. This process iterates until the desired number of nodes is reached. Some verifications are also made during this process in order to generate the exact number of nodes and to not create any fog host with no children *(lines 14 and 24)*.

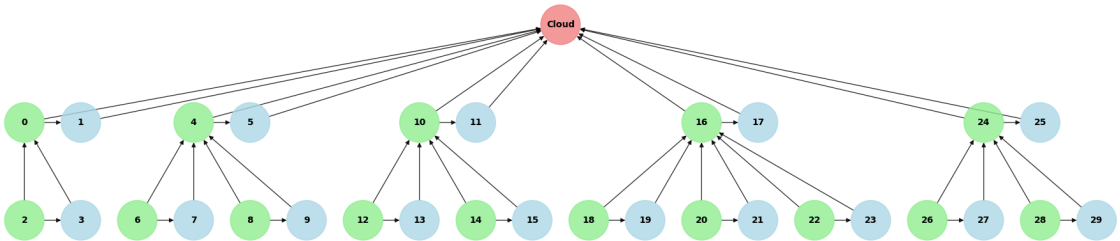
Running this algorithm for 30 nodes resulted in the architecture shown in Figure 6.10 where in red we see the cloud host, in green the hosts and in blue the replicas.




(a) 3 Fog Nodes.




(b) 4 Fog Nodes.




(c) 5 Fog Nodes.

Figure 6.10: Architecture of the third scenario. Some examples with 30 nodes.

```
hosts_counter = 0 # Number of hosts already generated
while hosts_counter < num_hosts:
    # Remaining hosts to generate
    remaining_hosts = num_hosts - hosts_counter

    # FOG
    fog_id = hosts_counter
    generateHostAndReplicas("fog", fog_id, None)
    hosts_counter += 2
    remaining_hosts -=2

    # Calculate max children for this fog.
    # Number maximum of children is proportional to the number of IPS
    fog_max_children = calculateMaxChildren(fog_id)

    # EDGE
    if remaining_hosts < 6:
        # There are not enough hosts to create a new fog with 1 edge
        # Add all remaining hosts to the last fog
        n_edge = remaining_hosts//2
    else:
        n_edge = randint(1, min(fog_max_children,remaining_hosts//2))

    for _ in range(n_edge):
        generateHostAndReplicas("edge", hosts_counter, fog_id)
        hosts_counter += 2

    # If the remaining hosts are less than 4, add them to the last fog
    if 0 < num_hosts - hosts_counter < 4:
        generateHostAndReplicas("edge", hosts_counter, fog_id)
        hosts_counter += 2
```

Listing 6.1: Dynamic tree building process.

For this final scenario, analogous to the previous one, the hosts had a range of resources in order to have more variability in terms of hosts' resources. The novelty here is that the range was increased and the cloud is considered to have infinite resources, so there is no need to simulate a cloud replica. The resources of the hosts are shown in Table 6.6.

Table 6.6: Scenario 3 Hosts Configuration.

| **Layer** | **Name** | **MIPS** | **Memory (GB)** | **Disk (GB)** |
|---|---|---|---|---|
| EDGE | B1ms - B4ms & B2ms | 2054 - 8192 | 2 - 8 | 4 - 16 |
| FOG | B12ms - B20ms | 24576 - 40960 | 48 - 80 | 96 - 160 |
| CLOUD | - | $\infty$ | $\infty$ | $\infty$ |

Though both the CPU and RAM faults datasets were generated, only the CPU faults dataset was used for the models, since it was the one that had the best results on the models

There were generated 3 different datasets, one containing only the data from the edge hosts, one containing only the data from the fog hosts and one containing the data from both the edge and fog hosts. The reason for this is that the edge hosts have different resources and workloads than the fog hosts, so it is expected that the models will learn different patterns for each one.

## 6.4 Summary

In this chapter, the implemented simulation scenarios were described. The scenarios evolved from a simple linear chaining of hosts to a more complex tree architecture. The scenarios were used to generate the datasets, making use of the fault injection mechanism, so the datasets and corresponding exploratory data analysis were also presented in this chapter.

# Chapter 7

# AI Models, Configuration and Data Preprocessing

This chapter presents the different models used in this thesis. The models are divided into two categories: classification and regression. The classification models are used to predict the intensity of the faults, while the regression models are used to predict the actual MIPS of the faults. The models are presented in the order they were tested. The first model is the simplest one, and the last one is the most complex.

## 7.1 Random Forest

The first model tested is the random forest. The random forest is a supervised learning algorithm that can be used for both classification and regression.

The Random Forest Classifier was the model chosen to be the baseline model because it is accurate and fast to train.

In an attempt to try to better improve these results, fine-tuning of the RFC parameters was performed. Therefore, a grid search was performed using the *GridSearchCV* class from the *scikit-learn* library. The parameters that were tuned and their values are shown in Listing 7.1.

Finally, in the last version of the dataset, the Random Forest Regressor was also tested. The RFR is a supervised learning algorithm having the same principal as the RFC, but instead of predicting a class, it predicts a continuous value.

```
param_grid = {
    "n_estimators": [50, 100, 200, 500],              # default 100
    "criterion": ["gini", "entropy", "log_loss"],     # default "gini"
    "min_samples_split": [2, 5, 10],                  # default 2
    "min_samples_leaf": [1, 2, 5],                    # default 1
    "max_features": [None, "sqrt", "log2"],           # default "sqrt"
    "bootstrap": [True, False],                       # default True
}
```

Listing 7.1: Parameters used in the fine-tuning of the RFC for scenario 1.

## 7.2 Neural Networks

The second model tested is the Neural Network. The NN is a supervised learning algorithm that was used for the classification problem.

For the NNs, four different architectures were explored, one more complex than the other. The architectures are shown in Listing B.1.

Since it is a good practice to normalise the data for NNs, the *StandardScaler* class from the *scikit-learn* library was trained with the training data and used to normalise the training, testing, and validation data. The models were trained for 100 epochs, with a batch size of 128, using the *Adam* optimizer and the *categorical_crossentropy* or *binary_crossentropy* loss functions, depending on the problem. There was also used the *EarlyStopping* callback to stop the training if the validation loss did not improve for 10 epochs, as well as the *ReduceLROnPlateau* callback to reduce the learning rate if the validation loss did not improve for 5 epochs.

## 7.3 Convolutional Neural Networks

The third and last model tested is the Convolutional Neural Network. The CNN is a supervised learning algorithm that was used for the classification problem.

For the CNNs, only one architecture was implemented, but, two different forms of normalisation were explored. The architecture is shown in Figure 7.1.

CNNs are mainly used for image classification, but they can also be used for time series classification. Preprocessing the data is a central step for the CNN training and evaluation. There is a need to reshape the data to a 3-dimensional array, where the data is represented as grey-scale images. For that, there is a need to normalise the data and, in this work, two different normalisation approaches were explored:

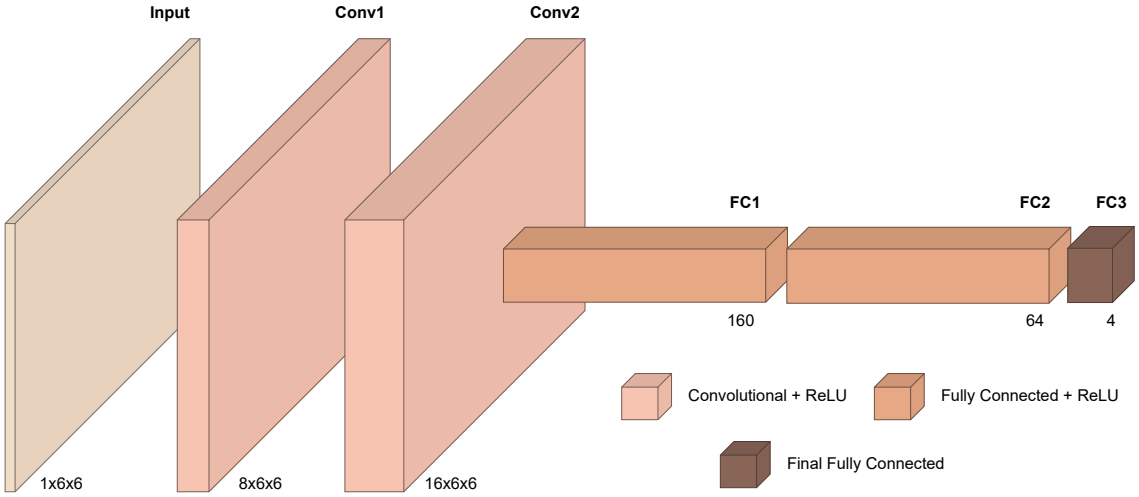



Figure 7.1: Architecture of the CNN used for fault prediction.

- **Local normalisation**: normalise the data according to IPS capacity of the CPU. Because of that, `ipscap` was removed from the dataset. The number of containers was normalised according to the maximum number of containers in the host.

- **Global normalisation**: All the data, except the CPU usage, was normalised according to the maximum value of the feature for that layer in the training dataset.

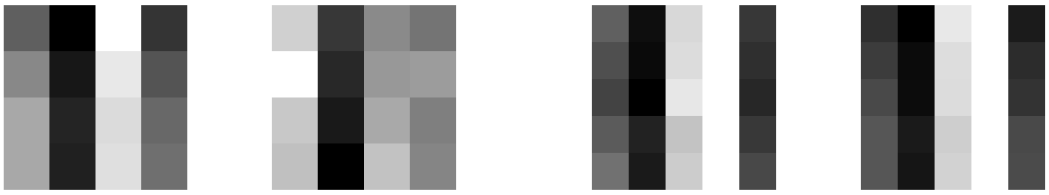
It is important to note that the CPU usage was normalised the same way for both approaches, dividing by 100. This is because the CPU usage is a percentage, and the values are between 0 and 100.

After the normalisation, images of 5 by 5 pixels were created, using 5 intervals for the first approach, where the `ipscap` was removed, or 6 by 6 pixels, for the second approach, using 6 time intervals. The label assigned to each image was the label of the last interval.

This means that this model is taking advantage of the time series nature of the data, as the CNN will be able to learn the patterns of the data in the time dimension by knowing the previous 4 or 5 values.

