João Pedro Simões Lopes

Online Failure Prediction in Containerized Environments

Dissertation Report
Master in Informatics Engineering
advised by Professor Dr. Nuno Antunes
and presented to the Department Informatics Engineering
of the Faculty of Sciences and Technology of the University of Coimbra

September 2018

• U C •

Universidade de Coimbra

Master in Informatics Engineering

Dissertation

Online Failure Prediction in Containerized Environments

Author:
João Pedro Simões Lopes
jplopes@student.dei.uc.pt

Supervisor:
Prof. Dr. Nuno Antunes
DEI–UC

Jury:
Prof. Dr. Marília Curado
DEI–UC

Prof. Dr. Mário Rela
DEI–UC

Date: 3rd September, 2018

FCTUC DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

This work is within the software engineering specialization area and was carried out in the Software and Systems Engineering (SSE) Group of the Centre for Informatics and Systems of the University of Coimbra (CISUC).

This work was partially supported by the project ATMOSPHERE, funded by the Brazilian Ministry of Science, Technology and Innovation (51119 - MCTI/RNP 4th Coordinated Call) and by the European Commission under the Cooperation Programme, H2020 grant agreement no 777154.

It is also partially supported by the project METRICS (POCI-01-0145-FEDER-032504), funded by the Portuguese Foundation for Science and Technology (FCT) through Programa Operacional Competitividade e Internacionalização - COMPETE 2020.

This work has been supervised by Professor Nuno Manuel dos Santos Antunes, Assistant Professor at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

This page is intentionally left blank.

Acknowledgements

I would like to express my gratitude to Prof. Nuno Antunes for his helpful guidance and for all the support which help to complete this work.

I would like to thank Gonçalo Pereira and his advisors Prof. Henrique Madeira and Prof. Raúl Barbosa for allowing the use of the fault injection tool *BugTor* in this work.

I would also like to thank João R. Campos for allowing the use of the machine learning tool *Propheticus* in this work.

I would also like to thank my colleagues and friends for all the support.

I would like to express my deep gratitude to my family for their support and patience, which always believed in me and made this all possible.

This page is intentionally left blank.

Abstract

Online Failure Prediction is very promising, as if it is possible to foresee the occurrence of failures, their consequences can be avoided or mitigated. Training prediction models requires failure-related data, which is scarce. Fault Injection has been used to create this data as the system evolves and to select an adequate Failure Prediction approach. However, existing approaches are hard to apply in practice: **systems are very complex with undefined boundaries, and the models created are very sensitive** to changes in the system. Containers promise isolation, stability across executions and boundaries defined by nature. Micro-service applications based in containers have the adequate characteristics to make Online Failure Prediction applicable.

The objective of this work is to **assess the feasibility of using Online Failure Prediction in containerized micro-services-based applications**, contributing to the applicability of the technique in this domain. For this, we started by analyzing different alternatives to monitor container-related metrics. It is possible to gather several metrics in a non-intrusive way, although these metrics differ from the ones used in past Online Failure Prediction experiments. A Docker setup with different hardware configurations was used to understand which of these metrics are relevant, and how they vary across similar executions in different setups. Next, Fault Injection was used to produce failure data in container's environment, for training purposes. Finally, we evaluated how effective the chosen Failure Prediction approach was in failures originated by the injected faults. The results show that it is possible to generate failure data, although data across different setups diverge significantly. They also show that the failure predictions approach are not effective when the configurations and parameters are not carefully selected for the application domain.

Keywords

Containers, Monitoring container variables, Online failure prediction, OS-level virtualization, Software fault injection.

This page is intentionally left blank.

Resumo

A previsão de avarias durante a execução é uma técnica promissora pois prevê a ocorrência de falhas, evitando ou mitigando as suas consequências. Treinar modelos de previsão necessita de dados de falhas, os quais são escassos. A injeção de falhas foi usada para criar esses dados à medida que o sistema evolui e para selecionar uma abordagem de previsão de avarias adequada. Contudo, as abordagens existentes são difíceis de aplicar na prática: **os sistemas são complexos com limites indefinidos e os modelos criados são sensíveis** às mudanças no sistema. Os contentores prometem isolamento, estabilidade e os limites são definidos pela sua natureza. As aplicações de micro-serviços baseadas em contentores possuem as características para tornar a previsão de avarias durante a execução aplicável.

O objetivo deste trabalho é **avaliar a viabilidade de usar previsão de avarias durante a execução em aplicações de micro-serviços baseadas em contentores**, contribuindo para a sua aplicabilidade neste domínio. Para isso, começámos por analisar alternativas para monitorizar métricas dos contentores. É possível recolher métricas não intrusivamente, embora estas sejam diferentes das usadas em experiências anteriores de previsão de avarias durante a execução. Uma configuração de Docker com diferentes configurações de hardware foi usada para entender quais dessas métricas são relevantes e como elas variam em execuções similares. Em seguida, usámos injeção de falhas para produzir dados de falhas nos contentores, com o fim de treinar. Finalmente, avaliámos a eficácia da abordagem de previsão de avarias escolhida em avarias originadas pelas falhas injetadas. Os resultados mostram que é possível gerar dados de avarias, embora os dados em diferentes configurações divirjam significativamente. Eles também mostram que a abordagem de previsão de falhas não é eficaz quando as configurações e os parâmetros não são cuidadosamente selecionados para o domínio da aplicação.

Palavras-Chave

Contentores, Injeção de falhas de software, Monitorização de variáveis dos contentores, Previsão de avarias durante a execução, Virtualização ao nível do SO.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

Acronyms

DEI Department of Informatics Engineering. 32, 33

FI Fault Injection. 2, 3, 5, 10, 24, 25, 31, 32, 35, 40–42, 44, 45, 47–49, 58

FIR Fault Injection Runs. 47

FP Failure Prediction. 1–5, 7, 9, 10, 24, 25, 33, 34, 39, 47, 48, 52–55, 57, 58

IT Information Technology. 21

ML Machine Learning. 3, 13–17, 33, 47, 50, 51, 53

OFP Online Failure Prediction. 1, 2, 5, 7, 8, 23, 24, 57

OS Operating System. 1, 11, 12, 17, 19, 21, 27, 29, 35

RAM Random Access Memory. 17

RBF Radial Basis Function. 49–52

RFE Recursive Feature Elimination. 14, 33, 49, 52

ROC Receiver Operating Characteristics. xiii, 10, 16, 51, 53

SMOTE Synthetic Minority Over-sampling Technique. 14, 33, 49–51

SoS Systems of Systems. 1

SVM Support Vector Machine. xiii, 15, 33, 34, 49–52, 55, 57

SWFI Software Fault Injection. 3

UC University of Coimbra. 12, 32, 33

UI User Interface. 21

VM Virtual Machine. xiii, 1, 17, 55

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

Failure Prediction (FP) is a very enticing proposition, as it would be very beneficial to avoid or mitigate the consequences of failures if we were able to foresee their occurrence with enough time to act. In practice, this technique only predicts failures using past failure data, where its main limitation is that it does not use any information about the system's current state [1]. **Online Failure Prediction (OFP)** aims at overcoming these limitations. The term "online" means that this technique can predict failures based in the information about the system's current state [2]. Thus, OFP methods monitor the target system at runtime, using both past failure and present data, where the present data is composed by the target system's current state information [2]. This technique allows the mitigation or avoidance of the consequences of the failures that occur in a given system.

OFP is hard to be implemented and is not applicable to complex systems in practice. It is known also that systems change and their complexity increases over time, and failures are likely to occur. Additionaly, the number of lines of code is increasing a lot and the systems have started to use external services to interact with other systems. Nowadays, we even have Systems of Systems (SoS), which consist in "large-scale integrated systems which are heterogeneous and independently operable on their own, but are networked together for a common goal" [3], being that they are systems with a high complexity. Virtual Machines (VMs) are a possible solution to improve the Online Failure Prediction's applicability, however they are not the ideal solution because their system's boundaries are unclear, VMs are a heavy virtualization solution and can also be very complex systems [4].

On the other end of the spectrum, containers are lightweight systems which have high portability and stability in the application context, isolation and have well-defined boundaries [4]. Thus, they provide an uniform context to the applications when executing in different Operating Systems (OSes) and hardware configurations, isolating the applications from their surroundings [4]. Containers are widely spread, as in the case of cloud environments. Their characteristics and composition allow them to be simple and easy to manage and replicate. Containers and VMs are identical at resource isolation and allocation-benefit levels, however the containers virtualize the OS (OS-level virtualization) while the VMs virtualize the hardware [4].

Containerized applications based on micro-services have the necessary and adequate characteristics to make Online Failure Prediction applicable in practice, because they are highly flexible and scalable [5]. Since Docker's release, users have started to massively adopt and use containers [6]. Thus, it would be possible to predict failures more simply in this kind of systems and applications, which could alert the user about

upcoming failures or even avoid them, maintaining the continuous delivery of the expected service without deviation.

Additionally, it would be useful if every user and company could be able to predict failures in a straightforward way, by taking into account the current state of the system as well as past failure data collected during the runtime. This way, it could be more precise in alerting the users when the OFP algorithm detects that something is going wrong, i.e. the behavior of the system is not the expected. Then, the system would be able to understand and detect when a given failure will occur, making it possible to mitigate the consequences from its occurrence at the present time.

In order to speed up the process of obtaining failure data to train and evaluate the Failure Prediction models, Fault Injection (FI) can be used. This is a well-known technique which has the goal of inserting faults in a given system to simulate the effect of real faults [7]. FI allows accelerating the training and validation of the models, thus avoiding taking weeks or even months. This training is necessary because it will allow to perform a more accurate failure prediction, since it already learned the required data and information in order to predict if a given failure is about to arise. Additionally, the number of false positives and false negatives associated with the application of FP algorithms will be lower, which means more rigorous and precise results.

The process of monitoring containers is challenging, not only because of the high portability of this virtual environment, but also due to the ability to run micro-service applications using a large number of containers, numbering tens or hundreds [8]. Thus, it is necessary to carefully choose the metrics that will be used in order to monitor containerized applications. Additionally, a monitoring tool will be chosen in order to collect that information from the containers. This way, it will be possible to get sensible and important information about the execution of this type of systems.

Therefore, **the main goal of this work is to evaluate the feasibility of the application of online failure prediction in containerized environments**, Docker specifically, trying to improve this way the applicability of this technique to containers. In order to perform this assessment, it is necessary to fulfill the following specific objectives:

- Analyze and select which variables can be extracted and monitored from Docker containers. Determine a strategy in order to monitor those variables from the containers and make an intersection between them and the metrics already used in past experiments.
- Study the type of variations of the monitored variables when implementing the Docker setup into different supporting systems and hardware configurations, thus performing a small experimentation for each one.
- Select, analyze and adapt fault injection methods in order to produce failure data based on the injection of faults into the containers. Then, it is necessary to understand how the injected faults interfere with the variables extracted from the containers.
- Select, examine and determine which failure prediction methods perform better when predicting the failures caused by the injected faults.

The goals previously described are sequential, which means that each one of them influences the following ones. Thus, it is essential to make sure that they are completely achieved in order to avoid a chain of problems between the objectives.

Then, after all the previous objectives are achieved, the remaining step is to verify if the results obtained from predicting failures demonstrate if it is feasible to use this type of technique. Thus, if it is possible, it will be necessary to present and describe the observed advantages and disadvantages associated with the application of the failure prediction. Otherwise, an explanation about the reasons and problems that make the application of this method impractical in something that presents such a dynamic nature.

It is important to understand that researching new algorithms for **Failure Prediction** was outside of the scope of this work. However, we believe that the work developed addressed challenging objectives.

1.1 Research Contributions

The most important research contributions from this work are the following:

- **A technique was developed to gather variables in a non-intrusive way from the Docker container using for this a containerized setup and a monitoring tool**. A comparison of monitoring tools was performed to determine which one was less intrusive, had native Docker support, was container specific, had an open source license, and what metrics it could collect from containers. The monitoring tool that best fit at these five characteristics was **Docker SDK for Python**.

- **An analysis of the Docker behavior containers in distinct environments**. This was necessary to study the type of variations of the monitored variables collected by running the Docker setup in different configured systems. Two different systems were used to gather the variables and an adequate statistical test, **Mann-Whitney U test**, was applied to compare the variables from both systems. It was verified that more than half of the total number of variables collected seem to have different distributions.

- **An approach to adapt a Fault Injection (FI) technique to the experiment, where faults generated by this technique were injected into the applications**. The impact of those faults was studied and understood in order to verify how they affect the variables gathered from the containers. The **Software Fault Injection (SWFI) tool *BugTor*** [9] was used in order to generate the faults which will be injected later into the containers. Also, we created an approach that attempts to increase the **representativeness of the faults** by selecting those that were activated more times when the Docker container starts. Five different failure modes following a classification based on and adapted from **CRASH Scale** [10] were identified when analyzing the data generated by the induced faults. Each injection process lasts at most around four minutes.

- **An approach to use Failure Prediction (FP) techniques with the objective of predicting the induced failures by training and testing a FP model**. The focus here was to assess the technique's capacity and effectiveness in predicting the failures produced by the injected faults. The **Machine Learning (ML) tool *Propheticus*** [11] was used to perform the FP experiments. We also created an approach to select the prediction parameters and data balancing techniques to be used in the experiments. A comparison was performed between the results obtained and the results from Irrera & Vieira [1],

1.2 Document Structure

The remainder of this document is organized as follows.

Chapter 2 presents an overview about dependable systems, online failure prediction, fault injection, machine learning and containers.

Chapter 3 presents and describes the main approach designed to achieve the goals of this work, as also the selection process of the monitoring tool. This chapter also describes the approach followed to study the type of the variables variations, the approach followed to create failure data, and the approach followed to evaluate FP methods.

Chapter 4 presents the processes of generation of failure and non-failure data. The first is generated by using fault injection. It also describes the process conducted to compare the variables across different systems.

Chapter 5 describes the failure prediction experimental campaign performed after selecting the appropriate data. The failure prediction results are also presented and discussed in this chapter.

Finally, **Chapter 6** presents the main conclusions of this work and future work.

Chapter 2

Background and Related Work

This chapter is organized as follows. Section 2.1 presents the key concepts as far as the dependability is concerned, while Section 2.2 gives an overview of the background and related work on Failure Prediction (FP) as well as Online Failure Prediction (OFP), how can FP be evaluated, why Fault Injection (FI) is used and the main concepts related to Machine Learning. Finally, Section 2.3 presents an overview about containers, the Docker container platform and tools for monitoring containers variables.

2.1 Dependable Systems

Nowadays, computer systems are present in most tasks, business, services, and places, making people all around the world more and more dependent of them. This leads to the questioning of the limits of computing system's dependability as stated in [12]. This way, if a "nation-wide computer-caused failure" occurs, it can lead to catastrophic consequences such as economics variances and endangering human lives [12].

According to [12], the behavior of a system is composed by the service that it delivers, being noticeable by its users. On the other hand, a user is the other system that interacts with the computer system, which can be human or a machine.

Computer systems present an unpredictable behavior when they grow in size and complexity, leading to the possibility of not performing as planned. Thus, it becomes almost impossible to achieve all benefits that these systems aim to provide, leading to the detriment of the service [1].

Correct service is defined as the expected service that a system must deliver [13]. However, when the observed service is not the same as the expected one, it can be concluded that a **service failure** occurred [13]. The deviation of the service is named **error**, and if it propagates, it will originate a failure. Nevertheless, it can be identified and handled by the system or it may not proliferate, becoming a latent error, wherein the system will present the correct behavior during the observation time [1]. Lastly, an error is caused by an internal or external **fault** to the system, i.e. it can be originated from another system where a failure occurred. A fault does not always cause an error, becoming dormant and not influencing the behaviour of the system. There are several types of faults, which are listed and specified in [13]. The fault-error-failure chain is summarized in Figure 2.1.

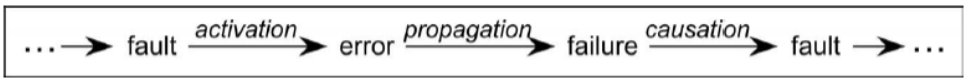


Figure 2.1: The chain of dependability threats (from [13]).

A definition of **dependability** is presented in [13], where it is defined as the capability of delivering a credible or trusted service knowing that the reliance of a system in another gives information about how each system's dependability influences the other. According to [13] and Figure 2.2, the decomposition of dependability comprises the following attributes, which influence it:

- **Availability** – the capability of a system to always be accessible when it is necessary, delivering the correct service.
- **Reliability** – a system's ability to keep providing the correct service.
- **Safety** – states that the system must not cause catastrophic consequences for its users, and the surrounding environment.
- **Integrity** – a system's ability to tolerate unauthorized modification attempts.
- **Maintainability** – the capability of a system that is able to undergo through changes and repairs.

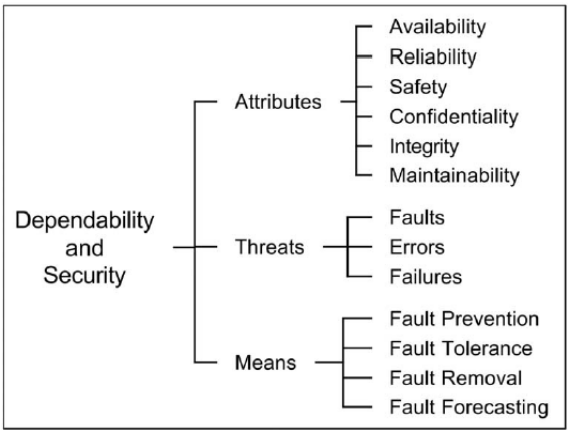


Figure 2.2: The dependability and security tree (from [13]).

According to [13] and to Figure 2.2, several means to achieve all the attributes that integrate dependability were matured over the last decades, which are arranged across four groups:

- **Fault prevention** – means which have the goal of keeping faults from occurring or being introduced into the system in order to attain a system with the possible minimum number of faults.
- **Fault tolerance** – means to prevent the system from failing when faults arise, making the system tolerant to them. This way, the system keeps providing the expected service even in the presence of faults.
- **Fault removal** – means which aim at decreasing the number of faults and mitigating the possible associated consequences that emerge from them.

- **Fault forecasting** – means which are necessary for measuring and evaluating the current and future number of faults and what consequences may arise when those faults occur.

Therefore, Failure Prediction can fit in fault forecasting because as it predicts failures based in past data, it can help to forecast fault occurrence in a system as each failure is strictly related to a given fault. On the other hand, predicting failures can also be associated with fault tolerance. In order for a system to have the capability of tolerating faults, it is necessary to learn and understand which failures can be originated from the occurrence of faults. This way, FP can assist it because it can foresee failures using previous information collected from a system.

2.2 Failure Prediction

Failure Prediction (FP) is the technique that is able to predict failures based on past failure data [1, 14]. Its main limitation is that it only predicts taking into account past failure data, using no information about the system's current state. A solution that solves this limitation is presented in Subsection 2.2.1, which is called Online Failure Prediction.

This section is organized as follows: a general overview of **Online Failure Prediction** is described in Subsection 2.2.1; an approach based in the contingency table from [15], which is summarized in Table 2.1, in order to evaluate **FP** algorithms, is presented in Subsection 2.2.2; and, lastly, Subsection 2.2.3 gives an overview about **Fault Injection**.

2.2.1 Online Failure Prediction

Online Failure Prediction (OFP) can be seen as the current stage of the evolution of Failure Prediction. "Online" means that this technique allows the prediction of failures using the system's current state [2]. Thus, OFP methods use information about the present (current state of the system), where they monitor the target system at runtime [2].

The taxonomy for Online Failure Prediction methods is presented in [15] and [2], where it is stated that these methods are based on:

- **Symptoms Monitoring** which has the objective of monitoring side-effects of errors which are named symptoms. The system's variables and parameters are monitored in order to detect strange system behavior. Techniques based in *time series analysis* and *function approximation* are used to monitor symptoms from a system. A large number of FP methods apply this technique, such as Multivariate State Estimation Technique (MSET) [16] and The Universal Basis Functions (UBF) [17]. According to [16], MSET consists in estimating the current system state by previously learning from the system states relations that occur between the parameters used to characterize each one of them. It is used vectors to define the states of the system, where it is the user that select them [16]. Lastly, UBF is based in Radial Basis Function (RBF), as stated in [17]. RBF is "a *nonlinear* data driven modeling technique" [17]. The authors from [17] claim that "RBF networks are one of the primary tools for interpolating multidimensional scattered data and are arguably one of the most popular methods for nonlinear regression". The difference between the Universal Basis Function (UBF) and Radial Basis Function (RBF) technique resides in the use of the G function: UBF uses a "flexible function (i.e. not necessarily

Gaussian) to adapt to specifics of the data space" [17] while RBF uses a "nonlinear transformation function" [17].

- **Undetected Errors Auditing** which consists in determining and identifying undetected errors trough the achievement of an audit. According to the authors of [2], it seems that there are no works related to this technique.
- **Detected Errors Reporting** which consists in writing reports to logs when an error is detected. Techniques based in *statistical tests*, *classifiers* and *pattern recognition* are used to report detected errors from a system. The Dispersion Frame Technique (DFT) [18] is an example of a method that is based on this category.
- **Failures Tracking** which is summarized as the process that uses tracking mechanisms to detect when a failure occurs. It is noted that tracking is frequently external to the system. Techniques based in co-occurrence and probability distribution estimation are used in order to track failures from a system. An example of a method that applies this technique is presented in [19].

