Hybrelastic: A Hybrid Elasticity Strategy with Dynamic Thresholds for Microservice-based Cloud Applications

José Augusto Accorsi, Rodrigo da Rosa Righi, Vinicius Facco Rodrigues, Cristiano André da Costa

Applied Computing Graduate Program,
Universidade do Vale do Rio dos Sinos
São Leopoldo, RS, Brazil
E-mail: jarcose@edu.unisinos.br, {rrrighi, vfrodrigues, cac}@unisinos.br

Dhananjay Singh

Electronics Engineering Department, Hankuk(Korea) University of Foreign Studies(HUFS)
E-mail: dsingh@hufs.ac.kr

Abstract: Microservices-based architectures aim to divide the application's functionality into small services so that each one of them can be scaled, managed, implemented, and updated individually. Currently, more and more microservices are used in application modeling, making them compatible with resource elasticity. In the literature, solutions employ elasticity to improve application performance; however, most of them are based on CPU utilization metrics and only on reactive elasticity. In this context, this article proposes the Hybrelastic model, which combines reactive and proactive elasticity with dynamically calculated thresholds for CPU and network metrics. The article presents three contributions in the context of microservices: (i) combination of two elasticity policies; (ii) use of more than one elasticity evaluation metric; and (iii) use of dynamic thresholds to trigger elasticity. Experiments with Hybrelastic demonstrate 10.31% higher performance and 20.28% lower cost compared to other executions without Hybrelastic.

Keywords: Elasticity; Reactive Elasticity; Proactive Elasticity; Scalability; Dynamic Thresholds; Microservices.

Reference to this paper should be made as follows: J. A. Accorsi, R. R. Righi, V. F. Rodrigues, C. A. Costa and D. Singh (2022) 'Hybrelastic: A Hybrid Elasticity Strategy with Dynamic Thresholds for Microservice-based Cloud Applications', *International Journal of Cloud Computing*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: José Augusto Accorsi completed his Bachelor's degree in Computer Science at the University of Vale do Rio dos Sinos (Unisinos, Brazil) in 2021. His research areas include cloud computing and distributed systems.

Rodrigo da Rosa Righi is assistant professor and researcher at the University of Vale do Rio dos Sino (Unisinos, Brazil). Rodrigo concluded his post-doctoral studies at KAIST — Korea Advanced Institute of Science and Technology, under

the following topics: RFID and cloud computing. He obtained his Ph.D. degree in Computer Science from the UFRGS University, Brazil, in 2009. His research interests include load balancing and process migration. He is a member of the IEEE and ACM.

Vinicius Facco Rodrigues is a researcher from the Software Innovation Laboratory – SOFTWARELAB at the University of Vale do Rio dos Sinos (Unisinos, Brazil) since 2016 working on research and industry projects. He completed a Bachelor's degree in Computer Science (Unisinos, 2012) and a Master's degree in Applied Computing (Unisinos, 2016) working with High-Performance Computing and Cloud. He finished his Ph.D. in Applied Computing (Unisinos, 2020) on a SIEMENS HEALTHINEERS project focused on distributed systems for Healthcare 4.0. During his Ph.D., Dr. Rodrigues spent six months working as a visiting researcher at the Machine Learning Data Analytics Lab (MadLab) (Erlangen, Germany) from the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). His research interests include distributed systems, computer networks, high-performance computing, health informatics, and artificial intelligence.

Cristiano André da Costa is a full professor and researcher at the University of Vale do Rio dos Sinos (Unisinos, Brazil). At Unisinos, he works at the Applied Computing Graduate Program and directs the SOFTWARELAB, Unisinos Laboratory of Software Innovation. Since 2012, he has been a Researcher Fellow at the National Council for Scientific and Technological Development (CNPq, Brazil). Dr. Costa has worked as a professor in higher education since 1997. He is a senior member of the ACM and the IEEE. He is also a member of the Engineering in Medicine Biology Society (EMBS), the International Association for the Development of the Information Society (IADIS), and the Brazilian Computer Society (SBC).

Dhananjay Singh is full professor in the Electronics Engineering Department at Hankuk University of Foreign Studies (HUFS), South Korea. His interesting include IoT, cloud computing and Future Internet.

1 Introduction

Cloud computing is a paradigm of computing systems that proposes significant changes in how the organisation and architecture of applications should be thought of (Li & Yang 2021). This procedure allows on-demand access to a range of computational resources, enabling fast and efficient provisioning, according to the use (Li 2020, Kumar et al. 2021). Cloud computing relies on end users to use their resources, as services anywhere, over the Internet. With all its computational resources, this paradigm offers crucial points to services in relation to speed, cost and integrity (Delfin 2019). Despite this, it depends heavily on the ability to transfer data between its services and its end users. Due to its benefits, it has been gaining more and more space in the market, as cloud providers provide all the infrastructure to keep an application running, abstracting all the complexity and maintenance of a corporate data centre.

Scalability is one of the main aspects that have made cloud computing so emerging (Pérez et al. 2018, Qi et al. 2021). According to Li (2020), it is the ability of a system, network, or process to handle the growth of its workload. A scalable system proportionally improves its performance as new resources are added. However, just the ability to increase

resources in an application is impractical, once the application workload returns to its normal state, and does not need all resources allocated, some of them will become idle. To solve it, cloud computing offers elasticity, which is one of the most striking features of this paradigm, and is really a differential of this distributed system (Righi 2013). Elasticity is the degree to which a system is able to adapt to changes in workload, increasing and decreasing resources, so that, at each moment, available resources match current demand as closely as possible (Venkatachalam et al. 2020). This characteristic allows the resizing of resources, in which the application elasticity processes are defined in the context in which at least one, or even several types of resources can be expanded or reduced, according to demand. Each feature type can be seen as a separate process dimension, with its own elasticity properties. Typically, an application's resources can only be provisioned in discrete units such as CPU cores, virtual machines (VMs), or physical nodes.

According to Arouk & Nikaein (2020), native cloud applications must follow three main design standards. The first says that they must have a microservice-based architecture, in which the application is divided into multiple services that operate independently, but together form a product. The second is containerisation, in which an application is packaged into one or several isolated containers and managed through a standard application programming interface (API). The third pattern is continuous integration and delivery (CI/CD), where the application goes through a series of rapid development, build, test, publish, and deploy cycles. A microservice-based application running in the cloud involves the execution of several services, each one can be implemented, deployed and updated independently, without compromising the integrity of the same (Bao et al. 2019). Microservices are usually exposed through containers, which have everything needed for the service to run.

This theme is explored in the literature both in monolithic applications and in applications based on microservices. In monolithic applications there is only one artefact on which the end user makes requests and they are just within the context of that artefact. On the other hand, in distributed applications, such as applications based on microservices, the functionality consists of exchanging requests between each microservice. Therefore, the use of architectures based on microservices can generate dynamic elasticity. As a result, in certain scenarios, a microservice may demand greater processing power, requiring more resources, while others do not require as much. Microservice-based architectures ease performing adjustments to configurations, in order to improve performance. However, these adjustments are not performed by themselves, they require external instructions for these decisions. Choosing the best time for an elasticity action is not a trivial task, as a late or early action results in overloaded or idle resources. To facilitate this task, thresholds are defined, based on metrics, which are responsible for determining when an elasticity action must be taken. Static thresholds are used by several authors (Nardin 2019, Fourati et al. 2019, Rossi et al. 2020, Chaturvedi et al. 2020, Gao et al. 2020, Nashold & Krishnan 2020, Bauer et al. 2019, Biswas et al. 2017), but their definition is not a simple task, as optimal thresholds for one application can be bad for another. For this, the creation of dynamic thresholds (Rodrigues et al. 2017), which adapt according to the application, solves this challenge.

