A Failure Detector Model Using Reinforcement Learning: A Cloud Computing Experiment

Maikel Dal Farra (20235851) BSc Computer Science

School of Computer Science

National University of Ireland Galway

Supervisors

Dr. Enda Howley

In partial fulfillment of the requirements for the degree of

MSc in Computer Science (Artificial Intelligence - Online)

May 2020

DECLARATION I, Maikel Dal Farra, do hereby declare that this thesis entitled A Failure Detector Model Using Reinforcement Learning: A Cloud Computing Experiment is a bonafide record of research work done by me for the award of MSc in Computer Science (Artificial Intelligence - Online) from National University of Ireland Galway. It has not been previously submitted, in part or whole, to any university or institution for any degree, diploma, or other qualification.

Signature: _____

To Lígia.

Thank you, my beloved wife, for years of partnership, agreements and thoughtful disagreements. You taught me a lot and made my life much better, and I will forever be grateful.

Acknowledgements

I would like to thank my supervisor D. Enda Howley. I can not thank you enough for all of the support you've given me. Your insight has helped me to find a focus in the right direction. Not to mention, I appreciated your input throughout the process. It significantly aided my work and helped me create a much better outcome.

I thank my family, especially my parents and my grandmother. You were always there for me. Supporting me and wanting the best for me. Thanks for believing in me. Especially in difficult times, I hope you know how much I appreciate it.

I also thank the board of lecturers, especially our lecturer and programme director Dr. James McDermott. All of you played an essential role in the formation of our knowledge and are part of this entire process that begins in our professional life.

Abstract

Distributed systems and serverless applications have become increasingly popular and highly adopted for the majority of successful business with the advent of cloud computing. Interconnected systems are key elements that rule the cloud mechanism. The complexity and overhead associated with this kind of system need to be managed, ensuring reliability and scalability.

The more companies discover the advantages of cloud computing, the more crucial it is for the business to being online and communicates with the cloud. Whether the customer needs any solution related, big cloud computing players already envisioned the opportunity and have their own products to offer. On the other hand, companies have created their own systems to minimise the risk of disruption.

This work discusses a practical solution for researchers and scientists to develop an effective failure detector using artificial intelligence (AI) models for cloud computing projects. An intelligent system failure detector may be able to learn from performance parameters and fit gracefully into the existing network architecture, therefore not affecting any system functionality.

Keywords: reinforcement learning, cloud computing, failure detection, failsafe systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The cloud is an important part of many technology companies strategy, whether storing data, running complex platforms, or providing infrastructure. The cloud is also the gateway to a new generation of artificial intelligence tools that will becoming more capable every year. The rise of services based on A.I. is fundamentally changing computer architecture. To put in perspective, this tendency is making possible for personal assistance and machine learning to be everywhere Lauterbach and Bonim (2016).

Cloud computing is also helping make services more reliable, and affordable. As stated by Lauterbach and Bonim (2016), the cloud went from a mere tool aiming to improve information technology economics to a game-changer with the power to transform the economy in less than a decade. "It delivers on-demand computing to any user who has access to the Internet." (Lauterbach and Bonim, 2016, p.2). There is a growth towards interconnected services since the beginning of cloud computing. In addition, the ability to scale up resources and scale out applications is one of the major benefits of migrating applications to the public cloud. Within this context, performance, scalability, and reliability are the true measures of the strength and success in any cloud system Liu (2011).

Reliability is one of the utmost concern for those building distributed systems, and it demands appropriate planning. One form of providing reliability is to introduce redundancy and formulate ways to handle failures gracefully. As formulated by Pannu et al. (2012), the success of cloud computing is up to its capacity to provide dependability at scale.

The greater the delay in response, the less reliable the system. According to Gupta et al. (2016), a failure to achieve the response on time results in a degraded response and the softness of the system is a measure of its delay. On the other hand, faulty equipment needs to be driven to return active and operative again without compromising the entire system. Failsafe is a state in which equipment or a system will not operate if a failure occurs. This mechanism is describe by Yan et al. (2021) as typically implemented with multiple redundant safety devices, some of which must fail before others engage to keep the critical system running. In addition, failsafe systems play a crucial role in monitoring systems availability, independent of human supervision. These can be designed to be able to continue running even after experiencing an issue.

Particularly, distributed systems have a fundamental strain. The way the whole system works as a unified entity depends upon messages reliably arriving at the right times and in the right order. However, networks are unpredictable. Failures may happen, even massive damages, but also ordinary failures. In a complex distributed system, the challenge is to maintain the sanity of the core as long as possible, regardless of the failures of its nodes (processes).

Independently how perfect is the infrastructure or how high skilled are the staff in charge of the business; there will always be failures. Besides, the companies are aware of the risks involving an incorrect failure detection policy. However, they are not the only ones. Cloud vendors are also providing solutions for handling this issue.

The purpose of this work is using Artificial Intelligence to acknowledge the problem of failure detection for Cloud Computing. The key to managing this situation is the ability to monitor the state of each node in a distributed system in anticipation of a possible degraded instance. The cluster task can remain operational long enough that the unhealthy instance can be replaced without interrupting normal business uses of the system. Which is something the companies have done using either their own systems or hybrid solutions provided by cloud vendors. In that regard, the proposed approach is using Reinforcement Learning to address the problem.

Chapter 2

Background and Literature Review

2.1 Cloud Computing

Cloud computing can be stated as a natural evolution of the local software services provided by companies on their premises. It can be briefly described as software, data storage, and processing handled remotely in a server that can be anywhere globally, aiming to provide computing as a service rather than a product as it was previously.

Conforming to Mell and Grance (2011) in the U.S. Recommendations of the National Institute of Standards and Technology publication, this computing paradigm might speed up business while reducing your IT costs and improve capacity to promptly create and deploy safe and secure enterprise solutions. "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." (Mell

and Grance, 2011, p.12). On top of it, the model is based on five essential characteristics, three service models, and four deployment models.

Essential characteristics:

- On-demand self-service. A consumer can provide computing resources such as server time and network storage on their own. These can be achieved automatically and without human interaction with any service provider.

- Broad network access. In the past, every database was installed in its own room and maintained by a distinct person (or a group) who detained the knowledge to do so. Nowadays, databases are on the network, which is also where the majority of the applications are. In addition to that, many applications create and update their own databases, so the database whereabouts it is unknown. These applications are accessed through standard mechanisms supported by so-called thin or thick client platforms.

- Resource pooling. Services are typically provided to users through a multi-tenant model. This model is essentially ubiquitous, highly available and on-demand, sharing a pool of computing resources which can be quickly provisioned and released with the smallest technical effort or service provider interaction. Examples of resources include storage, processing, memory, and network bandwidth.

- Rapid elasticity. Within this context, elasticity means the rapid allocation and de-allocation of resources in order to support business demands. Giving the customer the feeling that there are unlimited resources provided appropriately in any quantity at any time.

- Measured service. Resource usage is optimized according to each customer

demand. It can be monitored, gauged, controlled and reported granting transparency for both the provider and consumer of the used service.

Service Models:

- Software as a Service (SaaS). The Cloud Provider job is to supply every service to the customer, which means that cloud customers simply use the application provided by the cloud provided to accomplish their goals. The cloud provider controls all the infrastructure, such as network, servers, operating systems, storage, or even individual application capabilities.

- Platform as a Service (PaaS). This is a middle-tier option available to the customers, which are able to deploy their application and services, created using programming languages, libraries, services, and tools on top of the middleware components managed by the cloud provider.

- Infrastructure as a Service (IaaS). This is the lowest level of deployment that customers can request from a cloud provider, for instance, the customer can deploy any type of Virtual Machine (VM) they desire. Through this approach, customers end up with the most control over the cloud infrastructure. On the other hand, cloud providers are capable of provision processing, storage, networks, and other fundamental computing resources enabling the customer to do their job.

Deployment Models:

- Private cloud. This environment is considered a secure option, as long as the cloud infrastructure used is dedicated for exclusive use by a single organization. It may exist on or off-premises, maintained, controlled, and operated by the organization, a third party, or some combination.

- Community cloud. It is similar to a private cloud, with the main difference that it is provisioned for use by a specific group of cloud customers that have concerns in common (e.g., mission, security requirements, policy, and compliance considerations). It may exist on or off-premises, maintained, controlled, and operated by one or more organizations in the community, a third party, or some combination.

- Public cloud. As the name states, it is provisioned for use by general public. Cloud customers applications, regardless of data classification, implemented security controls, industry regulations, or any other external factor, exist on the same physical hardware as other cloud customers. Besides, it always exists on the premises of the cloud provider. Some of the well-known public cloud services include Amazon EC2, Amazon Web Services and Microsoft Azure, just to name a few.

- Hybrid cloud. Are combinations of mostly public and private clouds that allow data systems to be moved between the different environments almost perfectly. One common situation can occur when a load within a private cloud exceeds a certain threshold. Then, the customer can temporarily move some of the non-critical assets to a public cloud to alleviate the load. Once the load returns to normal, the migrated assets can be moved back into the private cloud.

Regularly, in cloud computing environments, applications can run in a container or a VMs as nodes within a system. These nodes might interact accordingly to provide so-called, microservices. This approach suggests modular components communicating with each other through APIs. In line with Kakivaya et al. (2018), normally, the loose coupling in an API based communication between microservices implies isolating the side-effects of a failure to only specific components.

It allows developers to more easily reason about the fault-tolerance of each microservice. On the other hand, a monolithic cloud application may have different parts affected, making fault-tolerance analysis considerably complex.

2.1.1 System Outage

According to Wang et al. (2020), multiple services can have the same functionality while changing their non-functional properties, so-called Quality of Service (QoS), known as an important criterion for service selection or recommendation. On top of that, the main reason for runtime anomalies of distributed component services is because the QoS values of component services may vary dynamically due to hardware/software updates or server workload changes. "Therefore, QoS value prediction may be inaccurate, and it is unavoidable that the unknown factors can cause some fluctuations. In other words, the performance of the service cannot always be consistent. Therefore, without proper modeling of volatile QoS values, the recommendation may be invalid. Then service selection for composite services may have unsatisfactory result." (Wang et al., 2020, p.1).

In fact, some services can suddenly fail and may need to be replaced by new ones, in other words, recovery. Therefore, quick response and adjustment are vital in a dynamic environment. For instance, in a simple Contact Center scenario, the connected nodes may guarantee the communication flowing through the network at a certain quality, previously determined by business policies.

Basically, the cloud can integrate into the telephony infrastructure, providing information from a CRM system about actors involved. It may also afford interaction with various message servers, publish/subscribers, databases and so on. Besides, it is extremely important that all the actors are available and performing as expected during the process, to ensure the system ability to acknowledge, process, track and report every type of communication received. Bearing in mind

that "Each task demands different amounts of computing and transmission resources, and each device owns different computing and transmission resources. Such heterogeneous nature leads to competition among multiple cloud centers to achieve the best latency performance" (Zhang et al., 2021, p.2).

According to Yan et al. (2020, p.2), "System reliability measures the ability or probability of the system to complete the specified function within the specified time and under the specified working conditions". It turns out that all systems eventually fail. The tradeoff between reliability and ability to react quickly is the key for successful cloud systems. In their work, Panev and Latkosky discussed some implication of failure. In a failure case, to guarantee the system's performance, it is essential to act proactively, providing alternative paths. Therefore, when a failure occurs, the failure detection mechanism should redirect the system without increasing signaling overhead. Panev and Latkoski (2020).

When thinking about failure in different contexts such as client-driven, asynchronous, replicated state machines, on systems in general, the first step is how to detect if a node has gone offline and taken the appropriate action. Identifying a machine outage at an application layer, in other words an inconsistency between parts has been a topic of open-ended research on several distributed systems articles.

2.2 Fault-tolerance

Fault-tolerant and recovery methods has been one of the important research topics in distributed systems. One of the main reasons that fault detection and recovery are bringing more attention over time is because many cloud computing services are very failure prone, including multi-tenancy, dynamic resource allocation, and dynamic code updates. The frequency of computer failures as well as

the impact on performance is an increasingly important concern to cloud stakeholders. Hence, the demand for fault detection and recovery in cloud computing has become increasingly significant over recent years.