These transformations of the data resulted in images like the ones of Figure 7.2.

The models were trained for a maximum of 100 epochs, stopping early if the validation loss had not improved for 10 epochs. The *Adam* optimizer was used, with a learning rate of 0.001, and the *categorical_crossentropy* loss function was used.



(a) Local Normalisation. (b) Global Normalisation.

Figure 7.2: Example of images generated for the CNN for the multiclass problem in scenario 3.

## 7.4 Evaluation Metrics

In order to evaluate and compare the trained models, the choice of metrics can significantly impact the analysis of the results.

The first, and most common, metric is accuracy which measures the ratio of correct predictions to the total predictions

The following equation presents the formula for the accuracy, where True Positives (TP) corresponds to the number of positive examples correctly classified, True Negatives (TN) the negative examples correctly classified, False Positives (FP) the number of positive examples wrongly classified and False Negatives (FN) the number of negatives examples wrongly classified.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{7.1}$$

Despite accuracy being one of the most used metrics in many scenarios, it does not give much information in an imbalanced dataset. This happens because, if a dataset has 98% of the data in class 0, if a model learns nothing and outputs 0 for every entry, it will predict right 98% of the time, resulting in 98% accuracy. This is where precision, recall, and the F1 score come into play. These metrics are particularly valuable in imbalanced datasets because they focus on different aspects of prediction accuracy that are crucial for specific applications.

Precision quantifies the proportion of true positive predictions out of all positive predictions made by the model.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{7.2}$$

Recall, also known as sensitivity or true positive rate, measures the ratio of true positive predictions to all actual positive instances in the dataset.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{7.3}$$

Finally, the F1 score is the harmonic mean of precision and recall, providing a balanced measure that considers both false positives and false negatives.

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{7.4}$$

The aforementioned metrics are widely used for classification tasks, but, when it comes to regression analysis, they do not work. One commonly used metric in regression is the Mean Squared Error (MSE).

MSE calculates the average of the squared differences between the actual and predicted values. Squaring these differences has the effect of giving more significant weight to larger errors, making MSE particularly sensitive to outliers. As a result, a lower MSE value indicates better predictive performance.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{7.5}$$

Where $n$ is the total number of data points, $y_i$ represents the actual target value for the $i$-th data point and $\hat{y}_i$ is the predicted value for the $i$-th data point. Consequently, $(y_i - \hat{y}_i)^2$ is the squared error for the $i$-th data point

## 7.5 Summary

In this chapter, the implemented models were presented as well as the training configuration. Besides that, some models needed a preprocessing of the data which was also addressed. Finally, the evaluation metrics were presented as well as their formula and importance.

# Chapter 8

# Results and Discussion

This chapter presents the results of the experiments in a sequential order and discusses them.

Section 8.1 presents the results for the RFC for the multiclass and binary problems for both CPU and RAM faults. It also presents the results when finetuning the RFC parameters.

Section 8.2 presents the results for the NNs and CNNs for the multiclass and binary problems focusing only on CPU faults. It also presents the results when using different normalisation approaches for the CNNs.

Section 8.3 presents the new labels for the dataset, where the IPS of the faults is also considered.

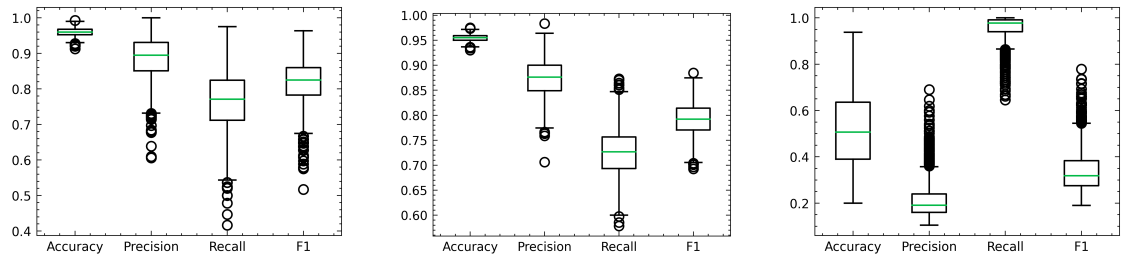
Finally, Section 8.4 presents the final results, where the same models were trained and evaluated with the new dataset. Only the multiclass problem was considered, as it is the one that changed when compared to the previous dataset. For simplicity, only the results for the CPU faults are presented. Then, at the end of the section, the regression problem is mentioned, where the RFR was trained and evaluated.

## 8.1 RFC results

For the preliminary results, the datasets generated were used to train and evaluate the models. The split was generally performed using 70% of the data for training and 30% for testing. In order to reduce statistical errors, at least for the RFC, they were trained and evaluated 50 times, and the results were averaged. The RFC was chosen because it is a simple model, fast to implement and train, and can be used as a baseline for the other models.

This section presents results for the multiclass problem, where the models were trained to predict the fault intensity, and for the binary problem, where the models were trained to predict if a fault occurred or not, converting any fault intensity greater than 0 to 1.

The first experimentations with ML were performed using a RFC over the first version of the dataset generated from the first scenario. Only the binary problem of the CPU faults dataset was used for this part. Three different models were trained: one trained and tested with the data regarding the edge host, another trained and tested with the data of all the hosts, and the last one trained with the data from the edge and fog hosts and tested with the data from the cloud host. The results of the performance of the RFC are shown in Figure 8.1.




(a) Trained and tested with the same host

(b) Trained and tested with all hosts

(c) Trained with edge and fog, tested with cloud

Figure 8.1: First results of the RFC for the CPU faults in scenario 1.

It can be seen that, as expected, a model trained for a specific host performs the best when tested with the same host. However, it is inviable to have a model for each host, as the number of hosts can be very high in a real scenario. Therefore, the model trained with all the hosts' information was used for the next experiments, since it is the one that performs the best when tested with all the hosts. The model trained with the edge and fog hosts and tested with the cloud host performed the worst, as the data from the edge and fog hosts is very different from the data from the cloud host. Figure 8.2 shows the difference between the true and predicted labels for the best-performing model of the 50 iterations.

After the first results, the first scenario was improved, as described in Section 6.1 Table 8.1 shows the results of the RFC for the CPU and RAM faults.

The results show that the RFC was able to achieve an average F1-score of 63% for the multiclass problem and 75% for the binary problem for the CPU faults. Despite having high accuracy, the other metrics show that the model is not performing that well, mainly predicting the RAM faults. This is due to the high imbalance of the dataset, where the majority of the data is from class 0, which represents the absence of faults.

Fine-tuning of the RFC parameters was performed resulting in the metrics of Table 8.2.

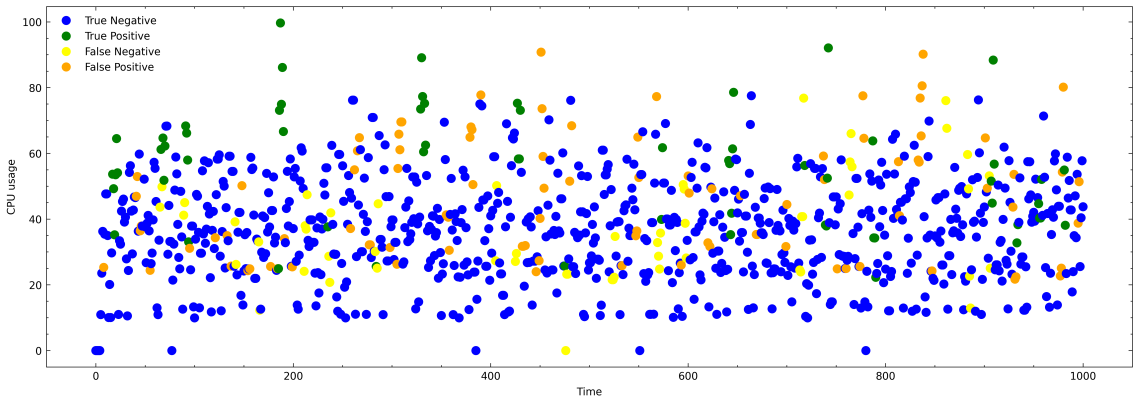



Figure 8.2: Difference between the predicted and the actual values for the CPU faults in scenario 1.

```
best_params = {
    'n_estimators': 200,
    'bootstrap': True,
    'criterion': 'entropy',
    'max_features': None,
    'min_samples_leaf': 1,
    'min_samples_split': 10,
}
```

Listing 8.1: Best parameters found for the RFC for the CPU faults multiclass problem in scenario 1.

Since the dataset was imbalanced, the accuracy metric was not a good metric to evaluate the models. Therefore, the F1-score was used and gave slightly better results when compared to the previous evaluation metric. The fine-tuning of the parameters allowed for improving the F1-score for the multiclass problem from 63% to 70% for the CPU faults. For the RAM faults, the F1-score had a slight improvement from 23% to 25%, but the model overfits the data, as the F1-score for the training set was 100%. Listing 8.1 shows the best parameters found for the RFC for the CPU faults multiclass problem in scenario 1.

For dataset 2, despite being a balanced dataset, the results were not as good as those from scenario 1. This happens because dataset 2, despite having more data, has more variability in all the features, which makes it harder for the models to learn, but a better representation of the real world. Table 8.3 shows the results of the RFC for the CPU and RAM faults.

Finally, for the third scenario, new approaches were introduced. Here, only the dataset with CPU faults was considered. There were 3 different versions of this dataset: one with only the data from the edge hosts, another with only the data from the fog hosts, and the last one with the data from the edge and fog hosts.

Table 8.1: Evaluation results for RFC in scenario 1.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Mean | 0.9318 | 0.7297 | 0.5922 | 0.6384 | 0.9459 | 0.8306 | 0.6780 | 0.7466 |
| Median | 0.9317 | 0.7359 | 0.5920 | 0.6388 | 0.9460 | 0.8302 | 0.6773 | 0.7462 |
| Std | 0.0012 | 0.0240 | 0.0119 | 0.0145 | 0.0010 | 0.0074 | 0.0069 | 0.0051 |

(a) CPU faults

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Mean | 0.8801 | 0.2511 | 0.2500 | 0.2355 | 0.8793 | 0.1341 | 0.0044 | 0.0085 |
| Median | 0.8801 | 0.2457 | 0.2500 | 0.2353 | 0.8794 | 0.1299 | 0.0042 | 0.0082 |
| Std | 0.0014 | 0.0180 | 0.0003 | 0.0006 | 0.0015 | 0.0316 | 0.0011 | 0.0022 |

(b) RAM faults

Initially, as for the other scenarios, the RFC was used to evaluate the dataset. Results are shown in Table 8.4.