The comparison of thresholds with metrics is the generating factor of the elasticity decision, which can be reactive or proactive. Some authors use reactive elasticity (Fourati et al. 2019, Chaturvedi et al. 2020, Gervasio et al. 2020, Biswas et al. 2017, Ye et al. 2017, Bouabdallah et al. 2016). However, this type of elasticity falls short when it is possible to determine a uniform behaviour in the application workload and it is not possible to anticipate decisions. On the other hand, proactive elasticity, used by other authors (Nardin 2019, Rossi

et al. 2020, Gao et al. 2020, Nashold & Krishnan 2020, Gervasio et al. 2020, Bauer et al. 2019, Kwan et al. 2019, Ye et al. 2017, Bouabdallah et al. 2016), favours scenarios that have a uniform behaviour. However, they cause incorrect decisions while this behaviour is not yet presented. In order to combine the best of both elasticity models, some authors use its combination to manage elasticity Gervasio et al. (2020), Biswas et al. (2017), Bouabdallah et al. (2016), Ye et al. (2017). The CPU metric (Nardin 2019, Fourati et al. 2019, Rossi et al. 2020, Gao et al. 2020, Nashold & Krishnan 2020, Gervasio et al. 2020, Bauer et al. 2019, Biswas et al. 2017, Kwan et al. 2019, Ye et al. 2017, Bouabdallah et al. 2016) is used constantly in elasticity managers. However, metrics such as network consumption (Gervasio et al. 2020) also represent the behaviour of the application and based on this it is possible to make decisions as precise as the CPU.

As a result, the proposed work presents the Hybrelastic, which is an elasticity management model for applications based on microservices architecture. Hybrelastic focuses on the horizontal modality of elasticity, combining reactive and proactive resource allocation policies through CPU utilisation and network consumption metrics. Its thresholds are dynamically created at runtime, based on workload. Hybrelastic's reactive elasticity compares the metrics of each microservice with the thresholds created. Proactive elasticity works with an auto-regressive integrated moving average (ARIMA) algorithm, that seeks to predict new metrics through the use of moving averages. First, Hybrelastic does a reactive analysis, and if it doesn't lead to any elasticity decision, the model performs another analysis, this time, proactive. This article presents the following contributions to the state-of-the-art on the microservice elasticity issue: (i) combination of two elasticity policies; (ii) use of more than one elasticity evaluation metric; and (iii) use of dynamic thresholds to trigger elasticity.

The present study is organised in the following sections: Section 2 analyses existing works which are directly related to the theme of this article. Section 3 presents the elaborated architecture, the developed model, both of the elasticity manager and the microservice and the evaluation environment. Section 4 presents the Hybrelastic evaluation methodology. Section 5 presents the results obtained, and their analysis. Finally, Section 6 summarises and concludes the work done.

2 Related Work

Currently, the development of mechanisms for managing elasticity in microservices is the target of different work fronts, whether carried out by professional or academic spheres. This section presents a collection of related articles resulting from a research strategy carried out on the *IEEE* (2021), *ResearchGate* (2021), *ACM* (2021), *Google Scholar* (2021) and *Springer* (2021) platforms, held in April 2021, using the following terms: "Elasticity in Microservices", " Elasticity in Cloud Computing", "Reactive Elasticity", "Proactive Elasticity", "Cloud Computing Elasticity Machine Learning", to filter by methods exploited in proactive strategies, and "Elasticity Forecast". Furthermore, they should indicate key points of its architecture, such as metrics, elasticity management, allocation policies and virtualisation model. Among the sources found, those present in Table 1 were selected due to the proximity of some factors.

The development of mechanisms to manage the elasticity of resources in cloud applications is largely seen in scientific works dealing with horizontal elasticity, most of which developed through proactive elasticity (Nardin 2019, Rossi et al. 2020, Gao et al.

Table 1 Related work comparison.

Article	Metrics	Elasticity Policy	Architecture	Virtualisation	Algorithm
Nardin (2019)	CPU	Proactive	Microservices	VM	ARIMA
Fourati et al. (2019)	CPU	Reactive	Microservices	Container	Own algorithm
Rossi et al. (2020)	CPU	Proactive	Microservices	VM	RL
Chaturvedi et al. (2020)	runtime	Reactive	Monolithic	VM	Own algorithm
Gao et al. (2020)	CPU and memory	Proactive	Monolithic	VM	ARIMA, BRR and LSTM
Nashold & Krishnan (2020)	CPU	Proactive	Monolithic	VM	SARIMA and LSTM
Gervasio et al. (2020)	CPU, memory and network	Proactive and reactive	Monolithic	VM	ARIMA
Bauer et al. (2019)	Runtime	Proactive	Monolithic	VM	SARIMA
Biswas et al. (2017)	CPU	Proactive and reactive	Monolithic	VM	LR
Kwan et al. (2019)	CPU and memory	Proactive	Microservices	Container	Own algorithm
Ye et al. (2017)	CPU	Proactive and reactive	Monolithic	Container	ARMA
Bouabdallah et al. (2016)	CPU	Proactive and reactive	Monolithic	VM	Time series algorithm

2020, Nashold & Krishnan 2020, Gervasio et al. 2020, Bauer et al. 2019, Kwan et al. 2019, Ye et al. 2017, Bouabdallah et al. 2016), while only three of the academic works analysed work with reactive elasticity (Fourati et al. 2019, Chaturvedi et al. 2020, Gervasio et al. 2020, Biswas et al. 2017, Ye et al. 2017, Bouabdallah et al. 2016). The vast majority of authors work with metrics related to application CPU utilisation (Nardin 2019, Fourati et al. 2019, Rossi et al. 2020, Gao et al. 2020, Nashold & Krishnan 2020, Gervasio et al. 2020, Bauer et al. 2019, Biswas et al. 2017, Kwan et al. 2019, Ye et al. 2017, Bouabdallah et al. 2016), as there is a clear relationship between CPU utilisation and service performance. The higher the usage, the lower its performance.

In addition to CPU, there are other metrics used for elasticity management in cloud computing, such as runtime (Chaturvedi et al. 2020), disk memory analysis (Gao et al. 2020, Gervasio et al. 2020, Kwan et al. 2019), network consumption (Gervasio et al. 2020) and response time (Bauer et al. 2019). The usage of microservice-based applications have increased over time, when it comes to cloud computing, according to Arouk & Nikaein (2020), there is a great recommendation for the use of microservices, as they provide advantages, mainly regarding maintenance and performance issues. Yet, the large part of the analysed works deal with elasticity in monolithic applications (Chaturvedi et al. 2020, Gao et al. 2020, Nashold & Krishnan 2020, Gervasio et al. 2020, Bauer et al. 2019, Biswas

et al. 2017, Bouabdallah et al. 2016), as this type of architecture is more common, therefore, more used. Already, the works of Nardin (2019), Fourati et al. (2019), Rossi et al. (2020), Kwan et al. (2019) deal with elasticity in microservices, besides, all the studies carried out in the referred academic works, with the exception of Fourati et al. (2019) and Kwan et al. (2019) use VMs as their virtualisation mechanism.

Mechanisms that proactively manage resource elasticity deal with the implementation of algorithms that take responsibility for decision making regarding the provisioning and deprovisioning of resources for the application. Many of them work with time series (Nardin 2019, Gao et al. 2020, Nashold & Krishnan 2020, Gervasio et al. 2020, Bauer et al. 2019, Kwan et al. 2019, Bouabdallah et al. 2016). Others work with machine learning models, as Bayesian Ridge Regression (BRR) (Gao et al. 2020, Biswas et al. 2017), Long Short Term Memory (LSTM) (Gao et al. 2020, Nashold & Krishnan 2020), Linear Regression (LR) (Biswas et al. 2017) and also with Reinforced Learning (RL) (Rossi et al. 2020).

In view of the related work, CPU is the most used metric to perform actions related to application elasticity, however it is not the only metric capable of helping in this type of decision. The network analysis can be a great indicator of the application workload growth, it is also noticed that there is a scarcity of mechanisms that work with the combination of reactive and proactive elasticity in microservices. In addition, proactive and reactive elasticity are widely used. However, they are used separately, leaving gaps in how they could be worked together, in the most diverse cloud providers. Taking these statements and related work into account, when it comes to microservice-based application and elasticity, it is clear that there is a gap in the state-of-the-art, and this gap can be defined as the lack of works that contribute to the context of elasticity in microservices, with a combination of policies for elasticity through more than one analysis metrics.