Most methods of failure detection rely on statistical learning models to understand what is causing a problem and when it might happen again. The basic idea behind these methods is that the training dataset is labeled, the training set is annotated such that it's possible to tell whether or not a failure event has occurred.

In general, labeled data are not always available in the real world. The deficiencies from unavailability of labeled data are especially evident in newly managed or deployed systems in which labelling is required. The current methods rely on historical data to recognize failures but do not consider how failures might occur in the future. Also, undetected failures are mostly not considered. Pannu et al. (2012).

2.2.1 Consistency

In their work, Ahsan and Gupta emphasizes the importance of providing consistency, which means a timing guarantee for how long two nodes can stay mutually inconsistent. This is crucial in many applications that we are dependant on nowadays, such as banking, stock markets, air traffic control, vehicle routing, etc. Consistent recovery actions need to be performed on these applications in an appropriate manner at various nodes. Reminding that any concurrent incorrect action by different existing nodes may cause severe application errors. Ahsan and Gupta (2020).

Considering that failure detectors are important for the continuity of the service and the circumstances are, in general, complicated, this work proposes using AI to detect and manage whether the system failure has occurred. Using rein-

forcement learning (RL), an agent can monitor the state of a node in a system. RL agents work by trial-and-error cases and learn by rewarding over time. The expectation is to compare an RL agent outcome with a very simple timeout based system and determine whether the RL agent can be more assertive in perceiving whether the node is offline and notifying the components accordingly.

2.3 Reinforcement Learning

Reinforcement learning is a learning from experience model in which an agent is going to be given a set of rewards or punishments to be able to learn how to perform sequential decision tasks by interacting with the environment. This means it learns what to do or what not to do in the future, based on its previous experiences. In short, the agent receives its input sequentially and is required to perform actions that maximize the reward signal in the long run. Van Otterlo and Wiering (2012).

This mechanism of states and actions and rewards was formalised as what is known as a Markov Decision Process.

2.3.1 Markov Decision Process

The intuition behind is that the agent is situated in their environment, where the environment is where they are going to make decisions. And its what is going to give them rewards or punishments for those decisions.

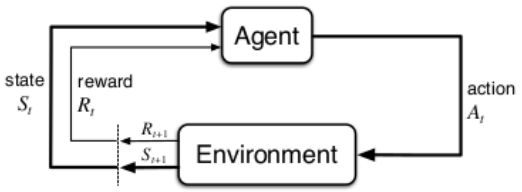

Figure 2.1: The agent–environment interaction
Markov Decision Process. Sutton and Barto (2018)

The environment is going to start by placing the agent in a state S_t. The state could be a game state that the agent is playing for instance. The agent, in turn, is in some sort of state, and it needs to choose to take an action A_t, among multiple actions they can choose from. As a result, the agent will usually ends up with two things in return, a new state S_{t+1}, and also they are given some sort of numerical reward R_t, either positive or negative, the negative one means punishment. And that is, in a sense, all the information the agent has available Sutton and Barto (2018).

The agent only receives one input at each time step, as opposed to batch learning, where the agent is provided with a whole sequence of inputs in a single interaction. These steps comply with the RL policy learning process, where sequential decision tasks refer to those that can be split down into a set of independent sub-tasks, which are learned one by one. To use reinforcement learning in the real world, it is usually needed for the agent to interact, or experiment with the environment and collect statistics about how its actions, gaining experience with everything that might affect the environment. Van Otterlo and Wiering (2012).

2.3.2 Episodic and Continuous Mechanism

As explained in their seminal introductory book, Sutton and Barto stated that the agent is operating overtime and producing sequences of actions aiming to maximize the cumulative reward received. More formally, the aim is to maximize the expected return, denoted by G_t. In addition, T is the final step, which is usually a random variable that can vary each episode, and the sequence of rewards indicated by R_{t+1}, R_{t+2}, R_{t+3} ,..., are received after time step t. Sutton and Barto (2018).

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T,$$

Figure 2.2: The return formulation.
Sutton and Barto (2018)

In each stage of each episode, an action is selected, and some reward is received. Ultimately there is a transition to a new state. The idea is to divide the interaction into episodes and then interpret the solution to each episode independently. This is, basically, the intuition behind *Periodic Tasks*.

With respect to the next concept, the interaction does not usually happen toward episodes and the maximization of rewards does not happen immediately. In fact, it is a sum of rewards over time, where γ is a parameter, $0 \leq \gamma \leq 1$, called the discount rate that determines the present value of future rewards. The return just signified in this notation by G_t, which is basically a sum.

Ultimately, the agent's goal is to update these values to select actions that maximize in expectation this long-term cumulative reward. Also the interaction does not end up with identifiable episodes but goes on continually as a sum over an infinite number of terms Sutton and Barto (2018). This is the intuition behind *Continuous Tasks*.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

Figure 2.3: The sum of the discounted rewards.
Sutton and Barto (2018)

There is an important function to start putting several of these pieces together. Meaning to estimate the reward value from taking an action in some state. The reasoning is presented below.

2.4 Q-Learning

By definition, *QLearning* is a model-free algorithm that aims to estimate the expected total of future rewards for an agent performing a certain action A in a state S. Model-free consists of learning the consequences of actions through experience, therefore adjusting the policy for optimal rewards. As opposed to Model-based, where an agent tries to understand the environment and create a model based on its interactions with this environment in a conservative way, so to speak.

The value of Q will be acknowledged over time, based on experience and trials that can lead to the learning process. Ultimately, the agent is expected to decide the best action to take. Overall, the best action is simply the action that results in the highest expected total return. Therefore, the agent might choose a policy that maximizes the future reward. The algorithm, as formulated by Sutton and Barto (2018) is the following:

Algorithm 1 Q-Learning

Initialize $Q(s, a)$ arbitrarily, and $Q(terminal - state) = 0$
for each episode **do**
    Initialize s
    for each step of episode **do**
        Choose a from s using policy derived from $Q(e.g., \epsilon - greedy)$
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a Q(s', a) - Q(s, a)]$
        $s \leftarrow s'$
    end for
end for

Q-Learning is know as an *off-policy* algorithm, given its exploratory nature. In other words, it takes different actions to achieve the most optimum policy. This is done by updating $Q - values$ by summing up the immediate reward to the discounted estimation of the action that returns the greatest reward.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big].$$

Figure 2.4: Q-learning. Sutton and Barto (2018)

To clarify, the future reward return can be determined by finding the *argmax* of the Q function over all possible actions at the state. Also, the learning rate, denoted by α, is responsible for adjusting the current value expectation of taking some action in a particular state. Lastly, the future rewards are being discounted, note the γ parameter, which means that future rewards are not as valuable as rewards immediately in this case.

Reminding that the agent's role is to explore solutions by itself rather than follow a predefined sequence of actions. Moreover, their decisions are solely based on the reward, however, this heuristic can be slightly tricky. For instance, if the agent takes a high level of reward for some action, it is also possible that some other action would have given more reward that was just not considered. It turns out that this is a challenge to explore the alternative actions while exploiting the information obtained so far, which is an inherent problem to RL.

In their paper, Dulac-Arnold et al. (2019) presented a number of interesting RL challenges. With respect to reward, although there is a discounted return taken into account, many research RL papers assess the average return to evaluate the quality of their agent. This makes sense for the context in which the optimization criteria is precisely the return itself. However, in the real world, it may be important to evaluate not only the average reward but also the performance for the worst case.

To put things in perspective, it is fair to say that every time the agent experience some new reward, it takes it into account. Thus, the estimation upon a certain action is updated. Consequently, in the future, the agent will be able to make decisions based on that information. One of the approaches to explore a

better estimate for every state and action is known as *ε-greedy Exploration Policy.* Zhang et al. (2021).

2.5 SARSA

This algorithm which stands for State–action–reward–state–action, was proposed by Rummery and Niranjan (1994). When comparing with Q-Learning, we notice some slightly differences. For instance, Q-Learning is more related to value iteration whilst Sarsa tends to be more of policy iteration. That means that the policy followed would affect value estimates. Thus, values estimates are calculated on top of observed rewards based on the selected actions.

In other words, it learns a sub-optimal policy while exploring. Besides, it pursue convergence allowing for penalties from exploratory actions. Which is known as an on-policy approach, different from the off-policy characteristic of Q-Learning. Essentially, Sarsa tend to be more *conservative* taking safe actions.

The cliff Walking image below from Sutton and Barto (2018) is perfect to explain the difference between on-policy (Sarsa) and off-policy (Q-Learning).

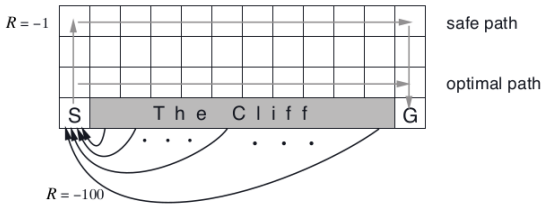

Figure 2.5: Cliff Walking task. Sutton and Barto (2018)

Briefly, the agent is suppose to walk from $S(tart)$ to $G(oal)$. The rewards are -1 for any transition and -100 if the agent falls off the cliff. Which is caused by an ϵ-greedy action eventually. The Sarsa path is the $safepath$, following its nature more conservative and consequently longer. In contrast, Q-Learning takes the *optimalpath*. This implies more risk. However, it gets to the goal first.

The results of the experiment can be seem in the following graphic:

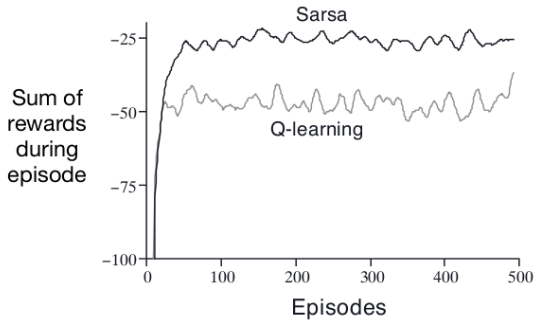

Figure 2.6: Cliff Walking task graph.
Sutton and Barto (2018)

The conclusion is that even though Q-Learning learns the optimal policy and walks on the cliff's edge, it is also being penalised more for having fallen off the cliff. Hence, its performance happens to be worse than Sarsa, which learns the safest policy. With this in mind, the assumptions about Sarsa can be perceived analysing the following algorithm, extracted from the Sutton and Barto (2018) book.

Algorithm 2 Sarsa

Initialize $Q(s, a)$ arbitrarily, and $Q(terminal - state) = 0$
for each episode **do**
    Initialize s
    Choose a from s using policy derived from $Q(e.g., \epsilon - greedy)$
    for each step of episode **do**
        Take action a, observe r, s'
        Choose a' from s' using policy derived from $Q(e.g., \epsilon - greedy)$
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    end for
end for

2.6 Related Work

Running any workload in the Cloud is challenging due to the high computational intensity, performance requirements and scalability of the application. The approach proposed by Sinha et al. (2019) uses VM migration between computing nodes during routine maintenance events to react to potential adverse events such as a hardware failure. Their system is based on a failure detector to manage non-functioning recoverable nodes on top of checkpoints to determine a healthy state point. Consequently, the program can be reverted to that state in case of any anomaly.

This is interesting, given that the instance is not automatically discarded after an interruption. It can be restored to a stable condition to work again. It is potentially a faster and cheaper process than instantiating a new node from scratch. According to the simulation, their results did fulfil expectations. The only caveat is the impact of unaccountable traffic blasts on precise fault detection, which needs further analysis.

Son and Buyya (2019) proposed taking the application latency into account to reduce the end-to-end network delay by using edge cloud resources. The system employs edge computing resources to respond to latency-sensitive applications, and cloud resources for less sensitive ones. Utilising Network Function Virtualization (NFV) to detect overloaded applications. When it happens, the algorithm duplicates the NFV at the same place if edge resources are appropriate.