It is important to note that **Fault Testing** is not considered for Online Failure Prediction because it consists in testing a specific part of the system. It is not necessary that the involved part is used by the system [2]. Thus, fault testing is a technique that aims at identifying faults through testing a given system. As OFP uses information about the system's current stage, it does not make sense to test a specific part of the system, because testing does not apply to the term "online".

Figure 2.3 presents the time relations in a task of Online Failure Prediction, where [15]:

- Δt_d represents the time interval between the past data gathered and the present time t.
- Δt_l, named *lead-time*, represents the interval of time where it is possible to predict the failure after present time t.
- Δt_w, called *minimal warning time*, represents the time interval while it is possible to conduct preventive actions.
- Δt_p, named *prediction period*, represents the interval of time where the prediction is valid.

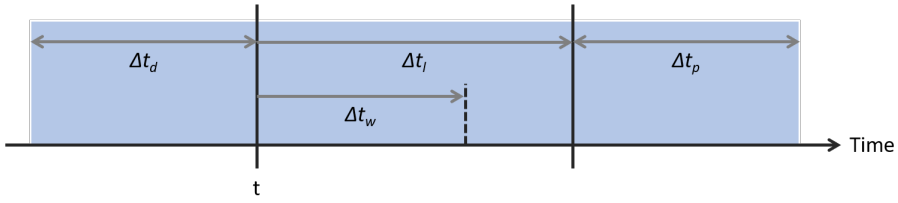


Figure 2.3: Time relations in Online Failure Prediction (adapted from [15]).

According to [15], the probability of correctly predicting a failure increases if the time interval Δt_p increases. However, if Δt_p increases too much, it will considerably reduce the accuracy of knowing when a failure will occur because it has more time to take into account. When analyzing Figure 2.3, it can be seen that the *lead-time* Δt_l cannot be shorter than *minimal warning time* Δt_w, because it is necessary to have sufficient time to

mitigate the consequences of the predicted failure or to completely avoid it [15]. Therefore, it is necessary to wisely choose the parameters from Figure 2.3 in order to obtain the best results when predicting failures.

Current approaches of Online Failure Prediction have already a large body of research, however they are yet to be applied in practice in systems such as containerized micro-service applications [20].

2.2.2 Evaluation of Failure Prediction

Failure Prediction (FP) algorithms need to be evaluated in order to select the one that fits better at predicting failures in a given context. Thus, in order to perform this selection, the definition and comparison of **figures of merit** are required [15].

Therefore, in order to conduct a more precise and rigorous prediction, it is necessary to forecast the maximum number of failures and to get the minimum number of false positives and false negatives [15]. When predicting failures, an algorithm can obtain the following possible results [15], which are summarized in Table 2.1:

- **True positive (TP)** which is defined as a true predicted failure, whose failure warning was correct.
- **False positive (FP)** which consists in detecting a failure that was not actually a failure, being a false alarm.
- **False negative (FN)** which is summarized as a failure that was not predicted as a failure, and no failure warning was raised.
- **True negative (TN)** which consists in not detecting a non-failure, which is correct.

Table 2.1: Contingency table (adapted from [15])

	Failure	No Failure	
Failure Predicted	**True Positive (TP)**	**False Positive (FP)**	Positives (POS)
No Failure Predicted	**False Negative (FN)**	**True Negative (TN)**	Negatives (NEG)
	Failures (F)	No Failures (NF)	Total (N)

This way, as stated in [15] and from observing Table 2.1, the total number of positives (**POS**) is obtained through the sum of the true positives and the false positives, while the total number of negatives (**NEG**) is obtained through the sum of the false negatives and the true negatives. Similarly, the total number of failures (**F**) is the sum of **TP** and **FN**, and the total number of no failures (**NF**) is the sum of the **FP** with the **TN**. A given instance of the contingency table is named **confusion matrix** [21].

The Failure Prediction algorithms have a binary result when predicting if some failure occurred or not. When they predict a failure, the result is *one*, and otherwise it is *zero*. When deciding if a given failure occurred, a **threshold** is used [1, 21]. If the result is above the threshold, the algorithm will classify it as a failure, where the result is equal to one. If it is below, it will classify it as a no failure, with the result being equal to

zero. According to [1, 21], varying the threshold has a direct impact in the accuracy and performance of the FP algorithm.

There are several metrics or figures of merit listed in [15], such as *Precision* and *Recall*, and these two specific metrics can be used in a method named **Precision/Recall Curve**. This method uses the above-mentioned threshold to decide if a failure is truly a failure. This value needs to be prudently chosen, because a lower value will increase the probability of detecting false positives, while a higher value will increase the probability of detecting false negatives [15]. Additionally, there is another method very popular called **Receiver Operating Characteristics (ROC)** [15], which is explained in subsection 2.2.4.

It is important to evaluate Failure Prediction algorithms in order to choose the one that obtains the best results when measuring the proportions between the true positives and the true negatives, and the false positives and the false negatives. It is clear that the best possible result is to classify all as **TP** and all non-failures as **TN**, having no **FN** and **FP**.

2.2.3 Fault Injection

According to [7], **Fault Injection (FI)** is defined as the technique that injects faults into a system in order to simulate the behavior of real faults and their impact. Thus, it can be considered as an acceleration technique which allows gathering fault and failure data faster, without needing to wait weeks or even months.

Prototype-based FI is the technique which aims at evaluating a system without it being necessary to have any premises about its design [22]. The **goals** of this technique, which are presented in [22], are the following ones:

- Determine existing barriers or obstacles which are jeopardizing the system's dependability.
- Understand and analyze the system behavior when faults are injected into it.
- Learn how well-developed and matured are the system capabilities in order to detect the errors caused by the injected faults, and to recover from failures originated by the propagated errors (fault-error-failure chain).
- Assess how effective and efficient are the fault tolerance mechanisms when detecting the injected faults.

The faults can be injected into a system at **software** or **hardware level**. This technique only analyzes emulated faults , and cannot provide dependability measures, as explained in [22].

Hardware Fault Injection is described as the technique that inserts faults in the system's hardware using additional hardware, as stated in [22]. It is divided in two categories named *hardware fault injection with contact* and *hardware fault injection without contact*. Examples of tools that inject faults at hardware level are *FIST* [23] (Fault Injection System for Study of Transient Fault Effect) and *MESSALINE* [7].

On the other hand, **Software Fault Injection** consists in introducing faults into a system at the software level [22]. Thus, the techniques based in this method are used with the purpose of injecting faults in applications without requiring expensive hardware, avoiding the associated costs [22]. Additionally, it is hard to inject faults into applications and

Operating Systems (OSes) using hardware fault injection techniques. Introducing faults into applications only requires inserting the fault injector into the application or into a layer between the application and the OS [22]. Nevertheless, inserting faults into the OS is more difficult, because it is hard to add a layer between the OS and the machine to implant the fault injector [22]. This technique has become very popular over the past two decades.

However, every technique has its **drawbacks** and software fault injection techniques are no exception. According to [22], these techniques have the following shortcomings:

- They only inject faults where accessible by software.
- They may perturb the workload if not well designed.
- They can be unsuccessful "to capture error behavior, like propagation" [22]. Thus, software fault injection techniques work better with long latency faults than short latency faults. Furthermore, in order to resolve the above problem, a *hybrid approach* should be used, which consists in a combination of "the versatility of software fault injection and the accuracy of hardware monitoring" [22]. Nevertheless, hardware monitoring can be expensive and reduce the flexibility associated to this approach, as it limits the "observation points and data storage size" [22].

Software injection techniques can be categorized in two groups according to the time that the faults are injected, as stated in [22]. The first category is named **compile-time injection**, which consists in inserting faults at compile-time, i.e. before the application is loaded and executed [22]. Thus, faults are injected into the source code of the application. Once the application executes, the faults will be activated [22]. This way, compile-time injection methods do not disturb the target application while it is executing, and are known for being easy to implement. However, they cannot introduce faults while the target application executes [22].

Runtime Injection is how the other category is called. According to [22], it consists in injecting faults during the execution of the target application, and it requires some type of mechanism in order "to trigger fault injection". These mechanisms can be the following ones [22]:

- **Time-out** which consists in introducing a fault when a software or hardware timer expires. It is a simple method and does not need to modify the target application. This method generates unpredictable effects to the application, because it works based on time and not based on "specific events or system state" [22].
- **Exception/trap** which is described as injecting faults when a specific event arises. The "exception" is related to hardware, while "trap" is related to software, where they "transfer control to the fault injector" [22].
- **Code insertion** which is summarized as the insertion of instructions or code into the target application, as the name suggests. This method injects faults while the application executes and inserts new code into the application, while not changing the already existing code [22].

Finally, examples of tools that use software fault injection are *Ferrari* (Fault and Error Automatic Real-Time Injection) [24], *Ftape* (Fault Tolerance and Performance Evaluator) [25], *Xception* [26] and *BugTor* [9].

Ferrari is a software fault injection tool, from 1992, which aims of using "software traps to inject CPU, memory and bus faults" [22]. This tool is composed by: "the initializer and activator, the user information, the fault-error injector, and the data collector and analyzer" [22].

Ftape is another example of a software fault injection tool, from 1995, which consists in a combination of workload generator and a fault injector, as stated in [25]. The workload generator includes a workload activity measurement tool. Through this combination, it is possible to "inject faults under high stress conditions based on workload activity" [25]. As this tool has a workload generator and a workload activity measurement tool, the authors from [25] claim that this aspect is the differentiator from other tools. Also, this functionality allows the injection of faults taking into account workload measurements [25].

Xception was developed in University of Coimbra in 1998, which aims at injecting more realistic software faults by using the information provided by the advanced debugging and modern processors features as the performance monitoring features [26]. The authors from [26] claim that the faults produced by this software fault injection and monitoring tool are injected with very low interference. Also, *Xception* does not alter the application when injecting faults to it. Lastly, as presented in [26], this tool does not use or rely in any OS service, because it works at the exception handler level. This way, all processes from the target system can be altered through the consequences produced by the faults introduced in the system [26].

BugTor was also developed in the University of Coimbra (UC) [9], in 2016. According to [9], this software fault injection tool generates software faults based in real faults made by programmers. The faults created by this tool are injected at the source code level. Also, it is important to note that those real faults, that can be emulated, are grouped in fault emulation operators, which in turn are listed in Table 2.2, as stated in [9]. Each operator has restrictions and is related to certain locations of the code, as presented in [9].

Table 2.2: *BugTor* fault emulation operators (adapted from [9]).

Operators	**Description**
MFC	Missing function call
MIA	Missing if construct around statements
MIEB	Missing if construct plus statements plus else before statements
MIFS	Missing if construct and surrounded statements
MLAC	Missing `and` sub-expr. in logical expression used in branch condition
MLOC	Missing `or` sub-expr. in logical expression used in branch condition
MLPA	Missing localized part of the algorithm
MVAE	Missing variable assignment with an expression
MVAV	Missing variable assignment with a value
MVIV	Missing variable initialization with a value
WAEP	Wrong arithmetic expression in parameters of function call
WPFV	Wrong variable used in parameter of function call
WVAV	Wrong value assigned to a variable

Lastly, there is also a recent hybrid approach named *HSFI* (Hybrid Software Fault Injection) [27]. By utilizing this hybrid software fault injection tool, it is possible to enable and disable the injected faults at the binary level, without being necessary to rebuild the code [27]. This way, the needed time to rebuild is shortened, allowing the *HSFI* to scale for applications with large code files [27]. It is also possible to use information about the context at the source level to inject the faults, as stated in [27]. The faults injected by

HSFI produce two versions of the code: the original code and the one with the code sections modified according to the fault injected. Thus, as presented in [27], each fault has a corresponding marker that will be noticed by the tool binary pass in order to decide which version corresponds to each code sections. Therefore, it is not necessary to rebuild the application because each injected fault has its marker, thus allowing easily its activation or deactivation [27].

2.2.4 Machine Learning

As stated in [28] in 1959, Machine Learning (ML) consists in the process where a computer behaves according to what it has learned as if it was an animal or a human being. In other words, ML is "about designing algorithms that allow a computer to learn" [29]. This way, ML already has some decades of research, not being a new subject. However, it is becoming very popular in the last years.

The algorithms from Machine Learning follow a classification taking into account what is the purpose of each algorithm [29]. There are six common classifications according to [29]:

- **Supervised learning** – a function or a classifier is created by the algorithm with the goal of mapping a given input to a requested output. Classification algorithms are an example of this.
- **Unsupervised learning** – a group of the inputs is modeled by the algorithm, because there are not any corresponding labeled examples.
- **Semi-supervised learning** – as the name implies, this classification is composed by the combination of the last two. This way, it is necessary to create a function or a classifier from the junction of the labeled and unlabeled inputs.
- **Reinforcement learning** – the action of the algorithm is controlled by a rule that was previously learned by the algorithm in order to act to a given observation of the environment. This way, the algorithm is leaded by the environment observations.
- **Transduction** – the output prediction is performed without the creation of a function or a classifier as in the case of supervised learning. It just tries to predict them based in the training inputs and outputs.
- **Learning to learn** – only previous knowledge or experience is used by the the algorithm to learn.

In order to get better results in Machine Learning, it is extremely important to take into account the data that will be used [30]. Then, if the data that will be used to train a supervised ML algorithm is not previously and properly treated, it will train the algorithm or classifier incorrectly [30]. This way, it is necessary to take into consideration and verify if the data has noise and repetitious or/and insignificant information [30]. It is stated in [30] that there is a relation between the performance of a supervised algorithm and the data pre-processing. If the latter is performed wrongly, it will substantially decrease the performance [30].

When pre-processing of the data is considered, the following processes can help to achieve it [30]:

- **Data cleaning** – if the data presents too much noise, it is necessary to reduce it dealing with the outliers [31]. To reduce the noise, it can use *clustering* and *regression* methods. On the other hand, if there are missing values from the data [32], it is required to handle them using a process from [33], as the case of *Most Common Feature Value*, *Mean Substitution* and *Regression or Classification Methods*.

- **Data transformation** – it can be achieved by performing the *Normalization* of the data, i.e. decreasing the difference between the maximum and minimum values observed in each feature. There are two methods most commonly used to normalize the features: *min-max normalization* and *z-score normalization*.

- **Feature extraction** – the extraction of the features is performed by combining them, where the combinations correspond to the new features that will be used [34]. The feature extraction methods can be supervised or unsupervised, being the unsupervised method *Principal Component Analysis (PCA)* [35] and the supervised method *Linear Discriminant Analysis (LDA)* [36] the most used [34].

- **Feature selection** – the irrelevant and redundant features need to be determined and removed from the data in order to increase the performance and the effectiveness of the ML algorithms. Feature selection methods can be classified into two main classifications [37]: *filter* and *wrapper*. In the case of filter methods, a score will be assigned to each feature, where this score is statistically calculated through comparing the features. *Mutual Information* [38] and *High Correlation Filter* [35] are examples of filter methods.

  In the case of wrapper methods, cross-validation is used in order to verify if a given feature is irrelevant or redundant. So, if after a given feature is removed and the result using cross-validation is worse, it could mean that the removed feature is not irrelevant neither redundant. Therefore, when the performance is considered, wrapper methods can be very slow an time consuming because making iteratively various combinations of the features. *Recursive Feature Elimination (RFE)* [39] is an example of wrapper methods.

- **Instance selection** – in order to reduce space and time complexities from the ML process, it is necessary to reduce the set of instances that will be used to predict [40]. This set of instances needs to have equal or better results, i.e. it needs to predict with equal or higher accuracy than the original set [40]. This way, the data used is reduced which increases the performance of the Machine Learning algorithms, through using one of the following data reduction approaches based on the selection of instances[40]: *Active Learning*, *Boosting*, *Prototype Selection* and *Sampling*. Also, it is stated in [40] that over-fitting can occasionally be prevented by selecting a smaller number of instances.

  On the other hand, *Random Oversampling* and *Random Undersampling* techniques can be used to handle imbalanced data [41]. As presented in [41], the first consists in the replication of random minority classes, and the latter is composed by the removal of random majority classes. Both attempt to adjust the data in order to decrease the discrepancy between the majority and minority classes [41].

  Synthetic Minority Over-sampling Technique (SMOTE) is a known oversampling technique, which differs from *Random Oversampling* because synthetic examples of the minority classes are generated, instead of being replicated [42].

According to [29], there are many algorithms types related to **supervised Machine Learning**, where many of them are classification algorithms: **Linear Classifiers** such

as **Logical Regression**, **Naïve Bayes Classifier**, **Perceptron** and **Support Vector Machine**, **Quadratic Classifiers**, **K-Means Clustering**, **Boosting**, **Decision Tree** and **Random Forest**, **Neural Networks** and **Bayesian Networks**.

The **Support Vector Machine (SVM)** uses an N-dimensional hyper plane which divides the data into two categories in order to classify them [29, 43]. This way, the SVM is a **binary classifier**. The SVM uses a kernel function with the goal of assigning a given point to a decision surface, as presented in [44]. *Radial Basis Function*, *Polynomial*, *Sigmoidal* and *Linear* kernels are examples of common kernel functions used in SVMs [44]. Also, the SVM can predict non-linear data using the *Kernel Trick*, and it is possible to use SVMs for regression problems too [44]. Figure 2.4 presents an example of a SVM with a polynomial kernel.

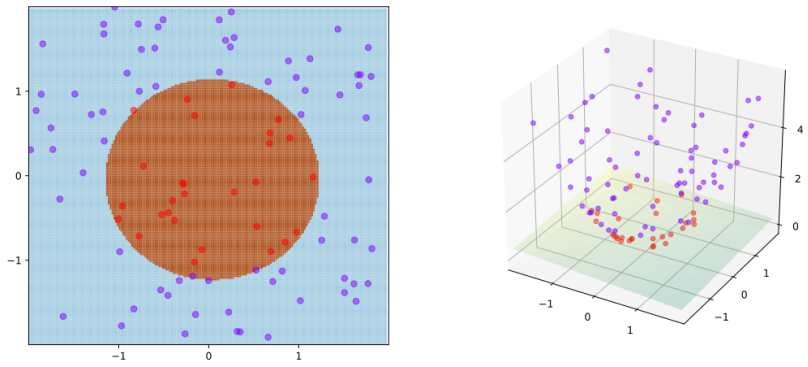


Figure 2.4: A training example of SVM with kernel given by $\varphi((a, b)) = (a, b, a^2 + b^2)$ (from [45]).

The evaluation of a ML algorithm is important to assess if it is predicting correctly or wrongly. Besides analyzing the number of correct and incorrect predictions, there are also other alternatives such as examining the temporal and space complexities of the ML algorithm [46]. It is important to note that a Machine Learning algorithm can predict with high correctness in a given dataset and low correctness in another [34]. This demonstrates the **No Free Lunch Theorem** [47]. Therefore, the evaluation of an algorithm always depends on the data used to train it.

Commonly, the dataset is separated in two different parts: the **training and validation datasets**. The first is used to train the algorithm by fitting the model and the latter is used for validation of the model [34]. Also, it is important to mention that this division is performed only for testing purposes [34].

Nevertheless, some problems can arise by dividing wrongly the dataset. The most common is **over-fitting** which happens when a classifier is randomly predicting or making generalizations, because the data used was not sufficient and/or contained noise, leading the classifier to "hallucinating" [48]. **Under-fitting** is another issue which consists in the opposite of over-fitting [48].

Bias and variance are related to the generalization error [49]. Figure 2.5 presents the decomposition of the generalization error into bias and variance in the context of the dart-throwing example [48]. **Bias** consists in frequently acquiring information the same wrong way, and **variance** occurs when the model learns in a random way instead of the correct way [48]. Usually, over-fitting is related with variance and under-fitting with bias.

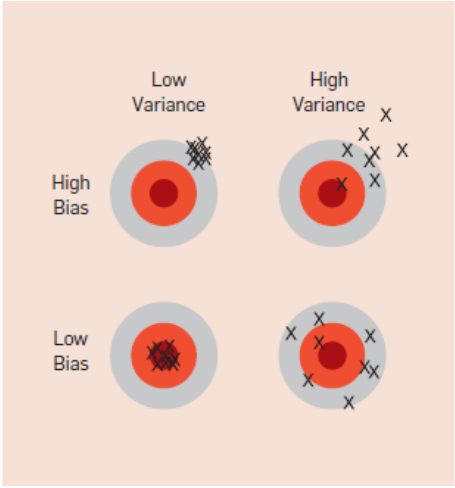


Figure 2.5: Bias and variance in dart-throwing (from [48]).

It is necessary to assess the performance of a ML algorithm, where it can be achieved by using classification performance metrics such as [34]:

- **Confusion Matrix** – is used when the Machine Learning algorithm is only predicting between two classes. It is an instance of a contingency table. The correct predictions are labeled as **True Positive (TP)** and **False Positive (FP)** and the incorrect predictions as **False Positive (FP)** and **False Negative (FN)**, where all are already defined in subsection 2.2.2.
- **Class Confusion Matrix** – is applicable when the ML algorithm is predicting more than two classes. It allows to visualize how a given class was predicted, i.e. how many times it was predicted as itself and as another class. This way, it is possible to verify, for example, which classes are repeatedly confused.
- **Receiver Operating Characteristics (ROC)** – consists in the combination of the TP and FP rates obtained by using different thresholds, as stated in [34].
- **Precision** – is calculated by dividing the number of TP by the sum of TP and FP for a given class. The closer the precision is to 1, the number of FP will be lower, tending to zero. However, it is important to note that the number of FN is not took into account.
- **Recall** – is obtained by dividing the number of TP by the sum of TP and FN for a given class. However, this metric does not consider the number of FP for that class. A recall corresponding to 1 means that there were not any FN, i.e. all samples from that class were predicted correctly.

