Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Eduardo Fernandez Afonso

An elasticity controller for applications orquestrated with Cloudify

October 2016

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Carlos Eduardo Fernandez Afonso

An elasticity controller for applications orquestrated with Cloudify

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
Miguel Matos
José Orlando Pereira

October 2016

ACKNOWLEDGEMENTS

I would like to thank my supervisor at University of Minho, Dr. Miguel Matos, my co-supervisors Professor Dr. José Pereira from University of Minho, and Professor Dr. Schahram Dustdar from Vienna University of Technology for all guidance provided throughout the dissertation. I am thankful for all their support and ideas.

I am very grateful to Dr. Georgiana Copil and Dr. Daniel Moldovan of Vienna University of Technology for their guidance and help during the project. I am very grateful for their constant availability for discussing the thesis direction and progress.

I would also like to thank Instituto de Engenharia de Sistemas e Computadores - Tecnologia e Ciência for providing me a work-space in their UMinho lab.

I would like to express my deepest appreciation to my colleague, Rui Miguel, for invaluable support in technical tasks, my friend Tiago Loureiro for our many technical discussions, and all my colleagues of the Edsger Dijkstra laboratory at University of Minho for their support.

I would also like to thank my parents, Francisco and Helena, as well as my sister, Mariana, for all their encouragement.

ABSTRACT

Cloud computing is a widely adopted model for the management of computing resources. Elasticity, an important characteristic of cloud computing, is the ability to allocate and release computing resources according to demand. An elasticity controller has the responsibility of performing decisions regarding when resources are provisioned or released and which types of resource are necessary. Developers have expectations of allocating the least amount of resources required and reaching optimal states using the least number of costly actions. Therefore, the task of controlling elasticity is challenging, especially in cases where the elastic application is composed of components with complex dependencies among themselves.

In this documentation, we introduce CEController, an elasticity controller for applications orchestrated with Cloudify. CEController introduces a novel elasticity strategy that takes into account dependencies between components and differences between metric dimensions. CEcontroller is evaluated in an environment created to test the controller, which includes the adaptation of a previously used application, and use of a load generation tool. Finally, we discuss the results obtained using CEController in a web application and discuss the results.

RESUMO

Cloud computing é um modelo de gestão de recursos computacionais amplamente utilizado. Elasticidade, uma importante propriedade de cloud computing, é a capacidade de alocar e libertar recursos computacionais segundo a procura. Um controlador elástico tem a responsabilidade de realizar decisões acerca de quando recursos deverão ser provisionados ou libertados assim como averiguar os tipos de recursos que são necessários. Programadores têm como objetivo a alocação da menor quantidade de recursos possível, assim como chegar a estados eficientes com o menor custo possvel em ações efetuadas. Desta maneira, realizar o controlo de elasticidade é uma tarefa complicada, especialmente em casos onde a aplicação elástica é composta por diversos componentes com dependências complexas entre estes.

Neste documento, introduzimos CEController, um controlador elástico para aplicações orquestradas com Cloudify. CEController introduz uma nova estratgia de controle elástico, que leva em conta dependências entre componentes assim como diferentes dimensões de métricas. De seguida, apresentamos o processo de criação de um ambiente de testes para o controllador. Incluindo a adaptação de uma aplicação previamente utilizada, e o uso de uma ferramenta para geração de carga. Por fim, apresentaremos os resultados obtidos ao utilizar CEController em uma aplicação web e discutiremos os resultados.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION

1.1 CONTEXTUALIZATION

Cloud Computing is a widely adopted resource management model. Currently there are several companies offering computing resources according to this model, such as Amazon Elastic Compute Cloud, Google Cloud Platform, and Microsoft Azure as well as countless clients that make use of these services.