Otherwise, the algorithm sets the new NFV in the cloud. After creating the new NFV, the algorithm creates a network latency map from the source node to the NFV in the cloud and on the edge. Ultimately, recently arrived packets are distributed by the new NFV according to the application latency state. The decision to forward packets to the edge is based on the latency requirement submitted by the application. The mechanism applies a binary classification that takes half

of the tighter delay as a critical and the other half as a normal application. The authors suggest that this can be improved by employing a linear approach.

The Adaptive Failure Detection (AFD), presented by Pannu et al. (2012) monitors the health states of physical servers, analysing performance metrics data of VMs. Therefore, AFD identifies possible failure states and reports them to operators for analysis. The outcomes happen to be input back to AFD for revamping. Briefly, AFD creates a hypersphere to categorise the majority of sample points, as such sample points outside the hypersphere are identified as potential failures. This work is in accordance with many studies that have been proposed in the literature for detecting faults based on certain thresholds and decision rules.

Chapter 3

Frozen Lake

3.1 Motivation

At this point, a practical experiment is proposed to highlight RL's key concepts and the mechanisms that govern it. For this purpose, the Frozen Lake problem is a feasible example of how an agent can interact with the environment and therefore learn its dynamics to achieve a particular goal. The aim is to observe the agent's behaviour performing both Q-Learning and Sarsa learning methods.

3.2 Summary

The problem we will use is relatively simple. We have an agent in a 5x5 grid world and observe its attempts to find the goal. A grid world environment is, by design, a matrix of numbers representing the state of each cell. It starts from the top-left position in the grid, and the goal is to find the bottom right. The agent can move in the four cardinal directions, "up", "down", "left", and "right". The environment has *holes* in the ice. Therefore, if the agent falls in, it will drown.

3.3 Setup

Given the set of possible directions, any action that causes it to land on a cell that is not its intended goal, meaning any regular move that has not reached the goal, implies a penalty of -1.0. Also, when falling into a hole, the penalty is more severe; -5.0 is applied in this case. On the other hand, when the target is achieved, the determined reward is +10.0 to encourage the agent to find the target.

In addition, other parameters are represented by the learning rate (alpha) which is responsible for updating weights: 0.5, and the discount factor (gamma) is 0.5. The step size is set to be 100 on top of 10000 training episodes and 5000 test episodes. Reminding that each episode is trained for each different epsilon, and *epsilon* means the probability of a random move as opposed to exploiting the utility function, for which *epsilon* and *max_epsilon* are set to 0.1, and *min_epsilon* to 0.001. Furthermore, there is an exponential decay rate equals to 0.001, acting in a way to balance exploration vs exploitation when trying to converge to the optimal action.

3.4 The Experiment

Each portion of the algorithms was carefully implemented to ensure correctness, on top of Q-Learning and SARSA algorithms presented in the Sutton and Barto (2018) book. Also, some graphs are showed for evaluation purposes, and their outcomes will be discussed accordingly.

Given that epsilon is the value that determines how much we explore in an ε-greedy policy, a decrease in epsilon values can be seen in Figure 3.1 Values are high at the beginning of the learning process when the agent knows little about the environment and then decrease with the number of steps performed until

reaching a plateau. Once the agent has performed an optimal policy, it does not waste steps exploring what is already known.

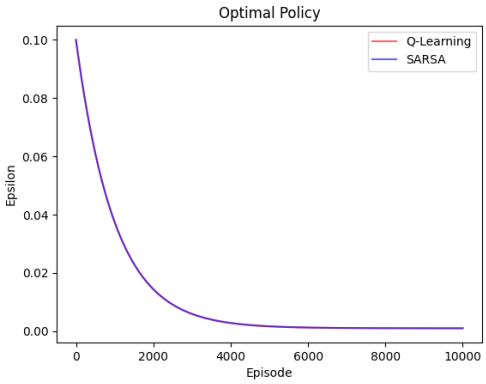

Figure 3.1: Epsilon curve.

The Figure 3.2 is an overview of the results of both algorithms during the training stage, representing the percentage of success. Scores were captured during a sampling of 500 episodes to get a sense of the behavior of each approach.

Figure 3.2: Train - Q-Learning vs SARSA

As expected, the score is very low in the training phase for both. The agent's choice of action appears to be quite random and modifications made to the epsilon

value don't appear to have any influence on the overall results. Indeed, agents are clearly not yet taking their environments into account. The mean value is perceived at 0.232 and 0.308 overall with Q-Learning performing slightly better.

In contrast, the testing phase portrays a more promising reality, where successful episodes have increased substantially. Agents met the goal more successfully over the episodes showing a consistent but gradual improvement. The mean value is now around 0.627 for SARSA and 0.413 for Q-Learning.

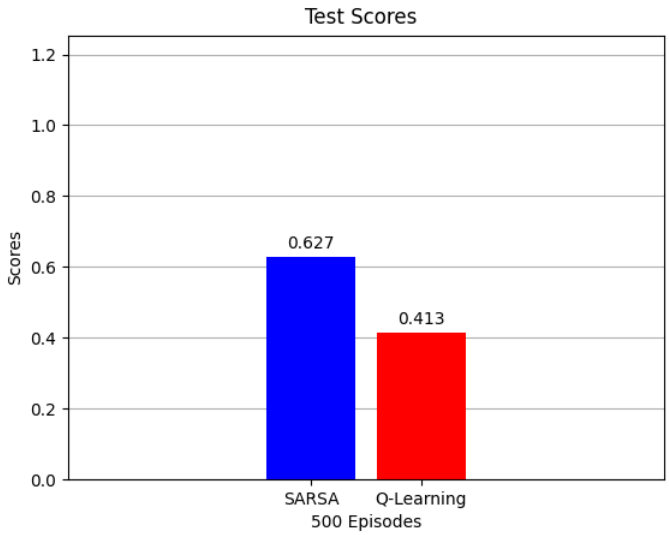

Figure 3.3: Test - Q-Learning vs SARSA

The training phase is a part of reinforcement learning that allows realizing the concept through testing. Both stages are equally important in the development and realization of a project. In the training phase, the algorithms for optimizing and achieving solutions for AI agents are designed. As such, this phase can hold instructions that will be carried throughout the testing phase.

Figure 3.4 endorses the same tendency that happened to the Gridworld experiment described in Sutton and Barto (2018) book. The confirmation that SARSA was more successful than Q-Learning is emphasized, mainly due to the conservative nature of the algorithm, which takes the action selection into ac-

count by design, and tends to minimize risks. As opposed to Q-Learning's more exploratory bias form that might prefer values for the optimal policy, so-called greedy action selection.

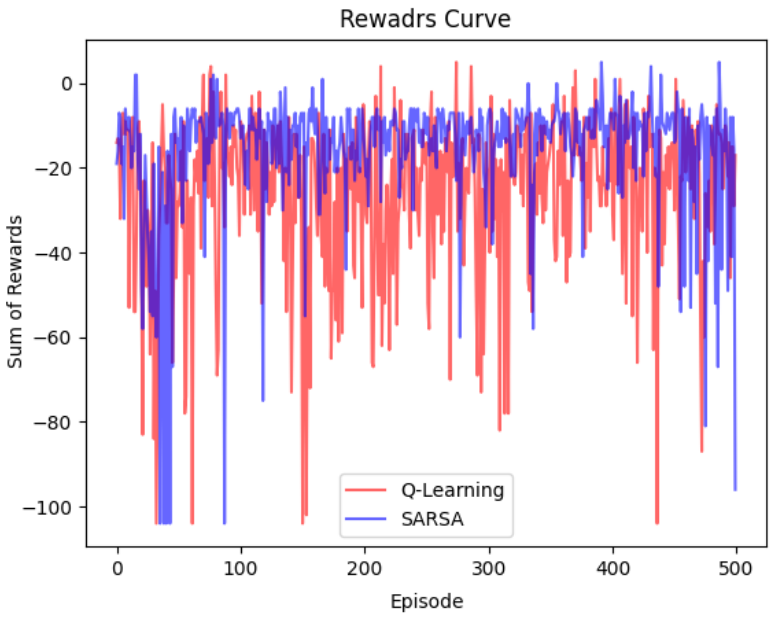

Figure 3.4: Q-Learning vs SARSA

Overall, the agent's behaviour seems to be quite reasonable. This can be explained by the fact that all agents' knowledge was focused to the extent that it let them achieve their best performance by the end of the training step. However, the number of training trials can be infinite, resulting in different results each time it is re-run.

Agents explored several new environments with different states and penalties. Consequently, different states were learned by trial and error, and they were able to maximize the expected reward by avoiding average states over time. Ultimately, the mission was accomplished and the agents reached the optimal exploration policy and exploited the best rewards.

Chapter 4

DynamicCloudSim

4.1 Overview

DynamicCloudSim is a framework presented by Bux and Leser (2015) for simulations of large-scale infrastructure on the cloud. Based on the work from Calheiros et al. (2011), the CloudSim simulator is an event-based simulation toolkit built in Java which allows researchers and developers to construct models of computer networks and apply them to different sorts of problems. For instance, evaluating the performance of network protocols simulating resource provisioning or analysing the impact of new network architectures by estimating operating costs for migration algorithms in data centers, given a VM topology.

The framework builds upon this model by placing additional runtime failures and perturbations inside the VMs themselves. It can model the influence of memory along with network bandwidth and CPU performance by inferring uncertainty at runtime. Hence it may more accurately portray the actual behaviour of complex applications. For that reason, taking advantage of different levels of capabilities of DynamicCloudSim and combining instability and service disruption to simulate workflow execution in dynamic cloud environments indicates to

be a more reasonable option for this work.

The research area of cloud simulation is growing over time. Hence the need to acknowledge the diversity of cloud environments. DynamicCloudSim proposes to address aspects of uncertainty more effectively than its peers. The authors claim that available cloud simulation toolkits rely on static models and infrastructure. Therefore they do not comprehend inhomogeneity and dynamic performance changes intrinsic to non-uniform and shared infrastructures.

4.2 Architecture

To put more context, the Figure 4.1 presents the CloudSim Architecture, which extends to DynamicCloudSim. Some of key aspects of the architecture are described as follows.

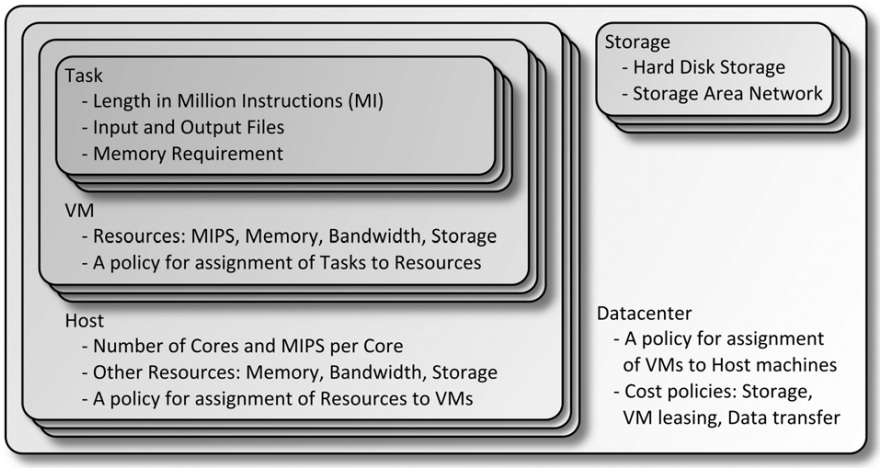

Figure 4.1: The architecture of the CloudSim framework. Calheiros et al. (2011)

4.2.1 Cloudlets

The simulation starts a number of data centers, each containing storage servers and physical host machines. A data center is a composite of storage servers

and host instances responsible for managing multiple VMs performing a number of tasks, known as *cloudlets*. In other words, Cloudlets represent the workload executed during the simulation. Hence the flow of control is distributed across a set of cloudlets and coordinates them so that hosts can dynamically join and leave the system, and cloudlets, in turn, are allowed to move across hosts dynamically. Calheiros et al. (2011)

4.2.2 Datacenter

The orchestration described previously is implemented on top of hosts, which governs the flow of VMs and maps its processing metrics. In fact, a Datacenter entity conducts the mapping process and is responsible for allocating hardware resources to all hosts and their respective VMs. It controls the use of memory, bandwidth, and storage devices and manages the hosts, ensuring access to the resources they need to function properly. Buyya et al. (2009)

4.2.3 Datacenter Broker

The Datacenter Broker class is accountable for handling interactions between applications and customers. Its primary purpose is to identify available resources and services that meet an application's quality requirements (Qos) and then negotiate for allocating those resources. This class can be tailored to assess and test specific brokering policies. Sundas and Panda (2020)

4.2.4 Storage

Storage is an essential component of any data center or cloud system, responsible for holding onto vast amounts of data. Storage frames can be employed for simulation purposes and simply for storing data safely. However, some caveats

may be considered carefully in order to design a feasible model: 1) Retrieval times depend heavily on network bandwidth availability; 2) Run-time file authorization can introduce delays in task execution.