3 Hybrelastic Model

This section presents the elasticity management model proposed by this work, the Hybrelastic. In subsequent subsections, design decisions for Hybrelastic and its architecture will be discussed. It will also be discussed about the blocks of the model, the Elasticity Manager and Metrics Manager, demonstrating how elasticity decisions and calculation of thresholds happen.

3.1 Design Decisions

Cloud providers such as *Amazon Web Services* (2021) (AWS) provide elastic application management, however, this management is based on scheduling or on previously established thresholds, using reactive elasticity through rules based on the use of application resources. This type of approach becomes simple, however, it is not always an easy strategy, as experience with the system is required in order to know the correct threshold for making elasticity decisions. In other words, these decision-making mechanisms with fixed thresholds do not really solve the application sizing problems. When they are configured so as not to represent optimal thresholds, the elasticity decision can be taken late, making it ineffective for certain scenarios.

Due to these problems arising from elasticity, this work proposes the Hybrelastic, a combination of reactive and proactive elasticity, creating a hybrid model and exploring the advantages of each of the two resource allocation policies. The model uses dynamically

defined thresholds, in order to abstract from the user the task of creating limits, according to the use of the application's resources. These thresholds are calculated based on the metrics collected on the last hour of analyses. The upper threshold is calculated from the 90% percentile, while the lower one is calculated from the median of these data. (Gervasio et al. 2020). The reactive part of the model will work directly with the dynamic thresholds calculated by the algorithm, while the proactive part will use a time series algorithm for workload anticipation, having CPU utilisation and network consumption as metrics. CPU metrics indicate the performance of an application. However, the network metric does not convey this idea, but the drop in performance can be identified with the constant increase in the network consumption metric. This mechanism indicates that the performance of the application can be compromised if an elasticity action is not taken.

Figure 1 presents a conceptual view of the Hybrelastic model. It demonstrates all the processes that are used in Hybrelastic. These processes work together so that at the end of their execution, an action of elasticity on a microservice that presents some bottleneck is taken. It also shows the model's execution hierarchy, starting from a metric reading and ending with an elasticity decision. Hybrelastic consists of two blocks: the Metrics Manager and the Elasticity Manager. The Metrics Manager will be responsible for collecting CPU and network values, in a pre-established interval and storing them in a database, besides being responsible for calculating the application's dynamic thresholds. The Elasticity Manager, in turn, will use the thresholds previously calculated by the Metrics Manager as a basis for the elasticity decision. The reactive part will compare the latest value of each metric against the thresholds, while the proactive part will consume the information that is stored in the database and, based on that, will anticipate the application workload change. Hybrelastic will be implemented in an application based on the microservices architecture that will be used for image processing. Thus, the present work considers that Hybrelastic:

- Acts in the infrastructure layer as a service in public cloud environments;
- Uses CPU utilisation and network consumption metrics for elasticity decision making;
- Uses reactive and proactive allocation policy;
- Works with microservices;
- Be able to grow and shrink VMs of each of the microservices according to its workload.

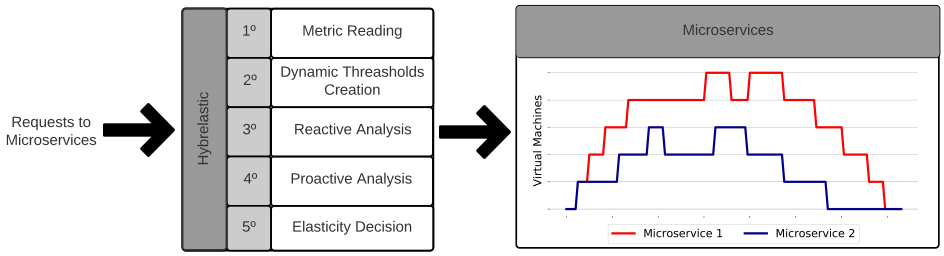

Figure 1 Conceptual view of the Hybrelastic model.

The operation of Hybrelastic in public clouds is considered because there is a growth and greater ease in its adherence. CPU and network metrics were selected from an analysis of microservices, showing that network and CPU would be available for all test scenarios.

3.2 Architecture

This subsection introduces the Hybrelastic architecture. The architecture proposes hybrid elasticity with CPU and network utilisation as metrics for applications based on microservices architecture. The limitations and gaps in reactive and proactive allocation policies are considered and a combination of the two is sought to exploit their benefits. It proposes a mechanism to manage the elasticity of applications based on microservices architecture, acting in the infrastructure as a service layer in cloud environments, combining reactive and proactive resource allocation policies, and using network consumption and CPU utilisation as metrics for elasticity decisions.

The system architecture is represented in Figure 2, along with a microservice-based application that Hybrelastic manages, demonstrating how it acts in the cloud in relation to the different actors in the system, which are divided into three: cloud manager, developer and user. The cloud manager is responsible for setting up Hybrelastic in a cloud environment as well as maintaining it. Its communication with Hybrelastic can be done via Secure Shell (SSH), since if it is necessary to change any parameter at run time, just access the server that Hybrelastic is hosted and update the parameters, without the need to take it down and bring it up with the new information. The developer is responsible for creating and configuring the microservices, which will be integrated with Hybrelastic, interacting directly with the microservices orchestrator. However, the user is only responsible for using the microservices submitted to the cloud.

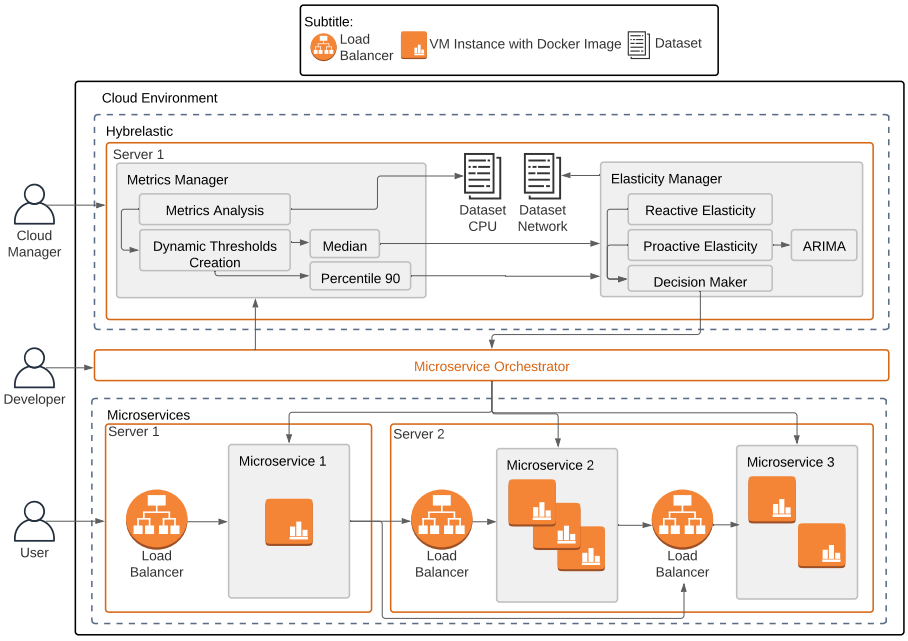

Figure 2 Hybrelastic architecture with a microservice-based application.

In addition, Figure 2 also shows the division of components within Hybrelastic. These components are the Metrics Manager and the Elasticity Manager. The Metrics Manager is

responsible for collecting and analysing the network consumption and CPU utilisation of each of the microservices, as well as defining the dynamic thresholds for these metrics. The Elasticity Manager, on the other hand, is responsible for the integration of reactive and proactive allocation polices, and for the decision of the system's elasticity. Hybrelastic reads the metrics for each of the microservices that it is managing. The Metrics Manager reads the information of each microservice, builds the data that is used for the Elasticity Manager. The Elasticity Manager is responsible for the allocation and deallocation of VMs for the microservices. These actions are made reactively and proactively. The reactive analysis is done via the comparison between metrics and thresholds, while the proactive is done via the comparison between the next predicted metrics and the thresholds. This predictions are done via an algorithm of time series denominated ARIMA.