Accuracy, **F-Score**, **Sensitivity** and **Specificity** are other examples of classification performance metrics.

Finally, to compare Machine Learning algorithms, it is necessary to perform **statistical tests**, where the null hypothesis is composed by if the algorithms have the equal performances, i.e. if the error rate is the same [34]. The **null hypothesis** is necessary because it cannot be proved, an it needs to be rejected by collecting the necessary evidence [50]. However, rejecting the null hypothesis does not mean that the **experimental hypothesis** will be proved, where it can be a **type I** error [50]. There is also another error named **type II** which consists is confirming that the null hypothesis is correct.

As presented in [34], to compare the performance of two ML classification algorithms, a statistical test such as **McNemar's Test** and ***K*-Fold Cross-Validated Paired *t* Test** can be used to achieve it.

2.3 Containers

According to [4], **containers** are "package software into standardized units for development, shipment and deployment", wherein they can be considered as an alternative to **Virtual Machines (VMs)**. A diagram with the comparison between the container architecture and the VM architecture is presented in Figure 2.6. A container has the following particular characteristics: it has all the necessary dependencies to execute, it does not need additional software in order to run, and it is very lightweight because it shares the OS kernel which leads to requiring less Random Access Memory (RAM) when compared to VMs. A container also isolates applications from their boundaries, wherein these can have the same behavior when the container is executing across different systems.

The containers originated in 1979 with the development of **chroot** as presented in [6]. In 2000, the **FreeBSD Jails** was launched. One year later, the **Linux-VServer** appeared and some years later, a project named Linux Containers also known as **LXC** [51]. However, more recently, users have started to massively use and adopt containers with the release of **Docker** in 2013 [6].

When comparing containers with virtual machines, which is present in Figure 2.6, it is clear that there are some similarities and differences between them. "Containers and virtual machines have similar resource isolation and allocation benefits, but function differently because containers virtualize the operating system instead of hardware" [4]. It is known that a VM tends to be more slow and heavy because it includes a full copy of the OS, i.e. an additional layer in relation to the containers who share the same OS kernel, as already mentioned [4]. This way, a container can boot and execute faster, and it has more portability, while occupying less space in disk than a VM.

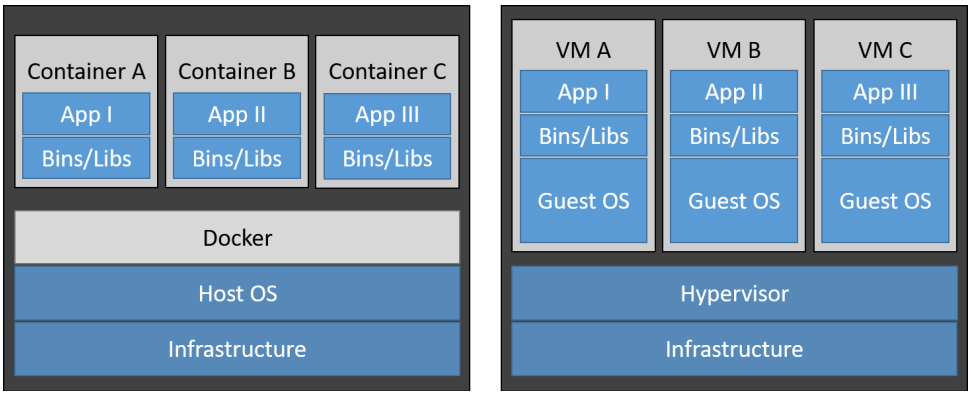


Figure 2.6: Comparison between a container (left) and a VM (right) (adapted from [4]).

2.3.1 Docker

It is known that Docker is one of the companies that has contributed the most to the container area, being the leader of the container movement. Docker claims that it is the only container platform provider that ensures the addressing of each application in the hybrid cloud [8]. An overview of hybrid cloud is present in [8], where it explains that

hybrid clouds require smooth portability of applications, which is accomplished through the container isolation that is provided. Thus, the applications become portable to any context or type of systems, which ends up with the problem of applications portability.

Docker was launched on March the thirteenth, 2013, being a recent technology. It features a functional API, called **Docker API**, which is very helpful in managing containers and containerized applications.

This way, Docker is a container virtualization platform [52] that claims to provide to the users [8]:

- **Agility** – the process of software development and distribution is very quick;
- **Portability** – the deployed applications can execute in any infrastructure;
- **Security** – the deployed applications are secured by using appropriate security mechanisms;
- **Cost savings** – users can save costs due to the optimization of the underlying infrastructure resources.

Therefore, the process of placing applications into containers increases the security, portability, agility and leads to cost savings as previously presented [8]. Currently, Docker containers support applications developed for Windows and Linux, and they allege that their containers follow open standards and are secured due to the provided container isolation, as stated in [4].

The **union file system** is the expression used by kernel developers for the copy-on-write model, being one of the technologies which Docker depends on, according to [52]. An image, as specified in "Docker language", is where a container is built, being a file system wherein one of its layers is dedicated for the dependencies, as the case of libraries, needed for the application to execute, and also the code and any package if used [52]. Thus, the result of building a container is a lightweight packaged software that only contains the necessary dependencies in order to execute the application. This way, it is like a "read-only file system with multiple layers" [52]. A Docker container can use cache too, making, in addition to the lightweight property, the process of rebuilding faster, as declared in [52]. This cache property can be visualized as the recycling and reusing of the layers in the file system of the container [52].

According to [52], Docker depends on two parts of the Linux kernel: ***namespaces*** and control groups or ***cgroups***. The first one is assigned by the kernel with the purpose of containing everything necessary for the creation of a container. A namespace isolates the processes, preventing them from knowing the existence of other process namespaces external to it. The second part is named *cgroups* and it is dedicated to the containers resource's supervision and management. It makes it possible to change the assigned resources of a container as the user wants [52].

Micro-services applications can be developed for Docker containers because of their light-weight characteristic [8]. So, it can easily create a single application made up of many containers. This way, according to [8], the process of creating, deploying, maintaining and managing applications based in micro-services can be simplified using containers.

Docker also allows the communication between hosts through **Docker Swarm**. It works like a "swarm of wasp". This way, Docker hosts can interact with other hosts. However, it

can have some implications in scalability and redundancy, being challenging for companies that want to implement this communication between different Docker hosts [52].

LXC (Linux Containers) and **rkt** are examples of Docker alternatives, being the latter the most important current competitor of Docker [53].

As stated in [54], the users can create and run multiple Linux containers by using the OS-level virtualization technology named **LXC**. Each container can execute a simple application or emulate something more complex: a host [54]. LXC features its own API, which helps the users manage their containers [55]. Also, the goal of Linux Containers is to emulate an environment similar to a standard Linux installation with the difference of not requiring its own kernel [55]. As presented in [54], Docker is seen as "an extension of LXC's capabilities". This way, Docker containers do not feature any OS [54]. Lastly, Docker has the following features that are not present in LXC [54]:

- The deployed applications are portable, which mean that they can be transferred and installed onto any machine with Docker installed.
- Docker can track the versions of a container, analyzing which differences are present between the versions, committing and/or rolling back the versions.
- Docker allows the reuse of Docker images (building and running) through different machines.
- Docker features a store [56] where various ready containers, plugins and Docker editions are available to be downloaded by the users.

On the other hand, **rkt** "features a pod-native approach, a pluggable execution environment, and a well-defined surface area that makes it ideal for integration with other systems", as presented in [57]. It uses pods which consist in one or multiple applications that are running in a context that is shared between them [57]. Also, rkt's standard container format is up-to-date and open source [57]. Nevertheless, it is stated in [57] that Docker container images (and not only) can be run with rkt. Last, but not the least, it is claimed that rkt is a very secure approach for the containers area, as it is stated in its slogan "A Security-minded Container Engine" from [57].

Therefore, there are some differences between Docker and its alternative, rkt, which are the following [53]:

- Docker is less secure when compared with rkt, as the latter was created taking into account the security flaws from Docker.
- Docker does not use an open source container format as the case of rkt.
- rkt is a more recent container technology, where its version 1.0 was released in February 2016, around three years after Docker 1.0 release.
- rkt does not feature and offer so many third party integrations as the case of Docker, being available on rkt's GitHub [58].

2.3.2 Monitoring Tools

In order to collect variables from the containers, it is necessary to have a tool that allows the achievement of this process. These variables correspond to the metrics that will be

used to monitor the containerized applications with the objective of gathering important information to be used to predict failures

The container's monitoring can be performed by using a monitoring tool or platform. Currently, there are many options available on the Internet, where a significant number of them are premium as the case of CoScale, Dynatrace, and New Relic. It is important to note that due to financial constraints, the premium monitoring tools will not be used nor tested in this work. However, they will be specifically analyzed in order to highlight which extra features they have, and which make them premium.

On the other hand, there are some open source monitoring tools such as cAdvisor, Metricbeat, Nagios, Sensu and Sysdig, wherein the analysis conducted by this work will fall on these tools with open source licenses.

It is also possible to use the **Docker API** as well the Docker Engine SDK available in Go and Python to obtain container metrics [59]. The first one is used to interact with the Docker daemon or Docker Engine API, being a RESTful API with an HTTP client. The second one aims at making the process of building and scaling Docker applications faster and easier [59]. It is as simple as using a stream function that exposes container metrics each second.

cAdvisor, or Container Advisor, is an open source monitoring tool developed by Google, in Go (or golang) programming language, which has the objective of collecting data about the resource usage and performance of containers [60]. It aims at aggregating, exporting, gathering and processing stats from containers. cAdvisor features Docker support and it states that it should support others container types. This monitoring tool is able to keep the exported collected metrics, wherein it can be composed by statistical data like histograms of resources usage [60]. Lastly, cAdvisor features a RESTful API that allows the obtainment of container stats, as well as a client developed in Go programming language.

CoScale is a premium monitoring tool with Docker certification, and features an SDK developed in Java programming language under the 3-clause BSD license. It claims that it goes further about Docker monitoring, being capable of recognizing services within containers [61]. This monitoring tool also claims that it has low overhead associated to container metrics gathering. As explained in [61], CoScale is a non-intrusive monitoring tool with minimal overhead as it uses lightweight agents to collect metrics and events from containers. It has the feature of detecting irregularities and inconsistencies that come from containers, as well the feature of full stack monitoring, i.e. capability to collect stats from both outside and inside of a container. Finally, CoScale presents that it is also possible for the users to create their own metrics, according to their needs [61].

Dynatrace is a premium monitoring tool that states it is very simple to monitor Docker containers because of its detecting ability when a container is created [62]. Since it claims that it monitors containerized applications and services without interfering with the images, it can be seen that it has very low intrusiveness in the containers. Dynatrace features a Docker agent under the MIT license. As explained in [62], Dynatrace can monitor applications based in micro-services because it was developed considering the dynamic environments associated with the container platforms. This monitoring also features the capacity of scaling when it detects new containers, being suitable for container orchestration and clustering [62]. Therefore, Dynatrace is great for monitoring micro-services because this kind of services tends to use a large number of containers. Lastly, it is also a certified Docker partner like the previous monitoring tool.

Metricbeat, from elastic, is written in Go programming language and in a summarized way, it consists in an open source shipper that collects metrics and dispatches them to Elasticsearch or Logstash [63], both also from elastic. It can gather system-level stats and it is container compatible. The process of monitoring Docker containers is performed through accessing and reading cgroups from Docker. Thus, the Docker API can be accessed in a straightforward way without the need to ask for any kind of authorization [63]. In order to store and process data, Metricbeat can ship the collected metrics to Elasticsearch, where the latter consists in "a distributed, RESTful search and analytics engine" as stated in [64], which features several client libraries written in different programming languages. Last but not least, it is important to note that the Docker module from Metricbeat is currently available as a beta version [65].

Nagios is an open source monitoring system whose goal is to deal with problems that arise from Information Technology (IT) infrastructures and systems [66]. This is a monitoring tool that has several features besides monitoring network, system metrics and applications, being able to launch alerts when it detects problems, to report the occurred problems, notifications and events, to perform maintenance and to predict when the failures will occur in order to avoid them while upgrading obsolete systems [66]. The main focuses of Nagios are infrastructure, network, services and systems. However, it is crucial to note that this monitoring tool does not directly support Docker, requiring a third-party plugin which can be found in Nagios Exchange [67].

New Relic is the last premium monitoring solution theoretically addressed by this work. It claims that it provides a complete monitoring for Docker containers and containerized applications [68]. This monitoring tool features deep visibility in containerized environments, which are known for their dynamic nature. It can also give a perception of the total cost of a Docker container to improve the business results.

Sensu is an open source monitoring framework that states it can easily collect and display metrics by providing a customizable platform [69]. It can detect if the systems which it is monitoring are working as expected or are having problems while executing, through the implementation of service checks. Sensu owns a RESTful JSON API that allows the access to the gathered metrics, featuring a key/value store. According to [69], this monitoring platform can also send alerts or notifications, and is both compatible with centralized and distributed monitoring. Sensu also has the ability to use external data through metric stats shipment [69]. Finally, to monitor Docker containers it is necessary to use a plugin, just like Nagios. However, this plugin provides native Docker support for gathering metrics from containers [70].

Sysdig is an open source monitoring tool that provides native support not only for Docker containers, but also for all the others Linux containers [71]. It claims that it offers deep system visibility, allowing the analysis of a system's and container's behavior. According to [71], this monitoring solution is compared to a system composed by *strace*, *tcpdump*, *htop*, *iftop*, *isof* and *Wireshark*. It also features User Interface (UI), named csysdig, that can be customized as desired [71]. So, Sysdig works at Linux kernel from OS level, which can belong to a physical or virtual machine. This way, it is possible to gather OS events which will be analyzed and filtered in order to find useful data for the container and system monitoring [71]. Additionally, Sysdig can execute as a container in order to collect metrics from containers.

Then, after presenting each one of the nine monitoring tools, the following can be summarized:

- All monitoring tools are container specific except **Nagios** and **Sensu**.

- There are two monitoring tools that do not feature native Docker support, which are **Nagios** and **Sensu**.
- **CoScale**, **Dynatrace** and **New Relic** feature a premium license.

Chapter 3

Online Failure Prediction in Containers

The main goal of this Master Dissertation is the assessment of the feasibility of applying online failure prediction to containerized applications based on micro-services. Thus, this work aims at supporting the improvement of the applicability of online failure prediction techniques in containers.

Also, it is known that the implementation of Online Failure Prediction (OFP) is difficult, even for containerized micro-services-based applications. In order to perform this assessment, it requires the fulfillment of four more specific goals already previously described (section 1), being in a summarized way the following ones: selection and analysis of the variables that can be monitored from containers, gathering of results from the observed behaviors when submitted to various environments, generation of failure data by injecting software faults into the setup and comparing the collected results in order to determine the effect of the injected faults in the experiment, and lastly, evaluation of the effectiveness of the chosen failure prediction algorithms. The approach followed in this work is summarized in Figure 3.1.

In order to achieve the **first specific objective**, it is necessary to use a containerized setup based in Docker and a monitoring tool. Thus, it is possible to divide this goal in two parts. The first one comprises the selection of the monitoring tool that fits best the following attributes or characteristics:

- Container specificity, i.e. if the monitoring tool features some functionality or property that are only specific to containers.
- Docker support, i.e. if the monitoring solution has native or official Docker support, or if it needs to use a third-party plugin, or, lastly, it does not feature any type of support.
- License which it can be open source, freeware and premium.
- Intrusiveness associated with the collection of metrics from the Docker containers. In other words, how much each monitoring tool interferes with the execution of the containerized application when gathering data from it.
- Number and relevance of the metrics that each monitoring solution is able to gather from containers.

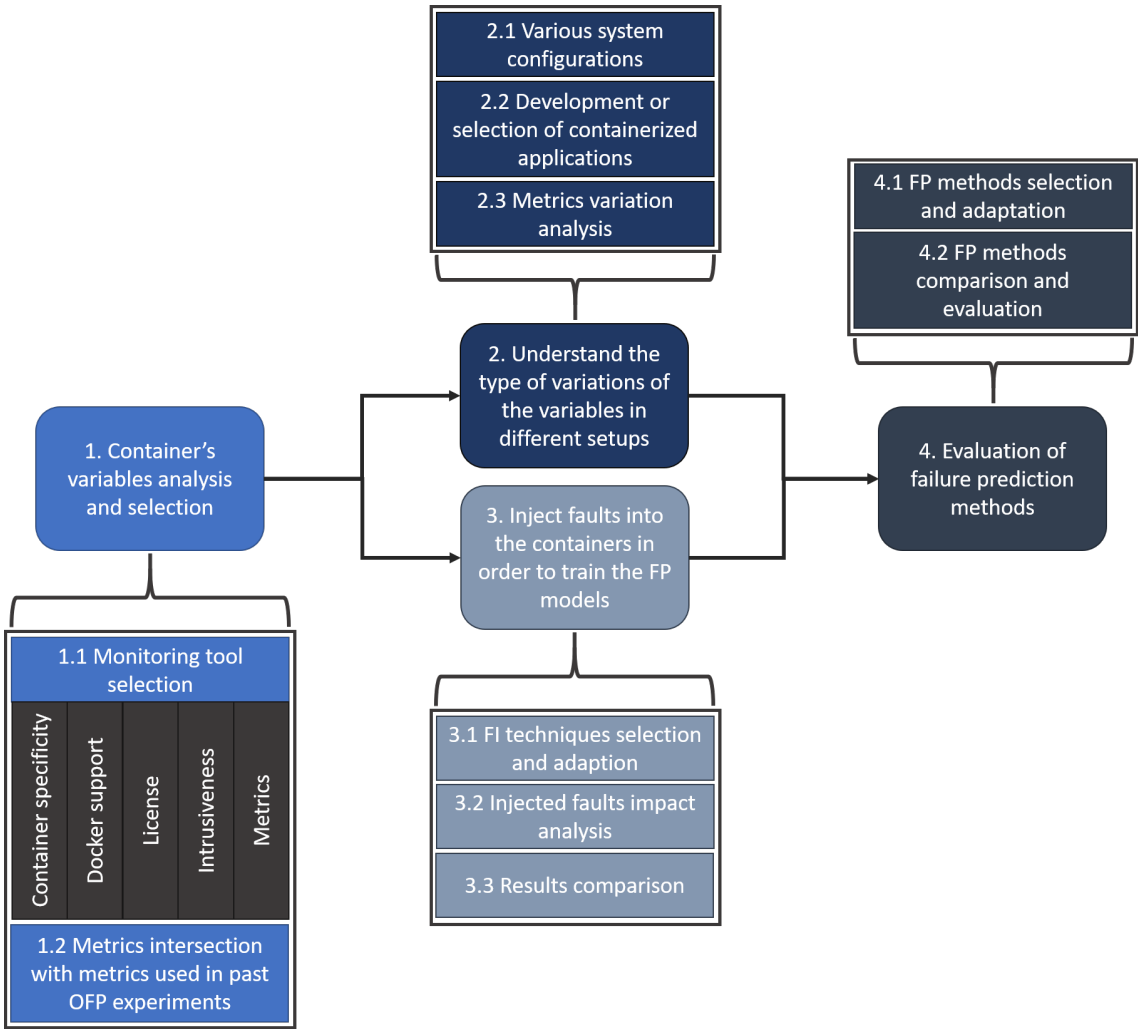


Figure 3.1: Overview of the research approach defined for this work.

The process of monitoring solution selection will be divided in two distinct parts in order to make the monitoring tools comparison easier. Firstly, a comparison between the first three attributes will be performed with the goal of discarding tools. Then, another comparison will be made considering the remaining attributes and monitoring tools. Lastly, after all the comparisons are completed, the tools that got the best results will be selected.

This work aims at using the metrics from Irrera & Vieira [1] as a basis for further comparisons with the container's metrics. Additionally, we will use the Fault Injection (FI) tools and OFP methods employed in Irrera & Vieira [1]. Adding, improving or formulating new concepts or algorithms to the Failure Prediction (FP) and FI domains is outside of the scope of this work.

Next, the second part of this monitoring tool selection is understood by a rigorous and accurate intersection between the selected monitoring tools metrics and the metrics used in the work presented in [1]. It is important to note that the monitoring solution which presents a better intersection between its metrics and the two hundred and thirty-three used in Irrera & Vieira [1] will be chosen to be used in this work. Finally, we can move on to the next goal.

The **second specific objective** is fundamental to assess the type of the variations of the metrics that can result from executing the Docker setup in various system configurations. This is relevant to understand how containers behave when executing in different environ-

ments, i.e. if they have identical behavior. It can be further subdivided into the following three goals:

- Set up various systems with distinct supporting systems and hardware configurations. Then, configure the Docker setup in each one.
- Development and selection of different containerized applications that will run inside a container in the Docker setup previously defined and configured.
- Analyze which variations occur between all the metrics collected from the containers running in each configuration, doing so a comparison.

After all the previous sub goals are achieved, arises the **third specific goal**, which will allow the understanding of how the fault injection techniques will influence the gathered metrics from containers. It consists in the following particular goals:

- Select and adapt fault injection techniques into the experimental setup with the purpose of generating failure data in the containers.
- Study in which way the injected faults affect the collected metrics from containers, understanding how the containers behave in the presence of their injection.
- Compare the results in order to determine which fault injection tool fits better in the configured Docker setup.