4.2.5 Virtual Machines

The component in charge of managing VMs during their lifetime is the Host entity. As stated before, multiple VMs can be hosted by a single host simultaneously, with cores being allocated based on a preset type of processor and policies of space-sharing and time-sharing. In addition, features like memory, processor, and the VM's internal scheduling policy are directly extended from the abstract Scheduler component.

4.2.6 Simulation

Simulating a data center using DynamicCloudSim is fairly straightforward, given that VMs, provisioning policies and hosts are independent objects. These objects do not require complex uncertainty management, given that thresholds bound the caps. In other words, the resource allocation decision is based on VMs demands and the specifics of the constraints to the provisioning policies that determine how to allocate VMs to hosts, the same way allocation happens in classic virtualized systems. Shukur et al. (2020)

To illustrate, there exist two hosts and three VMs. Hence those VMs are going to run within those hosts. If a fourth VM comes up, it can only run if there is an available host. This policy might be smoothly implemented with DynamicCloudSim, as it is independent of any complex uncertainty management.

The next chapter explains the experiment and how AI was inferred in the process, to improve the handling of cloud outages.

Chapter 5

Problem Formalisation

This section details the decision-making algorithm for the failure detection problem studied in this work. In addition, a formalisation of this problem and the relationship with the RL theory are presented. The formalisation is done by distinguishing between three components: 1) The machine learning algorithm, 2) An environment that maintains a reward signal, and 3) A measure to evaluates success or failure. This section details the decision-making algorithm for the failure detection problem discussed in this work.

The proposal is about a centralized system to maintain state information. Although this is not related to configuration management, which means not aiming to manage multiple versions of the same software, it is intended to acknowledge instance states across an independently operated network.

5.1 Reinforcement Learning Model

As depicted in Figure 2.1 reinforcement learning is a methodology used to train an agent to achieve a desired goal, learning from their interactions with the environment around them. On top of that are the policies π applied to maximize

the rewards r_t scrutinised over a certain period of time.

5.1.1 Policy

Fundamentally, the instance state ought to be monitored continuously, especially for those instances that turn out to be vital for the entire system. Either instances or the whole system can be modelled up to a classic RL paradigm. The algorithmic strategy can be expressed as a stochastic policy $\pi(a_t|o_t)$, that produces an action a_t according to the observation o_t at time t. The term *stochastic* refers to the probability of selecting a certain action in a particular state.

5.1.2 Value Function

In consonance with Sutton and Barto (2018), the Value Function v_π is the expected return condition on the states, also it is conditioned on a policy. Reminding that the goal of the agent is to maximize the expected cumulative rewards r_t by choosing actions a_t. Ultimately the notion of whether an action performed is the best option among all possible ones can be evaluated by the return of the value function. Briefly, it can be used to assess the value of states, also to select between actions.

Certainly, rewards that the agent can expect to receive in the future depend on what actions it will take. The importance of future rewards, is determined by the discount factor $\gamma[0, 1]$. Then $\gamma = 1$ indicates the preference for the accumulation of rewards in the future, so called *long-term rewards*. On the other hand, $\gamma < 1$ indicates the preference for *immediate rewards*. The *preference* cited here can be translated by *"putting higher weights on"*, mathematically speaking.

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t{=}s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t{=}s\right], \text{ for all } s \in \mathcal{S},$$

Figure 5.1: Value Functions. Sutton and Barto (2018)

5.1.3 Environment

The environment can be described as everything that is outside of the agents, and the environment responds, in a sense, by sending back observations. These outcomes, in turn, are analyzed by the agent in order to take the next action. It basically encompasses everything that can determine the node state.

While synchronous systems use timeout based mechanisms to detect failures in general, the asynchronous systems do not have this option available, once such timeout mechanism can not be implemented as there are no latency bounds Lundström et al. (2020). This means that it is challenging to identify whether a node has failed or it is just performing very slow for some reason.

In addition, considering a very complex environment as a cloud distributed system which comprises features such as dynamism, asynchronism, scalability, elasticity, heterogeneity, to name a few, the challenge becomes even greater. Moreover, it becomes even more crucial to understand each component of this system in order to obtain a better picture of the whole process.

For notation purposes, the *state* showed previously in Figure 2.1 can be interpreted as an *observation* here. In order to understand the mechanism in this particular context, it is also necessary to scrutinise observations, actions and rewards.

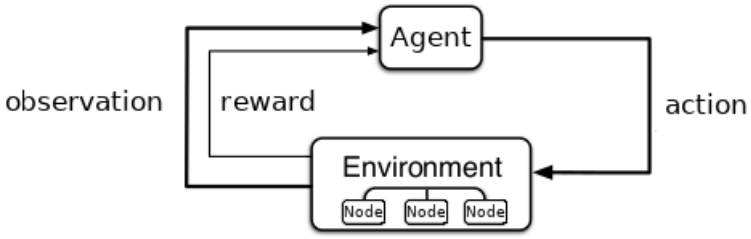

Figure 5.2: Fail Detector Model.

For the scope of this work, some particularities were implemented in order to best suit the goals of the project. The forked DynamicCloudSim can be found

on Dal Farra (2022a). It is based on the original project of Bux and Leser (2015) available on `https://code.google.com/p/dynamiccloudsim`.

5.1.4 Observations

To put in perspective, observations may be represented by the mean of the last four CPU utilisation percentiles at the time of a current task. This capability allows the agent to analyse a broader range of the instance performance behaviour in order to determine its state. That way, an assessment can be done on multiple VMs that ideally should share common attributes, which is the CPU utilisation in this case.

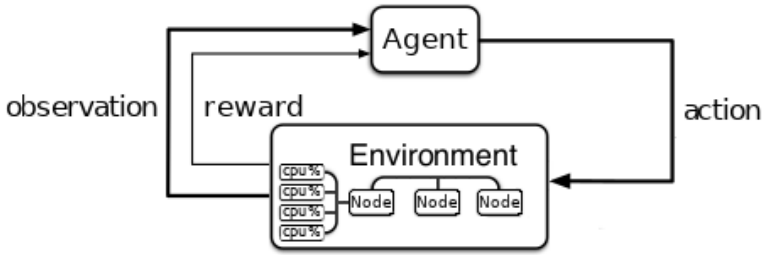

Figure 5.3: Fail Detector Model - Mean of four CPU percentiles.

CPU utilisation formula:

$$cpu_{(VM)} = mips_{(VM)} / \sum_{i=1}^{N} |mips^{i}|. \quad (1)$$

To make it possible, the *DynamicVm* class from DynamicCloudSim framework, a specialization of the Vm class, has been extended to keep CPU percentiles. The diagram from Figure 5.4 was generated using the PlantUML Diagram Generator plugin for IntelliJ, available on https://github.com/kuehne-nagel

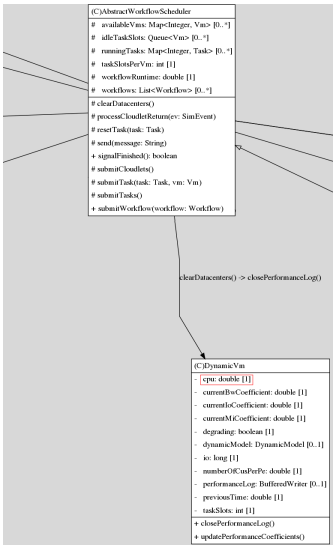

Figure 5.4: DynamicVm class.

The result is a CSV file with with either higher and lower CPU value, that will feed the agent, enabling decision making. In addition, to simulate instability of an instance, a *performance degradation* has been implemented on this particular DynamicCLoudSim version. The degradation step is performed during the *updateVmsProcessing* method, which is one of the core processes of the simulation. To illustrate, the internal processing of this method is shown in Figure 5.5 in the form of a sequence diagram.

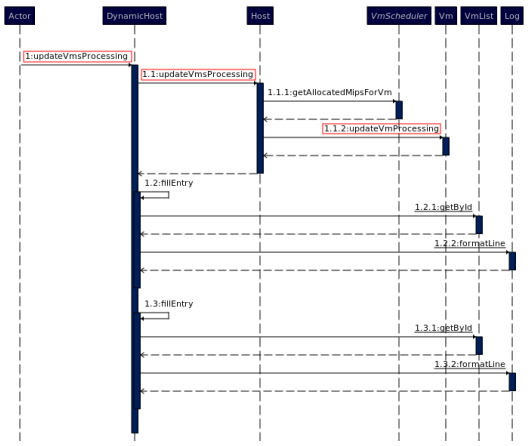

Figure 5.5: DynamicVm class.

In the end, the output is produced which, in this case, contains the information

required for the agent. The Table 5.1 shows a snippet of the output produced by the simulator. As can be seen, there are five lines of interest: The grey highlighted lines shows that the instance has a CPU which the performance is growing over time. Therefore it is considered in degrading state. This can cause the instance to stop responding when it reaches 100% CPU and therefore threatening the entire system. There is also additional framework information, with respect to other tasks being concluded and such.

40997.53144514096: ERA: VM #5 completed Task 10351 "CPU "
40997.53144514096: ERA: Mips: 190.5 RAM: 512 CPU: 16.35% VM
VM: #1 CPU: 59.62 Performance degrading: true
VM: #1 CPU: 66.24 Performance degrading: true
VM: #1 CPU: 73.60 Performance degrading: true
41000.74690350092: ERA: VM #5 completed Task # 11119 "network "
41000.74690350092: ERA: Mips: 190.5 RAM: 512 CPU: 16.35% VM
VM: #1 CPU: 81.78 Performance degrading: true
VM: #1 CPU: 90.87 Performance degrading: true

Table 5.1: Agent output.

5.1.5 Agent

It is the agent's responsibility to perform a task, monitor its progress, and report the state of an instance as required to prevent future outages due to failures of instances or applications. To ensure stability, there were modelled two discrete actions (a_t) in the *action space* to be reported by the agent; *Functional* and *Degraded*. Both represent the instance state when performing a certain task. An agent executing its failure detection strategy may be able to apply the following steps:

1. Updates the state of instances based on the observation o_t coming from the environment.

2. Executes the policy $\pi(a_t|o_t)$ which results in an action a.

3. Performs the action a.

4. Next time step $t \rightarrow t + 1$, go back to step 1.

In particular, this work proposes an agent capable of consuming a CVS file generated based on the DynamicCLoudSim simulator output, presented on the Table 5.1. The data extracted from the simulator output may contain three columns. One is an index, representing the time-step when the simulation is run. The next two columns represents the maximum and minimum running CPU load of the computer, respectively. The following Table 5.2 shows the structure.

	High	Low
1	16.9082493174502	12.3261137524212
2	25.7708418190066	18.7869436860558
3	39.2788322191839	28.6342686877851
4	59.8671425380032	43.6431469102043
5	16.9082493174502	12.3261137524212
6	23.1937576371059	12.3261137524212
7	18.7869436860558	13.6956819471347
8	28.6342686877851	20.8743818733953
9	31.815854097539	12.3261137524212
10	23.1937576371059	16.9082493174502

Table 5.2: DynamicCloudSim output.