3.2.1 Metrics Manager

According to Gervasio et al. (2020), IaaS cloud resources management can be used efficiently by predicting workloads or resource utilisation patterns. The public cloud workload, on the other hand, is not deterministic; therefore, it is necessary to use some techniques to predict the type of nature of workloads. In the literature, the workload is usually measured through application metrics (Fourati et al. 2019, Chaturvedi et al. 2020, Gervasio et al. 2020), however, the works that seek to have a more realistic control of their applications try to predict metrics of their VMs, with the aim of having a more coherent control of its resources. Hybrelastic comprises the Metrics Manager that periodically collects metrics from the microservices instances and compute dynamic thresholds. Algorithm 1 shows the main activities the Manager performs.

Algorithm 1 Metrics Analysis, in Metrics Manager, for each Microservice

Require: $microservices$
Ensure: metrics and thresholds calculated for each microservice

```
1: while True do
2:     for microservice in microservices do
3:         cpu ← 0
4:         network_in ← 0
5:         network_out ← 0
6:         for instance in microservice do
7:             cpu ← cpu + get_cpu(instance)
8:             network_in ← network_in + get_bytes_network_in(instance)
9:             network_out ← network_out + get_bytes_network_out(instance)
10:        end for
11:        microservice.cpu ← determine_total_cpu(cpu)
12:        microservice.network ← network_in + network_out
13:        create_thresholds(microservice)
14:        save_dataset(microservice)
15:    end for
16:    wait(periodical_interval)
17: end while
```

The Metrics Manager employs CPU utilisation and network consumption as the two fundamental metrics so that the Elasticity Manager can make its decisions regarding the

elasticity of the test microservice. The CPU analysis is usually done through percentage, which can vary from 0% to 100%, and no additional treatment is required in this metric. However, for network analysis, cloud environments provide data for network input and output, measured in bytes, with only a minimum value, not maintaining a maximum value for network consumption. In addition, the Metrics Manager is responsible for creating dynamic thresholds that will be used later for elasticity analysis. Thresholds are calculated based on the last hour of analysis. The upper threshold is calculated from the 90% percentile, while the lower one is calculated from the median of these values (Gervasio et al. 2020). The metric values of each microservice is analysed from each of its instances. Equation 1, refers to the calculation of CPU usage where n refers to to the number of VMs of each microservice. The value for the network is calculated from the network average of each microservice VM.

$$total\_cpu = \frac{\sum_{i=n}^{n} CPU\_utilisation\_per\_VM}{n} \tag{1}$$

3.2.2 Elasticity Manager

As shown in Figure 2, the Elasticity Manager is divided into two blocks, the reactive model and the proactive model. The reactive model is responsible for the reactive analysis of the elasticity of microservices, comparing the values of current metrics with the dynamically created thresholds. The proactive model, on the other hand, uses time series through the ARIMA algorithm, making the prediction of future values, with the purpose of predicting the change in the workload, before the reactive component needs to make any decision.

3.2.3 Reactive Model

The use of the reactive resource allocation policy is widely accepted in the market due to its simplicity of configuration and for being native to the cloud management system (Chaturvedi et al. 2020, Fourati et al. 2019), but the proactive elasticity has been well accepted in the academy (Nardin 2019, Rossi et al. 2020, Gao et al. 2020, Nashold & Krishnan 2020, Gervasio et al. 2020, Bauer et al. 2019), as it seeks to anticipate the decisions of reactive elasticity and overcome its bottlenecks. As a result, the Elasticity Manager of the proposed model was designed based on the adoption of two resource allocation policies, making use of the benefits of both. However, one of the problems that reactive elasticity presents refers to the decision thresholds, since these do not change according to the workloads, causing oscillation problems (Gervasio et al. 2020), often changing the number of VMs to handle the workload of a given service.

The model analyses the metrics collected by the Metrics Manager, along with the dynamic thresholds previously calculated for the reactive analysis of the microservices system. Based on this analysis, it is possible to determine whether the microservices that were submitted to Hybrelastic need to allocate or deallocate VMs, optimising processing time, because once the reactive model is triggered, the proactive model is ignored, since the elasticity was decided reactively.

3.2.4 Proactive Model

The Elasticity Manager also provides proactive resource allocation, this component will be responsible for the demand anticipation strategy, using the ARIMA data forecasting method. To achieve this prediction, the ARIMA algorithm will use the metrics, previously

saved by the Metrics Manager, which have already been analysed by the reactive part of the code, and compare them with the thresholds for each of the evaluation metrics. The prediction of this data occurs during the execution of the system and it tries to predict the next reading. As a result, it is essential that the data prediction algorithm does not consume a lot of time, as the prediction is triggered through the use of two computational threads, one for processing each metric.

As the prediction of future values occurs during the execution of the system, the ARIMA training should not take too long, as the data undergoes periodic training every time a new metric is read. In this way, they are divided into training sets and test sets. The training set is composed of 66% of the total historical data sample, and the test set is composed of 34% of the total historical data sample. After training the data, the forecast of time resource consumption $t+$ is performed for each metric, and if the accuracy is acceptable, it is used to make the elasticity decision. Data analysis is performed using the *auto_arima* (2021) library. A library that seeks to identify the most suitable parameters of p, q and d for the data set in question (Yermal & Balasubramanian 2017). Thus, the library tests all possible combinations of variables and selects the best model found, optimising processing and testing time for the various possible models. After dividing the collected data into testing and training, the analysis is done, since the algorithm uses the model selected by auto_arima to start predicting future data, whose time is $t + 1$.

The model's precision is calculated by the value of the mean absolute percentage error (MAPE), this percentage is equivalent to the percentage of errors that the model found; therefore, the model's precision can be calculated by Equation 2, where n refers to the number of samples, e_t is the absolute error of the last forecast, defined as the modulus of the difference between the latest prediction and the latest measurement, and d_t, which can be considered the latest prediction, or the next workload value.

$$Precision = 100 - \frac{1}{n}\sum \frac{|e_t|}{d_t} \tag{2}$$

3.2.5 Combining the Reactive and Proactive Models

The Hybrelastic model works based on the integration of the two managers discussed above, the Elasticity Manager and the Metrics Manager. According to the Algorithm 2, firstly, before the initialisation of the model, it is assumed that the thresholds have already been previously calculated by the Algorithm 1. Algorithm 2 starts with a loop that will always be executed while the Hybrelastic be running. After that, there is a loop through all the microservices in which Hybrelastic is managing. At lines 3 and 4, the values for the counter responsible to trigger an elasticity decision are assigned to the local variables $lower\_counter$ and $upper\_counter$. From lines 5 to 6, the reactive analysis happens. The Hybrelastic checks if the microservice CPU ($ms.cpu$) or the microservice network ($ms.network$) violate any threshold; if so, the counter responsible for triggering a resource allocation or deallocation is incremented in one unit. After that, the lines 12 and 16 check of the counter is equal to 3. If these counters reach the value of three, an elasticity decision is triggered. This type of approach is performed in order to discard any false positives, as there may be fluctuations in the reading of the metrics and triggering the elasticity actions (Gervasio et al. 2020) erroneously. If the counter values are equal to 3, lines 13 and 17 trigger the deallocation and allocation of VMs to the microservice, increasing or decreasing a unit of VM to the microservice. Then a cool down starts, an amount of time the microservices analytics are suspended for the new instance, or the instance that was just removed, to change

in the VM cluster. This type of mechanism is used to prevent the system from oscillating. Then, the counters are set to zero.