This way, this objective intends to produce failure data through the injection of faults that will be used by failure prediction methods to train their models.

Lastly, the **fourth and last specific goal** is very important and crucial to verify how the online failure prediction methods perform when applied to containerized environments. It can be divided in two sub-objectives:

- Select and adapt online failure prediction methods to the Docker setup in order to predict the failures originated by the induced faults.
- Compare how the online failure prediction methods perform and how effective they are at predicting the failures.

After all four specific goals are accomplished, we can conclude if it is feasible to use online failure prediction in containerized environments, accomplishing this way the main goal of this work.

The following subsections details the key points of this approach. An analysis and comparison of monitoring tools variables to select the tool to be used in this work is performed in Section 3.1. A comparison between the variables collected from containers running in different setups is accomplished by using an appropriate statistical test in Section 3.2. The failure data is created by using a FI tool, and a failure modes classification was proposed in Section 3.3. The FP experimental campaign and the results obtained are presented and discussed in Section 3.4.

3.1 Monitoring Container Variables

As presented in Section 2.3.2, the selected monitoring tools that will be analyzed and compared in this work are `cAdvisor`, `Metricbeat`, `Nagios`, `Sensu` and `Sysdig`. A simple implementation of the `Docker SDK`, which communicates with the `Docker API`, will also be used. It is important to note that these monitoring tools were selected through a meticulous process selection, where the ones that best suited our purposes were chosen. There are more premium monitoring solutions such as `Datadog` [72], `Sematext` [73] and `Pingdom` [74] which could be also introduced in Chapter 2. They feature an open source Docker agent. However, it is required to have a specific key which can only be obtainable by creating a **paid account**.

Following the approach described in Section 3, the monitoring solution selection starts with the analysis of the container specificity, Docker support and license. These characteristics are listed and summarized for each monitoring tool in Table 3.1.

Table 3.1: Comparison of the monitoring tools about their container specificity, Docker support, and license attributes.

Monitoring Tool	Container Specific	Docker Support	License
cAdvisor	Yes	Native	Open source
Docker SDK for Python	Yes	Native	Open source
Metricbeat	Yes	Native (beta)	Open source
Nagios	No	Third-party plugin	Open source
Sensu	No	Native via plugin	Open source
Sysdig	Yes	Native	Open source
CoScale	Yes	Native	Premium
Dynatrace	Yes	Native	Premium
New Relic	Yes	Native	Premium

As far as these characteristics are concerned, we can identify two main differences between the analyzed monitoring solutions. As already explained in Section 3, the premium tools are only listed in Table 3.1 as a curiosity. One of the significant differences that can be identified is the reduced number of monitoring tools that are not container specific: `Nagios` and `Sensu`. The other difference is determined by verifying that the previous identified monitoring tools do not support Docker directly too, requiring a plugin to it. However, all the other monitoring solutions analyzed feature native Docker support. Finally, we can also state that the tools are divided across two main groups: one is composed by the open source solutions an the other one with the premium ones. So, we can conclude that the tools from both groups look similar when considering their license property.

Since the first part of the monitoring solutions comparison is completed, we are able to discard the following tools:

- `Nagios` and `Sensu` because they are not container specific, i.e. they do not feature any functionality designed for the containers, and it is required to use a plugin in order to have Docker support. In the case of `Sensu`, the Docker support is native when using that plugin [70].
- `CoScale`, `Dynatrace` and `New Relic` because they are premium monitoring tools. So, it is necessary to pay for an account which is impossible due to financial constraints.

- `Metricbeat` only because its Docker module is available as a beta version and we seek more-stable tools.

This way, we can proceed to the next step after discarding the previous six tools. Another comparison will need to be carried out in order to select the monitoring solutions that fit better for this work. This step corresponds to the second part of the monitoring tool selection, as already explained in the approach presented in Section 3. The associated intrusiveness, and the metrics that can be extracted from containers by applying the tools, will be used to perform this comparison. Table 3.2 gathers the information from the remaining monitoring solutions related to these two characteristics.

Table 3.2: Comparison of the monitoring tools about their intrusiveness and metrics.

Monitoring Tool	**Intrusiveness**	**Metrics**
cAdvisor	None	`97`
Docker SDK for Python	None	`155`
Sysdig	None	`86`

As far as intrusiveness and metrics are concerned, we can only identify differences between the number of variables that can be extracted from containers with the application of each monitoring tool. The presented number of metrics is based on a previous performed analysis of the relevance of each metric in the container's context. The `Docker SDK for Python`, which interacts with the `Docker API`, is the tool that can extract more metrics from containers. On the other hand, `Sysdig` is the one that can monitor less metrics, being not far from `cAdvisor`. Thus, at this moment, we can state that `Sysdig` is the monitoring tool between the three analyzed that is more likely to be discarded. However, it is still possible that `Sysdig` can have more metrics correlated with the two hundred and thirty-three from [1] than the other tools.

When the intrusiveness of the monitoring solutions is considered, we verified that all the three were not intrusive. Firstly, `cAdvisor` is a monitoring tool that gathers most of the metrics from cgroup's tree, which leads to not needing to interact with the containers. Next, the `Docker SDK for Python` communicates with the `Docker Engine API` in order to get metrics from Docker containers. The `Docker API` is a `RESTful API` and features an `HTTP client` that allows communication through HTTP requests. This API features a method that allows to stream the container's metrics in each second. This way, the `Docker SDK` is not intrusive because it is a functionality of Docker. Lastly, **Sysdig** executes at Linux kernel where it collects and filters Operating System (OS) events. As it does not need to interact with the containers in order to gather metrics, it is not intrusive when it is monitoring.

Therefore, a further more detailed metrics analysis will be carried out in the next subsection, in order to understand which tool fits better in this work.

In order to select the monitoring tool that fits best in this work, the selection process needs to be rigorous. Thus, it is necessary to analyze the metrics gathered by each one of the three solutions summarized in Table 3.2, performing a meticulous and accurate intersection between them and the metrics used in the PhD Thesis *Fault Injection for Online Failure Prediction Assessment and Improvement* from Irrera & Vieira [1]. Those metrics are combined across fourteen different groups, which are listed in Table 3.3.

A prior analysis was conducted in order to study a possible intersection between the previous metrics and the ones gathered from containers using a monitoring tool. We have

Table 3.3: Metrics used in the Irrera & Vieira [1].

Object Name	Total
`.NET CLR Exceptions`	5
`.NET CLR LocksAndThreads`	10
Cache	27
Job Object Details	27
Job Object	13
Logical Disk	23
Memory	29
Objects	6
Paging File	2
Physical Disk	21
Process	27
Processor	15
System	16
Thread	12

confirmed that there is some metric types that or could not be applied to the container's context as the case of the `.NET CLR`, or that cannot be extracted from containers such as `Job Object` and `Paging File`. It was also verified that there are not metrics for both `Logical` and `Physical Disk`. Thus, we decided to combine these two types in only one named `Disk`, which had twenty-one metrics repeated. However, the initial number of two hundred and thirty-three will be used in the next comparisons that will be conducted in order to analyze the metrics gathered by these three monitoring solutions.

Figure 3.2 shows a Venn diagram which presents the results from comparing the two hundred and thirty-three metrics from [1] with the one hundred and fifty-five metrics that can be collected using the `Docker API` and with the ninety-seven metrics that cAdvisor can gather.

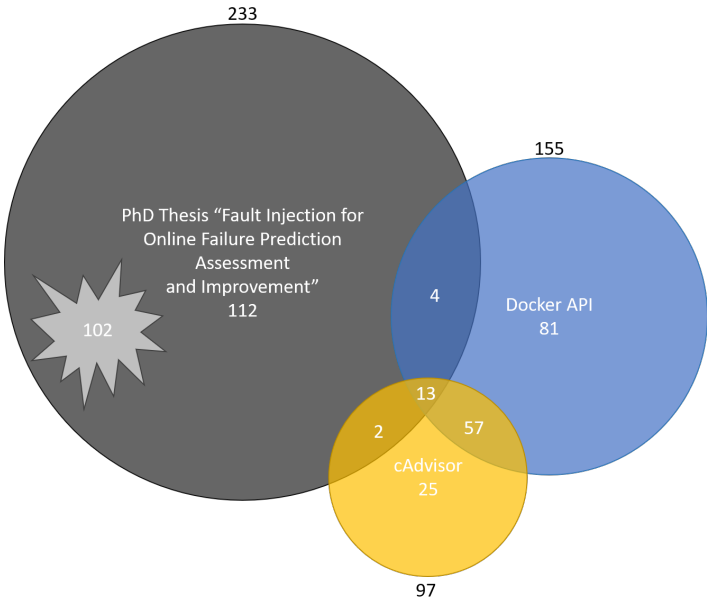


Figure 3.2: Intersection between metric sets from Irrera & Vieira [1], `cAdvisor` and `Docker API`.

When analyzing Figure 3.2, we can state that there is a low number of identical metrics between all the three lists, being only thirteen. On the other hand, we can verify that seventy-two percent of the `cAdvisor`'s metrics are identical to the ones from the `Docker API`. Thus, we are able to conclude that these two tools are not very different between each other. Additionally, we verified that the `cAdvisor` metrics types and the `Docker API` metrics types are within the same groups which are five, as we can see in Table 3.4. Finally, seven distinct metrics groups with a total of one hundred and two metrics do not have any correspondence with the two monitoring solutions used in this comparison.

A intersection between the metric sets present in [1], the eighty-six metrics that `Sysdig` can monitor from containers and, lastly, the one hundred and fifty-five from `Docker API`, was carried out and the results are shown as a Venn diagram in Figure 3.3.

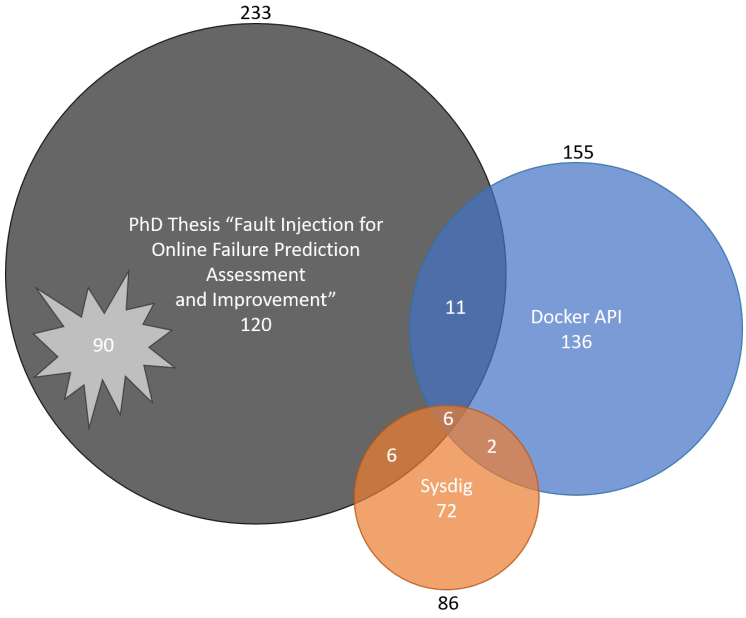


Figure 3.3: Intersection between metric sets from Irrera & Vieira [1], `Docker API` and `Sysdig`.

As `Sysdig` is a monitoring tool that provides deep system visibility by gathering and filtering OS events, by executing at Linux kernel, which leads to a bigger concentration of the gathered metrics along the following groups: `System`, `Process` and `Thread`. On the other hand, the metrics monitored by the other solutions, `cAdvisor` and `Docker API`, are more concentrated in groups such as `Disk`, `Memory` and `Processor`. Thus, the low number of identical metrics between `Sysdig` and the `Docker API` is understandable, as they focus in different types of metrics in order to monitor containers. The same reason can be appointed for the intersection between all the three metrics lists analyzed in Figure 3.3, concentrating only six identical variables. These observations are identified in Table 3.4.

Furthermore, we found out that six different groups with a total of ninety variables have not any correspondence when considering `Sysdig` and `Docker API`. It is also important to note that `Sysdig` has twelve metrics in common with the two hundred and thirty-three used in [1]. Therefore, we can conclude that the `Docker API` features a bigger correspondence, being seventeen different variables.

Finally, Figure 3.4 presents, also in a Venn diagram, how much identical are the metrics collect with `Sysdig and cAdvisor`, and with the ones used in the PhD Thesis "Fault Injection for Online Failure Prediction Assessment and Improvement" [1]. In order to accomplish it, a comparison of the variables that could be monitored by these two monitoring

solutions was performed.

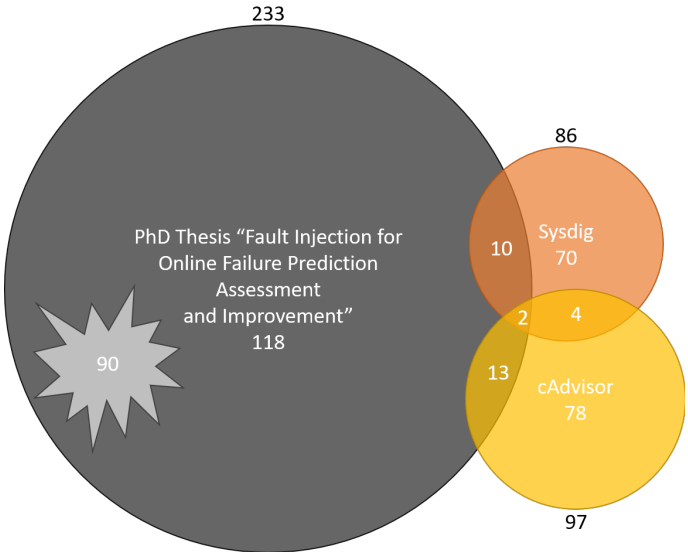


Figure 3.4: Intersection between metric sets from Irrera & Vieira [1], `cAdvisor` and `Sysdig`.

When analyzing Figure 3.4, we can state that the presented results are not very different from the ones summarized in Figure 3.3. As `cAdvisor` and the `Docker API` present a big intersection percentage between them, it can be a potential reason to explain the identical results when comparing these two monitoring tools with `Sysdig`. As far as the intersection of the three metrics lists presented in Figure 3.4 is concerned, a total of two metrics is common to all, being less eleven than in Figure 3.2 and less four than in Figure 3.3.

However, when considering the equivalence between the gathered metrics using `Sysdig` and `cAdvisor`, we verify that is only less two than in the previous comparison presented in Figure 3.3. Thus, following this perspective, we can conclude that `cAdvisor` and `Sysdig` metrics have a better intersection than the `Docker API` and `Sysdig` metrics, because `cAdvisor` is equivalent to have approximately sixty-three percent of the `Docker API` metrics. Finally, as in the case of Figure 3.3, there are also six distinct groups with a total of ninety metrics, which have no intersection with the considered monitoring solutions.

Table 3.4: Intersection between the metric sets from Irrera & Vieira [1] and the monitoring tools.

Object Name	**Irrera & Vieira [1]**	**cAdvisor**	**Docker API**	**Sysdig**
Cache	27	0	0	0
Disk	23	7	7	1
Job Object Details	27	0	0	0
Job Object	13	0	0	0
Memory	29	3	3	0
Objects	6	0	0	0
Paging File	2	0	0	0
Process	27	1	3	5
Processor	15	3	3	2
System	16	1	1	2
Thread	12	0	0	2
Total	**212**	**15**	**17**	**12**

Therefore, we can conclude that the monitoring tool that obtained the best results was the `Docker SDK for Python`, which gathers metrics using the `Docker API`. This way, it will be the tool that will be used in the remaining parts of this work. It is important to note that the gathered results from the comparisons are summarized in Table 3.4.

As the monitoring tool is already chosen, we can state this specific goal as accomplished. Thus, we can move on to the next step of this work, which consists in performing experiments composed by executing a Docker setup across various system configurations.

3.2 Understand the Variation of Variables across Setups

In order to analyze and study the Docker containers behaviour when running in different system configurations, it was necessary to perform experiments in different system configurations for the selected applications. This way, it was required to collect the data generated by running runs without any fault injected which are called **golden runs**.

The second and third specific objectives can be simultaneously done, so the FI tool was already selected which is presented in Section 3.3. The selection of the FI tool implies the use of applications which are developed in the targeted programming language, in this case the `C` programming language. Those applications will run inside Docker containers in order to perform the experiments to get failure data.

Then, after doing some research to find out which applications could be the best options to use in this work, we verified that the **Apache HTTP Server (Apache httpd)** [75], **NGINX Server** [76] and **PostgreSQL** [77] were the best candidates. The applications will run separately inside a Docker container. It is important to note that all the mentioned applications have the Core code developed using the `C` programming language.

We performed the experiments which are detailed in Section 4.1 and the variables were collected considering all scenarios: two applications and two system configurations, also presented in Section 4.1.

Lastly, it was necessary to use a statistical test to compare the variables gathered from the different systems configurations running each application inside a container. As the gathered variables were **not continuous**, **independent**, their distribution was not known (**non-parametric**) and were collected **from two different systems**, the **Mann-Whitney U test** [78] could be used. The comparison between the runs from each scenario was also performed in Section 4.1.

In the Mann-Whitney U test, the **null hypothesis**, $\mathbf{H_0}$, states that both x and y *samples* have the same distribution. On the other hand, the **alternative hypothesis**, $\mathbf{H_1}$, states the opposite, i.e. x and y *samples* have different distributions.

After understanding which variations occur between the variables gathered from different system configurations, we needed to move to the next objective which is composed by generating failure data using Fault Injection (FI).

3.3 Using Fault Injection to Generate Failure Data

The Fault Injection (FI) tools that were considered for this work were *HSFI* [27] and *BugTor* [9], both already presented in 2.2.3. Nevertheless, as the focus of this work is the use of applications which allow an easy integration of microservices, and as we already have

some experience with the *BugTor* fault injection tool, this tool looks more promising for the available time to perform the generation of failure data. We also have took into account that *BugTor* was developed in a Master Dissertation by *Gonçalo Pereira* at Department of Informatics Engineering (DEI), University of Coimbra (UC) [9]. This way, as we can only use one fault injection tool to generate failure data due to time constraints, we have chosen the *BugTor* to perform the fault injection in this work.

Section 4.2.1 presents the work performed to produce the faults by using the FI tool. After generating all the patches that contain the faults to be injected into the applications, it was necessary to select those that were more representative, i.e. the faults that were more activated when running the experiments. The selection of the faults is described in Section 4.2.2.

When all FI experiments were performed, we analyzed all the runs to identify which failures occurred. This part of the work is presented and described in Section 4.2.3. In order to identify the failures, we wrote the following failure modes classification based on and adapted from the **CRASH Scale** from [10]:

- **Catastrophic/Crash**: The Docker container stops running before the expected time, because the application that was running inside it crashed.
- **Restart/Hang**: When the system does not answer any request and/or wrongly answers all requests after a certain time instant.
- **Abort**: When a request that was performed to the application that is running inside the container and returned an error, or did not give the expected output, or, lastly, a timeout occurred (ten seconds).
- **Repeated Abort**: When many aborts happen during a certain time interval.
- **Hindering/Delay**: When a request lasts more time than the usual, passing a given threshold.

We have also created the following detection approach which is used in Section 4.2.3 in order to detect each failure:

- **Catastrophic/Crash**: The `Bash` script that is generating the workload, already mentioned in Section 4.1, it takes at least thirty-three minutes, corresponding respectively to one thousand, nine hundred and eighty seconds. When this script detects that it has already reached the expected time, always waits for the last requests made. This way, this can take up to two additional minutes at most, totaling thirty-five minutes. So, if the container stops running before reaching the thirty-three minutes, we can state that occurred a crash of the application that is executing inside the container. On the other hand, to verify if the application crashed during the additional time, we need to analyze the logs from the application that are persisted using a Docker volume. Finally, the `time stamp` corresponding the time instant the application crashes will be classified as a **crash**.
- **Restart/Hang**: It is necessary to verify if, at a certain time instant, all the requests made to the application failed. It should be noted that the time instant above referred must be at least thirty seconds from the end of the run. In order to analyze if a request failed, we need to confirm the output obtained from it. If there is not an output corresponding to that request or the output is different from the original

one, we can assert that that request failed. This way, we only need to verify if all requests performed after a certain time instant failed, and if so, we need to classify the first request `time stamp` as a **hang**.

- **Abort**: To find out if any request failed, we need to compare the number of correct outputs, i.e. outputs that are equal to the original ones, with the number of requests made to the application. So, if the number of correct outputs is less than the total number of requests, we can state that there were requests that failed. All these requests are classified as **aborts**.
- **Repeated Abort (RA)**: If many aborts occur in a given time frame, all will be classified as repeated abort (RA). It requires at least three consecutive aborts in which they must be separated by a maximum interval of two seconds between them. It is important to note that the time frame can be the entire run, being different from a hang, because there are successful requests between and/or after the ones that failed.
- **Hindering/Delay**: A log file with some discriminated times is created for each request performed. Those times are presented in the `workload script` in Section 4.1. So, a threshold of twice the sum of the average of the golden runs' requests times and corresponding standard deviation was set for each application and machine, totaling four distinct thresholds. Thus, every requests that lasted longer than the corresponding threshold will be classified as **hindering/delay**.