The agent itself is implemented in Python, taking advantage of the tools provided by the language and its versatility for domain-specific problems. It uses Gym from OpenAi Brockman et al. (2016), one of the most popular open-source libraries that implement many tools and algorithms for working with reinforcement learning. It has been extensively used in academic research.

5.1.6 Actions and Rewards

Fundamentally, an agent is expected to act in a given environment. Thus, rewards are taken into account. In this work, the environment comprises a cloud computing scenario orchestrated by DynamicCloudSim, as stated previously. As such, there is instability and uncertainty in the simulation where virtual machines perform tasks that are not always completed successfully, if completed.

In this case, the uncertainty and instability are determined by the CPU measurements of the virtual machines. An approach to model this concept considers the mean of the last four CPU utilisation percentiles. Whether this value is above a threshold, it might be considered a *Degraded* state or *Functional* otherwise. Hence, the *action space* comprises these two discrete actions.

The reward applied is *+5* for each successful prediction and *-10* for failures. For instance, if the agent classified the node as *Functional* and the CPU value is closer than 100%. Likewise, if the node classification was determined *Degraded* and the CPU value is very low. Both situations described might be potentially misclassification cases. Therefore the agent might be penalised by -10.

The RL package used was proposed by Raffin et al. (2021), so called Stable Baselines 3. Stable Baselines has a set of algorithms built-in, making it simple to train and compare as required. In addition, trained models can be persisted, allowing further improvement by training on larger datasets as well as transfer learning. For that reason, the process is no longer limited to training huge models and datasets. Ultimately, Stable Baselines abstracts complexity regarding model training and optimizing hyperparameters, allowing users to focus more on research and development and less on infrastructure issues.

The algorithm for the RL model for the failure detector on cloud environments is based on Proximal Policy Optimization proposed by Schulman et al. (2017) in their work. Using a stochastic policy-oriented approach, PPO trains a sampling

policy by exploring actions based on the latest version of its stochastic policy. The degree of randomness in action selection depends on initial conditions and the training procedure. During the training, the policy typically becomes gradually less random, as the update rule might stimulate to exploit rewards found previously. Schulman et al. (2017)

In this experiment, the model is trained on historical data, including CPU performance range values. It then predicts the probability that a given instance will be indicated as Functional or Degraded, and the output of this agent can be used to inform decision-making around different stages at the cloud monitoring process.

Chapter 6

Experiment Results

This chapter suggests some visualisation tools to understand the models debugging and optimisation processes. To accomplish this goal, it was the chosen Tensorboard from the Tensorflow project. Abadi et al. (2015). This tool provides mechanisms for visualisation and analysis from either training or testing steps. Besides that, It allows for assessment metrics like loss and accuracy, to name a few.

The idea is first to start from a first scenario and then evolve it. Analyse the metrics individually and observe how each differs from the other. Then, examine the properties of the datasets using graphic tools, and also compare the results. The experiment has been divided into two scenarios. Initially, the first attempt to achieve the model conversion was not entirely successful. Results have shown poor conversion. On the other hand, the second attempt proved to be promising. The reasons will be discussed in detail over the following topics.

6.1 Scenario 1

Initially, RL techniques were presented and evaluated considering the following scenario:

Features	Value
Inputs	31000
Agents	1
Steps per episode	100
Likelihood of failure	2%
Virtual Machines Total	7
Virtual Machines Disrupted	1 (fixed)
Deprecation Percentage	10%

Table 6.1: DynamicCloudSim Scenario.

To begin with, a single agent was responsible for monitoring all the seven VMs from the environment. Meanwhile, a single VM was deliberately flagged as degraded, which means that its CPU is overloaded and might not be able to finish the current task assigned. Overall, the task might increase the CPU workload by about 10% each step until reaching 100% or more and is consequently dropped from the pipeline, as shown in the code snippet from *Datacenter* Java class below. Ultimately, the agent is expected to capture and report the impact of those low-performing virtual machine.

```
1 private void terminatingDegradedVms(Host host) {
2     List<DynamicVm> dVmList = new ArrayList<>();
3     for (Vm vm : getVmList()) {
4       DynamicVm dVm = (DynamicVm) vm;
5       if (dVm.getDegrading() && dVm.getCpu() >= 95) {
6         Log.formatLine(CloudSim.clock() + ": " + getName() +
7             ": Terminating VM #" + dVm.getId() +
8             " CPU: %.2f%%" +
9             " HOST #" + host.getId(), dVm.getCpu());
```

```
10          host.vmDestroy(dVm);
11          dVmList.add(dVm);
12        }
13      }
14      for (DynamicVm dVm_ : dVmList) {
15        getVmList().remove(dVm_);
16      }
17    }
```

Listing 6.1: Datacenter class

By design, a plot's primary purpose is to communicate the outcomes of an experiment in a way that is simple to understand. It may be suitable for identifying how uncertainty and bias are handled in model training. In other words, understanding how the model works and also what conclusions those models suggest. This section explains the plots generated utilising Tensorboard during the training and evaluating phases. The experiment results of each scenario can be accessed on the Tensorboard cloud. Dal Farra (2022b).

6.1.1 Training Step

6.1.1.1 Loss Measures

According to artificial intelligence literature, the value of a state is given by a probability distribution over the states available to it. Sutton and Barto (2018). To illustrate, the set of states for which a learning algorithm has correct and incorrect expectations happens to be captured by the next two plots.

The *Value Loss* is the measure of the difference between the learning algorithm's expectation of the value of a state and the observed value of that state. Briefly, the smaller it is, the better it predicts how accurately it performs. Preferably decreasing when the reward stabilizes.

Given these facts, when looking at the curve in the Figure 6.1, it is clear that at the beginning the trend was progressing downwards, as expected. However at the 0.12 coordinate on step 6.144 it started to go up and then to wobble. This means that, unfortunately, it is still not learning accordingly.

Figure 6.1: Value Loss

In line with the previous plot, the *Loss* (Figure 6.2) presents an unsatisfactory trend, oscillating between 0.03 and 0.12. In contrast, it should be progressing towards zero, pointing out that the model is making fewer errors. That is clearly not the case. The line pivots up and down, which means the model inaccurately predicts the outcome.

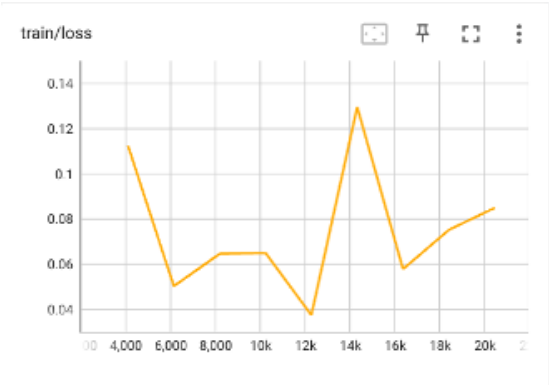

Figure 6.2: Loss

6.1.2 Testing Step

The purpose of this stage is to encourage the agent to continue testing as many accessible state transitions as possible, finding long sequences of actions to perform such tests. The analysis will take place by verifying rewards accumulated over time and checking the errors as consequences of any possible incorrect state transition selected.

6.1.2.1 Average Rewards

Reminding from the RL theory, the average rewards calculation shows the expected return of rewards obtained from the state s, firstly taking action a and then following policy π. Briefly, it is a definition of the optimal function $Q(s, a)$. In other words, it points to the trade-off between the maximum reward and the minimum penalty for each step. To visualise it in numbers, the following graphic, shown in Figure 6.3, presents the results of the training phase, achieved over 20k steps.

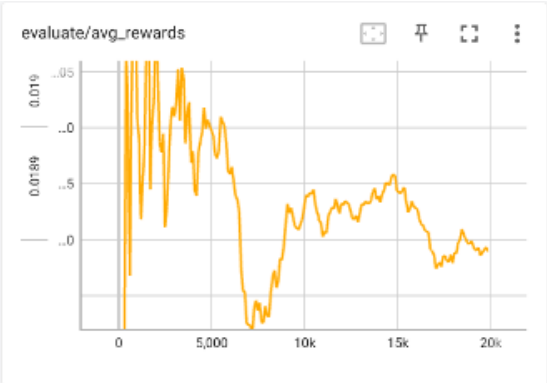

Figure 6.3: Average rewards function

The plot indicates no plausible trend at first. Then it stabilizes to some extent, giving a visual intuition of a possible, though not ideal, tendency. One thing to notice is that the slope is abrupt at first and continue nearly flat. Usually, plots like this give the sense that nothing relevant happens over time. Similar to

looking at the data table numbers, it will somehow produce a false impression of a trend.

6.1.2.2 Errors

All pieces of evidence presented so far are confirmed by looking at the error graph. There were 556 errors over 20k steps in this simulation. According to the previous parameters, the situation might worsen with time. Alternatively, some approaches will be used to mitigate the scenario design flaws, which will be detailed as follows.

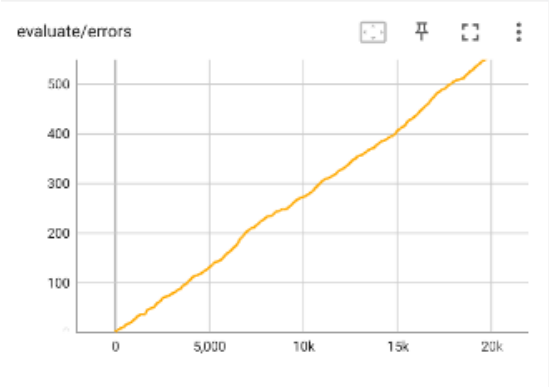

Figure 6.4: Errors

6.1.3 Analysis

Particularly, high learning rate is likely to produce overfitting, which is probably one problem encountered. Therefore, one approach was to decrease the learning rate. Though, this move alone had no practical effect. Then, after some time studying the input data, it was decided to generate a more diverse dataset.

It turns out that the previous input was poorly diverse and very biased. The likelihood of failure, which is a parameter of the DynamicCloudSim simulator, was increased 10x. Also the number of steps per episode are 5x higher than

before, and the VMs are degrading randomly, to avoid patterns that may infer bias to the agent learning process.

These particular decisions were crucial to improving results. The lesson learned is that spending more effort on Feature Engineer may save energy in tuning the model and make it more robust. All the insights started due to the visualisation of data, which emphasises the need to analyse visually the data to understand how the model behaves.

In that case, Facets, an open source dataset visualization tool hosted by github.io/facets, has been used to identify patterns and possible relationships in data that would not be obvious otherwise. For instance, in the Figure 6.5 it is possible to identify features with unbalanced distributions. For instance, like CPU4 highlighted in red below.

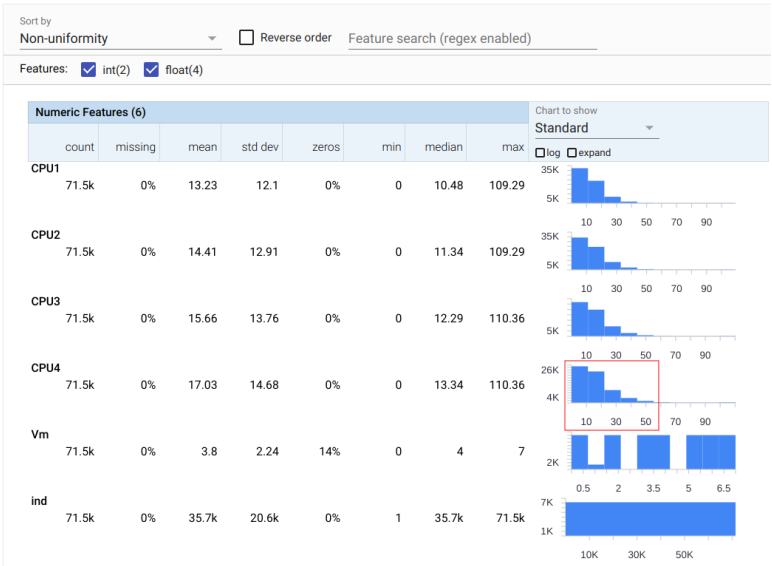

Figure 6.5: Facet - Scenario 1

A more detailed analysis provided by the tool revealed a distribution in Figure 6.6 with greater concentration at high values and little representation of low values. This, in itself, represents a skewed dataset, and potentially might contribute to poor results.