Cloud computing itself is a model for enabling ubiquitous, convenient, on demand network access to a shared pool of configurable computing resources, such as networks, servers, storage, applications, and services. These resources can be quickly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of several service models, such as PaaS (Platform as a Service), IaaS (Infrastructure as a Service), SaaS (Software as a Service) and five essential characteristics [36]: on-demand self-service, broad network access, resource pooling, elasticity, and measured service. We will focus on the elasticity characteristic.

Elasticity captures an essential feature of cloud computing: when limited resources are offered for potentially unlimited use, providers must manage them elastically by scaling up, down, in, and out as needed [28]. Elasticity can be defined as the ability of an application to automatically adjust the infrastructure resources it uses to accommodate varied workloads and priorities, while maintaining availability and performance[39].

Elasticity does not encapsulate only cloud management, but any reconfiguration that affects application performance or cost[25]. Elastic applications are managed by elasticity controllers, through exposed elasticity capabilities. Some examples of this kind of capabilities are: tool configuration and re-configuration (e.g., changing load balancer balancing strategy), data placement management, container management, and process level management.

1.2 MOTIVATION

Cloud computing is widely used and researched. In 2011, Smith et al. [40] have referred to cloud computing as maturing, but still the most hyped topic in IT. When discussing the characteristics of elasticity, the authors refer to elasticity as ”one of the more hyped aspects of cloud computing. It is rising rapidly toward the Peak of Inflated Expectations. There is little understanding of what elasticity truly is in the market, and even less of how it can be achieved consistently and automatically” [40]. Currently, there are expectations of cloud computing solutions only using the required amount of resources, with the potential for almost limitless capacity. We believe that the research on elasticity, as well as elastic solutions that are robust and provide meaningful ways to reconfigure the controlled applications, are very important to fulfill these expectations.

However, the implementation of elasticity in cloud computing solutions is not an easy task. There are several issues that need to be accounted for such as, the evaluation of whether systems are truly behaving elastically, as in adapting to the requirements and changes in workloads, and implementation of elasticity capabilities beyond the addition and removal of computing resources. Another issue is how to properly evaluate the performance of systems in runtime, and whether the system is behaving in optimal ways, and which of the possible set of actions should be taken in order to correct this behavior. This sets of expectations and issues is an important topic with a large amount of potential of improvement and innovation.

Cloudify is a software orchestration tool developed by Gigaspaces[12]. Cloudify is installed in on top of an Infrastructure as a Service, and allows for the orchestration and maintenance of cloud applications. It is possible to create an elasticity controller for these cloud applications using only the features present in Cloudify. However, we believe that an elasticity controller using only Cloudify’s features has the some limitations.

Consequently, we address these limitations by creating a new, dedicated, elasticity controller for Cloudify. This new controller, CEController, implements some features from state of the art elasticity controllers and addresses the previously mentioned limitations.

1.3 METHODOLOGY

The dissertation was performed along with the Erasmus Programme, which happened during the second academic semester. The dissertation project is a collaboration between University of Minho’s Department of Informatics and Vienna University of Technology’s Faculty of Informatics.

Because Erasmus Programme was developed during the second semester and had a duration of four months, the first part of the project was developed at University of Minho, with remote communication to Vienna University of Technology. During the second semester the project was developed in Vienna, and the project's conclusion as well as the public defense will occur at University of Minho.

The methodology for this dissertation includes:

- Study and analysis of the state of the art and the available technological offerings.
- Hypothesis formulation based on the previous analysis.
- Validation of previous hypothesis in an experimental setting, which will contribute to further analysis.

1.4 OBJECTIVES

This work has two major objectives. The first one is to understand what elasticity means, what role elasticity plays in application design, from development phase up to operation phase. The second objective is the creation of an elasticity controller. The controller must be able to be used with applications orchestrated by Cloudify. The controller, CEController, implements features from commonly used elasticity controllers and integrates well with the features provided by Cloudify, such as the diamond metric collection system and the Scale workflow. If possible, novel features are to be implemented in order to present contributions to the current state of the art.