Therefore, we can move on to next step of this work as all the data was generated. The last objective consists in selecting the appropriate data to be used in the Failure Prediction (FP) experimental campaign, as also the prediction parameters and methods for feature selection and data balancing.

3.4 Evaluation of Failure Prediction Algorithms

As all the non-failure and failure data was generated, we could start this part of the work by **selecting the appropriate data to be used to train and test the Failure Prediction (FP) model**. We also needed to verify which failure modes from the initial five would be used in order to predict. Then, the approach followed for the data and failure modes selection is described in Section 5.1.

Once the data and features are chosen, we selected sets of parameters and Machine Learning (ML) techniques and we performed some experiments to chose the parameter combination with best results. In order to perform the FP experiments, we used the ML tool *Propheticus* [11] developed by João Campos, a PhD student at DEI, UC. This tool is still under development, but already does what is necessary for this work. The datasets need to follow a specific model in order to be used in the tool. Filtering by feature value, normalizing data, balancing data (random oversampler, random undersampler and Synthetic Minority Over-sampling Technique (SMOTE)) and reducing the dimensionality (correlation, F-Score, Recursive Feature Elimination (RFE) and variance) are examples of how this tool can process data. This part of the work is also described in Section 5.1. This ML tool can also analyze the data through, for example, class distribution, feature box plot, feature correlation plot and descriptive analysis. Lastly, it has clustering and classification algorithms to process the datasets. Decision Tree, Gaussian Process, Logistic Regression, Neural Network, Random Forests and Support Vector Machine (SVM) are examples of the classification algorithms that can be used.

After selecting the data, we performed the FP experimental campaign using the **SVM supervised classifier** and all the parameters and techniques chosen in Section 5.1. The experimental campaign is detailed in Section 5.3. The results are also presented in Section 5.3, where they are analyzed and discussed.

Lastly, it is important to note that a comparison of the results with is performed Irrera & Vieira [1] to **evaluate the effectiveness of the configuration chosen** for this work.

Chapter 4

Data Generation and Analysis

This chapter presents all the work performed related to the generation of both failure and non-failure data, where the first is created by using a Fault Injection (FI) tool and the second by running golden runs. This chapter is organized in three sections. Section 4.1 presents the system configurations used in this work and the setup created to run the experiments. A comparison between the variables collected from the containers running in the system configurations is also performed and described in this section. Section 4.2.1 describes the process conducted to produce failure data using for that a FI tool. This section is organized in three subsections. Section 4.2.1 presents how the faults were created for each application by using the FI tool. Section 4.2.2 describes the approach followed to select the faults taking into account their representativeness. Finally, Section 2.2.3 presents the process of generation failure data by injecting the faults. The process composed by the detection of the failures following a failure modes classification that was created in Section 3.3 is also describes in this section.

4.1 Metrics Variation Analysis

In order to perform the experiments, two machines were used whose specifications are listed in Table 4.1. The selection process of the applications that will run inside the containers took into account which programming languages the selected fault injection tool was able to inject faults.

Table 4.1: Hardware and Software specifications of the machines used in this work.

Specification	**Machine I**	**Machine II**
CPU	Intel(R) Xeon(R) CPU E5506 @ 2.13GHz	Intel(R) Core(TM) i3-4330 CPU @ 3.50GHz
RAM	12GB	16GB
OS	Ubuntu 16.04	Ubuntu 16.04
HDD	4x459GB	294GB
SSD	-	110GB

As presented in Section 3.2, the best candidates to be used as applications in this work are **Apache httpd**, **NGINX Server** and **PostgreSQL**.

In order to allow HTTP requests, a static website template called *Elements* was selected from [79] and will be hosted in each web server (Apache HTTP Server and NGINX Server), allowing HTTP requests to its HTML pages.

As far as Docker is concerned, the same version will be installed and used, `17.12.0-ce`, in both machines from Table 4.1. The containers will be built on top of Debian 9-slim [80], a lightweight version of Debian 9 (*Stretch*) which allows a faster download and installation. So, we coded the following `Dockerfiles` for building the following Docker images:

- To install all server dependencies and configure the Apache httpd or the NGINX Server. The goal is to avoid spending time, as these actions are always the same for each run. Then, it can be seen as the configuration Docker image for each application.
- To replace the code with the one with the fault injected and install the server (Apache httpd or NGINX Server), using the corresponding configuration Docker image.

Also, it is necessary to install the monitoring tool in each machine from Table 4.1, in order to gather the variables from the containers. A `Python` script was developed using and adapting the Docker `SDK` for `Python` from [81]. This script collects the metrics from containers, using a function from the `SDK` that streams them, and stores them into a `SQLite` database.

Lastly, the following `Bash` scripts were developed to organize and facilitate the execution of the experiments:

- The **main script** iterates over all the selected patches which contain faults, and for each run it applies the patch, builds and runs the container, launches a new process running the Python script already explained in Section 4.1 in order to monitor the container, runs the workload script and, finally, when the workload finishes, it removes the patch. This script is summarized in Figure 4.1.
- The **workload script** that submits HTTP requests to the container using the `curl` command, saving errors, outputs and times into logs. The time variables displayed by the `curl` command are the following, as presented in [82]:
  - **time_appconnect** – "The time, in seconds, it took from the start until the SSL/SSH/etc connect/handshake to the remote host was completed".
  - **time_connect** – "The time, in seconds, it took from the start until the TCP connect to the remote host (or proxy) was completed".
  - **time_namelookup** – "The time, in seconds, it took from the start until the name resolving was completed".
  - **time_pretransfer** – "The time, in seconds, it took from the start until the file transfer was just about to begin. This includes all pre-transfer commands and negotiations that are specific to the particular protocol(s) involved".
  - **time_redirect** – "The time, in seconds, it took for all redirection steps including name lookup, connect, pretransfer and transfer before the final transaction was started. `time_redirect` shows the complete execution time for multiple redirections".
  - **time_starttransfer** – "The time, in seconds, it took from the start until the first byte was just about to be transferred".
  - **time_total** – "The total time, in seconds, that the full operation lasted".

  After analyzing them, we decided to not count the time_namelookup, because the name resolving is not influenced by the application that runs inside the Docker container.

This script consists in seven phases. It begins with single requests to random HTML pages, from the static website, during three minutes, where each request is made between one and three seconds after the previous one. It stops and waits for five minutes, then it continues to perform between two and fifteen parallel requests during also five minutes. At this stage, it will repeat the pause and the parallel requests mentioned above. Then it will pause again during five minutes and lastly, it will make between twenty and thirty parallel requests to the container.

This script is summarized in Figure 4.1 and has a duration of at least thirty-three minutes. It is important to note that all requests have ten seconds of timeout, i.e. they will wait at least ten seconds for the application response, and otherwise, an error is returned.

- The **diff script** verifies the output of all requests submitted to the container. In order to do that verification, it uses the `diff` command between each request's output and the corresponding `HTML` page hosted in the application that is running inside the container.

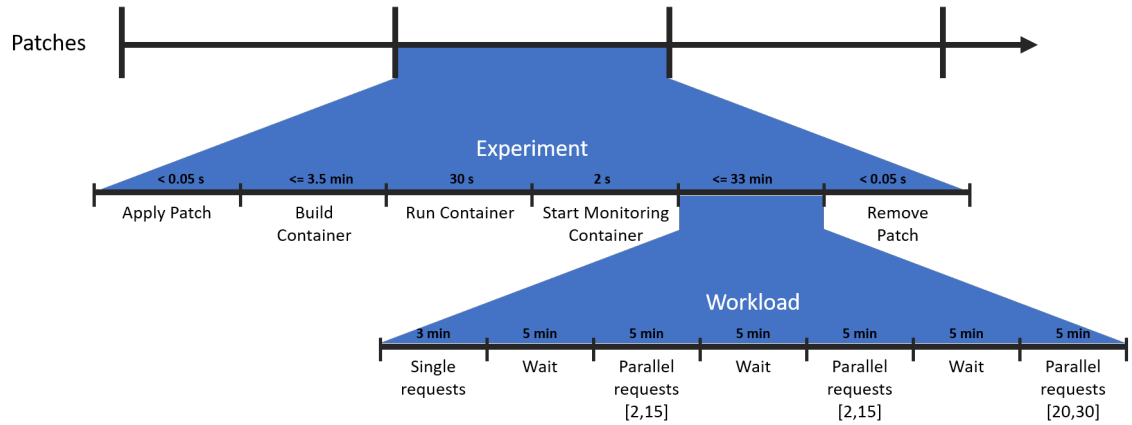


Figure 4.1: Overview of the experiment phases and workload submitted to the Docker container.

Thus, taking into account the duration of each phase from the experiment, we can conclude that each run lasts at most thirty-seven minutes, where each run corresponds to the injection of one fault.

Nevertheless, it is also necessary to generate non-failure data to train the failure prediction models. This way, we decided to run ten golden runs for each scenario, where each scenario corresponds to an application: Apache httpd, NGINX Server and PostgreSQL. Each golden run will execute use the scripts and the times already established. The only difference resides in the fault injection that will be not necessary for the golden runs case.

The same seed value was took into account and was used in the golden runs and in the fault injection runs (FIR), because `Bash` scripting uses **pseudorandom** generators. This way, the same sequence of random numbers was used for the time between requests and for the HTML page requested by a given request.

After running ten golden runs for each application, we could perform a comparison between the gathered variables from the runs from each machine. We needed to use a statistical test to perform this comparison. As already presented and described in Section 3.2, the **Mann-Whitney U test** [78] was used because the variables were **not continuous**, **independent**, **non-parametric** and were collected **from different systems**. This way, we

used this statistical test in each variable of the one hundred and forty-nine non string type metrics collected by the monitoring tool `Docker SDK for Python`, from both machines.

In order to apply the Mann-Whitney U test, we used the `Python` package `mannwhitneyu` from SciPy [83]. It is important to note that we defined the alternative hypothesis as **two-sided** or **two-tailed** to calculate the p-value, so the mean of the *x samples* was lesser or greater than the *y samples*. This way, if the p-value calculated was lesser than five percent we could rejected the null hypothesis for the corresponding variable.

Table 4.2 presents seventeen example variables from the one hundred and forty-nine variables collected when running Apache httpd server, and the corresponding U statistic and p-value obtained from using the `Python` package [83] mentioned above. The package raises an error when all x and y samples are identical, i.e. all values from both samples are equal. The variables corresponding to this case were marked with "-".

Table 4.2: Example of the U statistic and p-value calculated by applying the Mann-Whitney U test to the Apache httpd golden runs variables.

Variable	**U statistic**	**p-value**
BLKIO service bytes recursive read major	-	-
BLKIO service bytes recursive read minor	0	0
BLKIO service bytes recursive read value	63318179.0	0
BLKIO merged recursive read major	-	-
BLKIO merged recursive read minor	0	0
BLKIO merged recursive read value	206618178.0	$2.36e^{-24}$
CPU total usage	309064315.0	0
System CPU usage	411133372.0	0
CPU throttling data periods	-	-
Memory usage	169108405.0	$4.65e^{-210}$
Memory active file	129739788.5	0
Memory cache	163924174.0	$1.27e^{-273}$
Memory dirty	205788782.5	0.84
Memory pgfault	407496779.5	0
Network eth0 rx bytes	218164177.0	$1.17e^{-26}$
Network eth0 rx packets	194301478.5	$1.21e^{-21}$
Network eth0 rx dropped	-	-

When the p-value is zero, it means that all values from one sample are greater than the values from the another sample, or the opposite. Lastly, when the p-value is lesser than five percent means that there is enough evidence to reject the null hypothesis H_0 for a given variable, otherwise the null hypothesis cannot be rejected.

On the other hand, Table 4.3 also presents the same seventeen example variables and the corresponding U statistic U and p-value obtained by using the same Python package from SciPy [83], in which the variables were gathered by running the NGINX Server inside a Docker container.

Finally, Table 4.4 presents a summary of all p-values obtained for each variable from both scenarios: Apache httpd and NGINX servers. As far as the variables collected in the Apache httpd scenario are concerned, we concluded by viewing the values presented in Table 4.4 that about forty-four percent of the variables have equal distributions in both machines against about fifty-three percent that have different distributions. Only four variables in a total of one hundred and forty-nine did not allow the rejection of the null hypothesis because their p-value is greater than five percent.

Table 4.3: Example of the U statistic and p-value calculated by applying the Mann-Whitney U test to the NGINX golden runs variables.

Variable	**U statistic**	**p-value**
BLKIO service bytes recursive read major	-	-
BLKIO service bytes recursive read minor	31783367.5	0
BLKIO service bytes recursive read value	138428228.5	0
BLKIO merged recursive read major	-	-
BLKIO merged recursive read minor	31783367.5	0
BLKIO merged recursive read value	-	-
CPU total usage	279527082.0	0
System CPU usage	397464030.0	0
CPU throttling data periods	-	-
Memory usage	279386552.5	0
Memory active file	144942577.5	0
Memory cache	157691680.5	$2.19e^{-279}$
Memory dirty	198393211.5	0.76
Memory pgfault	187386846.5	$5.49e^{-23}$
Network eth0 rx bytes	218164177.0	$5.17e^{-49}$
Network eth0 rx packets	194301478.5	$1.28e^{-48}$
Network eth0 rx dropped	-	-

Table 4.4: Summary of p-values obtained by applying the Mann-Whitney U test for each variable from Apache httpd server and NGINX server scenarios.

p-value	**Apache httpd server**	**NGINX server**
-	66	51
< 0.05	79	95
≥ 0.05	4	3

By continuing to analyze Table 4.4, the NGINX Server scenario presented more discrepant values than the Apache httpd scenario, where about thirty-four percent of the variables had equal values in both machines which leads to the conclusion that they have the same distribution. In contrast, about sixty-four percent of the total one hundred and forty-nine variables presented different distributions in both machines as their corresponding p-value is lesser than five percent. Lastly, the null hypothesis was not rejected in only three variables, as we can see in Table 4.4.

Therefore, we can conclude by analyzing the data collected from the Apache httpd and NGINX Server golden runs, that more than half of the total number of variables collected from both machines (one hundred and forty-nine) seem to have different distributions. This way, we needed to take it into account, because the values of these variables seem to highly depend on the machine where the Docker container is running.

As the comparison of variables gathered from different systems is finished, we can move on to the next specific goal which consists in applying a fault injection tool to generate failure data to be used later in the failure prediction, to train and test Failure Prediction (FP) models.

4.2 Using Fault Injection to Generate Failure Data

4.2.1 Fault Generation

As stated in 3.3, *BugTor*, currently hosted in `ucx.dei.uc.pt` [84], was the selected fault injection tool for this work. It is of simple use, being only necessary to upload the applications code and to run the tool in each code file in order to generate the patches which contain the fault that will be injected later. It is important to note that this tool formats the code files, and generates the patches considering them.

We started to generate all patches for a selection of C code files from Apache httpd, NGINX and PostgreSQL source code using the selected FI tool. The selection of code files was performed taking into account a prior analysis from which folders and files could be more interesting:

- In the case of Apache, we used most of all `C` files from its `Server` folder. Files from `mpm/mpmt_os2`, `mpm/netware` and `mpm/winnt` folders were not considered.
- In the case of NGINX, all `C` files from its `core`, `events` and `http` folders were selected. In the latter two, the `C` files from `modules` were not considered.
- In the case of PostgreSQL, we considered all the `C` files inside `lib` and `nodes` folder and `main C` file from `backend` folder. Moreover, all files from `bin` folder were also chosen.

A summary of all generated patches is presented in Table 4.5. We can observe that the number of `C` files is not so much different when comparing Apache with NGINX. However, PostgreSQL has approximately four times more `C` files than Apache httpd and six times more than NGINX. Nevertheless, the number of selected `C` files does not follow a proportion for all applications, fifteen percent of Apache files were selected, thirty-seven percent for NGINX and only nine percent for PostgreSQL. As already mentioned, the selection approach takes into account the files that seemed more interesting to inject faults. Then, Apache httpd Server is the application with less selected files and generated patches, and on the other hand, PostgreSQL is the one with more selected files and created patches.

Table 4.5: Summary of total generated patches, selected and total `C` files for Apache HTTP Server, NGINX Server and PostgreSQL.

Application	**Patches**	**Selected `C` Files**	**Total `C` Files**
Apache httpd	8266	42	276
NGINX	35885	73	199
PostgreSQL	53057	109	1204

The software fault injection tool *BugTor* generates patch files for a given code file written in `C` programming language, as stated in [9]. Each patch file contains a modification for a portion of the original code, which corresponds to a fault. In order to inject the fault, we only need to apply the patch to the original code. As stated above, this tool is hosted in `ucx.dei.uc.pt` [84] and requires a registered account in order to use it. Then, we uploaded the source files presented in Appendix A: Table A.1 as Apache HTTP Server is concerned, in Tables A.2, A.3 and A.4 in the case of NGINX Server, and, lastly, in Tables A.5 and A.6 as PostgreSQL is concerned.

4.2.2 Fault Selection

The selected Fault Injection (FI) tool, *BugTor*, has some known limitations with code that contains macros, as presented in [9]. As various code files from Apache HTTP Server and NGINX Server present conditional compilation, this arises as a problem. This way, the tool creates patches with errors when the code presents macros in places other than the beginning of the code. In order to fix this problem, it was necessary to identify and set a strategy. In [9], two ways are suggested to deal with this problem:

- Using the command `gcc -e file` to solve the macros. However, as most of the files from Apache HTTP, NGINX Server and PostgreSQL have dozens of dependencies, it will be very difficult to get it to work.
- Analyzing separately each file and select the most important parts or functions. However, we wanted to submit the whole files to the tool and then study and select which patches were the best for this work.

So, the best approach we have identified consisted in doing a prior analysis about which functions execute when these applications start and when they receive and process HTTP requests. In order to reduce complexity and due to time constraints, we needed to select less applications in order to save some time. From this moment, we will only consider Apache HTTP Server and NGINX Server, leaving PostgreSQL for future work.

In order to try to increase the **representativeness of the faults**, i.e, injecting faults in parts of the code that execute one or more times in the situations described above, we needed to do a statistical analysis to choose the files and patches which were more representative. Then, we added a `fprintf` to the beginning of each function of the files. Every time a function executes, it will add a line with the corresponding file and function identifiers to a text file inside the Docker container, which is persisted by using a Docker volume. Therefore, the `C` code files were selected using a profiling technique for both applications.

Figure 4.2 and Figure 4.3 present the twenty-three functions that were executed the most. These results were obtained by following the `fprintf` approach described above. We will give more importance to the results associated with the performed HTTP requests, which some of them are presented in Table 4.3, because we want to minimize the possibility of the containers crashing at the beginning of their execution. Thus, we can have more data monitored from the containers as they will last longer.

On the other hand, Figure 4.4 and Figure 4.5 also present the twenty-six `fprintf` that had more occurrences, however those are related to Apache httpd Server.

After analyzing which functions and code files run more times when the application starts running or receives requests, we selected eleven `C` code files and twenty patches for each file. The best candidates for the Apache httpd Server were the following `C` code files: **config.c**, **core.c**, **core_filter.c**, **event.c**, **fdqueue.c**, **log.c**, **main.c**, **protocol.c**, **request.c**, **scoreboard.c** and **util_filter.c**. On the other hand, the best candidates for the NGINX Server were the following eleven files: **nginx.c**, **ngx_array.c**, **ngx_buf.c**, **ngx_connection.c**, **ngx_hash.c**, **ngx_list.c**, **ngx_open_file_cache.c**, **ngx_palloc.c**, **ngx_rbtree.c**, **ngx_string.c** and **ngx_times.c**.

Therefore, we will use a total of two hundred and twenty different patches in this work for each application. The patches selection took into account the following approach:

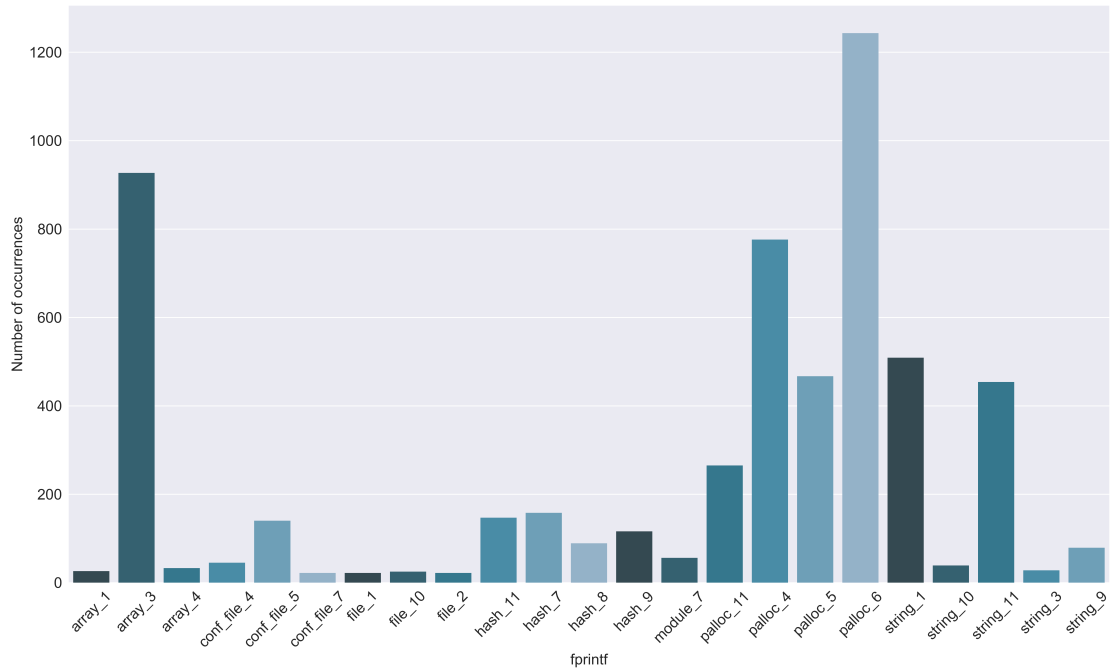


Figure 4.2: Plot of `fprintf` data with more occurrences from NGINX Server when it starts running.