It is important to note that the dataset is very imbalanced, as the majority of the data is from class 0, which represents the absence of faults. This scenario is also the one with the most variability in the data, complicating the learning process. Therefore, the RFC was not able to learn the patterns of the data.

## 8.2 NN and CNN results

In an attempt to find better results, there were explored two new types of models, NNs and CNNs.

For the NNs, the data was split into 70% for training, 20% for testing and 10% for validation.

The evaluation results of the NNs are presented in Table 8.5. Figure 8.3 shows the evolution of the F1-score and loss for the training and validation sets along the epochs, for the first NN on the multiclass problem for the fog hosts.

The results show that the NNs were able to achieve better results when compared to the RFC. The first 2 models performed close to each other, achieving, for the fog, an F1-score of 60% for the multiclass problem and 80% for the binary problem, and for the edge, an F1-score of 55% for the multiclass problem and 80% for the binary problem.

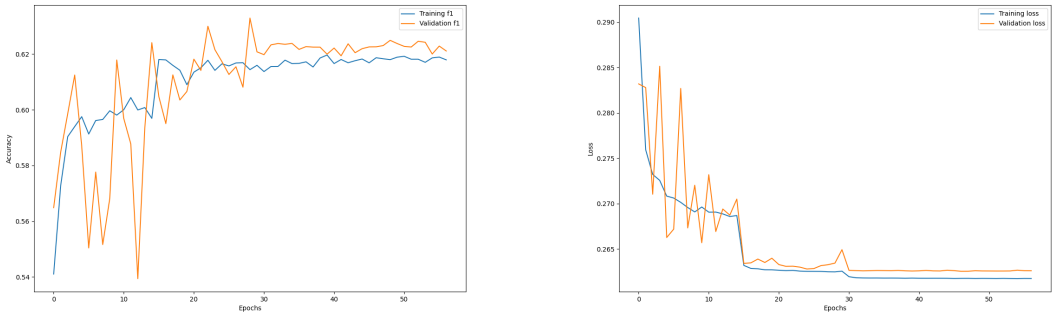
Table 8.2: Mean evaluation results for the fine-tuned versions of RFC in scenario 1.

| **Scoring** | **MULTICLASS** | | | | **BINARY** | | | |
|---|---|---|---|---|---|---|---|---|
| **function** | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| F1 | 0.94 | 0.78 | 0.65 | 0.70 | 0.95 | 0.91 | 0.84 | 0.87 |
| Accuracy | 0.94 | 0.77 | 0.61 | 0.66 | 0.95 | 0.91 | 0.83 | 0.86 |

(a) CPU faults

| **Scoring** | **MULTICLASS** | | | | **BINARY** | | | |
|---|---|---|---|---|---|---|---|---|
| **function** | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| F1 | 0.78 | 0.26 | 0.26 | 0.26 | 0.79 | 0.50 | 0.50 | 0.50 |
| Accuracy | 0.88 | 0.22 | 0.25 | 0.23 | 0.88 | 0.44 | 0.50 | 0.47 |

(b) RAM faults




(a) F1 score

(b) Loss

Figure 8.3: Evolution of the F1 score and loss for NN 1 on the multiclass classification task in fog hosts of scenario 3.

Finally, in order to take advantage of the time series nature of the data, CNNs were trained. The evaluation metrics for this model are shown in Table 8.6.

The results show improvements in the F1-score when compared to the RFC and the NNs. The best-performing model was the CNN using the second approach for the normalisation, which was able to achieve an F1-score of 73% for the multiclass problem and 87% for the binary problem.

Despite all the efforts, the models were not able to achieve better results for the multiclass problem. By taking a look at the confusion matrix of the best-performing model, shown in Figure 8.4, it is possible to see that the model is confusing classes with the closer ones. For example, the model is confusing class 2 with class 1, and class 3.

Table 8.3: Evaluation results for RFC in scenario 2.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Mean | 0.6538 | 0.6490 | 0.6539 | 0.6510 | 0.9058 | 0.9371 | 0.9373 | 0.9372 |
| Median | 0.6538 | 0.6491 | 0.6541 | 0.6511 | 0.9058 | 0.9371 | 0.9372 | 0.9373 |
| Std | 0.0017 | 0.0016 | 0.0016 | 0.0016 | 0.0010 | 0.0011 | 0.0013 | 0.0007 |

(a) CPU faults.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Mean | 0.2654 | 0.2654 | 0.2654 | 0.2654 | 0.7413 | 0.7517 | 0.9783 | 0.8501 |
| Median | 0.2653 | 0.2653 | 0.2653 | 0.2653 | 0.7413 | 0.7512 | 0.9782 | 0.8502 |
| Std | 0.0018 | 0.0018 | 0.0018 | 0.0018 | 0.0014 | 0.0014 | 0.0009 | 0.0009 |

(b) RAM faults.

This led to the following question: Is it a problem with the models or the dataset? To answer this question, the dataset was analysed.

Table 8.4: Evaluation results for RFC in scenario 3.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Mean | 0.8863 | 0.6431 | 0.4897 | 0.5395 | 0.9073 | 0.7635 | 0.5298 | 0.6255 |
| Median | 0.8863 | 0.6433 | 0.4900 | 0.5397 | 0.9075 | 0.7635 | 0.5301 | 0.6257 |
| Std | 0.0004 | 0.0053 | 0.0035 | 0.0041 | 0.0004 | 0.0025 | 0.0026 | 0.0019 |

(a) Edges only.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Mean | 0.8913 | 0.6937 | 0.4968 | 0.5519 | 0.9111 | 0.7793 | 0.5460 | 0.6421 |
| Median | 0.8913 | 0.6922 | 0.4966 | 0.5516 | 0.9111 | 0.7790 | 0.5462 | 0.6421 |
| Std | 0.0007 | 0.0131 | 0.0045 | 0.0058 | 0.0006 | 0.0036 | 0.0033 | 0.0024 |

(b) Fog only.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Mean | 0.8876 | 0.6543 | 0.4924 | 0.5440 | 0.9083 | 0.7668 | 0.5346 | 0.6300 |
| Median | 0.8877 | 0.6539 | 0.4920 | 0.5435 | 0.9083 | 0.7671 | 0.5343 | 0.6300 |
| Std | 0.0004 | 0.0049 | 0.0027 | 0.0031 | 0.0003 | 0.0022 | 0.0019 | 0.0014 |

(c) All the hosts.

Table 8.5: Evaluation results for NNs in scenario 3.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Neural Network 1 | | | | | | | | |
| Fog | 0.8992 | 0.7375 | 0.5408 | 0.6055 | 0.9181 | 0.8775 | 0.7690 | 0.8098 |
| Edge | 0.8935 | 0.7056 | 0.4956 | 0.5559 | 0.9128 | 0.8694 | 0.7529 | 0.7951 |
| Neural Network 2 | | | | | | | | |
| Fog | 0.8995 | 0.7346 | 0.5333 | 0.6007 | 0.9170 | 0.8739 | 0.7639 | 0.8050 |
| Edge | 0.8935 | 0.7387 | 0.4858 | 0.5491 | 0.9135 | 0.8681 | 0.7543 | 0.7959 |
| Neural Network 3 | | | | | | | | |
| Fog | 0.8988 | 0.7312 | 0.5386 | 0.6059 | 0.9183 | 0.8816 | 0.7622 | 0.8058 |
| Edge | 0.8901 | 0.5221 | 0.4660 | 0.4830 | 0.9122 | 0.8544 | 0.7623 | 0.7981 |
| Neural Network 4 | | | | | | | | |
| Fog | 0.8562 | 0.2141 | 0.2500 | 0.2306 | 0.8533 | 0.4267 | 0.5000 | 0.4604 |
| Edge | 0.8924 | 0.7140 | 0.4761 | 0.5433 | 0.8537 | 0.4269 | 0.5000 | 0.4605 |

Table 8.6: Evaluation results for CNNs in scenario 3.

| | MULTICLASS | | | | BINARY | | | |
|---|---|---|---|---|---|---|---|---|
| | **Accuracy** | **Precision** | **Recall** | **F1** | **Accuracy** | **Precision** | **Recall** | **F1** |
| Local Normalisation | | | | | | | | |
| Fog | 0.9040 | 0.7151 | 0.5323 | 0.5936 | 0.9135 | 0.8685 | 0.8009 | 0.8333 |
| Edge | 0.9118 | 0.7730 | 0.5332 | 0.5749 | 0.9302 | 0.8893 | 0.8151 | 0.8464 |
| Global Normalisation | | | | | | | | |
| Fog | 0.9248 | 0.8067 | 0.6570 | 0.7177 | 0.9363 | 0.9084 | 0.8230 | 0.8584 |
| Edge | 0.9305 | 0.8178 | 0.6761 | 0.7330 | 0.9438 | 0.9169 | 0.8485 | 0.8781 |