Algorithm 2 Elasticity Decision Making in Elasticity Manager Using the Combination of Reactive and Proactive Elasticity

Require: $microservices$
Ensure: elasticity decision for each microservice
1: **while** True **do**
2: **for** ms in $microservices$ **do**
3: $lower\_counter \leftarrow ms.lower\_counter$
4: $upper\_counter \leftarrow ms.upper\_counter$
5: **if** $ms.cpu < ms.cpu\_lower\_threshold$ OR $ms.network < ms.network\_lower\_threshold$ **then**
6: $lower\_counter \leftarrow lower\_counter + 1$
7: $return$
8: **else if** $ms.cpu > ms.cpu\_upper\_threshold$ OR $ms.network > ms.network\_upper\_threshold$ **then**
9: $upper\_counter \leftarrow upper\_counter + 1$
10: $return$
11: **end if**
12: **if** $lower\_counter = 3$ **then**
13: $deallocate\_vm()$
14: $cool\_down()$
15: $clear\_counters(lower\_counter, upper\_counter)$
16: **else if** $upper\_counter = 3$ **then**
17: $allocate\_vm()$
18: $cool\_down()$
19: $clear\_counters(lower\_counter, upper\_counter)$
20: **end if**
21: **if** $(ms.predicted\_cpu < ms.cpu\_lower\_threshold$ AND $ms\_cpu\_accuracy \geq 70)$ OR $(ms.predicted\_network < ms.network\_lower\_threshold$ AND $ms\_network\_accuracy \geq 70)$ **then**
22: $lower\_counter \leftarrow lower\_counter + 1$
23: $return$
24: **else if** $(ms.predicted\_cpu > ms.cpu\_upper\_threshold$ AND $ms\_cpu\_accuracy \geq 70)$ OR $(ms.predicted\_network > ms.network\_upper\_threshold$ AND $ms\_network\_accuracy \geq 70)$ **then**
25: $upper\_counter \leftarrow upper\_counter + 1$
26: $return$
27: **end if**
28: **if** $lower\_counter = 3$ **then**
29: $deallocate\_vm()$
30: $cool\_down()$
31: $clear\_counters(lower\_counter, upper\_counter)$
32: **else if** $upper\_counter = 3$ **then**
33: $allocate\_vm()$
34: $cool\_down()$
35: $clear\_counters(lower\_counter, upper\_counter)$
36: **end if**
37: $ms.lower\_counter \leftarrow lower\_counter$
38: $ms.upper\_counter \leftarrow upper\_counter$
39: **end for**
40: $wait(periodical\_interval)$
41: **end while**

If the reactive analysis does not increment the counters, they will never reach value 3, it means the reactive elasticity does not happen, thus, the rest of the algorithm will be

responsible for the proactive analysis of Hybrelastic. Line 21 and 24 try to identify if the next metric predicted by the $ARIMA$ algorithm passes the limits of defined the upper and lower thresholds, for each of the metrics and the precision obtained by it is greater than or equal to 70% (Gao et al. 2020), according to the Equation 2. Next, the algorithm checks if any of the counters have reached 3 and, if this check is true, elasticity decisions are assigned followed by a cooling in the microservice in question and clearing the value on the counters. After that, the algorithm retrieves to the microservice, the values for its counters. The main loop that starts on line 2 runs for each microservice, and after that, Hybrelastic waits for a periodical interval before starting its next analysis. The Algorithm 2, is executed every metric reading, which was defined in a one minute interval; therefore, every minute, the Algorithm 1, will perform a metrics analysis and, in the sequence, the Algorithm 2 will be responsible for the elasticity decision for the microservices of Hybrelastic test.

Figure 3 presents the sequence of steps involving the components of the architecture. The sequence diagram assumes a scenario where the elasticity action takes place after a true proactive modulus analysis. It starts in the cloud manager, enabling Hybrelastic to read the microservices metrics, then the Metrics Manager is responsible for analysing the metrics (creating the network consumption metric), persisting them in the data set and creating the thresholds. With the metrics finished, the Elasticity Manager checks if the current metrics exceed any of the thresholds through the reactive model. As this verification was false, the analysis of the prediction of the next metrics against the thresholds is done by the proactive module, and this analysis becomes true, indicating that the Hybrelastic needs to take an elasticity action. Action is then taken in the Elasticity Manager and cooling is triggered.

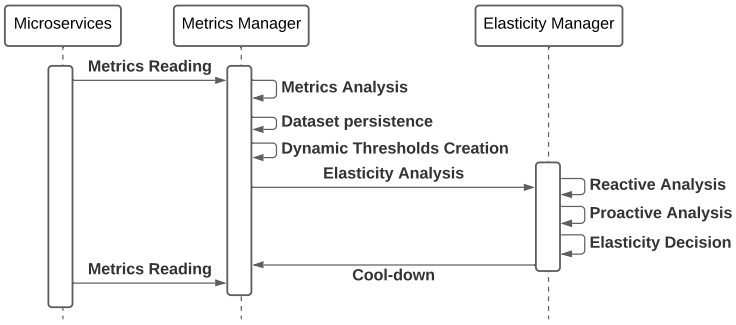

Figure 3 Hybrelastic sequence diagram of one interval of analysis that generates an elasticity action.

4 Methodology

In this section, the evaluation methodology used by Hybrelastic will be discussed, as well as the microservice that was submitted to the model tests, will be presented the experiments that were carried out with the objective of evaluating the elasticity solution. Therefore, the purpose of the evaluation is to analyse the behaviour of Hybrelastic.

4.1 Evaluation Microservice

Hybrelastic was developed to work under applications based on microservices architecture. And, for purposes of testing the proposed model, a microservice application that works with image processing was developed. The image processing theme was chosen so that each microservice works hard, as required, in order to stimulate the creation of new instances. The Microservice was developed using *Docker* (2021), as it facilitates the installation in virtual machines. An example implementation of this architecture is shown in Figure 4. It corroborates the application that will be subjected to testing by Hybrelastic, which uses three different microservices, one to be the home page, one for image processing and one for the database, which stores the created and edited images.

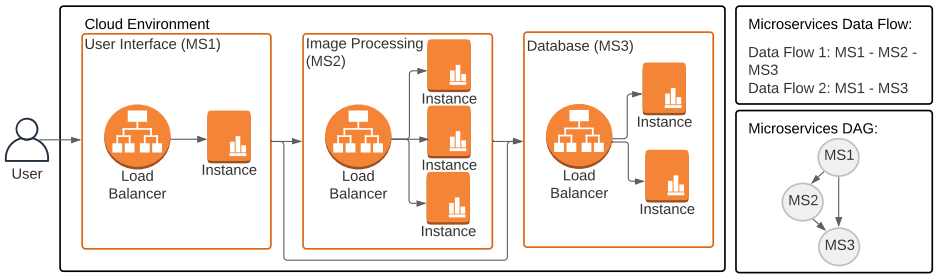

Figure 4 Image processing microservice that was used for Hybrelastic evaluation.

The analysis of Figure 4 shows that each microservice can have a different number of instances, this is because the application runs in parallel, as it can be accessed by countless users at the same time. Thus, image processing must require more complex processing than other microservices, taking longer to process, and requiring more running instances. As a result, there is a load balancer to distribute requests to the instances of each microservice. One way for Hybrelastic to manage the application's microservices would be to know exactly the behaviour of each part of it; however, analysis of this magnitude tend to be very complex, as they require a very detailed analysis of the microservice and, in a given period, there may be application overloads, while in other periods it does not. In view of this, the most suitable approach for Hybrelastic is to manage each microservice individually, regardless of their relationships, just the resources they are using. In this approach, the bottleneck of the system will be that microservice that is showing a low performance.

As mentioned earlier, the microservice shown in Figure 4 has three stages, in which they communicate with each other. The first stage refers to the user interface, which would be exposed so that the end user can interact with the application and, from there, work with image processing and writing and reading in the database. The second stage is represented by the image processing microservice, which is responsible for modifying an image that the user loads into the interface microservice and saving it in the database. This microservice allows rotation, compression, blurring, edge contouring and the creation of a thumbnail for each image executed, in addition to the creation of a Mandelbrot Set. Finally, the third stage presents a database microservice, which is responsible for saving the images created by the image processing microservice and returning them to the user interface microservice.

To deploy each of the microservices to the AWS cloud provider, three *Auto Scaling Groups* (2021) were created. These groups which can be defined as a set of *EC2* (2021)

instances, which are treated as a logical grouping for management and automatic scalability purposes. These Auto Scaling Groups increase or decrease one of their EC2 VMs according to Hybrelastic elasticity decisions. Each of these groups communicates with the others through *Elastic Load Balancing* (2021), which are responsible for redirecting the workload to the VMs of each of the microservices.