- Verify the lines where the function starts and ends in the code file that was previously formatted by the FI tool.
- Find all the patches that contain modifications between the lines mentioned above.
- Lastly, choose the twenty patches that look more suitable and interesting for this experiment. A profiling technique was also used. So, the patches were carefully selected following the distribution presented in Table 4.6 whenever possible. Table 4.6 presents the software faults types generated by *BugTor* [9] for the systems under testing: Apache httpd Server and NGINX Server, based in field data study [85]. To track if a given fault is activated, we added a `fprintf` with an identifier to all patches.

  Due to time restrictions, it was required to select the best candidate faults to be injected later in the experiments.

Table 4.6: Fault types generated by *BugTor* [9] and their incidence taking into account the field data study from [86].

Fault Type	Incidence (%)
Missing `if` around statements (**MIA**)	4.32 %
Missing `if` construct plus statements plus else before statements (**MIEB**)	3.20 %
Missing `if` construct plus statements (**MIFS**)	9.96 %
Missing `AND sub-expr` in expression used as branch condition (**MLAC**)	7.89 %
Missing `Or sub-expr` in expression used as branch condition (**MLOC**)	4.70 %
Missing small and localized part of the algorithm (**MLPA**)	1.32 %
Wrong variable used in parameter of function call (**WPFV**)	1.50 %
Total	**32.89 %**

When the patches selection was completed, it was necessary to generate those patches by modifying the original file accordingly to each patch and using the `diff` command between

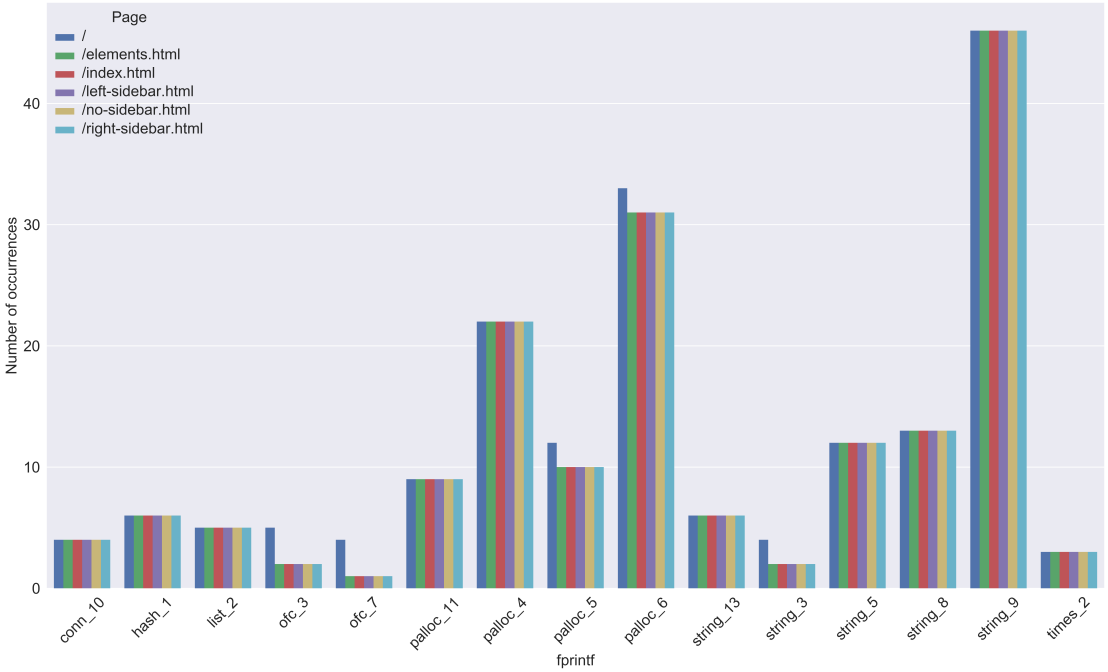


Figure 4.3: Plot of `fprintf` data with more occurrences from NGINX Server for each HTTP request.

the modified file and the original one. So, this action was performed both for Apache httpd and NGINX two hundred and twenty times, totalling four hundred and forty patches. It is important to note that it was a very exhausting and time consuming task.

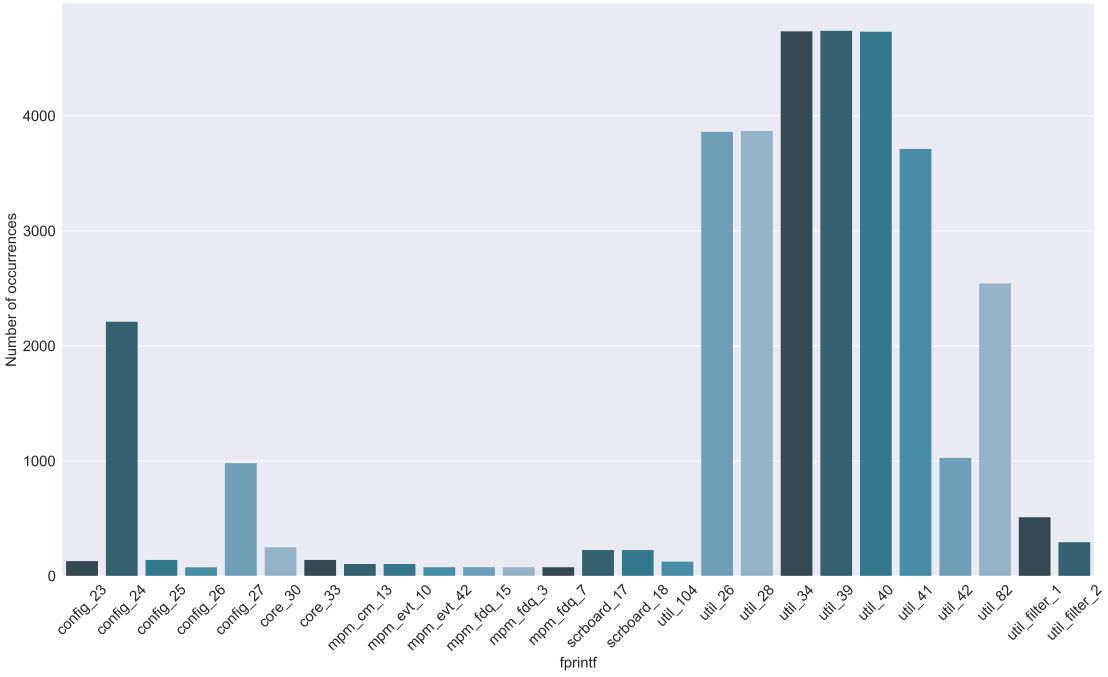


Figure 4.4: Plot of `fprintf` data with more occurrences from Apache httpd Server when it starts running.

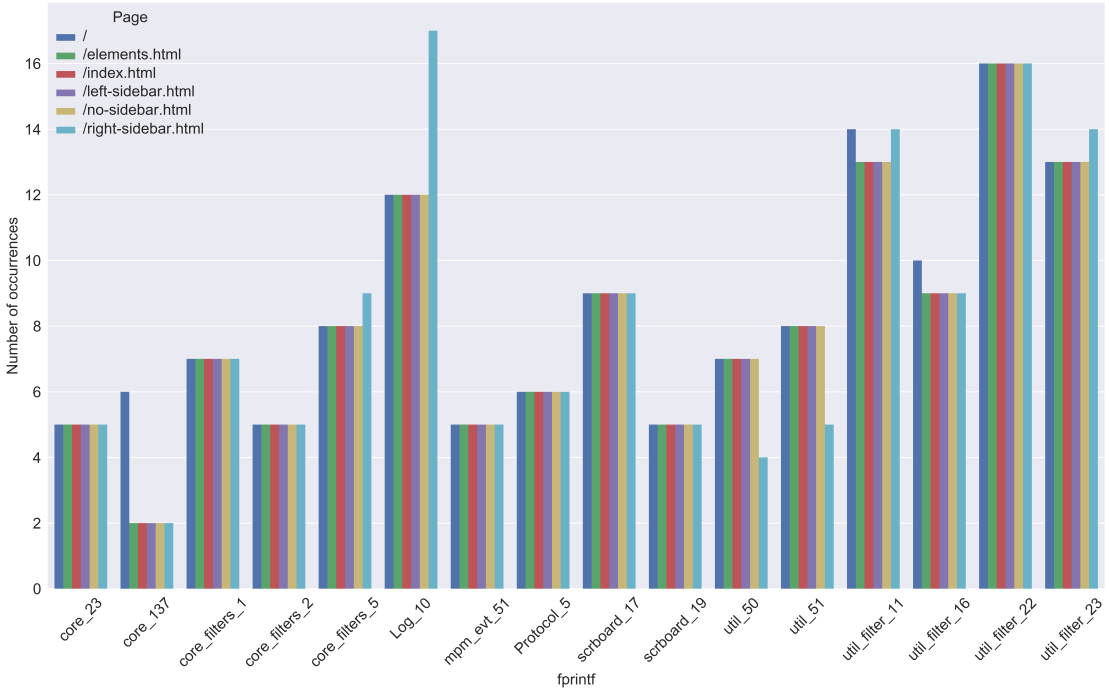


Figure 4.5: Plot of `fprintf` data with more occurrences from Apache httpd Server for each HTTP request.

4.2.3 Fault Injection

As all the patches that will be used in the experiments are already created and chosen, we can start doing the FI experiments. As stated above, we will inject a fault in each run. Therefore, our experiments consist in two hundred and twenty runs for **Apache httpd Server** and for **NGINX Server**, as we decided to select eleven files in both applications and twenty patches in each file.

Each run should last at most thirty-seven minutes, as presented in Figure 4.1. This way, as already stated in Subsection 2.3.2, the selected monitoring tool, `Docker SDK for Python` gathers variables from the containers every second, there will be at least one thousand, nine hundred and eighty samples per run.

After running all the experiments in both machines presented in Table 4.1, we analyzed which failures occurred in each run. These failures follow a failure modes classification which is listed and described in Section 3.3. It is important to note that the failure modes classification was based on and adapted from the **CRASH scale** from [10]. We have also established a detection approach to detect if a given failure occurred. This detection approach is described in Section 3.3.

Afterwards, everything was ready to perform the Fault Injection campaign, consisting of the injection of two hundred and twenty faults in two different applications, Apache httpd and NGINX servers, replicating this campaign in two different machines. This way, the **total number of runs** is eight hundred and eighty, where each one can execute at most about thirty-three minutes, not counting the build time of the container. So, it will take about twenty-nine thousand and forty minutes to finish, which is equivalent to more than twenty days.

A summary of the results gathered from the FI campaign are presented in Table 4.7. More than one failure type can occur in a certain Fault Injection run (FIR) except the Hang and

the Crash, because we only consider the samples until the instant a hang has occurred. For every scenario ten golden runs (GR) and two hundred and twenty FI runs (FIR) were performed, as already mentioned in this section.

Table 4.7: Summary of the failures detected in the Fault Injection campaign in the Apache HTTP Server and NGINX Server scenarios.

Application	**Machine**	**GR**	**FIR**	**Failures**					
				Hang	**Crash**	**RA**	**Abort**	**Delay**	**Total**
Apache httpd	I	10	220	28	10	14	46	58	156
	II	10	220	28	10	14	46	125	223
NGINX Server	I	10	220	24	28	12	57	25	146
	II	10	220	25	28	14	56	95	218

When analyzing the results presented in Table 4.7, it can be verified that the average occurrence of failures was approximately one failure per run in the case of both applications running in Machine II. On the other hand, the number of failures detected in Machine I was lower which in turn is related with the number of runs where delays occurred. This turns out to be a bad result when compared with the results from Machine II.

As the failure data was already generated, collected and classified, we can move on to the next specific goal, which consists in selecting and submitting the adequate data to a failure prediction algorithm and evaluate how effective it is at predicting the failures generated by the faults injected.

This page is intentionally left blank.

Chapter 5

Experimental Campaign on Containers Failure Prediction

To evaluate the effectiveness of the existing failure prediction algorithms in containers environment, we devised an experimental campaign. The goal was to understand how well these algorithms fared in this context, and to understand if the results they obtained were aligned with the ones obtained in other domains.

First, it was necessary to select the data by analyzing the **Fault Injection Runs (FIR)** for each failure mode and selecting the ones that are more suitable for this part of the work. This way, the failure modes consisted in the ones presented and identified in the selected FIR. Then, we needed to verify which variables have more information for training and testing the Failure Prediction (FP) model, discarding the remaining variables.

After the data selection is performed, it was necessary to configure the Failure Prediction (FP) setup by choosing a supervised machine learning algorithm to classify the samples. We used the Machine Learning (ML) tool *Propheticus* [11] to perform all the FP experiments necessary for this work. This way, it was required to select the prediction parameters for the FP algorithm and ML techniques for feature selection and data balancing of the datasets.

Finally, the results were obtained by running the Failure Prediction (FP) experiments and then analyzed and discussed. A comparison was performed with the results from Irrera & Vieira [1] in order to assess how effective was the chosen FP configuration and algorithm.

5.1 Data Selection

In the supervised learning, the first step resides in analyzing and treating the data, being one of the most important steps. If this step is performed incorrectly, it will influence the future results because it will wrongly train the algorithms or classifiers.

We chose the supervised learning because the goal is to use classification algorithms, where all samples are labeled with non-failure or with one failure mode (Crash, Hang, Abort, Repeated Abort, Delay). As presented in Subsection 2.2.4, the goal of supervised learning is the creation of a classifier through the mapping of the input into a requested output.

This way, we started by verifying which Fault Injection (FI) runs were the best candidates

to be used in FP. In order to achieve this, we decided to not use runs where all requests failed, because there is non-failure initial data to train the model. Also, we will not use runs where no failure was detected, as this does not indicate with absolute certainty that no failure occurred. Lastly, the runs that the application crashed at the first request will not be considered too. The number of runs that will be used for predicting is presented in Table 5.1, for both applications scenarios: Apache httpd and NGINX Server.

As all the crashes occurred at the first request, these are not valid Crash data to train and test the models so, we decided to discard the failure mode Crash from this part of the work. This can be directly related to the representativity and complexity of the applications selected, i.e. in other applications it could be possible to detect crashes later in the execution of the runs.

Also, we verified that the way we are detecting the failure mode Delay is not totally "accurate", because it can be detected in golden runs too. In order to reduce the underlying complexity and the time required to run the Failure Prediction experiments, we decided to discard this failure mode too, because it was the one with more occurrences.

Therefore, the failure modes that will be used in this part of the work are Abort, Hang and Repeated Abort.

Table 5.1: Number of Fault Injection runs selected for Failure Prediction in Apache HTTP Server and NGINX Server.

Application	Machine	FIR	FIR selected (%)
Apache httpd	I	220	14 (6.36 %)
	II	220	93 (42.27 %)
NGINX Server	I	220	12 (5.45 %)
	II	220	56 (25.45 %)

Analyzing Table 5.1, we can observe that the Fault Injection runs (FIR) selected from Machine I are much less than the ones from Machine II. This is directly related with the number of runs where at least a Delay was detected, with many more being observed on the Machine II.

We have also performed an analysis of the collected variables. We observed that many of them are incrementers or accumulators, which is confirmed by the metrics documentation from Docker [87]. This may become a problem as they can influence the model to make false predictions, because the model can associate the increase of a metric value to a given failure, which could be wrong. As stated in [88], the illegitimate usage of data can be considered data leakage. We decided also to discard some variables that are always equal to zero and others that present information like the number of CPUs available.

Therefore, we decided to discard those variables, having the direct advantage of decreasing the time complexity. Table 5.2 presents the total number of metrics and the number of discarded metrics per Docker metrics' groups. From now on, we will consider only eighty-seven variables from the initial one hundred and forty-nine.

However, there is a problem when collecting all BLKIO variables in the Apache httpd scenario. It happened in some golden runs (GR), four in Machine I and one in Machine II, and also in some FI runs, one in Machine I and thirty-five in Machine II. All or most of the corresponding values are equal to -1 in those runs. In order to resolve this problem, one of the following approaches may be used:

Table 5.2: Variables discarded per variable group.

Variables Group	# Variables	# Discarded Variables (%)
Block I/O	96	32 (33.3 %)
CPU	9	9 (100 %)
Memory	34	11 (32.4 %)
Network	8	8 (100 %)
Other	2	2 (100 %)
Total	**149**	**62 (41.6 %)**

- Discard those runs, which means that there will remain thirteen FIR from Machine I and fifty-eight from Machine II, as we can see in Table 5.2;
- Discard all block I/O variables from all golden and FI runs, leading to a decrease of the total number of eighty-seven variables to only twenty-three, as presented in Table 5.2;
- Keep the runs and all block I/O variables, as this problem can occur at any time when collecting these metrics, so the model will be more prepared to handle this when it occurs.

As the first and second approaches greatly reduce the information available for training the model, we decided to keep the runs and all block I/O variables. Thus, we will be training the model with errors that can arise in the variables stated when monitoring the containers, being in this way more realistic.

5.2 Failure Prediction Setup

Due to time restrictions, we decided to only use the **Support Vector Machine (SVM)** as the supervised machine learning algorithm. We started by coding in Python programming language using useful libraries and packages from scikit-learn [89] to perform small experiments and to learn more how it works. Then we used the tool *Propheticus* as already stated in Section 3.4.

In order to select the parameters which seem to be the best or the most appropriate for the SVM, we performed some small experiments with the FI data and feature and instance selection techniques, the kernel [90] and the kernel coefficient (γ) [90].

As the time available was less and less, we did those small experiments only in one random combination of Δt_l (*lead-time*) and (*prediction period*), fifty and thirty seconds respectively. Thus, four different kernels were tested: Linear, Polynomial, Radial Basis Function (RBF) and Sigmoid. For each kernel, small experiments were performed in order to get the best data balancing techniques (Random Oversampling, Random Undersampling, Synthetic Minority Over-sampling Technique (SMOTE)) or otherwise, none of them, and also, the best value for the (γ) parameter. It is noteworthy that each kernel has its own parameters, where can be equal to the parameters of other kernel. For example, as stated in [90], Polynomial kernel has a parameter named degree, and the kernel coefficient (γ) is for RBF, Polynomial and Sigmoid kernels.

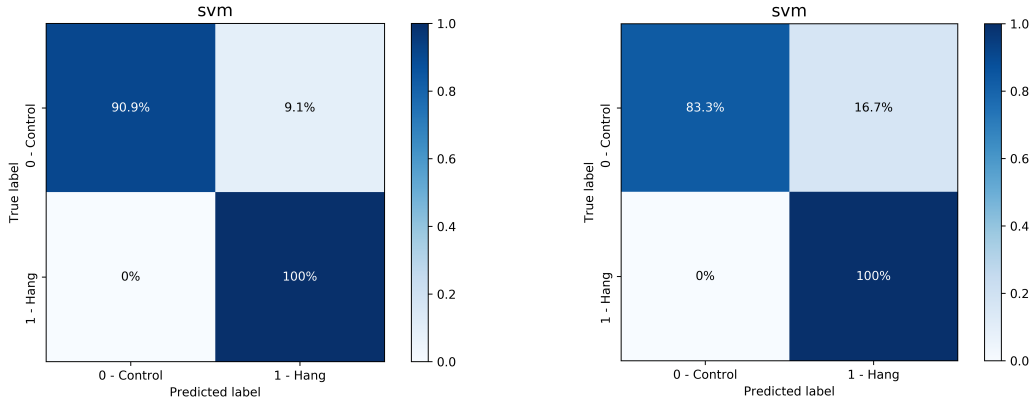
Due to time restrictions, we could not test for different feature selection methods and different values for the C parameter [90]. This way, we decided to use the Recursive

Feature Elimination (RFE) and variance methods for feature selection and the default value for the C parameter ($C = 1.0$) as presented in [90].

As stated in [90], it is important to note that the the time complexity associated with the fit of the model to the training data is greater than the quadratic complexity. So, it will take a lot of time for larger datasets containing more than ten thousand samples, as presented in [90].

The average number of samples in the four datasets generated through fault injection is approximately thirty-five thousand, seven hundred and ninety-eight samples. The average number of samples was calculated taking into account only the runs related with the three failure modes already mentioned in Subsection 5.1 (Abort, Hang and Repeated Abort), and all the golden runs. Therefore, only one dataset from the initial four was used, being the one generated in the NGINX Server and Machine I scenario.

The Support Vector Machine (SVM) is a binary supervised classifier [43], so the experiments will performed separately for each failure mode, even being possible to make multiclass classification with SVM. Figure 5.1 presents two examples of confusion matrices obtained for different configurations and for the failure mode Hang, while Figure 5.2 for the failure mode Abort, and lastly, Figure 5.3 for the failure mode Repeated Abort. These figures were created using the ML tool.