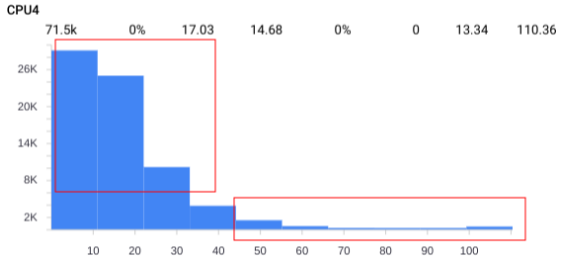

Figure 6.6: Facet - Scenario 1 CPU4

6.2 Scenario 2

In order to improve the results, a new scenario has been proposed based on observations mentioned in topic 6.1.3, as can be seen below.

Features	Value
Inputs	41680
Agents	1
Steps per episode	500
Likelihood of failure	20%
Virtual Machines Total	7
Virtual Machines Disrupted	7 (randomly)
Deprecation Percentage	10%

Table 6.2: DynamicCloudSim Scenario.

In comparison to the Table-6.1, there were increasing of Inputs, along with extra Steps per episode and also higher Likelihood of failure. Besides, to assure uncertainty, seven virtual machines are simulating outages randomly. As opposed to just one, like the first scenario.

Overall, these changes were made on the Simulator side. On the other hand, changes in the Agent were also made in order to achieve better performance. For instance, a learning rate decay function has been implemented instead of a fixed one, inspired on stable.baselines3.readthedocs.io.

The way it was computed, the observation average did not capture any feasible behaviour. Therefore, this function was slightly changed to calculate the average between the *minimum* and maximum observation values.

6.2.1 Training Step

6.2.1.1 Value Loss

The following graph is a more accurate view than the one shown in Figure 6.1 of the relationship between learning algorithms' expectations and its current loss. In fact, the convergence has also been perceived in the previous plot to some extent. However, the curve below clearly depicts a more progressive trend towards zero..

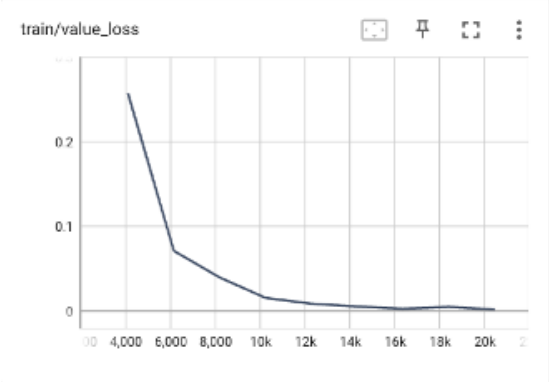

Figure 6.7: Value Loss

The next graph, Value Loss comparison, emphasizes a substantial improvement considering the conditions within the new scenario. The curve shaped as a downward slope is a step in the right direction. It accurately identifies and represents loss patterns in a more diverse environment.

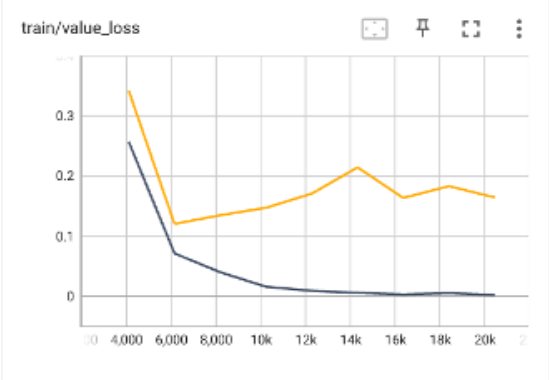

Figure 6.8: Value Loss comparison

Regarding the Loss itself, essentially, the previous chart displayed a lack of tendency, likely due to overfitting, as shown in Figure 6.2. Like the function Loss, this one is supposed to show a tendency towards zero at some point and then progress flat and closer than zero for a valid classifier.

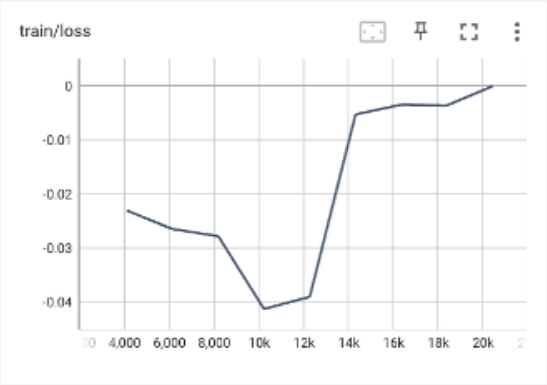

Figure 6.9: Loss

In comparison, the new curve indicates more accurate progress, verging on zero as time progresses. Showing that the modifications made resulted in considerable improvements in result.

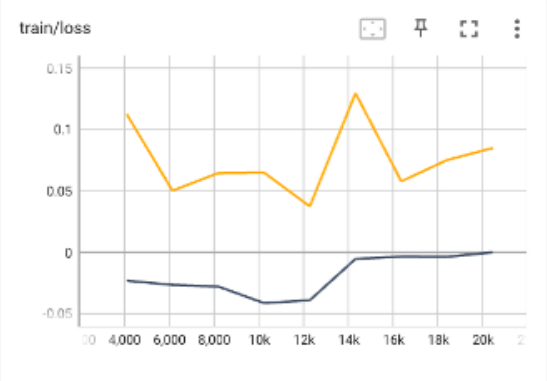

Figure 6.10: Loss comparison

6.2.2 Testing Step

6.2.2.1 Average Rewards

The average rewards function improved substantially, as can be compared in the Figure 6.11 against Figure 6.3. The new graph better correlates the decision making process with rewards over time. Therefore, from the curves, it is clear that value and policy converge to some extent as time progresses.

Considering that the model's goal is to find the policy that maximizes the long-term expected average reward from the distribution, and reminding that the long-term expected reward is defined as a sum of future rewards discounted by specific rates. It is fair to say, analysing the growing trend perceived in the Figure 6.11, that the model is improving,

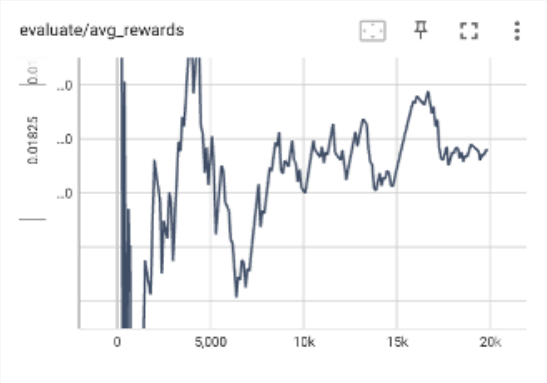

Figure 6.11: Average rewards function

In the comparison graph, both results appear similar, due to scaling. However, when analysing each graph separately, as shown in Figure 6.3 and Figure 6.11, the difference becomes apparent.

Figure 6.12: Average rewards function comparison

6.2.2.2 Errors

The Error plot is a fundamental metric for understanding the accuracy of predictions. However, as opposed to the previous one, this has shown potentially more satisfactory results, as seen below.

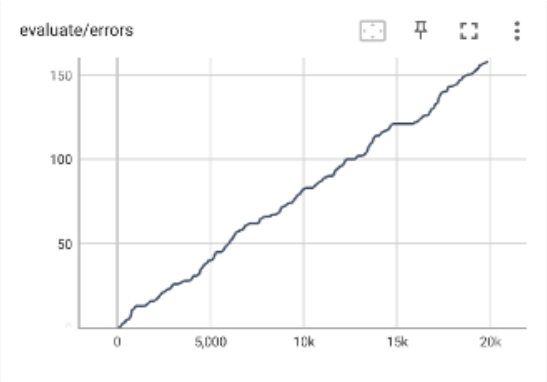

Figure 6.13: Errors

The comparison suggests that there was a significant decrease in errors. It indicates that potentially the model is learning more effectively at this point. To put it in numbers, the previous error total was 556, over 20k steps. Now the value is 158, which is definitely better performance.

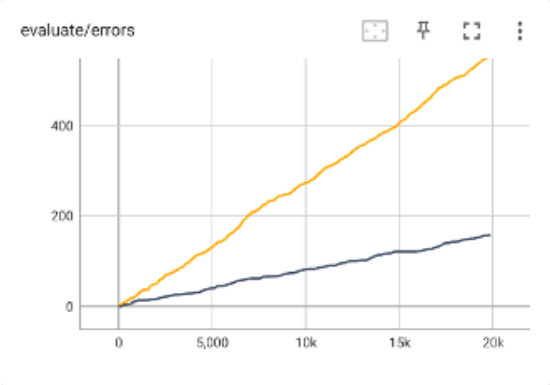

Figure 6.14: Errors comparison

6.2.3 Analysis

6.2.3.1 Extending the Training Set

The analysis provided was based on the intuition described on topic 6.1.3. Basically, one approach was to include more data. In other words, expanding the training set to increase the model's accuracy. Eventually, providing different observations to reduce the dominant relationship among the input and output variables. Specifically, more relevant data, from the simulator has been injected into the model.

6.2.3.2 Data augmentation

According to many authors, including Aggarwal et al. (2018), this strategy helps avoid overfitting. In summary, build up new training sets generated through transformations on the original sets. For this experiment, CPU values were either increased or decreased in order to achieve a more suitable distribution of values.

6.2.3.3 Agent Modifications

As described at the beginning of the topic 6.2, the Agent was slightly improved to achieve convergence. Minor but significant modifications were made until it

reached better results.

For the analysis, as previously, the Facets tool was used. The difference is easily perceived. The Figure 6.15 comprises the representation of the two scenarios. The right side shows the previous distribution, already presented in Figure 6.5. Hence, the left side shows the outcomes after implementing the measures described above. Ultimately, a more gracefully distribution may be noticed. For instance, on the left side, the *CPU4* was very restricted to a small range of values. Otherwise, on the right, the range is clearly less skewed after the modifications.

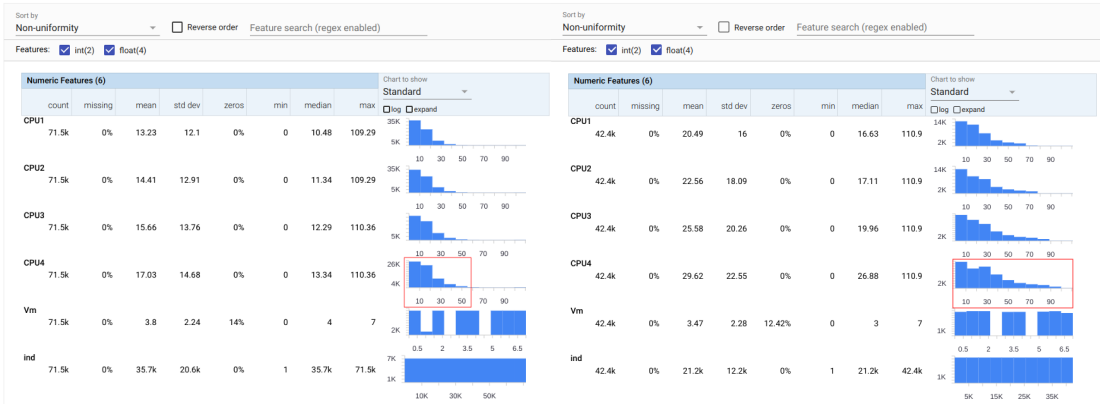

Figure 6.15: Facet - Scenario 2 vs Scenario 1

The Figure 6.16 shows in detail the effect of the improvement comparing CPU4 values.