4.2 Prototype

This section details the Hybrelastic implementation. The model was implemented in Python, as the testing microservice was hosted on AWS, it was necessary to use *Boto3* (2021), a standard Amazon SDK that provides a dedicated API for communicating its components with peripherals. Hybrelastic, through this API, is able to collect the metrics of the microservices that were created in EC2 instances, with *Cloud Watch* (2021), at runtime, and make its analysis through the metrics manager. Hybrelastic collects data from microservices every minute. The test microservice was hosted at an Amazon datacenter in the city of São Paulo – SP, Brazil, and the requests were made through a script written in Python, which executed calls to the microservices every 20 minutes. AWS elasticity was configured only through the CPU utilisation metric, as without dynamic thresholds, network consumption does not present a uniform behaviour, making it impossible to detect precise thresholds Gervasio et al. (2020). This script made requests to the microservices and stored information regarding the request in a local database. This information contained the latency, the total response time, and the HTTP status, which indicated whether the response occurred as expected.

After the elasticity action, the Hybrelastic enters in a cool-down period where it stops its readings until a certain amount of time has passed. This type of approach should be used, as soon after an instance is allocated or deallocated, it goes through a process of creation or shutdown, and this process causes oscillations in the VMs, causing variations in their metrics and, consequently, hindering future analysis . Based on these situations, the Hybrelastic prototype assumes that the following parameters are followed:

- Interval of 60 seconds between each of the microservices metric readings;
- It takes three elasticity decisions in a row (three to increase or three to decrease) for Hybrelastic to take an action;
- After an elasticity action the Hybrelastic will go into a three-minute cooling.

The Hybrelastic execution parameters mentioned above were defined from tests performed. An interval of at least 60 seconds is required during readings, as Cloud Watch updates microservices metrics every minute. Three elasticity decisions in a row are required before Hybrelastic takes action to avoid false positives, as addressed by Gervasio et al. (2020). A cool-down period is also required after an elasticity action because when an instance is created, its metrics oscillate as it goes through a cycle of testing and installing the microservice image, before it is ready for use.

4.3 Infrastructure and Scenarios

The testing infrastructure for Hybrelastic was built on the AWS public cloud. To do so, microservices were created using an Auto Scaling Group, EC2 instances and Cloud Watch. Auto Scaling Group is a cluster that manages and organises EC2 instances, which was used

for the simplicity of increasing or decreasing a microservice virtual machine. Cloud Watch is a metrics manager for instances stored in AWS, which is consumed by Hybrelastic to read CPU and network metrics values for each microservice. It was utilised instances EC2 type t2.micro to build the microservice. This type of instance was featured with 1 virtual CPU (vCPU), which is the default and maximum number of vCPUs available, with 1 GibiByte (GiB) of memory, and 3.3 GHz Intel Scalable Processor.

The installation of the microservice image onto the EC2 instance was done via a Docker Image. Every time an allocation of a new VM was triggered by the Hybrelastic, the Auto Scaling Group runs an script that install in the EC2 instance, the image of Docker and the *GitHub* (2021). After that, the instance clones a GitHub repository that contains the code for the microservice, and install it via Docker. After that, the microservice is ready for use. To analyse the Hybrelastic behaviour, was defined the following scenarios:

- Scenario 1: Test with reactive elasticity provided by the AWS cloud provider, without Hybrelastic;
- Scenario 2: Test with only the network consumption metric, with Hybrelastic;
- Scenario 3: Test with only CPU utilisation metric, with Hybrelastic;
- Scenario 4: Test with network consumption and CPU utilisation metrics, with Hybrelastic.

It is important to emphasise that for all the tests, the workload was generated by the script mentioned beforehand. the Hybrelastic does not act on Scenario 1, so the thresholds of 70% of CPU utilisation for new resource allocation and 30% for new resource deallocation (Fourati et al. 2019, Chaturvedi et al. 2020) were configured. The elasticity decision for the scenario in question is assigned by AWS. Furthermore, it is important to note that each test scenario was run for five hours, in order to have greater precision in its definitions

4.4 Evaluation Metrics

Hybrelastic results will be compared with each of the aforementioned scenarios. The metrics taken into account for the evaluation of the proposed model were the response time of Hybrelastic, in order to measure its performance in comparison to the elasticity offered by AWS, and also the cost of each test routine, for the purpose to measure the financial viability of the model. The response time of each scenario was calculated through the sum of time spent divided by the number of requests made, according to Equation 3, where n means the number of requisitions and the a_i means the time spent by the requisition. The main objective of the evaluation is to analyse the behaviour of Hybrelastic, considering its performance and cost. For such, the analysis of its performance was calculated through the equation 4. This equation compares the response time of Scenarios 2, 3, and 4 ($Scenario_n$), defined beforehand with the response time of Scenario 1 ($Scenario\_A\_W\_S$), which is the AWS one. In it, according to Gervasio et al. (2020), positive values represent a performance improvement in relation to execution without Hybrelastic, while negative values represent a loss.

The cost was calculated from the Equation 5, where n refers to number of VMs used in the scenario, $PricePerVM$ refers to the hourly demand price of $0.0186 and the $ConsumedTimeUnit$ is calculated from a flat fee over which can be about minutes or hours. A flat rate per minute implies that a pricing unit is charged for each minute the

instance is available. The hourly rate counts 60 units every hour the instance is available. However, it is counted from the first minute it is used; therefore, an instance that is available for 61 minutes will be billed for 120 units. All of the test scenarios mentioned above are based on the fixed hourly rate, this rate is the default rate for instances provided by AWS.

$$AverageRequisitionTime = \frac{1}{n}\sum_{i=1}^{n} a_i \tag{3}$$

$$Performance = (1 - \frac{Scenario_n}{Scenario\_A\_W\_S}) \times 100 \tag{4}$$

$$Cost = (\sum_{i=1}^{n} ConsumedTimeUnits) \times PricePerVM \tag{5}$$

5 Results

This section will address the results obtained by Hybrelastic, which were based on the evaluation methodology presented in Section 4. At first, an analysis of the performance of Hybrelastic in comparison with the elasticity of AWS will be presented, according to the four scenarios referenced in subsection 4.2. Then, a cost analysis will be shown and it will end with a metrics analysis.

5.1 Performance Analysis

The performance analysis of Hybrelastic, in comparison with the elasticity offered by AWS, was performed from the analysis of the response time of each request made to the test microservice.

The performance analysis was based on requests through the User Interface microservice (MS1), which redirects calls to the other two microservices: Image Processing (MS2) and Database (MS3). Table 2 interprets the communication flow between MS1, MS2 and MS3 as Data Flow 1, while Data Flow 2 represents the communication between MS1 and MS3. As represented in Figure 4, the $AverageRequisitionTime$ is the sum of the two flows, with the purpose of knowing what is the average time in which the entire application is executed, while the Mean represents the average of all the requisitions and the Standard Deviation represents the variation between the requisitions.

Table 3 shows the performance calculation of the test scenarios against AWS default elasticity. It can be seen in Table 3 that Hybrelastic, with only the CPU metric (Scenario 2), presents a loss of 13.99% of $Performance$ in relation to the elasticity of AWS (Scenario 1) and, only with the network metric (Scenario 3), it obtains a gain of 6.63%. On the other hand, the complete Hybrelastic (Scenario 4), with the CPU and network metrics, presents a $Performance$ gain of 10.31% in relation to the AWS elasticity, decreasing the average time of requests by approximately eight seconds.

The combination of reactive and proactive elasticity across CPU and network metrics made Hybrelastic stand out from the other three test scenarios, particularly against the AWS scenario. This performance improvement was expected, as the combination of elasticities and the use of dynamic thresholds meant that at each new reading of microservices, the thresholds were recalculated, providing an understanding of the application behaviour by

Table 2 Statistical analysis of each test scenario.

Scenario	**Data Flow**	**Standard Deviation (s)**	**Mean (s)**	$AverageRequisitionTime$ **(s)**
Scenario 1	1	16.48	49.73	77.11
	2	24.84	27.38	
Scenario 2	1	11.36	51.82	87.9
	2	25.3	36.08	
Scenario 3	1	24.36	47.03	71.99
	2	22.42	24.96	
Scenario 4	1	9.46	50.22	69.16
	2	18.98	18.94	

Table 3 Hybrelastic performance compared to Amazon AWS.