(a) Parameters: RBF kernel and $\gamma = 0.1$, Random Oversampling and Random Undersampling.

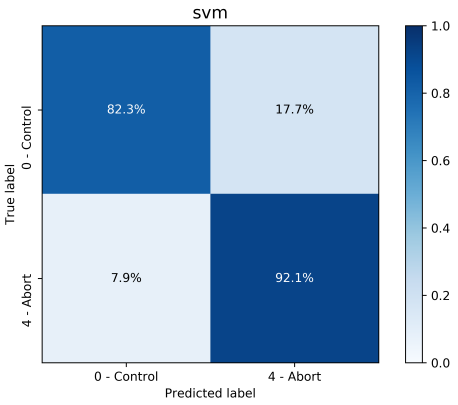
(b) Parameters: RBF kernel and $\gamma =' auto'$, Random Undersampling.

Figure 5.1: Examples of confusion matrices obtained using SVM for (Δt_l= 50, Δt_p= 30) and failure mode Hang.

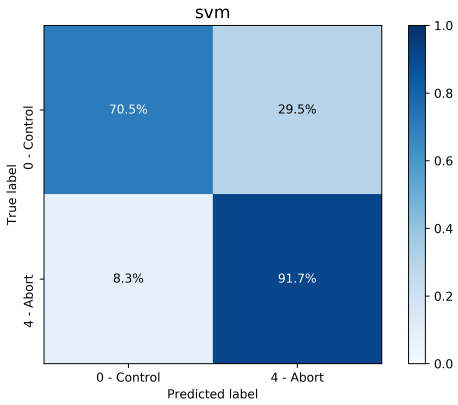
After analyzing the results, a tie was verified between the following parameter combinations:

- SVM with RBF kernel and $\gamma = 0.1$, Random Oversampling and Random Undersampling for Instance selection;
- SVM with RBF kernel and $\gamma = 0.1$, Random Undersampling and SMOTE for Instance selection;

Both combinations obtained the best results, taking into account the approach described above. The first will be used from now on, being summarized in Table 5.3, and the reason for this selection is due to the latter having the SMOTE for instance selection, which can add some overhead as it creates synthetic examples of the minority classes [42]. The

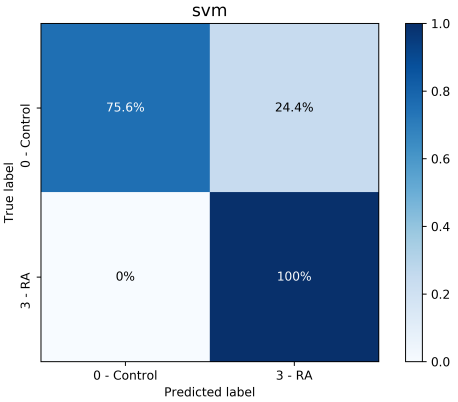


(a) Parameters: RBF kernel and γ = 0.1, Random Oversampling and Random Undersampling.

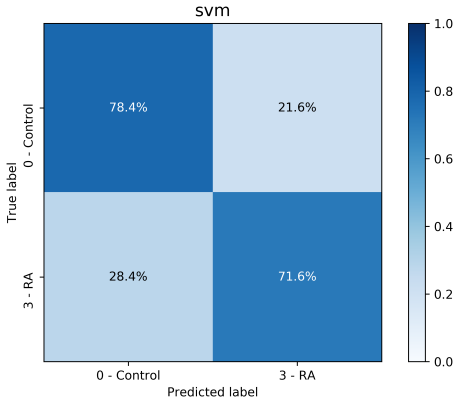


(b) Parameters: Polynomial kernel and $degree = 2$, Random Oversampling and SMOTE.

Figure 5.2: Examples of confusion matrices obtained using SVM for (Δt_l= 50, Δt_p= 30) and failure mode Abort.



(a) Parameters: RBF kernel and γ = 0.1, Random Oversampling and Random Undersampling.



(b) Parameters: RBF kernel and γ = 0.1, Random Undersampling.

Figure 5.3: Examples of confusion matrices obtained using SVM for (Δt_l= 50, Δt_p= 30) and failure mode Repeated Abort.

confusion matrices corresponding to the selected parameters are presented in Figure 5.1a for Hang, in Figure 5.2a for Abort, and lastly, in Figure 5.3a for Repeated Abort (RA).

Figure 5.4 presents the Receiver Operating Characteristics (ROC) curves obtained for each failure mode using the selected parameter combination with the dataset created in the NGINX Server and Machine I scenario. The Receiver Operating Characteristics (ROC) curves were also created using the Machine Learning (ML) tool. When analyzing Figure 5.4, it can be seen that the average area under the curve when considering the failure mode Hang is 0.99 (Figure 5.4b) which is a good result, and in the case of other two failure modes, the average areas are not bad, being 0.91 (Figure 5.4a) and 0.88 (Figure 5.4c).

Nevertheless, when analyzing Figure 5.5, which presents the Precision/Recall curves created using the ML tool, we can verify that the precision is low for all the three failure modes. In the case of the Hang, Figure 5.5b, the precision can be equal to 1.0, however the corresponding recall is low.

On the other hand, the other kernels did not obtain interesting results, having confused

Table 5.3: Parameters of the Failure Prediction campaign.

Parameter	Values
Failure Modes	Hang, Abort, Repeated Abort
Supervised Classifiers	SVM (kernel=RBF, $\gamma = 0.1$, $C = 1.0$)
Δt_l (lead-time)	10, 20, 30, 40, 50 seconds
Δt_l (prediction period)	10, 15, 20 seconds
Sliding Window	3 seconds
Feature Selection	RFE, Variance
Instance Selection	Random Oversampling, Random Undersampling
Results Validation	5-fold cross-validation

the failure data with the non-failure data, performing a bad classification. There are cases that all or almost all samples are classified as a failure or as control (non-failure). This can be explained by the weak parameters selection which can lead to over-fitting or under-fitting. However, the RBF kernel was the one with better results taking into account the parameters and techniques chosen.

For the Failure Prediction campaign we will consider only Δt_l (*lead-time*) and Δt_p (*prediction period*), leaving aside Δt_w (*minimal warning time*) in order to reduce the complexity of this task. The set of values that will be used for Δt_l and for Δt_p are presented in Table 5.3, which consist in the nine Δt_l and Δt_p pairs with best F-Measure results presented in Table 5.5 (b) from *Case Study: Benchmarking different failure prediction models*, from the PhD Thesis *Fault Injection for Online Failure Prediction Assessment and Improvement* from Irrera & Vieira [1]. Thus, a comparison between the results obtained and the results from Irrera & Vieira [1] will be performed in order to access if the SVM with the selected parameters predicts well the failures induced by the faults injected.

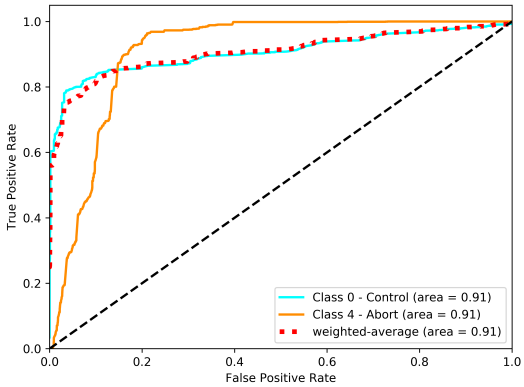
Lastly, to validate the results, we will use the **k-fold cross-validation** with five folds, i.e. dividing the samples into five equal size subsets and then, using each one at a time for validation and the others for training. The final result will be composed by the average of the five results.

As all the parameters necessary to conduct the Failure Prediction experimental campaign are listed in Table 5.3, we are able to move to the last step of this work, which is composed by gathering the results to verify the effectiveness of the selected Machine Learning algorithm with the parameter combination from Table 5.3. As already discussed, the results will be compared with the results from Irrera & Vieira [1].

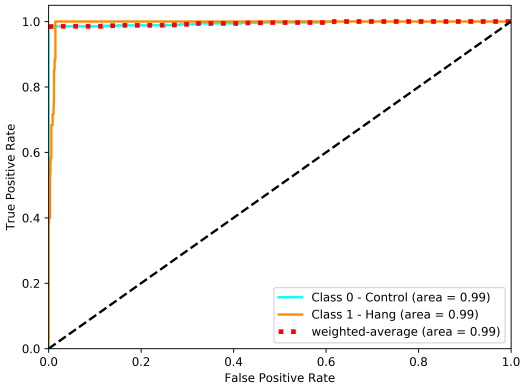
5.3 Results and Discussion

In order to perform the Failure Prediction (FP) experiments, it was necessary to execute, for each pair of Δt_l (*lead-time*) and Δt_p (*prediction period*) and for each failure mode, presented in Table 5.3, 30 runs to be able to draw conclusions and to generalize. The Machine Learning tool uses a different seed in each run to generate random values when necessary, for example, SVM needs it when shuffling the data [90].

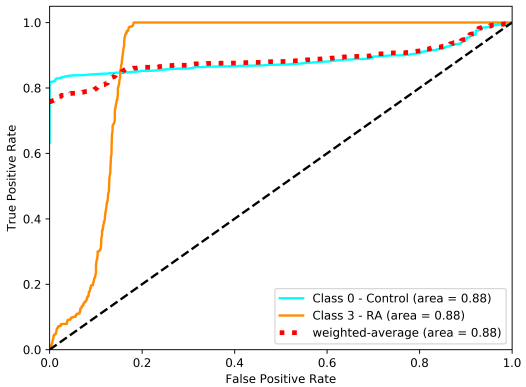
Also, we needed to repeat it for each dataset created by generating failure and non-failure data, as already described in Chapter 4: Apache httpd Machine I (**A1**) and Apache Machine II (**A2**), NGINX Server Machine I (**N1**) and NGINX Server Machine II (**N2**). And also considering and not considering the sliding window. Thus, three thousand, two



(a) Failure mode Abort.



(b) Failure mode Hang



(c) Failure mode Repeated Abort (RA)

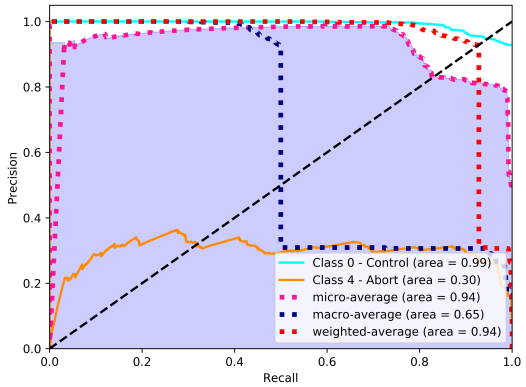
Figure 5.4: ROC curves corresponding to each failure mode using the selected parameters.

hundred and forty runs were performed using the classifier and configuration parameters from Table 5.3.

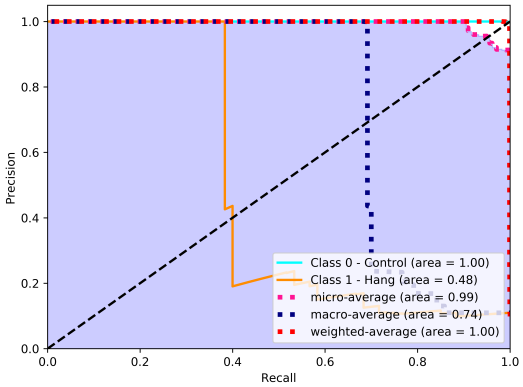
For the sliding window of three seconds, only the failure mode Hang will be considered, due to time constraints. This way, it was also necessary to execute one thousand and eighty additional runs.

The results obtained from running all the FP experiments are presented in Table 5.4. The **F-measure** can also be named as F-score, and consists in the harmonic mean of precision and recall as stated in [91]. When this measure is close to 1, it means that both precision and recall are very high, i.e. close to 1 too. Nevertheless, when the precision and/or the recall are closer to 0, it will lead to the F-measure be close to 0. The precision and the recall were already introduced in Subsection 2.2.4

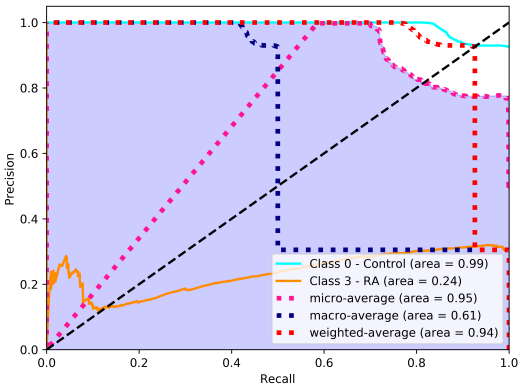
When analyzing Table 5.4, we verified that most of the values obtained for this measure are very low, which may reveal a rather weak prediction with the selected parameters. We also verified that there is only one very similar F-measure value when comparing with Irrera & Vieira [1], being the one from Table 5.4d, (Δt_l=40, Δt_p=20) and dataset A2. The corresponding F-measure value from Irrera & Vieira [1] is 0.963. Figure 5.6 presents the corresponding **Confusion Matrix**, **ROC Curve** and **Precision-Recall Curve** and was created by using the ML tool. Taking into account the results from all the thirty runs, the number of True Positives (TP) is 84038, False Positives (FP) is 142, False Negatives (FN) is 9 and the number of True Positives (TP) is 1191. These values reveal that the failure prediction model was very accurate at classifying the non-failures and the failures.



(a) Failure mode Abort.



(b) Failure mode Hang



(c) Failure mode Repeated Abort (RA)

Figure 5.5: Precision/Recall curves corresponding to each failure mode using the selected parameters.

The dataset A2 from Table 5.4d is the one with overall better results when comparing with all the F-score values obtained, as all the corresponding F-measure values are higher than 0.8. We can also conclude by analyzing Table 5.4 that the same dataset is the one with best results in all scenarios. This, in turn, may be highly related with the selection of this dataset to test and choose the parameters to be used in this work.

Therefore, the F-measure results obtained are very weak. Nevertheless, the number of samples with failure data is very low when compared to the number of samples with non-failure data (control data). This way, even if the percentage of false positives is very low, its number can be much larger than the number of true positives, leading to a low precision which in turn leads to a low F-score. This situation was observed frequently along the pairs of Δt_l and Δt_p and the datasets.

This way, if the proportions of samples containing failure and non-failure were more balanced, the results could be more satisfactory. However, changing the proportions can lead to completely different results when the FP model predicts, because the datasets information can be very divergent.

It is important to note that the Failure Prediction (FP) results obtained in this work can only reveal that the parameters and data balancing methods chosen were not the most adequate. As already stated, we could not test for several prediction parameter combinations due to time constraints.

The proportions of samples containing failure data and samples containing non-failure data

Table 5.4: F-measure values obtained for each set of Δt_l and Δt_p.

Δt_l	Δt_p	**Datasets**			
		A1	**A2**	**N1**	**N2**
10	20	0.274	0.685	0.443	0.399
20	20	0.245	0.649	0.517	0.377
30	15	0.327	0.622	0.427	0.374
30	20	0.255	0.607	0.456	0.334
40	15	0.272	0.603	0.425	0.384
40	20	0.255	0.577	0.437	0.325
50	10	0.327	0.571	0.396	0.379
50	15	0.258	0.574	0.431	0.343
50	20	0.259	0.557	0.455	0.294

(a) Failure mode Abort, no sliding window

Δt_l	Δt_p	**Datasets**			
		A1	**A2**	**N1**	**N2**
10	20	0.030	0.702	0.012	0.055
20	20	0.020	0.712	0.009	0.057
30	15	0.032	0.706	0.007	0.036
30	20	0.049	0.683	0.008	0.061
40	15	0.014	0.664	0.004	0.027
40	20	0.013	0.821	0.010	0.041
50	10	0.006	0.733	0.006	0.011
50	15	0.010	0.670	0.012	0.027
50	20	0.014	0.824	0.018	0.031

(b) Failure mode Hang, no sliding window

Δt_l	Δt_p	**Datasets**			
		A1	**A2**	**N1**	**N2**
10	20	0.152	0.551	0.354	0.630
20	20	0.138	0.529	0.359	0.605
30	15	0.131	0.494	0.341	0.582
30	20	0.140	0.523	0.356	0.590
40	15	0.134	0.484	0.344	0.562
40	20	0.147	0.507	0.359	0.570
50	10	0.095	0.406	0.323	0.535
50	15	0.120	0.454	0.346	0.543
50	20	0.130	0.503	0.363	0.540

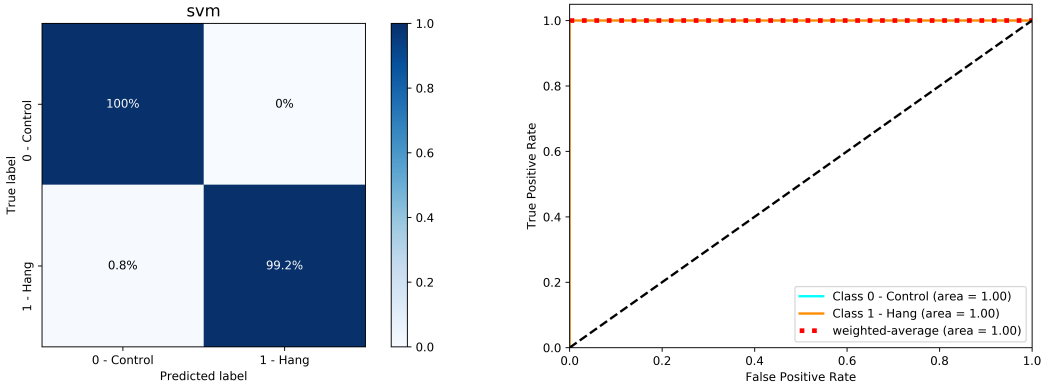
(c) Failure mode Repeated Abort (RA), no sliding window

Δt_l	Δt_p	**Datasets**			
		A1	**A2**	**N1**	**N2**
10	20	0.035	0.836	0.012	0.050
20	20	0.030	0.824	0.008	0.078
30	15	0.027	0.854	0.005	0.090
30	20	0.026	0.852	0.009	0.122
40	15	0.053	0.825	0.008	0.036
40	20	0.011	0.940	0.009	0.045
50	10	0.005	0.855	0.011	0.031
50	15	0.009	0.842	0.007	0.041
50	20	0.037	0.907	0.017	0.028

(d) Failure mode Hang, sliding window 3s

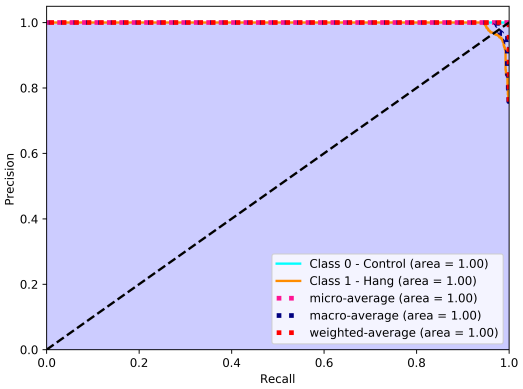
turned out as a problem. As the Support Vector Machine (SVM) is a binary classifier, we have separated the failures which increased the proportion of non-failure data per dataset.

Therefore, the comparison of the results with the results from Irrera & Vieira [1] reveal that most of the results are bad. Only one F-measure value was very identical. Although it is very unlikely, the chosen supervised learning classifier, SVM, may work for Virtual Machines (VMs) and not to Docker containers. In order to evaluate this, it is necessary to test with different parameter combinations, and also, with other Failure Prediction (FP) methods as already stated.



(a) Confusion Matrix.

(b) ROC Curve



(c) Precision-Recall Curve

Figure 5.6: Classification results for ($\Delta t_l = 40$, $\Delta t_p = 20$, A2, $window = 3s$).

Chapter 6

Conclusions and Future work

This work has the main objective of assessing the feasibility of applying Online Failure Prediction (OFP) to containerized micro-services-based applications. These applications have the necessary characteristics to make OFP methods applicable in practice.

For this, it was necessary to select monitoring solutions to be able to collect metrics from those applications, with minimum intrusiveness. The comparison performed between the monitoring tools revealed that `Docker SKD for Python` was the tool with better results when considering the relevance and quantity of the metrics compared with past work on Online Failure Prediction (OFP). However, the vast majority of the metrics are quite different from the ones traditionally used.

A representative setup was defined in order to make the comparison of the types of the monitored variables variations across two different system configurations. The comparison was accomplished by using the statistical test Mann-Whitney U test. The results obtained from the comparison revealed that more than half of the monitored variables seem to have different distribution.

Fault injection was used to generate representative failure data to be used for training and testing failure prediction models. An approach was conducted with the objective of increasing the representativeness of the faults used in this work, thus increasing the probability of faults being activated more often. A failure modes classification was created to classify five different failure modes.

A Failure Prediction (FP) experimental campaign was performed using the supervised learning classifier Support Vector Machine (SVM), and the prediction parameters and data balancing techniques selected by following an approach which attempted to choose the best parameter set. In order to assess how effective the Failure Prediction (FP) method was when predicting the failures caused by the injected faults. The results show that the prediction parameters and the data balancing methods were not the most appropriate. Also, the number of samples containing non-failure data was very large when compared with the number of failures to predict. This turned out as a problem even having a very low rate of false positives, because the number of false positives will tend to be much bigger than the number of true positives.

Therefore, the chosen Failure Prediction (FP) approach revealed a satisfactory effectiveness when predicting the failures. Nevertheless, the proportion of samples containing failures and non-failures in combination with a not very suitable parameters selection, led to the observed results.