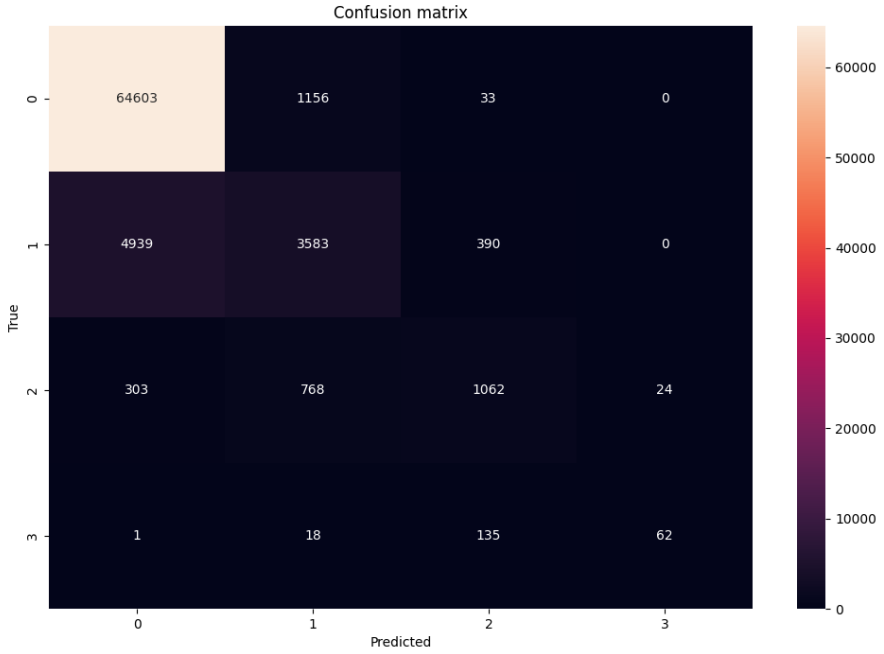
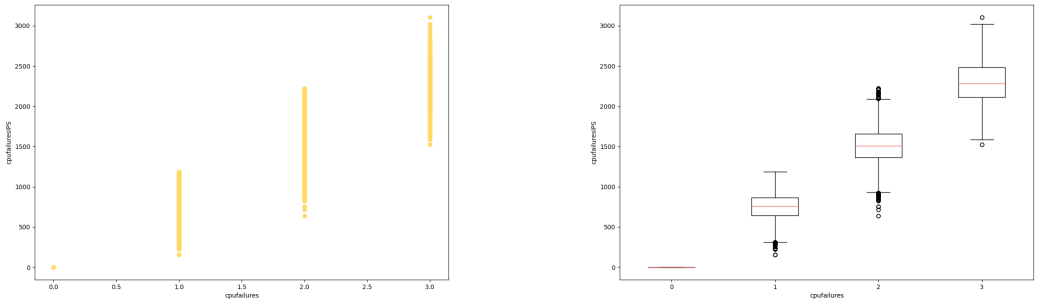



Figure 8.4: Confusion matrix for the CPU fault prediction using the best-performing model in scenario 3.

## 8.3 New Labels

When running the simulations, in order to get the fault intensity labels only the number of phantom containers is considered. However, at this point, the number of phantom containers is not the only factor that influences the fault intensity. Because in complex scenarios the containers have a large range of IPS to be treated, the IPS that each phantom container is assigned to, is also a factor that influences the fault intensity.

Therefore, a new dataset was generated in order to get that value. By plotting the IPS of the phantom containers against the fault intensity, it is possible to see that, there are faults of intensity 1 with a large IPS than some faults of intensity 2, and even of intensity 3. This is shown in Figure 8.5.



(a) Scatter plot

(b) Boxplot

Figure 8.5: Fault intensity for IPS faults in dataset from scenario 3.

Figure 8.6 shows the distribution of the IPS of the phantom containers. By previous analysis of the dataset it was seen that, for this simulation parameters, 79% of the faults were of intensity 1, 19% of intensity 2, and 2% of intensity 3. The vertical lines in the Figure 8.6 represent the 79% and 98% percentiles.

Therefore, the new labels were defined as follows:

- Normal behaviour: IPS of the faults is 0. It is the same as the previous label 0.
- Fault intensity 1: IPS of the phantom containers is below the 79% percentile.
- Fault intensity 2: IPS of the phantom containers is between the 79% and 98% percentiles.
- Fault intensity 3: IPS of the phantom containers is above the 98% percentile.

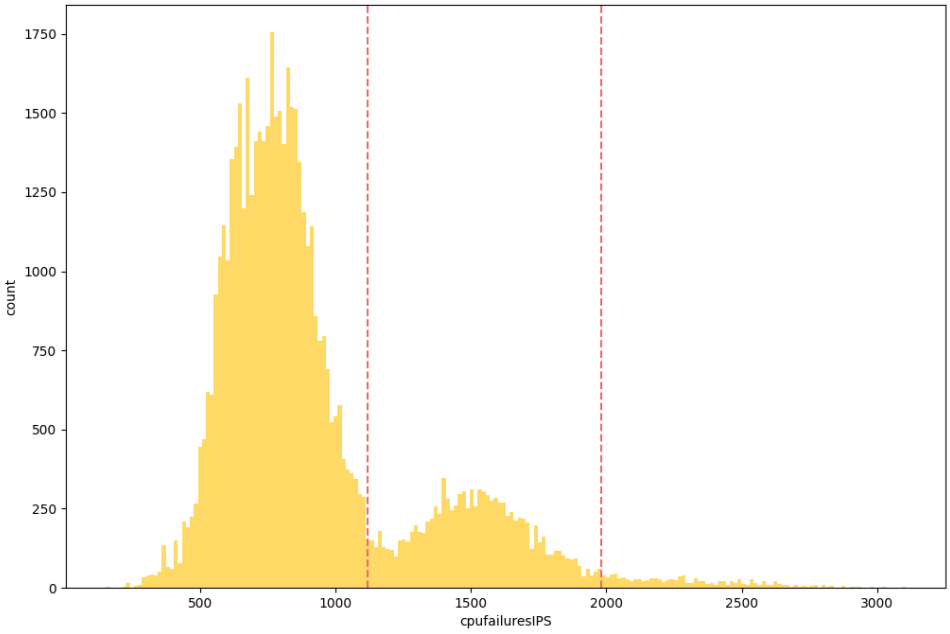



Figure 8.6: Distribution of IPS of faults in dataset from scenario 3.

## 8.4 Final Results

The same models were trained and evaluated with the new dataset. Only the multiclass problem was considered, as it is the one that changed when compared to the previous dataset. Table 8.7 shows the results of the NNs.

The results show that the NNs were able to achieve better results when compared to the previous version, achieving improvements of 2% in the F1-score for the fog and 6% for the edge, however, the results are still not good and the improvement is not that significant.

For the CNNs, the results are shown in Table 8.8.

Since the simulator was modified in order to also give the IPS of faults, and not only the number of phantom containers, the regression problem was also explored. A RFR was trained and evaluated, due to its simplicity and fast training time. The results are shown in Table 8.9.

Despite the bad results from the RFR, it opens the door to new approaches, like the use of other regressors for fault prediction, which was not explored in this work.

Table 8.7: Evaluation results for NNs in scenario 3 with the new labels.

| | **Accuracy** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| Neural Network 1 | | | | |
| Fog | 0.9012 | 0.7266 | 0.5585 | 0.6208 |
| Edge | 0.8940 | 0.7116 | 0.5471 | 0.6066 |
| Neural Network 2 | | | | |
| Fog | 0.8992 | 0.7601 | 0.5582 | 0.6268 |
| Edge | 0.8935 | 0.7106 | 0.5400 | 0.6012 |
| Neural Network 3 | | | | |
| Fog | 0.8991 | 0.7160 | 0.5596 | 0.6188 |
| Edge | 0.8901 | 0.6513 | 0.5947 | 0.6136 |
| Neural Network 4 | | | | |
| Fog | 0.8999 | 0.7423 | 0.5652 | 0.6295 |
| Edge | 0.8918 | 0.7063 | 0.5519 | 0.6101 |

Table 8.8: Evaluation results for CNNs in scenario 3 with the new labels.

| | **Accuracy** | **Precision** | **Recall** | **F1** |
|---|---|---|---|---|
| Local Normalisation | | | | |
| Fog | 0.9036 | 0.7145 | 0.5171 | 0.5782 |
| Edge | 0.9098 | 0.7418 | 0.5506 | 0.5965 |
| Global Normalisation | | | | |
| Fog | 0.9230 | 0.7548 | 0.7228 | 0.7322 |
| Edge | 0.9265 | 0.8159 | 0.6645 | 0.7267 |

## 8.5 Summary

This chapter presented the results of the experiments and discussed them.

The results show that the CNNs were able to achieve the best results, with an F1-score of 70% for the multiclass problem for the CPU faults. For the CNNs, the global normalisation approach was the one that performed the best. However, the results are still not good enough.

The results also show that the RFR was not able to achieve good results for the regression problem, but it opens the door to new approaches, like the use of other regressors for fault prediction.

Table 8.9: MSE results for RFR in scenario 3.

| | Test | | | Train | | |
|---|---|---|---|---|---|---|
| | **Mean** | **Median** | **Std** | **Mean** | **Median** | **Std** |
| Edge | 3971 | 3969 | 21 | 554 | 554 | 2 |
| Fog | 54086 | 54100 | 397 | 7554 | 7557 | 30 |
| All | 16844 | 16861 | 101 | 2352 | 2352 | 9 |

# Chapter 9

# Conclusion

This work has been focused on the development of a self-organising engine for the Cloud-to-Edge continuum that can automate the fault management process.

A simulator was presented and modified to support the proposed solution. SFC were implemented in order to simulate a flow of containers from the edge to the cloud.

Failure detection and mitigation mechanisms were designed and implemented. The failure detection mechanism was based on the CPU and RAM usage of the hosts. The mitigation mechanism was based on the migration of containers to replicas of the host.

The design and implementation of a fault injection mechanism were also presented. It was implemented in the simulator and can inject recurrent or cumulative faults in the RAM or CPU of the hosts.

Three different scenarios were implemented in the simulator, each one with more complexity than the previous one and closer to a real-world scenario.

The modified version is public and available in a public repository. With this version of the simulator, there can be generated datasets for Cloud-to-Edge computing scenarios.

The datasets generated from the implemented scenarios were used to train ML models to predict faults. Three different types of classificators were trained, RFC, NN and CNN. CNN presented the best results, predicting CPU failures with an f1 score of around 90% for binary classification problem and 71% for multi-class classification problem.

Finally, a regressor model was trained to predict the number of IPS that the faults are consuming. The regressor used was RFR which, besides presenting a high mean squared error, opens the door to further exploration of regressor models for fault prediction, especially the ones that take advantage of the time series nature of the data.

## 9.1 Limitations

Throughout the development of this work, several limitations were found and overcome.