Scenario	$AverageRequisitionTime$ **(s)**	$Performance$ **(%)**
Scenario 1	77.119	
Scenario 2	87.9	-13.99
Scenario 3	71.99	6.63
Scenario 4	69.16	10.31

Hybrelastic. The combination of elasticities serves so that, even if no threshold is crossed in the present analysis, there is an attempt to anticipate metrics in the future, in order to make an early and accurate decision, if necessary, based on historical data. Therefore, both metrics correctly represent the application's behaviour, and as the CPU reached its cutoff values, the network followed it. Figure 5 demonstrates how the dynamic thresholds behaved according to the workload in the MS2 microservice. Note that as the metrics go up, the upper and lower thresholds are recalculated, with the purpose of always reflecting the current workload. The use of dynamic thresholds solves the problem that very high thresholds provide to the application. Very high thresholds take longer to be reached, causing the instance to work at its limit, until this threshold is reached, so that an elasticity action is applied, as represented in Figure 6, which covers the MS2 of the AWS scenario.

5.2 Cost Analysis

In addition to the performance analysis of the Hybrelastic, a cost analysis was carried out, with the intention of identifying whether the model proposed by the work is financially viable. To measure the total cost of each scenario, it is necessary to take into account all the VMs available during the scenario.

Table 4 presents a cost analysis for each of the test scenarios. This analysis is done using the Equation 5, which displays the cost that the test application implies for the two types of charges that AWS supports: hourly charge or minute charge. As Hybrelastic was configured to work with charges per minute, and does not take into account the time the instance is running for its analysis, the values in Table 4, which refer to the charge per hour, were made through estimates. However, the values per minute were obtained in the tests.

The $ConsumedTimeUnit$ column indicates how many units will be charged by the scenario, the more units, the more costly it will be. In an analysis, it is possible to verify

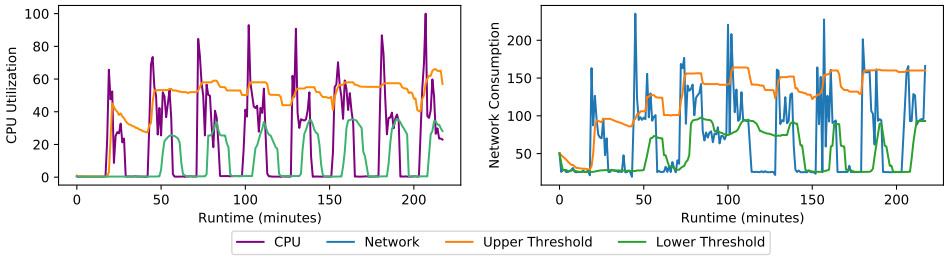

Figure 5 Definition of dynamic thresholds from the MS1 microservice's workload through Hybrelastic.

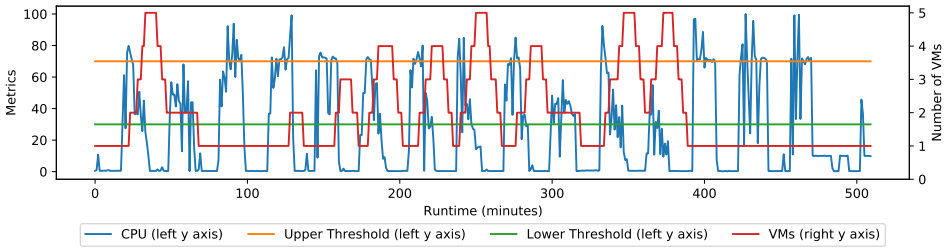

Figure 6 Number of VMs in MS2 microservice with fixed thresholds.

Table 4 Costs per test scenario.

Scenario	**Total number of VMs**	**Billing Type**	$ConsumedTimeUnit$	$Cost$ **($)**
Scenario 1	41	Hours	3000	55.8
		Minutes	1336	24.84
Scenario 2	22	Hours	1860	34.59
		Minutes	1012	18.82
Scenario 3	26	Hours	2220	41.29
		Minutes	1095	20.36
Scenario 4	33	Hours	2520	46.87
		Minutes	1065	19.8

that Hybrelastic stands out in relation to AWS, both in the type of hourly charge, as well as in the minute charge, making it a competitive model also financially, since it has a value 16% more economical financially per hour and 20.28% more affordable per minute.

5.3 Metrics Analysis

The use of different types of metrics to manage resource elasticity is not very common in the literature (Gervasio et al. 2020, Gao et al. 2020). As a result, Hybrelastic proposed an approach in which more than one metric is used, in order to apply them as a basis for the elasticity decision for different types of microservices. Figure 7 shows the variation of VMs in one of the microservices. In dashed black is shown two times when Hybrelastic triggers resource allocation, this allocation is done later, based on the growth of CPU and network metrics. In dashed red, two moments are ratified when there is a deallocation of resources,

which is also made based on the decrease in metrics. It is important to note that CPU and network metrics are directly proportional, as one grows, the other tends to grow as well.

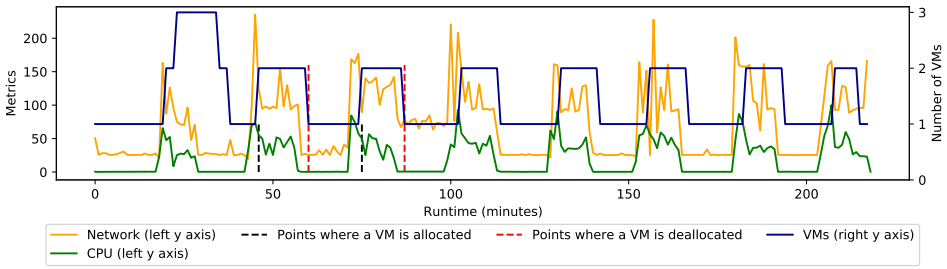

Figure 7 Number of VMs in MS1 microservice with dynamic thresholds.

Figures 8 and 9 show the behaviour of Hybrelastic based on the two metrics for microservices MS2 and MS3, respectively. The CPU utilisation metric is evident in MS2, as this microservice works with image processing, requiring a high processing of its virtual machines. On the other hand, the network metric was more used in MS3, as reading and writing database consume more network than CPU, as pointed out by the tests.

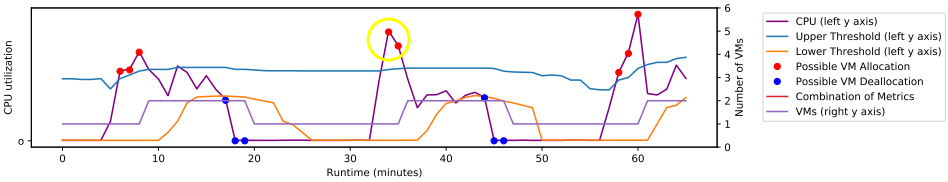

Figure 8 Elasticity actions triggered by Hybrelastic from CPU utilisation metric in MS2.

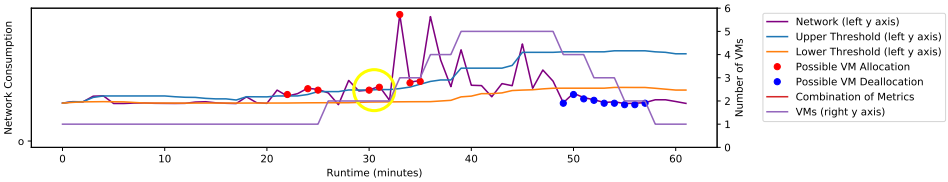

Figure 9 Elasticity actions triggered by Hybrelastic from network consumption metric in MS3.

In the graphs it is possible to analyse red and blue dots, which indicate a situation in which, possibly, an allocation of resources is necessary. As discussed in Section 4, it takes three elasticity decisions in a row for an action to be taken. Figures 8 and 9 show this behaviour in the coloured dots. However, the yellow circle indicates that the present elasticity decision occurred through a combination of metrics. Thus, it is clear that combining metrics for elasticity management is a valid approach, as each application may have a metric that best represents it, in addition, there are scenarios in which two metrics make better decisions than just one.