Future work includes applying these methodologies to micro-service based applications, replacing the web application used in this study. Injecting fault in applications with different representativeness and complexity may lead to different failure modes detected.

It also includes broadening the application of fault injection techniques to several tools, comparing them, to mitigate the impact that the tool may have in the generated data. The impact of those faults must be studied and understood in order to compare these techniques and to choose the one with the best results.

The Failure Prediction (FP) experiments were performed using only one FP method with the objective of predicting the induced failures. Other FP methods could be used to allow the comparison of their capacity and effectiveness when predicting failures.

Finally, we plan to complete and submit a scientific publication containing the results obtained from the Fault Injection (FI) and Failure Prediction (FP) experiments.

References

[1] Ivano Irrera. *Fault Injection for Online Failure Prediction Assessment and Improvement*. PhD thesis, 2016.

[2] Felix Salfner and Miroslaw Malek. Architecting dependable systems with proactive fault management. *Architecting dependable systems VII*, pages 171–200, 2010.

[3] M. Jamshidi. *Systems of Systems Engineering: Principles and Applications*. CRC Press, 2017.

[4] Docker Inc. What is a Container. `https://www.docker.com/what-container`. Accessed: 2017-12-28.

[5] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. Performance evaluation of microservices architectures using containers. In *Network Computing and Applications (NCA), 2015 IEEE 14th International Symposium on*, pages 27–34. IEEE, 2015.

[6] Imesh Gunaratne. The Evolution of Linux Containers and Their Future - DZone Cloud. `https://dzone.com/articles/evolution-of-linux-containers-future`. Accessed: 2018-01-18.

[7] J. Arlat, Y. Crouzet, and J. C. Laprie. Fault injection for dependability validation of fault-tolerant computing systems. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 348–355, 1989.

[8] Docker Inc. What is Docker? `https://www.docker.com/what-docker`. Accessed: 2017-12-30.

[9] Goncalo Silva Pereira. Evaluating the robustness of the Cloud. In *Evaluating the robustness of the Cloud*, 2016.

[10] Philip Koopman, Kobey Devale, and John Devale. *Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project*, chapter 11, pages 201–226. Wiley-Blackwell, 2008.

[11] João R. Campos. Tools - Propheticus. `http://joaodecampos.com`, 2018. Accessed: 2018-09-02.

[12] Jean-Claude Laprie. Dependable computing: Concepts, limits, challenges. In *Special Issue of the 25th International Symposium On Fault-Tolerant Computing*, pages 42–54, 1995.

[13] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[14] Fares A Nassar and Dorothy M Andrews. *A methodology for analysis of failure prediction data*. Center for Reliable Computing, Computer Systems Laboratory, Depts. of Electrical Engineering and Computer Science, Stanford University, 1985.

[15] Felix Salfner, Maren Lenk, and Miroslaw Malek. A survey of online failure prediction methods. *ACM Computing Surveys (CSUR)*, 42(3):10, 2010.

[16] Ralph M Singer, Kenny C Gross, James P Herzog, Ronald W King, and Stephen Wegerich. Model-based nuclear power plant monitoring and fault detection: Theoretical foundations. Technical report, Argonne National Lab., IL (United States), 1997.

[17] Guenther Hoffman and Miroslaw Malek. Call availability prediction in a telecommunication system: A data driven empirical approach. In *Reliable Distributed Systems, 2006. SRDS'06. 25th IEEE Symposium on*, pages 83–95. IEEE, 2006.

[18] Ting-Ting Yao Lin. Design and evaluation of an on-line predictive diagnostic system. 1988.

[19] Tadashi Dohi, Katerina Goseva-Popstojanova, and Kishor S Trivedi. Analysis of software cost models with rejuvenation. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 25–34. IEEE, 2000.

[20] Tim Zwietasch. Online failure prediction for microservice architectures. Master's thesis, 2017.

[21] David Martin Powers. Evaluation: from precision, recall and f-measure to roc, Informedness, Markedness and Correlation. 2011.

[22] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[23] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *[1989] The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pages 340–347, 1989.

[24] Ghani A Kanawati, Nasser A Kanawati, and Jacob A Abraham. Ferrari: A tool for the validation of system dependability properties. In *FTCS*, pages 336–344, 1992.

[25] Timothy K. Tsai and Ravishankar K. Iyer. Measuring fault tolerance with the ftape fault injection tool. In Heinz Beilner and Falko Bause, editors, *Quantitative Evaluation of Computing and Communication Systems*, pages 26–40, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[26] Joao Carreira, Henrique Madeira, João Gabriel Silva, et al. Xception: Software fault injection and monitoring in processor functional units. *Dependable Computing and Fault Tolerant Systems*, 10:245–266, 1998.

[27] E. v d Kouwe and A. S. Tanenbaum. HSFI: Accurate Fault Injection Scalable to Large Code Bases. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 144–155, 2016.

[28] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[29] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. In *New advances in machine learning*. InTech, 2010.

[30] SB Kotsiantis, D Kanellopoulos, and PE Pintelas. Data preprocessing for supervised leaning. *International Journal of Computer Science*, 1(2):111–117, 2006.

[31] Charu C. Aggarwal. Outlier Analysis. In *Data Mining*, pages 237–263. Springer, Cham, 2015.

[32] Jerzy W Grzymala-Busse and Ming Hu. A comparison of several approaches to missing attribute values in data mining. In *International Conference on Rough Sets and Current Trends in Computing*, pages 378–385. Springer, 2000.

[33] Kamakshi Lakshminarayan, Steven A Harp, and Tariq Samad. Imputation of missing data in industrial databases. *Applied intelligence*, 11(3):259–275, 1999.

[34] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[35] Rosaria Silipo, Iris Adae, Aaron Hart, and Michael Berthold. Seven techniques for dimensionality reduction. *White Paper by KNIME. com AG*, pages 1–21, 2014.

[36] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[37] Pat Langley et al. Selection of relevant features in machine learning. In *Proceedings of the AAAI Fall symposium on relevance*, volume 184, pages 245–271, 1994.

[38] Muhammad Aliyu Sulaiman and Jane Labadin. Feature selection based on mutual information. In *IT in Asia (CITA), 2015 9th International Conference on*, pages 1–6. IEEE, 2015.

[39] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. Gene selection for cancer classification using support vector machines. *Machine learning*, 46(1-3):389–422, 2002.

[40] Jose Ramon Cano, Francisco Herrera, and Manuel Lozano. *Strategies for Scaling Up Evolutionary Instance Reduction Algorithms for Data Mining*, pages 21–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[41] Bee Wah Yap, Khatijahhusna Abd Rani, Hezlin Aryani Abd Rahman, Simon Fong, Zuraida Khairudin, and Nik Nik Abdullah. An application of oversampling, undersampling, bagging and boosting in handling imbalanced datasets. In *Proceedings of the First International Conference on Advanced Data and Information Engineering (DaEng-2013)*, pages 13–22. Springer, 2014.

[42] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. Smote: synthetic minority over-sampling technique. *Journal of artificial intelligence research*, 16:321–357, 2002.

[43] Luis Gonzalez, Cecilio Angulo, Francisco Velasco, and Andreu Catala. Unified dual for bi-class svm approaches. *Pattern Recognition*, 38(10):1772–1774, 2005.

[44] Armin Shmilovici. Support vector machines. In *Data mining and knowledge discovery handbook*, pages 231–247. Springer, 2009.

[45] Shiyu Ji. A training example of svm with kernel given by $\varphi((a,b)) = (a, b, a^2 + b^2)$. `https://en.wikipedia.org/wiki/Support_vector_machine#/media/File:Kernel_trick_idea.svg`. Accessed: 2018-08-15.

[46] Peter D Turney. Types of cost in inductive concept learning. *arXiv preprint cs/0212034*, 2002.

[47] David H Wolpert, William G Macready, et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[48] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[49] Pedro Domingos. A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning*, pages 231–238, 2000.

[50] Andy Field. *Discovering statistics using IBM SPSS statistics*. sage, 2013.

[51] thildred. The History of Containers. `https://rhelblog.redhat.com/2015/08/28/the-history-of-containers/`. Accessed: 2018-01-08.

[52] C. Anderson. Docker [Software engineering]. *IEEE Software*, 32(3):102–c3, 2015.

[53] UpGuard. Docker vs CoreOS Rkt. `https://www.upguard.com/articles/docker-vs-coreos`. Accessed: 2018-08-09.

[54] UpGuard. Docker vs LXC. `https://www.upguard.com/articles/docker-vs-lxc`. Accessed: 2018-08-09.

[55] Canonical Ltd. Linux Containers - LXC - Introduction. `https://linuxcontainers.org/lxc/introduction`. Accessed: 2018-08-09.

[56] Docker Inc. Docker Store. `https://store.docker.com`. Accessed: 2018-08-09.

[57] CoreOS. rkt, a security-minded, standards-based container engine. `https://coreos.com/rkt`. Accessed: 2018-08-09.

[58] CoreOS. CoreOS. `https://github.com/coreos`. Accessed: 2018-08-12.

[59] Docker Inc. Develop with Docker Engine SDKs and API. `https://docs.docker.com/develop/sdk/`. Accessed: 2017-12-31.

[60] Google. cAdvisor: Analyzes resource usage and performance characteristics of running containers. `https://github.com/google/cadvisor`. Accessed: 2018-01-02.

[61] CoScale. Docker Monitoring. `https://www.coscale.com/docker-monitoring`. Accessed: 2018-01-02.

[62] Dynatrace. Docker Monitoring. `https://www.dynatrace.com/technologies/cloud-and-microservices/docker-monitoring/index-language.html`. Accessed: 2018-01-03.

[63] elastic. Metricbeat. `https://www.elastic.co/products/beats/metricbeat`. Accessed: 2018-01-04.

[64] elastic. Elasticsearch. `https://www.elastic.co/products/elasticsearch`. Accessed: 2018-01-04.

[65] elastic. Docker module | Metricbeat Reference [6.1] | Elastic. `https://www.elastic.co/guide/en/beats/metricbeat/6.1/metricbeat-module-docker.html`. Accessed: 2018-01-04.

[66] Nagios. Nagios Overview. Nagios.org. `https://www.nagios.org/about/overview/`. Accessed: 2018-01-03.

[67] timdaman. Check Docker - Nagios Exchange. https://goo.gl/yAFWKT. Accessed: 2018-01-03.

[68] New Relic. New Relic: New Relic Docker Monitoring. https://newrelic.com/partner/docker. Accessed: 2018-01-03.

[69] Sensu. Sensu | Monitoring Platform. https://sensuapp.org/features. Accessed: 2018-01-03.

[70] Sensu. sensu-plugins-docker: This plugin provides native Docker instrumentation for monitoring and metrics collection, including: container status, container number, and container metrics via 'docker ps'. https://github.com/sensu-plugins/sensu-plugins-docker. Accessed: 2018-01-03.

[71] Sysdig. Container Troubleshooting and Linux Visibility | Sysdig. https://www.sysdig.org/wiki/sysdig-overview/. Accessed: 2018-01-03.

[72] Datadog. Getting Started with Datadog. https://docs.datadoghq.com/integrations/docker_daemon/. Accessed: 2018-01-09.

[73] Sematext. Docker Monitoring & Logging. https://sematext.com/docker/. Accessed: 2018-01-09.

[74] Pingdom. Docker Monitor ˜ Pingdom Server Monitor. http://server-monitor.pingdom.com/plugin_urls/19761-docker-monitor. Accessed: 2018-01-09.

[75] Apache Software Foundation. Welcome! - The Apache HTTP Server Project. https://httpd.apache.org. Accessed: 2018-08-20.

[76] Inc NGINX. NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy. https://www.nginx.com. Accessed: 2018-08-20.

[77] PostgreSQL Global Development Group. PostgreSQL: The world's most advanced open source database. https://www.postgresql.org. Accessed: 2018-08-20.

[78] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[79] Html5webtemplates.co.uk. Html5 Webtemplate Elements. https://www.html5webtemplates.co.uk/templates/elements. Accessed: 2018-07-10.

[80] Docker Inc. debian - Docker Store. https://store.docker.com/images/debian. Accessed: 2018-07-14.

[81] Docker Inc. Docker SDK for Python — Docker SDK for Python 2.0 documentation. https://docker-py.readthedocs.io/en/stable. Accessed: 2018-07-14.

[82] curl. curl - How To Use. https://curl.haxx.se/docs/manpage.html. Accessed: 2018-08-21.

[83] SciPy. scipy.stats.mannwhitneyu — SciPy v1.1.0 Reference Guide. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html#scipy.stats.mannwhitneyu. Accessed: 2018-08-19.

[84] ucXception. Software Faults | ucXception. https://ucxception.dei.uc.pt/index.php/software-faults. Accessed: 2018-07-14.

[85] J. A. Duraes and H. S. Madeira. Emulation of Software Faults: A Field Data Study and a Practical Approach. *IEEE Transactions on Software Engineering*, 32(11):849–867, 2006.

[86] J. Duraes and H. Madeira. Definition of software fault emulation operators: a field data study. In *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, pages 105–114, 2003.

[87] Docker Inc. Runtime metrics | Docker Documentation. `https://docs.docker.com/config/containers/runmetrics`. Accessed: 2018-08-22.

[88] Shachar Kaufman, Saharon Rosset, Claudia Perlich, and Ori Stitelman. Leakage in data mining: Formulation, detection, and avoidance. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(4):15, 2012.

[89] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[90] scikit learn. sklearn.svm.SVC — scikit-learn 0.19.2 documentation. `http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html#sklearn.svm.SVC`. Accessed: 2018-08-31.

[91] Yutaka Sasaki et al. The truth of the f-measure. *Teach Tutor mater*, 1(5):1–5, 2007.

Appendices

Appendix A

Faults Generated per Application

Table A.1: Faults generated for each `C` code file from Apache httpd `server` source folder.

`C` Code File	Patches
util_buildmark.c	0
config.c	434
util_connection.c	0
core.c	1385
core_filters.c	228
eoc_bucket.c	1
eor_bucket.c	13
error_bucket.c	3
gen_test_char.c	28
listen.c	83
log.c	341
main.c	474
mpm/event/event.c	1207
mpm/event/fdqueue.c	189
mpm/prefork/prefork.c	512
mpm/worker/fdqueue.c	146
mpm/worker/worker.c	609
mpm_common.c	105
mpm_unix.c	121
protocol.c	442
provider.c	1
request.c	209
scoreboard.c	54
util.c	188
util_cfgtree.c	9
util_charset.c	0
util_cookies.c	15
util_debug.c	0
util_ebcdic.c	0
util_expr_eval.c	312
util_expr_parse.c	264
util_expr_scan.c	550
util_fcgi.c	0
util_filter.c	83
util_md5.c	0
util_mutex.c	11
util_pcre.c	0
util_regex.c	0

C Code File	Patches
util_script.c	71
util_time.c	9
util_xml.c	0
vhost.c	169
Total	**8266**

Table A.2: Faults generated for each C code file from NGINX core source folder.

C Code File	Patches
nginx.c	457
ngx_array.c	24
ngx_buf.c	147
ngx_conf_file.c	670
ngx_connection.c	384
ngx_cpuinfo.c	0
ngx_crc32.c	10
ngx_crypt.c	0
ngx_cycle.c	707
ngx_file.c	439
ngx_hash.c	422
ngx_inet.c	373
ngx_list.c	26
ngx_log.c	168
ngx_md5.c	968
ngx_module.c	74
ngx_murmurhash.c	46
ngx_open_file_cache.c	527
ngx_output_chain.c	257
ngx_palloc.c	130
ngx_parse.c	242
ngx_parse_time.c	127
util_proxy_protocol.c	115
util_queue.c	20
util_radix_tree.c	127
util_rbtree.c	181
util_regex.c	87
util_resolver.c	2478
util_rwlock.c	0
util_sha1.c	2157
util_shmtx.c	18
ngx_slab.c	544
ngx_spinlock.c	0
ngx_string.c	752
ngx_syslog.c	134
ngx_thread_pool.c	256
ngx_times.c	292
Total	**13359**

Table A.3: Faults generated for each `C` code file from NGINX `events` source folder.

`C` Code File	Patches
ngx_event.c	421
ngx_event_accept.c	384
ngx_event_connect.c	131
ngx_event_openssl.c	1972
ngx_event_stapling.c	0
ngx_event_pipe.c	435
ngx_event_posted.c	9
ngx_event_timer.c	39
Total	**4535**

Table A.4: Faults generated for each `C` code file from NGINX `http` source folder.

`C` Code File	Patches
ngx_http.c	869
ngx_http_copy_filter_module.c	62
ngx_http_core_module.c	2091
ngx_http_file_cache.c	1798
ngx_http_header_filter_module.c	269
ngx_http_parse.c	1217
ngx_http_postpone_filter_module.c	41
ngx_http_request.c	1567
ngx_http_request_body.c	601
ngx_http_script.c	748
ngx_http_special_response.c	301
v2/ngx_http_upstream.c	2307
v2/ngx_http_upstream_round_robin.c	635
v2/ngx_http_v2.c	2446
v2/ngx_http_v2_filter_module.c	1289
v2/ngx_http_v2_huff_decode.c	11
v2/ngx_http_v2_huff_encode.c	52
v2/ngx_http_v2_module.c	147
v2/ngx_http_v2_table.c	156
v2/ngx_http_variables.c	1238
v2/ngx_http_write_filter_module.c	146
Total	**17991**

Table A.5: Faults generated for each `C` code file from PostgreSQL `backend` source folder.

`C` **Code File**	**Patches**
lib/binaryheap.c	70
lib/bipartite_match.c	56
lib/dshash.c	295
lib/hyperloglog.c	30
lib/ilist.c	4
lib/knapsack.c	39
lib/pairingheap.c	77
lib/rbtree.c	203
lib/stringinfo.c	60
main.c	231
nodes/bitmapset.c	216
nodes/copyfuncs.c	4279
nodes/equalfuncs.c	2001
nodes/extensible.c	19
nodes/list.c	346
nodes/makefuncs.c	286
nodes/nodeFuncs.c	1569
nodes/nodes.c	0
nodes/outfuncs.c	1196
nodes/params.c	112
nodes/print.c	198
nodes/read.c	119
nodes/readfuncs.c	7093
nodes/tidbitmap.c	468
nodes/value.c	4
Total	**18971**

Table A.6: Faults generated for each `C` code file from PostgreSQL `bin` source folder.

`C` Code File	Patches
initdb/findtimezone.c	188
initdv/initdb.c	1424
pg_archivecleanup.c	120
pg_basebackup/pg_basebackup.c	947
pg_basebackup/pg_receivewal.c	395
pg_basebackup/pg_recvlogical.c	597
pg_basebackup/receivelog.c	376
pg_basebackup/streamutil.c	238
pg_walmethods.c	323
pg_config.c	126
pg_controldata.c	297
pg_ctl.c	653
pg_dump/common.c	722
pg_dump/compress_io.c	39
pg_dump/dumputils.c	306
pg_dump/parallel.c	303
pg_dump/pg_backup_archiver.c	1779
pg_dump/pg_backup_custom.c	252
pg_dump/pg_backup_db.c	253
pg_dump/pg_backup_directory.c	270
pg_dump/pg_backup_null.c	70
pg_dump/pg_backup_tar.c	452
pg_dump/pg_backup_utils.c	23
pg_dump/pg_dump.c	3466
pg_dump/pg_dump_sort.c	506
pg_dump/pg_dumpall.c	947
pg_dump/pg_restore.c	466
pg_resetwal.c	923
pg_rewind/copy_fetch.c	46
pg_rewind/datapagemap.c	25
pg_rewind/fetch.c	3
pg_rewind/file_ops.c	95
pg_rewind/filemap.c	264
pg_rewind/libpq_fetch.c	218
pg_rewind/logging.c	36
pg_rewind/parsexlog.c	88
pg_rewind/pg_rewind.c	375
pg_rewind/timeline.c	73
pg_test_fsync.c	206
pg_test_timing.c	74
pg_upgrade/check.c	449
pg_upgrade/controldata.c	462
pg_upgrade/dump.c	48
pg_upgrade/exec.c	104
pg_upgrade/file.c	109
pg_upgrade/function.c	65
pg_upgrade/info.c	301
pg_upgrade/option.c	239
pg_upgrade/parallel.c	21
pg_upgrade/pg_upgrade.c	318
pg_upgrade/relfilenode.c	45
pg_upgrade/server.c	106
pg_upgrade/tablespace.c	21
pg_upgrade/util.c	54
pg_upgrade/version.c	115
pg_waldump/compat.c	6

C Code File	Patches
pg_waldump/waldump.c	370
pg_waldump/rmgrdesc.c	0
pgbench.c	2373
pgevent.c	17
psql/command.c	1246
psql/common.c	812
psql/conditional.c	21
psql/copy.c	233
psql/crosstabview.c	272
psql/describe.c	2921
psql/help.c	1016
psql/input.c	19
psql/large_obj.c	121
psql/mainloop.c	309
psql/prompt.c	124
psql/startup.c	567
psql/stringutils.c	97
psql/tab-complete.c	923
psql/variables.c	124
scripts/clusterdb.c	202
scripts/common.c	198
scripts/createdb.c	231
scripts/createuser.c	308
scripts/dropdb.c	152
scripts/dropuser.c	133
scripts/pg_isready.c	158
scripts/reindexdb.c	282
scripts/vacuumdb.c	430
Total	**34086**