The first limitation was the lack of datasets for fault management in Cloud-to-Edge computing scenarios. There are some works that tackle the problem of fault management in Cloud-to-Edge computing, but most of those datasets are not publicly available. This problem was overcome by the development of a simulator that can generate datasets for Cloud-to-Edge computing scenarios. This allowed the implementation of several scenarios and the generation of different datasets for each one of them.

The second limitation was the lack of a simulator that could simulate a Cloud-to-Edge computing scenario. COSCO was the chosen simulator but, despite being able to simulate a Cloud-to-Edge computing scenario, it was not designed for fault management. This limitation was overcome by modifying the simulator to support all the necessary features for fault management. This took a lot of time and effort but, in the end, it was possible to implement all the necessary features and have an in-depth knowledge of the simulator.

Finally, the last limitation was the computational power needed to generate the datasets and train the ML models. The simulator was not designed to be fast, and it took a lot of time to generate the datasets. This problem was overcome by optimising the simulator and the generation of the datasets. Besides that, access to new computational resources also helped to reduce the time needed to generate the datasets and, mainly, to train the ML models by using Graphics Processing Unit (GPU).

## 9.2 Future Work

Regarding the future work, there are several points that can be further explored.

The simulator was a huge part of this work and it can be further improved. Several code optimisations can be done, like the use of thread pools to reduce the time of generation of the datasets. The configuration of the simulator can also be improved, transforming it into a more user-friendly tool.

Regarding the scenarios, they could be more complex. The SFC could be bidirectional instead of unidirectional. It could also go to the sides, communicate with hosts from the same layer, and not only up. Edge nodes could have more than one fog parent. Closer nodes could share replicas therefore reducing the number of replicas needed.

Finally, relating to the ML models. CNNs can be further explored for fault prediction. The use of CNN for fault prediction presented the best results in this work. Pretrained models could be used to improve the results and reduce training time. Other time series models, like LSTM, could also be used.

The use of time series models for CPU and/or RAM usage forecasting could be used to predict a failure and pre-migrate the containers in order to avoid the failure.

There can be way more exploration of regressor models for fault prediction. The use of regressor models for fault prediction was almost not explored in this work and, in the final version, the simulator can give the number of IPS that the faults are consuming. This value could be predicted with regression models.

Finally, the use of reinforcement learning for fault prediction could also be explored, not only for fault prediction but also for other problems like container placement. The big advantage of having a simulator is that it can be used to train reinforcement learning models and, since now the simulator is faster than before, it can be used to train more complex models.

# References


[1] Estefanía Coronado et al. 'Zero Touch Management: A Survey of Network Automation Solutions for 5G and 6G Networks'. In: *IEEE Communications Surveys & Tutorials* 24.4 (2022), pp. 2535–2578. ISSN: 1553-877X. DOI: 10.1109/COMST.2022.3212586.

[2] Shreshth Tuli et al. 'COSCO: Container Orchestration Using Co-Simulation and Gradient Based Optimization for Fog Computing Environments'. In: *IEEE Transactions on Parallel and Distributed Systems* 33.1 (Jan. 2022), pp. 101–116. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3087349.

[3] Kehua Su, Jie Li and Hongbo Fu. 'Smart city and the applications'. In: *2011 International Conference on Electronics, Communications and Control (ICECC)*. Sept. 2011, pp. 1028–1031. DOI: 10.1109/ICECC.2011.6066743.

[4] Jayant Kumar Singh and Amit Kumar Goel. 'Data Security Through Fog Computing Paradigm Using IoT'. en. In: *Proceedings of Academia-Industry Consortium for Data Science*. Ed. by Gaurav Gupta et al. Advances in Intelligent Systems and Computing. Singapore: Springer Nature, 2022, pp. 95–103. ISBN: 9789811668876. DOI: 10.1007/978-981-16-6887-6_9.

[5] ETSI. *ETSI GS NFV 002 V1.1.1 (2013-10)*. Tech. rep. Accessed on 04/11/2022. ETSI, Oct. 2013. URL: https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf.

[6] Raouf Boutaba et al. 'A comprehensive survey on machine learning for networking: evolution, applications and research opportunities'. In: *Journal of Internet Services and Applications* 9.1 (June 2018), p. 16. ISSN: 1869-0238. DOI: 10.1186/s13174-018-0087-2.

[7] Mazeiar Salehie and Ladan Tahvildari. 'Self-adaptive software: Landscape and research challenges'. In: *ACM Transactions on Autonomous and Adaptive Systems* 4.2 (May 2009), 14:1–14:42. ISSN: 1556-4665. DOI: 10.1145/1516533.1516538.

[8] Paul Robertson and Robert Laddaga. 'Model Based Diagnosis and Contexts in Self Adaptive Software'. en. In: *Self-star Properties in Complex Information Systems*. Ed. by Ozalp Babaoglu et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 112–127. ISBN: 978-3-540-32013-5. DOI: 10.1007/11428589_8.

[9] Rogério de Lemos and José Luiz Fiadeiro. 'An architectural support for self-adaptive software for treating faults'. In: *Proceedings of the first workshop on Self-healing systems*. WOSS '02. New York, NY, USA: Association for Computing Machinery, Nov. 2002, pp. 39–42. ISBN: 978-1-58113-609-8. DOI: 10.1145/582128.582136. URL: https://doi.org/10.1145/582128.582136.

[10] Harald Psaier and Schahram Dustdar. 'A survey on self-healing systems: approaches and systems'. en. In: *Computing* 91.1 (Jan. 2011), pp. 43–73. ISSN: 1436-5057. DOI: 10.1007/s00607-010-0107-y.

[11] Debanjan Ghosh et al. 'Self-healing systems — survey and synthesis'. en. In: *Decision Support Systems*. Decision Support Systems in Emerging Economies 42.4 (Jan. 2007), pp. 2164–2185. ISSN: 0167-9236. DOI: 10.1016/j.dss.2006.06.011.

[12] IBM. *What is machine learning?* en-us. Accessed on 04/01/2023. URL: https://www.ibm.com/topics/machine-learning.

[13] IBM. en-us. Accessed on 05/02/2023. URL: https://www.ibm.com/topics/supervised-learning.

[14] Corinna Cortes and Vladimir Vapnik. 'Support-vector networks'. en. In: *Machine Learning* 20.3 (Sept. 1995), pp. 273–297. ISSN: 1573-0565. DOI: 10.1007/BF00994018.

[15] Chandradip Banerjee. *ANOVA and Chi-Square*. Accessed on 15/05/2023. June 2021. URL: https://medium.com/@chandradip93/anova-and-chi-square-aea693c4eb96.

[16] Zhiheng Zhong et al. 'Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions'. In: *ACM Computing Surveys* 54.10s (Sept. 2022), 217:1–217:35. ISSN: 0360-0300. DOI: 10.1145/3510415.

[17] Qingfeng Du, Tiandi Xie and Yu He. 'Anomaly Detection and Diagnosis for Container-Based Microservices with Performance Monitoring'. en. In: *Algorithms and Architectures for Parallel Processing*. Ed. by Jaideep Vaidya and Jin Li. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 560–572. ISBN: 978-3-030-05063-4. DOI: 10.1007/978-3-030-05063-4_42.

[18] Tao Zhang, Kun Zhu and Ekram Hossain. 'Data-Driven Machine Learning Techniques for Self-Healing in Cellular Wireless Networks: Challenges and Solutions'. en. In: *Intelligent Computing* 2022 (Sept. 2022), pp. 1–8. ISSN: 2771-5892. DOI: 10.34133/2022/9758169.

[19] C.S. Hood and Chuanyi Ji. 'Proactive network fault detection'. In: *Proceedings of INFOCOM '97*. Vol. 3. Apr. 1997, 1147–1155 vol.3. DOI: 10.1109/INFCOM.1997.631137.

[20] O.P. Kogeda, J.I. Agbinya and C.W. Omlin. 'A Probabilistic Approach To Faults Prediction in Cellular Networks'. In: *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL'06)*. Apr. 2006, pp. 130–130. DOI: 10.1109/ICNICONSMCL.2006.29.

[21] Okuthe Kogeda and Johnson Agbinya. *Prediction of Faults in Cellular Networks Using Bayesian Network Model*. UTS ePress, Jan. 2006. ISBN: 978-0-9775200-0-8. URL: https://opus.lib.uts.edu.au/handle/10453/2943.

[22] Jianguo Ding et al. 'Predictive fault management in the dynamic environment of IP networks'. In: *2004 IEEE International Workshop on IP Operations and Management*. Oct. 2004, pp. 233–239. DOI: 10.1109/IPOM.2004.1547622.

[23] Yong Wang, Margaret Martonosi and Li-Shiuan Peh. 'Predicting link quality using supervised learning in wireless sensor networks'. In: *ACM SIGMOBILE Mobile Computing and Communications Review* 11.3 (July 2007), pp. 71–83. ISSN: 1559-1662. DOI: 10.1145/1317425.1317434.

[24] Xu Lu et al. 'Using Hessian Locally Linear Embedding for autonomic failure prediction'. In: *2009 World Congress on Nature & Biologically Inspired Computing (NaBIC)*. Dec. 2009, pp. 772–776. DOI: 10.1109/NABIC.2009.5393880.

[25] Alessandro Pellegrini, Pierangelo Di Sanzo and Dimiter R. Avresky. 'A Machine Learning-Based Framework for Building Application Failure Prediction Models'. In: *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. May 2015, pp. 1072–1081. DOI: 10.1109/IPDPSW.2015.110.

[26] Zhilong Wang et al. 'Failure prediction using machine learning and time series in optical network'. EN. In: *Optics Express* 25.16 (Aug. 2017), pp. 18553–18565. ISSN: 1094-4087. DOI: 10.1364/OE.25.018553.

[27] Yash Kumar, Hasan Farooq and Ali Imran. 'Fault prediction and reliability analysis in a real cellular network'. In: *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*. June 2017, pp. 1090–1095. DOI: 10.1109/IWCMC.2017.7986437.

[28] R.A. Maxion. 'Anomaly detection for diagnosis'. In: *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*. June 1990, pp. 20–27. DOI: 10.1109/FTCS.1990.89362.