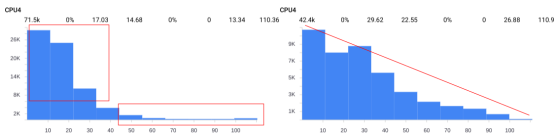

Figure 6.16: Facet - Scenario 2 vs Scenario 1 CPU4

Although the previous results may vary, a similar trend will likely be observed in each plot when repeating the followed steps. On top of it, rewards and losses might be sufficient to identify tendencies and assess the model outcomes in future research. The plots can be accessed in the Tensorboard cloud. Dal Farra (2022b).

6.3 Summarising Data

6.3.1 Overview

This section is related to understanding a few statistics computation steps to obtain the relevant results. This descriptive statistics process aims to reduce the information in the data into summaries that give a clearer understanding of what is happening. As stated by Fisher and Marshall (2009), descriptive statistics is the process of converting raw data into meaningful summaries. It reveals information about what's in the data and how it is distributed. It also tells us where the data comes from, the biases, and the methods we should use to estimate information.

6.3.2 Numerical Summaries

The sample in the Figure 6.17 encompasses the numerical summary of Centre, represented by the *Mean*. In addition, the summary of the Spread, denoted by the *Variance*. The mean, computes as the sum of the values divided by the total of observations. On the other hand, the variance shows how the observations may vary related to the mean. Zukhra and Ogiloy (2022).

Vm <dbl>	CPU1 <dbl>	CPU2 <dbl>	CPU3 <dbl>	CPU4 <dbl>	Variance <dbl>	Mean <dbl>	Status <chr>
5	12.95	37.93	42.15	46.83	228.59	34.96	DEGRADED
3	8.07	35.61	39.57	43.97	261.96	31.80	DEGRADED
6	0.02	0.03	0.03	0.03	0.00	0.03	FUNCTIONAL
3	6.29	6.29	60.56	63.08	1029.01	34.06	DEGRADED
6	14.11	14.56	53.65	56.47	554.30	34.70	DEGRADED
4	0.83	0.87	0.89	0.90	0.00	0.87	FUNCTIONAL
6	13.66	13.94	14.99	49.09	304.77	22.92	DEGRADED
3	2.40	2.42	2.44	2.47	0.00	2.43	FUNCTIONAL
6	1.87	1.89	1.91	1.93	0.00	1.90	FUNCTIONAL
4	26.24	27.99	29.15	74.73	552.12	39.53	DEGRADED

Figure 6.17: Data Set Table Sample

It is important to note that those metrics presume the variables are independent of each other, and the sample is a plausible estimate of the population. For instance, an observation given by CPU values can have substantial variance in

itself. However, it might be assumed that the other observations are close to the mean. Hence that observation ends up having little effect on the mean. The implication is that given enough observations, and if the variables are independent of each other, it is possible to extract relevant conclusions about the population from that sample.

The Figure 6.18 presents the VM population stratified by Status. Giving a sense of the behaviour of each VM during the experiment. In addition, it is possible to get a notion of the capacity of the Simulator to generate VMs with predefined settings and redesign it, if necessary, to achieve results accordingly. However, the following figure is legit for the experiments conducted in this work.

	DEGRADED (N=14999)	FUNCTIONAL (N=16000)
VMs		
0	2354 (15.7%)	2462 (15.4%)
1	361 (2.4%)	333 (2.1%)
2	2133 (14.2%)	1307 (8.2%)
3	1708 (11.4%)	2922 (18.3%)
4	2175 (14.5%)	2112 (13.2%)
5	1917 (12.8%)	2635 (16.5%)
6	2036 (13.6%)	2731 (17.1%)
7	2315 (15.4%)	1498 (9.4%)

Figure 6.18: VMs per Status Table Sample

The next Figure 6.19, presents the CPU statistic values stratified by Status, which not only gives a notion of the symmetry in each variable but also helps to understand the decisions implemented in the agent to decide what might be a feasible range for both *Degraded* and *Functional* instances. As evidenced by the figure below, it is fair to say that there is a growing trend from top down, for both Statuses. This information happened to be helpful when implementing the agent's logic. However, it is not restricted to this aspect. In both cases, the agent analyses a combination of metrics to determine its decision.

	DEGRADED (N=14999)	FUNCTIONAL (N=16000)
CPU1		
Mean (SD)	18.3 (12.1)	1.85 (1.81)
Median [Min, Max]	15.7 [0.0554, 86.6]	1.21 [0.000350, 22.8]
CPU2		
Mean (SD)	29.6 (21.0)	1.88 (1.82)
Median [Min, Max]	21.7 [0.408, 111]	1.24 [0.000388, 22.8]
CPU3		
Mean (SD)	42.1 (23.4)	1.91 (1.83)
Median [Min, Max]	39.9 [0.482, 111]	1.27 [0.000432, 22.8]
CPU4		
Mean (SD)	56.1 (19.9)	1.94 (1.84)
Median [Min, Max]	50.1 [23.2, 111]	1.30 [0.000480, 22.9]

Figure 6.19: CPU Statistics Table Sample

The following table shows the more detailed correspondence between the CPU state values stratified by two variables, in which case they are nested - Status and VM. The goal is to analyze CPU statistics for each Status group split by VM to understand each variable's symmetry in detail.

	DEGRADED							
	0 (N=2354)	1 (N=361)	2 (N=2133)	3 (N=1708)	4 (N=2175)	5 (N=1917)	6 (N=2036)	7 (N=2315)
CPU1								
Mean (SD)	19.5 (11.9)	19.1 (17.2)	18.5 (11.5)	16.1 (11.6)	18.7 (12.6)	19.0 (12.8)	17.0 (10.9)	18.9 (11.7)
Median [Min, Max]	17.6 [0.313, 84.5]	13.4 [1.76, 75.6]	15.7 [2.27, 82.4]	13.8 [0.0554, 83.3]	15.1 [0.412, 86.6]	17.6 [1.59, 83.9]	14.8 [0.375, 84.2]	16.1 [1.12, 84.2]
CPU2								
Mean (SD)	30.8 (20.7)	30.1 (22.8)	30.0 (20.8)	27.6 (21.2)	29.4 (20.8)	30.5 (21.6)	27.9 (20.1)	30.4 (21.3)
Median [Min, Max]	24.0 [1.41, 110]	19.9 [1.96, 104]	22.1 [2.50, 109]	20.0 [1.46, 110]	21.1 [0.434, 111]	22.5 [1.74, 111]	19.5 [0.408, 109]	22.2 [1.55, 110]
CPU3								
Mean (SD)	43.2 (23.0)	43.2 (24.5)	41.9 (22.9)	40.4 (23.9)	41.9 (23.4)	43.4 (23.9)	40.6 (22.9)	42.7 (23.8)
Median [Min, Max]	41.1 [4.96, 111]	39.2 [2.17, 104]	39.9 [2.74, 111]	38.4 [2.34, 110]	39.2 [0.482, 111]	41.3 [1.92, 111]	38.8 [0.782, 111]	40.7 [1.61, 110]
CPU4								
Mean (SD)	56.8 (19.6)	56.8 (21.3)	55.7 (19.1)	54.9 (20.3)	56.1 (20.1)	57.1 (20.2)	54.2 (19.4)	57.1 (20.0)
Median [Min, Max]	51.0 [23.5, 111]	50.7 [25.6, 110]	50.1 [24.3, 111]	48.5 [23.5, 111]	49.8 [25.1, 111]	51.4 [24.8, 111]	47.8 [24.5, 111]	51.0 [23.2, 111]

	FUNCTIONAL							
	0 (N=2462)	1 (N=333)	2 (N=1307)	3 (N=2922)	4 (N=2112)	5 (N=2635)	6 (N=2731)	7 (N=1498)
CPU1								
Mean (SD)	1.67 (1.62)	1.66 (1.75)	2.43 (1.93)	1.86 (1.75)	1.79 (1.81)	2.00 (1.66)	1.44 (1.91)	2.27 (1.96)
Median [Min, Max]	1.09 [0.000520, 14.2]	0.979 [0.000724, 14.0]	2.01 [0.00601, 15.1]	1.20 [0.00146, 14.2]	1.10 [0.000350, 15.7]	1.54 [0.00135, 11.7]	0.677 [0.000539, 15.4]	1.83 [0.0182, 22.8]
CPU2								
Mean (SD)	1.70 (1.63)	1.68 (1.76)	2.46 (1.94)	1.89 (1.76)	1.82 (1.82)	2.03 (1.67)	1.47 (1.92)	2.30 (1.96)
Median [Min, Max]	1.12 [0.000565, 14.2]	1.00 [0.000796, 14.0]	2.05 [0.00639, 15.1]	1.22 [0.00155, 14.3]	1.13 [0.000388, 15.7]	1.58 [0.00148, 11.7]	0.705 [0.000592, 15.4]	1.86 [0.0188, 22.8]
CPU3								
Mean (SD)	1.73 (1.64)	1.71 (1.77)	2.49 (1.95)	1.92 (1.77)	1.85 (1.83)	2.06 (1.67)	1.49 (1.93)	2.33 (1.97)
Median [Min, Max]	1.15 [0.000614, 14.3]	1.04 [0.000874, 14.0]	2.08 [0.00680, 15.1]	1.25 [0.00173, 14.3]	1.17 [0.000432, 15.7]	1.62 [0.00163, 11.8]	0.737 [0.000650, 15.4]	1.88 [0.0194, 22.8]
CPU4								
Mean (SD)	1.76 (1.65)	1.74 (1.78)	2.53 (1.95)	1.95 (1.77)	1.88 (1.84)	2.09 (1.68)	1.52 (1.94)	2.37 (1.98)
Median [Min, Max]	1.19 [0.000667, 14.3]	1.09 [0.000961, 14.1]	2.11 [0.00724, 15.3]	1.28 [0.00192, 14.3]	1.20 [0.000480, 15.7]	1.65 [0.00171, 11.8]	0.770 [0.000715, 15.6]	1.92 [0.0200, 22.9]

Figure 6.20: CPU Statistics per VM Table Sample

6.3.3 Graphical Summaries

A suitable graphical representation is delivered by Boxplots, which provides valuable insights into the sample. The Figure 6.21 below shows CPU1 and CPU4 side by side for comparison. Notably, there is substantially more variability in degraded statuses than functional statuses. It is also clear, the presence of outliers in both statuses for both samples.

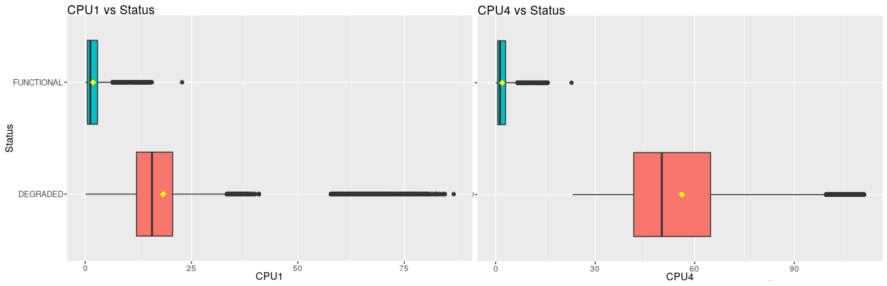

Figure 6.21: CPU1 vs CPU4

The next graph is the Barchart. It may be convenient to provide proportions, percentages and sample sizes. The intuition combines those three measures in one plot and effectively conveys the information. The graphic reveals similar sizes and proportions, indicating that the sample is composed evenly of the two types of samples defined.