6 Conclusion

Applications based on microservice architecture are gaining more space in the market. This is due to the emerging evolution of cloud computing and all the benefits it brings to users. Elasticity management is typically provided natively by cloud providers; however, they are designed to act in monolithic applications and not in microservice-based applications. Therefore, this study presented Hybrelastic, an elasticity management model for applications based on microservices architecture.

The model proposed in this study, the Hybrelastic, combines reactive and proactive resource allocation policies through network consumption and CPU utilisation metrics. And it makes use of dynamic thresholds, which are created based on the application's behavioural history. The model was designed to work in applications based on microservices architecture. Hybrelastic was analysed based on its behaviour and performed 10.31% higher and 20.28% lower cost compared to other runs without its use. The research methodology proposed a model that combined two elasticities to make the best of each and, based on the results, this objective was completed. The results obtained by the model are encouraging for further research on this topic. They showed how applications behave for different types of metrics, indicating trends, and highlighted the benefits that the combination of two elasticity policies provide.

While the model has achieved its goals, it also has some limitations. Hybrelastic was tested with only one microservice, the ideal would be to expand the tests to more robust microservice systems, in which they addressed transactional microservices and that these were used in the market, in real applications. Another limitation highlighted was that the proactive algorithm, which works with workload predictions, is not always able to adequately predict the next load correctly, oscillating at the beginning of the analysis. As a result, there is an opportunity for future work. Therefore, the following suggestions present opportunities to improve the work: (i) use of other types of metrics for elasticity decisions, since other metrics can satisfy different types of microservices; (ii) proactive approach that uses more than one algorithm to predict future values, so that Hybrelastic can analyse the results of each algorithm and determine which is the most suitable at the moment; (iii) migration from Hybrelastic to other cloud providers for analytics outside of AWS; (iv) use of vertical elasticity and migration modalities, providing Hybrelastic with the choice of what type of modality to use and; (v) migrate Hybrelastic to a microservices architecture and use it to manage its elasticity, through recursive management.

Acknowledgement

The Authors would like to Thank to the Following Brazilian Agencies: CNPq, FAPERGS and CAPES.

References

ACM (2021). Accessed in August 2021.
URL: *https://www.acm.org/*

Amazon Web Services (2021). Accessed in August 2021.
URL: *https://aws.amazon.com/*

Arouk, O. & Nikaein, N. (2020), 'Kube5G : A Cloud-Native 5G Service Platform', pp. 3–8.

auto_arima (2021). Accessed in August 2021.
URL: *https://alkaline-ml.com/pmdarima/modules/generated/pmdarima.arima.auto_arima.html*

Auto Scaling Groups (2021). Accessed in August 2021.
URL: *https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html*

Bao, L., Wu, C., Bu, X., Ren, N. & Shen, M. (2019), 'Performance modeling and workflow scheduling of microservice-based applications in clouds', *IEEE Transactions on Parallel and Distributed Systems* **30**(9), 2114–2129.

Bauer, A., Herbst, N., Spinner, S., Ali-Eldin, A. & Kounev, S. (2019), 'Chameleon: A hybrid, proactive auto-scaling mechanism on a level-playing field', *IEEE Transactions on Parallel and Distributed Systems* **30**(4), 800–813.

Biswas, A., Majumdar, S., Nandy, B. & El-Haraki, A. (2017), 'A hybrid auto-scaling technique for clouds processing applications with service level agreements', *Journal of Cloud Computing* **6**(1), 29.
URL: *https://doi.org/10.1186/s13677-017-0100-5*

Boto3 (2021). Accessed in August 2021.
URL: *https://boto3.amazonaws.com/v1/documentation/api/latest/index.html*

Bouabdallah, R., Lajmi, S. & Ghedira, K. (2016), 'Use of reactive and proactive elasticity to adjust resources provisioning in the cloud provider', pp. 1155–1162.

Chaturvedi, A. K., Sengar, P. & Sharma, K. (2020), 'Measuring effective scheduling interval against vm load with rapid elasticity in cloud computing'.

Cloud Watch (2021). Accessed in August 2021.
URL: *https://aws.amazon.com/cloudwatch/*

Delfin, S. (2019), 'Fog Computing : A New Era of Cloud Computing', (Iccmc), 1106–1111.

Docker (2021). Accessed in August 2021.
URL: *https://www.docker.com/*

EC2 (2021). Accessed in August 2021.
URL: *https://aws.amazon.com/ec2/*

Elastic Load Balancing (2021). Accessed in August 2021.
URL: *https://aws.amazon.com/elasticloadbalancing/*

Fourati, M. H., Marzouk, S., Drira, K. & Jmaiel, M. (2019), 'Dockeranalyzer : Towards fine grained resource elasticity for microservices-based applications deployed with docker'.

Gao, J., Wang, H. & Shen, H. (2020), 'Machine learning based workload prediction in cloud computing'.

Gervasio, I., Castro, K. & Araujo, A. P. F. (2020), 'A hybrid automatic elasticity solution for the iaas layer based on dynamic thresholds and time series'.

GitHub (2021). Accessed in August 2021.
URL: *https://github.com/*

Google Scholar (2021). Accessed in August 2021.
URL: *https://scholar.google.com/*

IEEE (2021). Accessed in August 2021.
URL: *https://ieeexplore.ieee.org/Xplore/home.jsp*

Kumar, M., Dubey, K. & Pandey, R. (2021), 'Evolution of emerging computing paradigm cloud to fog: Applications, limitations and research challenges'.

Kwan, A., Wong, J., Jacobsen, H.-A. & Muthusamy, V. (2019), 'Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres', pp. 80–90.

Li, K. (2020), 'Quantitative Modeling and Analytical Calculation of Elasticity in Cloud Computing', *IEEE Transactions on Cloud Computing* **8**(4), 1135–1148.

Li, Z. & Yang, L. (2021), 'An electromagnetic situation calculation method based on edge computing and cloud computing', pp. 232–236.

Nardin, I. F. d. (2019), 'Elergy: Elasticidade em nuvem para gerenciar desempenho e energia em aplicações baseadas na arquitetura de microsserviços'. Accessed in June 2021.
URL: *http://www.repositorio.jesuita.org.br/handle/UNISINOS/9506*

Nashold, L. & Krishnan, R. (2020), 'Using LSTM and SARIMA Models to Forecast Cluster CPU Usage', *arXiv* .

Pérez, A., Moltó, G., Caballer, M. & Calatrava, A. (2018), 'Serverless computing for container-based architectures', *Future Generation Computer Systems* **83**(February), 50–59.

Qi, S., Kulkarni, S. G. & Ramakrishnan, K. K. (2021), 'Assessing container network interface plugins: Functionality, performance, and scalability', *IEEE Transactions on Network and Service Management* **18**(1), 656–671.

ResearchGate (2021). Accessed in August 2021.
URL: *https://www.researchgate.net/*

Righi, R. D. R. (2013), 'Elasticidade em cloud computing: conceito, estado da arte e novos desafios', *Revista Brasileira de Computação Aplicada* **5**(2), 2–17.

Rodrigues, V. F., da Rosa Righi, R., Rostirolla, G., Victória Barbosa, J. L., André da Costa, C., Alberti, A. M. & Chang, V. (2017), 'Towards enabling live thresholding as utility to manage elastic master-slave applications in the cloud', *Journal of Grid Computing* **15**(4), 535–556.
URL: *https://doi.org/10.1007/s10723-017-9405-3*

Rossi, F., Cardellini, V. & Presti, F. L. (2020), 'Self-adaptive threshold-based policy for microservices elasticity'.

Springer (2021). Accessed in August 2021.
URL: *https://www.springer.com/br*

Venkatachalam, A., Lathamanju, R., Shobana, M. & Sandanakaruppan, A. (2020), 'Improving elasticity in cloud with predictive algorithms', *Proceedings of the International Conference on Smart Technologies in Computing, Electrical and Electronics, ICSTCEE 2020* pp. 585–588.

Ye, T., Guangtao, X., Shiyou, Q. & Minglu, L. (2017), 'An auto-scaling framework for containerized elastic applications', pp. 422–430.

Yermal, L. & Balasubramanian, P. (2017), 'Application of auto arima model for forecasting returns on minute wise amalgamated data in nse'.