[29] Sudarshan Rao. 'Operational Fault Detection in cellular wireless base-stations'. In: *IEEE Transactions on Network and Service Management* 3.2 (Apr. 2006), pp. 1–11. ISSN: 1932-4537. DOI: 10.1109/TNSM.2006.4798311.

[30] J.S. Baras et al. 'Automated network fault management'. In: *MILCOM 97 MILCOM 97 Proceedings*. Vol. 3. Nov. 1997, 1244–1250 vol.3. DOI: 10.1109/MILCOM.1997.644967.

[31] Karwan Qader, Mo Adda and Mouhammd Al-Kasassbeh. 'Comparative Analysis of Clustering Techniques in Network Traffic Faults Classification'. en. In: *International Journal of Innovative Research in Computer and Communication Engineering* 5.4 (2007), p. 13.

[32] Azzam I. Moustapha and Rastko R. Selmic. 'Wireless Sensor Network Modeling Using Modified Recurrent Neural Networks: Application to Fault Detection'. In: *IEEE Transactions on Instrumentation and Measurement* 57.5 (May 2008), pp. 981–988. ISSN: 1557-9662. DOI: 10.1109/TIM.2007.913803.

[33] H. Hajji. 'Statistical analysis of network traffic for adaptive faults detection'. In: *IEEE Transactions on Neural Networks* 16.5 (Sept. 2005), pp. 1053–1063. ISSN: 1941-0093. DOI: 10.1109/TNN.2005.853414.

[34] Umair Sajid Hashmi, Arsalan Darbandi and Ali Imran. 'Enabling proactive self-healing by data mining network failure logs'. en. In: *2017 International Conference on Computing, Networking and Communications (ICNC)*. Silicon Valley, CA, USA: IEEE, Jan. 2017, pp. 511–517. ISBN: 978-1-5090-4588-4. DOI: 10.1109/ICCNC.2017.7876181. URL: http://ieeexplore.ieee.org/document/7876181/.

[35] Karamjeet Kaur, Veenu Mangat and Krishan Kumar. 'A comprehensive survey of service function chain provisioning approaches in SDN and NFV architecture'. en. In: *Computer Science Review* 38 (Nov. 2020), p. 100298. ISSN: 1574-0137. DOI: 10.1016/j.cosrev.2020.100298.

[36] Sandra Herker et al. 'Data-Center Architecture Impacts on Virtualized Network Functions Service Chain Embedding with High Availability Requirements'. In: *2015 IEEE Globecom Workshops (GC Wkshps)*. Dec. 2015, pp. 1–7. DOI: 10.1109/GLOCOMW.2015.7414158.

[37] Ahmed M. Medhat et al. 'Resilient orchestration of Service Functions Chains in a NFV environment'. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. 2016, pp. 7–12. DOI: 10.1109/NFV-SDN.2016.7919468.

[38] Karthik Karra and Krishna M. Sivalingam. 'Providing Resiliency for Service Function Chaining in NFV systems using a SDN-based approach'. In: *2018 Twenty Fourth National Conference on Communications (NCC)*. Feb. 2018, pp. 1–6. DOI: 10.1109/NCC.2018.8600121.

[39] Isaac Lera, Carlos Guerrero and Carlos Juiz. 'YAFS: A Simulator for IoT Scenarios in Fog Computing'. In: *IEEE Access* 7 (2019), pp. 91745–91758. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2927895.

[40] rishabv90. *B-series burstable - Azure Virtual Machines*. en-us. Accessed on 24/07/2023. Sept. 2022. URL: https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable (visited on 25/07/2023).

[41] Corey Sanders. *Introducing B-Series, our new burstable VM size | Azure Blog | Microsoft Azure*. en-US. Sept. 2017. URL: https://azure.microsoft.com/en-us/blog/introducing-b-series-our-new-burstable-vm-size/.

[42] Accessed on 15/08/2023. URL: https://archive.vn/20130205075133/http://www.tomshardware.com/charts/cpu-charts-2004/Sandra-CPU-Dhrystone,449.html.

[43] Python. Accessed on 24/07/2023. July 2023. URL: https://github.com/imperial-qore/COSCO.

[44] Siqi Shen, Vincent van Beek and Alexandra Iosup. 'Statistical characterization of business-critical workloads hosted in cloud datacenters'. In: *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. CCGRID '15. Shenzhen, China: IEEE Press, 2015, pp. 465–474. ISBN: 978-1-4799-8006-2. DOI: 10.1109/CCGrid.2015.60. URL: https://dl.acm.org/doi/10.1109/CCGrid.2015.60.

[45] Accessed on 16/08/2023. URL: http://gwa.ewi.tudelft.nl/datasets/gwa-t-12-bitbrains.

[46] Eli Cortez et al. 'Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms'. en. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. Shanghai China: ACM, Oct. 2017, pp. 153–167. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132772. URL: https://dl.acm.org/doi/10.1145/3132747.3132772.

[47] Accessed on 16/08/2023. Aug. 2023. URL: https://github.com/Azure/AzurePublicDataset.

[48] Accessed on 17/08/2023. URL: https://manpages.ubuntu.com/manpages/xenial/man1/stress-ng.1.html.

[49] Qingfeng Du et al. 'An Approach of Collecting Performance Anomaly Dataset for NFV Infrastructure'. en. In: *Algorithms and Architectures for Parallel Processing*. Ed. by Jaideep Vaidya and Jin Li. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 59–71. ISBN: 978-3-030-05057-3. DOI: 10.1007/978-3-030-05057-3_5.

[50] Carla Sauvanaud et al. 'Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned'. In: *Journal of Systems and Software* 139 (May 2018), pp. 84–106. ISSN: 0164-1212. DOI: 10.1016/j.jss.2018.01.039.

[51] Carla Sauvanaud et al. 'Anomaly Detection and Root Cause Localization in Virtual Network Functions'. In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. Oct. 2016, pp. 196–206. DOI: 10.1109/ISSRE.2016.32.

[52] Mbarka Soualhia, Chunyan Fu and Foutse Khomh. 'Infrastructure fault detection and prediction in edge cloud environments'. In: *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. SEC '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 222–235. ISBN: 978-1-4503-6733-2. DOI: 10.1145/3318216.3363305. URL: https://doi.org/10.1145/3318216.3363305.

# Appendices

# Appendix A

# Exploratory Data Analysis

## A.1 Scenario 1: Linear Chaining

Table A.1: Description of the dataset for scenario 1.

| | CPU usage | Number of containers | Base IPS | IPS available | IPS cap | Apparent IPS | Fault intensity |
|---|---|---|---|---|---|---|---|
| count | 90090 | 90090 | 90090 | 90090 | 90090 | 90090 | 90090 |
| mean | 17.6 | 7.3 | 755.2 | 8659.0 | 9414.0 | 1431.0 | 0.1 |
| std | 8.1 | 2.5 | 406.0 | 4829.0 | 5019.0 | 686.7 | 0.4 |
| min | 0.0 | 0.0 | 0.0 | 2249.4 | 4029.0 | 0.0 | 0.0 |
| 25% | 11.8 | 6.0 | 458.9 | 3675.2 | 4029.0 | 924.0 | 0.0 |
| 50% | 16.2 | 7.0 | 682.4 | 7397.4 | 8102.0 | 1304.0 | 0.0 |
| 75% | 22.2 | 9.0 | 980.0 | 14807.9 | 16111.0 | 1825.0 | 0.0 |
| max | 65.1 | 20.0 | 3440.0 | 16111.0 | 16111.0 | 5678.0 | 3.0 |

(a) CPU metrics.

| | RAM usage | Number of containers | RAM size | RAM read | RAM write | Available RAM size | Available RAM read | Available RAM write | Fault intensity |
|---|---|---|---|---|---|---|---|---|---|
| count | 90090 | 90090 | 90090 | 90090 | 90090 | 90090 | 90090 | 90090 | 90090 |
| mean | 4. | 7.3 | 506.5 | 1.4 | 1.1 | 18105.2 | 368.1 | 256.1 | 0.1 |
| std | 3.7 | 2.5 | 502.9 | 4.8 | 4.6 | 12171.2 | 8.2 | 43.1 | 0.4 |
| min | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 2852.3 | 290.0 | 197.1 | 0.0 |
| 25% | 1.2 | 6.0 | 186.8 | 0.0 | 0.0 | 4129.3 | 360.0 | 200.0 | 0.0 |
| 50% | 2.9 | 7.0 | 333.4 | 0.0 | 0.0 | 16799.5 | 371.9 | 266.7 | 0.0 |
| 75% | 5.7 | 9.0 | 604.7 | 0.2 | 0.2 | 33291.6 | 375.5 | 304.7 | 0.0 |
| max | 33.6 | 20.0 | 4966.6 | 74.0 | 69.6 | 34360.0 | 376.5s | 305.0 | 3.0 |

(b) RAM metrics.