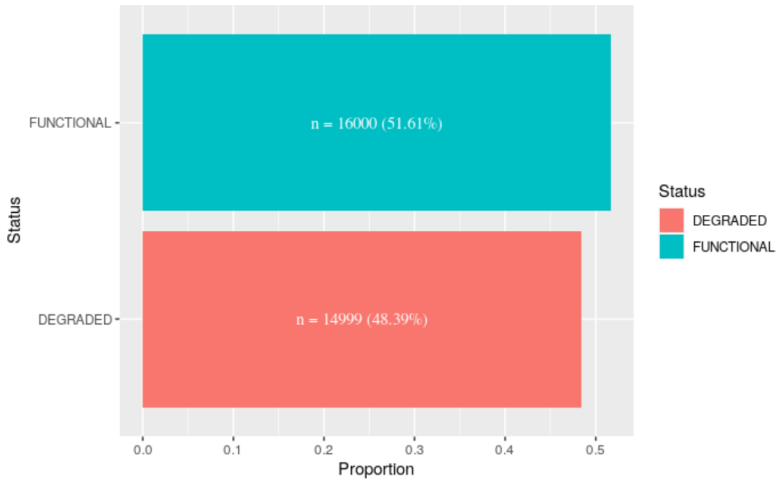

Figure 6.22: Status vs Proportion Barchart

6.4 Discussion

One of the most difficult challenges faced by cloud computing businesses according to Islam et al. (2021) on their study about anomaly detection in cloud, is how to mitigate risks, including outages. Ideally, companies demand cloud instances that can be rapidly provisioned and released with minimal management effort or service provider interaction. This work's idea came from practical situations experienced while working on cloud computing systems on a daily basis.

The results of the experiments presented so far show that the RL agents have the capabilities to improve upon traditional failure detector methods based on thresholds provided by cloud vendors. Even though it is challenging to predict CPU utilization of host machines, there might be several possible combinations with other resources that the agent can learn altogether, like memory status, timeout response latency, hardware and software parameters, to name a few. Furthermore, simulations of cloud instances with inferred instability and uncertainty using tools such as DynamicCloudSim presented in the Chapter 4, have produced plausible inputs to train the agent.

There are several approaches for failure detection on the market, for instance, based on network latency (Sinha et al. (2019)), machine uptime (Nassif (2014)), metrics monitoring (Pannu et al. (2012)), and so on. Nevertheless, they might not give a clear picture of the actual state of machines available for allocation in cloud infrastructure and indeed cause cloud deployment problems. Therefore, these two-part scenarios of experiments demonstrated in Chapter 6 attempted to show in detail how machine learning can upgrade failure detectors in cloud environments.

The analysis only focuses on the CPU, considering that it may be an essential factor in cloud instance reliability and can be estimated at a higher rate than other aspects, for instance, memory utilization. As explained in Section 5 in more

detail. Furthermore, memory utilization may be correlated with CPU utilization to some degree and then can be combined in the agent's logic.

The experiment results in Chapter 6 showed that spending more effort on the Feature Engineer might be worth improving the model. All the insights from Section 6.1.3 started due to the data visualization. With these outcomes, it was possible to structure and tune the data to aid the further investigation. In summary, Feature Engineering helped explore the input data to derive more straightforward representations.

Ultimately, descriptive statistics, presented in Section 6.3 added meaning and transparency to the data analyzed. The information provided allowed evaluate the inconsistencies and verify the decision-making about which channel produced the best results for a particular purpose.

Chapter 7

Conclusion

Cloud computing is a relatively new and disruptive technology. As such, it is subjected to profound scrutiny by the community. Because it is built upon proven and solid concepts, like virtual machines, which have been around since the sixties, many companies and individuals have embraced the idea. Though most companies might not fully understand its implications.

Nevertheless, there are several flaws in distributed system that are often unnoticed by both providers of cloud services and their consumers. This work moves forward with key issues that might be addressed when cloud computing comes into play in a company's infrastructure. It has been proposed to explore a different environment, in addition to games and robotics, typically used with RL techniques. The main goal was achieved, which was, through RL, to anticipate possible failures of cloud systems. Through non-intrusive monitoring of its components.

The task of solving reinforcement learning can be defined as finding a good policy that maximizes the long-term expected average distribution reward. In this context, the basic rule of the policy improvement model is that after each interaction, the agent tries to figure out a better action than those in its current

policy. This mission has also been accomplished.

Fundamentally, this study is expected to motivate the research community to investigate further the feasibility of using RL agents for more capable failure detectors. We hope combined approaches can utilise resources more efficiently and perform better than previous methods.

7.0.1 Contributions

Our study has shown that machine learning can be used to predict CPU utilization of cloud instances and create failure detectors that could be very helpful in handling cloud outages. Furthermore, and more importantly, we have been able to identify some interesting relationships between the parameters of an instance and its CPU utilization.

At the same time, getting out of the comfort zone of board games and robotics regarding this technique has been a challenge in itself. Despite these challenges, this work has proven that it is possible to open the perspective and try tackling a non-conventional problem. As long as it is suitable for the case.

In addition, The system presented is decoupled from proprietary technology and may gracefully fit into any cloud project. The source of a failure detector agent is entirely transparent to any external entity, and it can be used in every stage without coordination with any other system components.

7.0.1.1 Work in Progress

Since this is a work in progress, to whom it may concern, both R Summary Data and the RL agent project links references are Dal Farra (2022d) and Dal Farra (2022c), respectively.

7.0.2 Future Work

Simulations using RL agents show promising results in terms of improving decision-making over time. For instance, papers showing techniques to simulate the behaviour of financial markets are quite common. Likewise, researchers could leverage RL agents to anticipate and predict instances in a cloud while incorporating the uncertainty that comes with this kind of environment.

Besides, consumers have no control over cloud computing resources, and cloud providers must ensure those resources' quality, availability, reliability and performance. In this context, we look forward to seeing RL agents integrated into a commercial software solution for Service Level Agreements (SLAs) management and how this technology may be applied in other forms of computing and storage industries.

Lastly, there is much more room for improvement, especially with respect to this work. It may be interesting to increase the list of parameters to be monitored in order to give more context to the RL agent. The decision to use RL to inspect the hosts was made because of various considerations mentioned in the previous sections, based on the information at that moment. Even though the approach sounds promising, further investigations might be required to make it more effective.

References

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `https://www.tensorflow.org/`. Software available from tensorflow.org. 38

Charu C Aggarwal et al. Neural networks and deep learning. *Springer*, 10:978–3, 2018. 50

Shegufta Bakht Ahsan and Indranil Gupta. A new fully-distributed arbitration-based membership protocol. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 716–725. IEEE, 2020. 10

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. 35

Marc Bux and Ulf Leser. Dynamiccloudsim: Simulating heterogeneity in computational clouds. *Future Generation Computer Systems*, 46:85–99, 2015. 25, 32

Rajkumar Buyya, Rodrigo N Calheiros, and A Beloglazov. Cloudsim: A framework for modeling and simulation of cloud computing infrastructures and services. *The Cloud Computing and Distributed Systems (CLOUDS) Laboratory.[Online].[Accessed 18 March 2022]*, 2009. 27

Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and experience*, 41(1):23–50, 2011. ix, 25, 26, 27

Maikel Dal Farra. Dynamiccloudsim. `https://github.com/maikeldf/dynamiccloudsim.git`, 2022a. 32

Maikel Dal Farra. Tensorboard. `https://tensorboard.dev/experiment/fmTxnnvnRg6emORd8gYhgg/#scalars`, 2022b. 40, 51

Maikel Dal Farra. fdm. `https://github.com/maikeldf/msc_ai/tree/main/project/fdm`, 2022c. 59

Maikel Dal Farra. rds. `https://github.com/maikeldf/msc_ai/tree/main/project/rds`, 2022d. 59

Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019. 15

Murray J Fisher and Andrea P Marshall. Understanding descriptive statistics. *Australian critical care*, 22(2):93–97, 2009. 52

github.io/facets. Facets - known your data. `https://pair-code.github.io/facets`. Accessed: 2022-08-14. 44

Amitava Gupta, Anil Kumar Chandra, and Peter Luksch. *Real-Time and Distributed Real-Time Systems: Theory and Applications*. CRC Press, Inc., USA, 2016. ISBN 1466598476. 2

Mohammad S Islam, William Pourmajidi, Lei Zhang, John Steinbacher, Tony Erwin, and Andriy Miranskyy. Anomaly detection in a large-scale cloud platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 150–159. IEEE, 2021. 56

Gopal Kakivaya, Lu Xun, Richard Hasha, Shegufta Bakht Ahsan, Todd Pfleiger, Rishi Sinha, Anurag Gupta, Mihail Tarta, Mark Fussell, Vipul Modi, et al. Service fabric: a distributed platform for building microservices in the cloud. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018. 7

Anastassia Lauterbach and A Bonim. Artificial intelligence: A strategic business and governance imperative. *NACD Directorship, September/October*, 54:57, 2016. 1

Henry H Liu. *Software performance and scalability: a quantitative approach*, volume 7. John Wiley & Sons, 2011. 1

Oskar Lundström, Michel Raynal, and Elad M Schiller. Self-stabilizing uniform reliable broadcast. In *International Conference on Networked Systems*, pages 296–313. Springer, 2020. 31

P. Mell and T. Grance. The nist definition of cloud computing: Recommendations of the national institute of standards and technology. *Public Cloud Computing: Security and Privacy Guidelines*, pages 97–101, 2011. 4

Lilian Noronha Nassif. Failure detection in network forensics for volatile data acquisition. In *ICCWS2014-9th International Conference on Cyber Warfare & Security: ICCWS*, volume 342, 2014. 56

Strahil Panev and Pero Latkoski. Sdn-based failure detection and recovery mechanism for 5g core networks. *Transactions on Emerging Telecommunications Technologies*, 31(2):e3721, 2020. doi: https://doi.org/10.1002/ett.3721. 9

Husanbir S Pannu, Jianguo Liu, Qiang Guan, and Song Fu. Afd: Adaptive failure detection system for cloud computing infrastructures. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, pages 71–80. IEEE, 2012. 2, 10, 19, 56

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 2021. 36

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. Citeseer, 1994. 16

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL `https://arxiv.org/abs/1707.06347`. 36, 37

Hanan Shukur, Subhi Zeebaree, Rizgar Zebari, Diyar Zeebaree, Omar Ahmed, and Azar Salih. Cloud computing virtualization of resources allocation for distributed systems. *Journal of Applied Science and Technology Trends*, 1(3): 98–105, 2020. 28

Bharati Sinha, Awadhesh Kumar Singh, and Poonam Saini. A failure detector for crash recovery systems in cloud. *International Journal of Information Technology and Computer Science (IJITCS)*, 11(7):9–16, 2019. 18, 56

Jungmin Son and Rajkumar Buyya. Latency-aware virtualized network function provisioning for distributed edge clouds. *Journal of Systems and Software*, 152: 24–31, 2019. 18

stable.baselines3.readthedocs.io. Stable baselines. 45

Amit Sundas and Surya Narayan Panda. An introduction of cloudsim simulation tool for modelling and scheduling. In *2020 International Conference on Emerging Smart Computing and Informatics (ESCI)*, pages 263–268. IEEE, 2020. 27

Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. ix, 11, 12, 13, 14, 15, 16, 17, 21, 23, 30, 40

Martijn Van Otterlo and Marco Wiering. Reinforcement learning and markov decision processes. In *Reinforcement learning*, pages 3–42. Springer, 2012. 11, 12

Futian Wang, Xunan Wang, Cheng Zhang, Qiang He, and Yun Yang. Fault tolerating multi-tenant service-based systems with dynamic quality. *Knowledge-Based Systems*, 195:105715, 2020. 8

Lu Yan, Tao Zhang, Ying Gao, Rongsheng Wang, and Shuxin Ding. Reliability analysis of station autonomous computer system based on fuzzy dynamic fault tree and markov model. *Engineering Reports*, page e12376, 2020. doi: https://doi.org/10.1002/eng2.12376. 9

Lu Yan, Tao Zhang, Ying Gao, Rongsheng Wang, and Shuxin Ding. Reliability analysis of station autonomous computer system based on fuzzy dynamic fault tree and markov model. *Engineering Reports*, page e12376, 2021. doi: 10.1002/eng2.12376. 2

Y. Zhang, B. Di, Z. Zheng, J. Lin, and L. Song. Distributed multi-cloud multi-access edge computing by multi-agent reinforcement learning. *IEEE Transactions on Wireless Communications*, 20(4):2565–2578, 2021. doi: 10.1109/TWC.2020.3043038. 9, 16

Khasanova Zukhra and Shamsiyeva Ogiloy. Describing data using numerical measures. *PEDAGOGS jurnali*, 9(1):27–35, 2022. 52