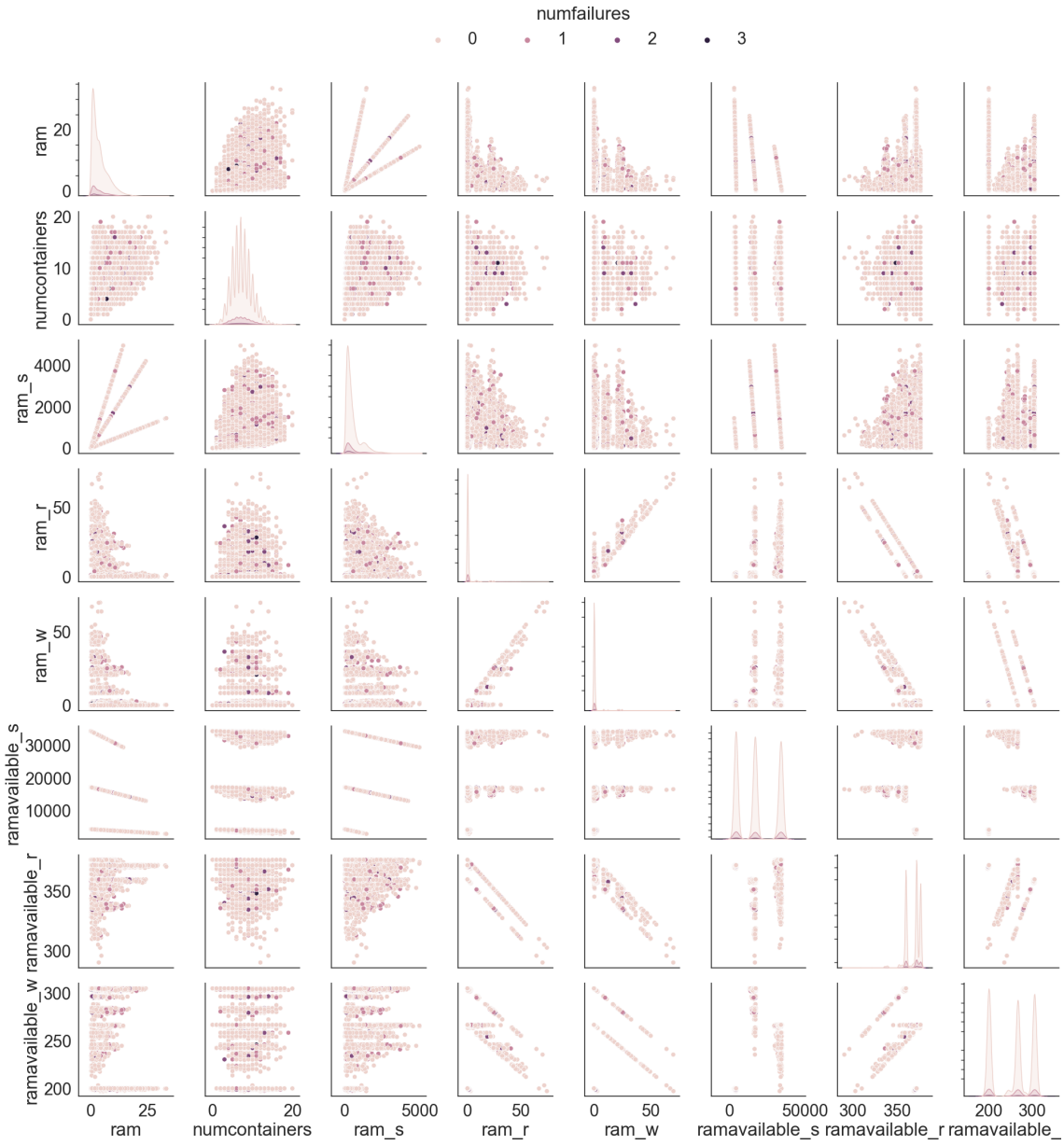



Figure A.1: Pairplot of the RAM metrics collected from the hosts in scenario 1.

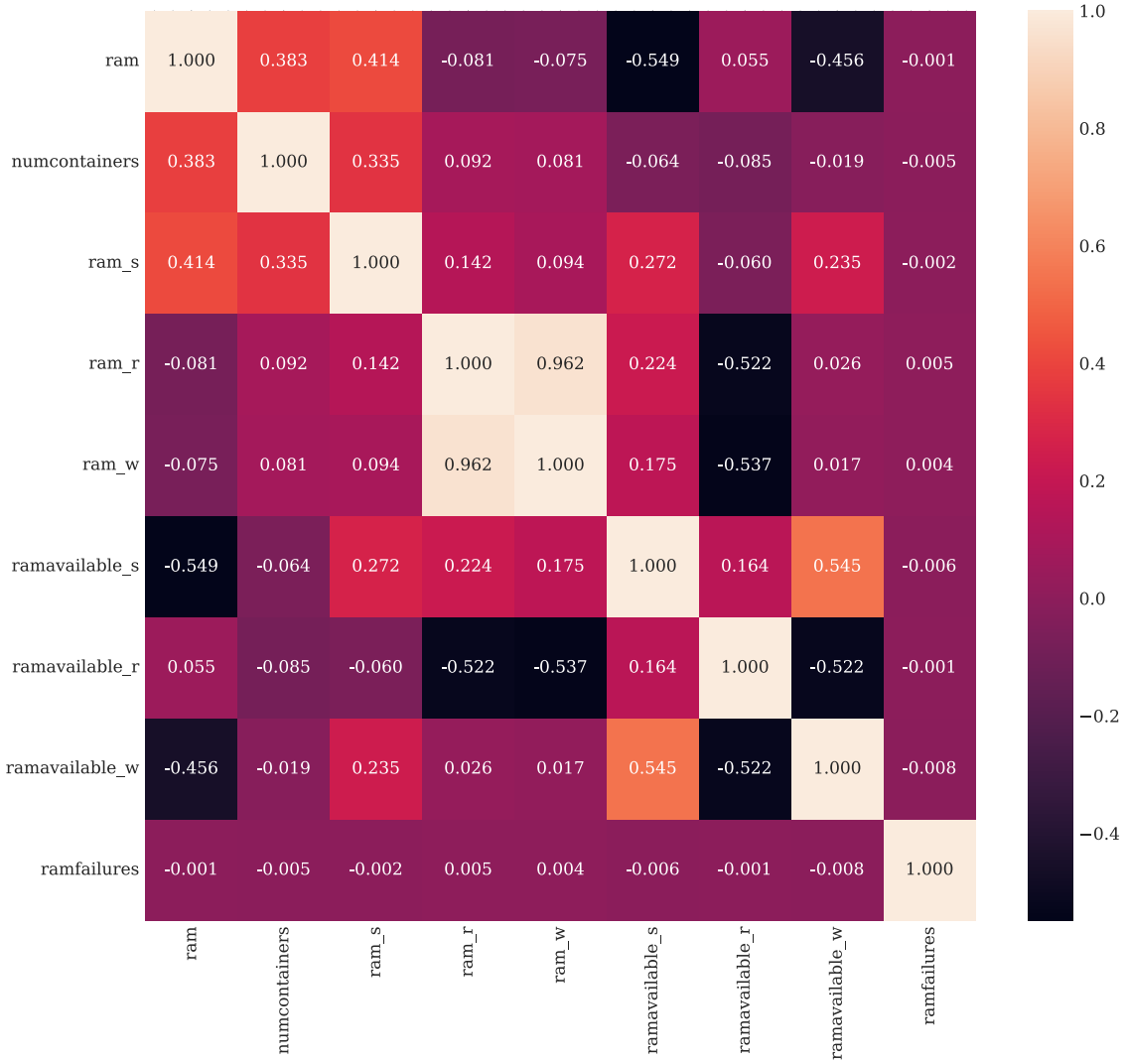



Figure A.2: Correlation matrix of the RAM metrics collected from the hosts in scenario 1.

## A.2 Scenario 2: Balanced Fixed Tree

Table A.2: Description of the balanced dataset for scenario 2.

| | CPU usage | Number of containers | Base IPS | IPS available | IPS cap | Apparent IPS | Fault intensity |
|---|---|---|---|---|---|---|---|
| count | 210210 | 210210 | 210210 | 210210 | 210210 | 210210 | 210210 |
| mean | 36.3 | 12.4 | 2613.2 | 10721.5 | 13334.7 | 5730.4 | 1.5 |
| std | 14.3 | 8.0 | 3694.1 | 10307.5 | 13603.2 | 7473.2 | 1.1 |
| min | 0.0 | 0.0 | 0.0 | 803.0 | 2049.0 | 0.0 | 0.0 |
| 25% | 25.8 | 7.0 | 309.6 | 2513.4 | 2848.0 | 862.0 | 0.0 |
| 50% | 34.9 | 10.0 | 526.4 | 3433.3 | 3782.5 | 1317.0 | 1.0 |
| 75% | 45.8 | 15.0 | 3356.7 | 18182.8 | 21299.0 | 7691.0 | 2.0 |
| max | 100.0 | 52.0 | 22736.8 | 40960.0 | 40960.0 | 39800.0 | 3.0 |

(a) CPU metrics.

| | RAM usage | Number of containers | RAM size | RAM read | RAM write | Available RAM size | Available RAM read | Available RAM write | Fault intensity |
|---|---|---|---|---|---|---|---|---|---|
| count | 210210 | 210210 | 210210 | 210210 | 210210 | 210210 | 210210 | 210210 | 210210 |
| mean | 7.1 | 12.4 | 1696.4 | 3.0 | 1.2 | 23618.6 | 366.2 | 238.3 | 1.5 |
| std | 4.2 | 8.0 | 2651.7 | 9.1 | 3.8 | 26449.2 | 11.0 | 46.1 | 1.1 |
| min | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1212.8 | 231.2 | 188.0 | 0.0 |
| 25% | 4.0 | 7.0 | 200.6 | 0.0 | 0.0 | 2654.0 | 359.9 | 200.0 | 0.0 |
| 50% | 6.5 | 10.0 | 360.5 | 0.1 | 0.1 | 3624.2 | 371.9 | 200.0 | 1.0 |
| 75% | 9.5 | 15.0 | 2001.0 | 0.6 | 0.4 | 42049.6 | 372.0 | 299.8 | 2.0 |
| max | 43.8 | 52.0 | 19598.2 | 145.4 | 78.7 | 81920.0 | 376.5 | 305.0 | 3.0 |

(b) RAM metrics.

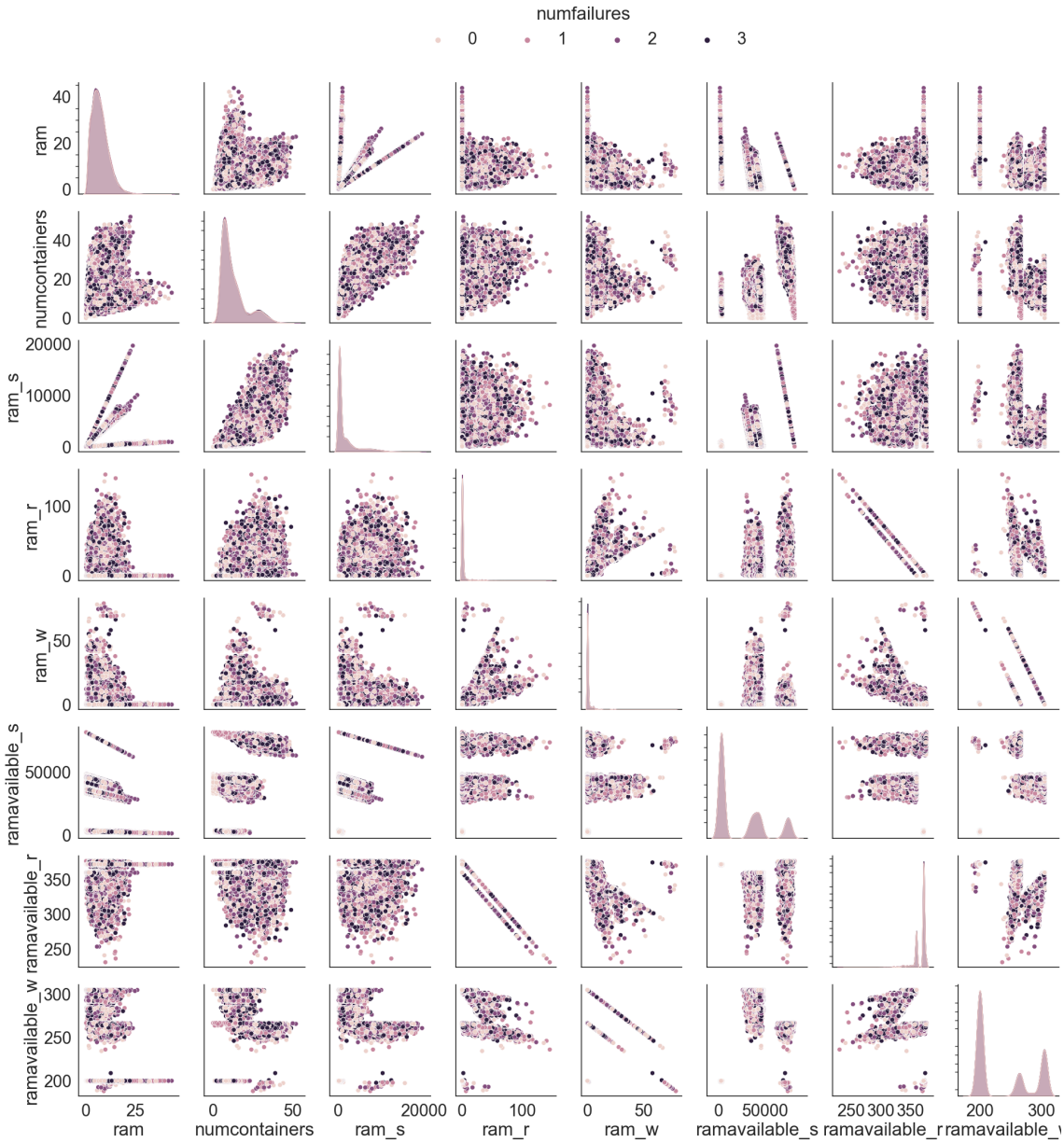



Figure A.3: Pairplot of the RAM metrics collected from the hosts in scenario 2.


Figure A.4: Correlation matrix of the CPU metrics collected from the hosts in scenario 1.

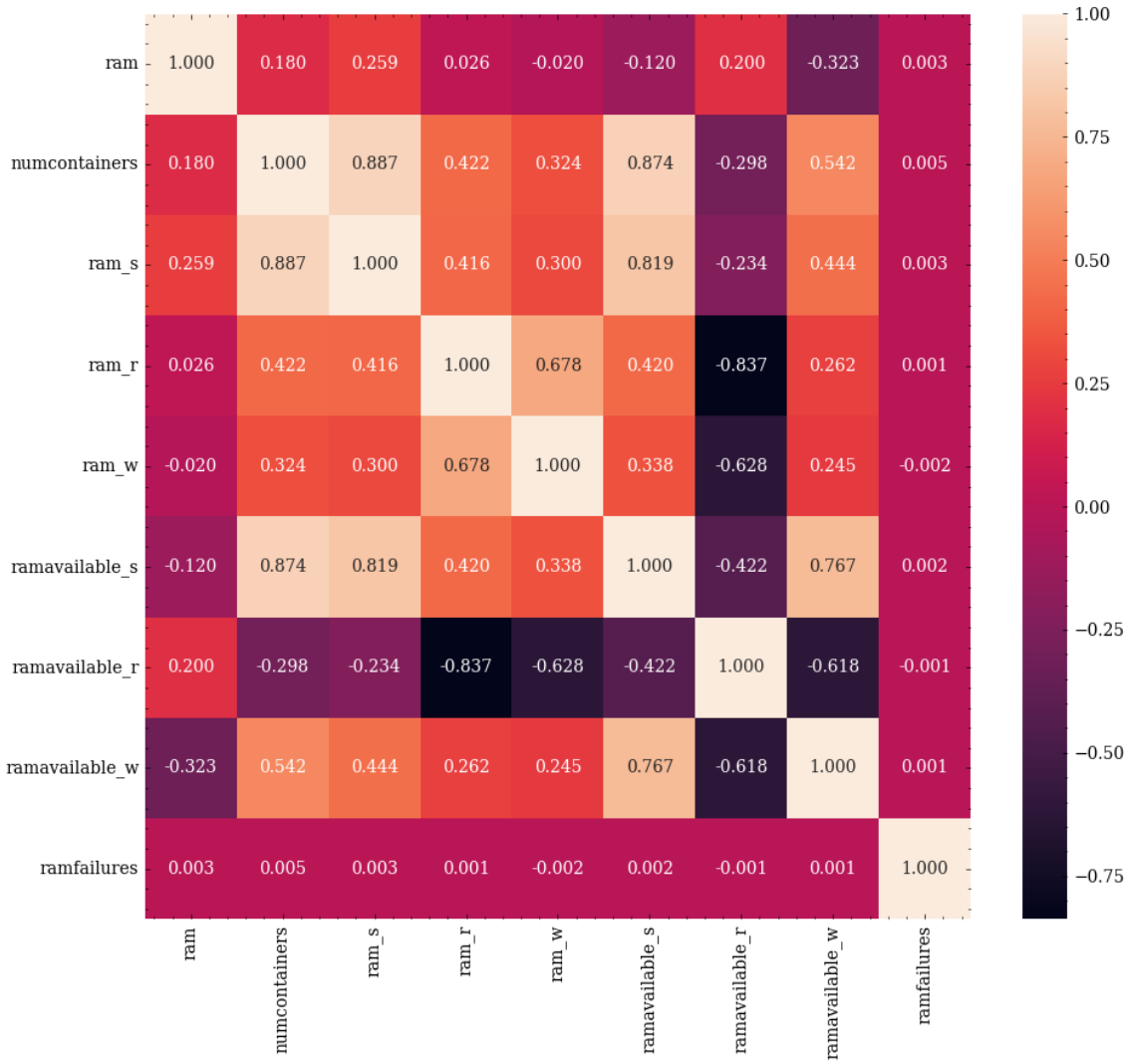



Figure A.5: Correlation matrix of the RAM metrics collected from the hosts in scenario 1.

Table A.3: Feature importance for CPU faults in scenario 2.

| | **Select K Best** | | **Feature Importance** |
|---|---|---|---|
| | **ANOVA** | **CHI2** | **Extra Tree Classifier** |
| cpu | **20881.14** | **2.69e+05** | **0.210** |
| apparentips | **678.92** | **1.96e+07** | **0.245** |
| numcontainers | 2.24 | 3.46e+01 | **0.145** |
| baseips | 1.62 | 2.53e+04 | 0.144 |
| ipscap | 0.37 | 1.55e+04 | 0.093 |
| ipsavailable | 0.30 | 9.03e+03 | 0.142 |

# Appendix B

# AI Models

```
## NN 1 - smaller
model1 = tf.keras.models.Sequential(
    [
        tf.keras.layers.Dense(
            64, activation="relu", input_shape=(X_train.shape[1],)
        ),  # 6 features
        tf.keras.layers.Dense(64, activation="relu"),
        tf.keras.layers.Dense(
            len(y.unique()), activation="softmax"
        ),  # 4 or 2 classes
    ]
)

## NN 2 - bigger
model2 = tf.keras.models.Sequential(
    [
        tf.keras.layers.Dense(
            32, activation="relu", input_shape=(X_train.shape[1],)
        ),  # 6 features
        tf.keras.layers.Dense(64, activation="relu"),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dense(256, activation="relu"),
        tf.keras.layers.Dense(256, activation="relu"),
        tf.keras.layers.Dense(32, activation="relu"),
        tf.keras.layers.Dense(
            len(y.unique()), activation="softmax"
        ),  # 4 or 2 classes
    ]
)
```

```
## NN 3 - even larger
model3 = tf.keras.models.Sequential(
    [   tf.keras.layers.Dense(
            16*6, activation="relu", input_shape=(X_train.shape[1],)
        ),  # 6 features
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(64*6, activation="relu"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(128*6, activation="relu"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(32*6, activation="relu"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Dense(64, activation="relu"),
        tf.keras.layers.Dense(16, activation="relu"),
        tf.keras.layers.Dropout(0.2),
        tf.keras.layers.Dense(
            len(y.unique()), activation="softmax"
        ),  # 4 or 2 classes
    ]
)
## NN 4 - the largest
model4 = tf.keras.models.Sequential(
    [   tf.keras.layers.Dense(
            32, activation="relu", input_shape=(X_train.shape[1],)
        ),  # 6 features
        tf.keras.layers.Dense(64, activation="relu"),
        tf.keras.layers.Dense(128, activation="relu"),
        tf.keras.layers.Dense(256, activation="relu"),
        tf.keras.layers.Dense(256, activation="relu"),
        tf.keras.layers.Dense(512, activation="relu"),
        tf.keras.layers.Dense(512, activation="relu"),
        tf.keras.layers.Dense(1024, activation="relu"),
        tf.keras.layers.Dense(1024, activation="relu"),
        tf.keras.layers.Dense(256, activation="relu"),
        tf.keras.layers.Dense(64, activation="relu"),
        tf.keras.layers.Dense(16, activation="relu"),
        tf.keras.layers.Dense(
            len(y.unique()), activation="softmax"
        ),  # 4 or 2 classes
    ]
)
```

Listing B.1: Architecture of the Neural Networks used in scenario 3.