A test environment is created by adapting a currently existing application into Cloudify. In order to do so, blueprints and scripts for the installation and setup of the application are created. In cases in which there are no previously made diamond monitors for relevant components of the application, custom monitors are adapted or created.

Lastly, evaluation of the elasticity controller is be performed, and results are discussed.

Another objective of the dissertation is to publish the proposed solution and preliminary results at the INForum 2016 symposium.

2

STATE OF THE ART

2.1 ELASTICITY CONCEPTS

In this section, we present and analyse publications in relevant topics for this dissertation, as well as relevant commercial solutions. The state of the art is divided in two parts: scientific component, in which we focus on existing literature, and the technological component, where we focus on relevant existing commercial technologies.

2.1.1 *Definitions of elasticity*

In the literature there are many definitions of elasticity. While these definitions are generally similar, there are some differences between them. In this section, we will compare some of these definitions.

Cloud computing is a model for enabling on demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [36]. This computing model has been very successful and one of its key features is Elasticity.

The definition of elasticity varies slightly across authors. According to Mell et al. [36] elasticity is defined as the existence of capabilities that can be elastically provisioned and released, in some cases automatically, to scale in or out according to the demand. This definition refers to the acquisition and release of resources that would affect the performance of the system, and these operations are performed according to the current or expected workload demand. It is noteworthy to point out that elasticity has to deal with the problems of not only acquisition of these resources but also shifting load to them and putting them to use. In this approach, resource efficiency also becomes a growing concern because of the popular pay-as-you-go pricing model.

A common question is the difference between scalability and elasticity. While these concepts are similar there are keys differences between them. Galante et al. [29] defines that scalability is a static property that describes the system's ability to be enlarged to a size or reach a certain scale, and elasticity, on the other hand, is a dynamic property that allows the system to scale on demand in an operation system.

Therefore, we can conclude that scalability acts in relation to the growth of a system, while elasticity translates in performing the allocation and release of resources, as well as eventual reconfigurations in order to adapt to workloads. Moreover, scalability is a concept in which the time required to achieve such growth is not taken into consideration, where is elasticity, time is a essential aspect, as it depends on the speed of response to certain changes in the workload.

Other works define elasticity differently. Dustdar et al. [28] define elasticity as a measure of a function's responsiveness or sensitivity to changes in parameters in a relative way. This approach opts for a less restrict definition and proposes that elasticity should be composed of three dimensions: resource elasticity, cost elasticity and quality elasticity.

Resource elasticity describes, much alike the other elasticity definitions, the response on resource usage in relation to changes in the workload. However, in this context resource elasticity does not encompass the entirety of elasticity capabilities, but only one of three dimensions of which elasticity is composed of.

Cost elasticity describes a resource provision's responsiveness to changes in the cost. This dimension of elasticity is used on two types of occasions: the first occasion is when the service providers define the price models for cloud computing systems. The second occasion is when clients take advantage of the pay-as-you-go model to use instances without a long term commitment, or in face of dynamic pricing of resources based of the current demand. In this last scenario, users would acquire additional resources when the price of these resources dropped below the maximum amount that they are willing to pay. In these models users that are not time constrained can acquire processing power for potentially much lower prices while cloud providers can distribute resource usage uniformly.

The last dimension is quality elasticity. In this dimension quality refers to specific objectives about the application's behaviour that the developer might desire. Some of these quality objectives might be response time of operations in general applications such as web servers, completion time of MapReduce operations, or result quality in an approximation-based computation model. Quality elasticity measures how the application's quality responds when the resource usage is changed. This elasticity comes from applications that experience quality increase in response to higher resource utilization. This allows the choice of the amount of resources based on the desired quality objective.

The later elasticity definition has been used in subsequent works to, among other uses, separate metrics between these dimensions [37], [24], analysing the relationships between metrics [38], and definition of elasticity capabilities for big data systems [41].

There is another set of definitions that we use in this work, and would like to present. Copil et al. [23] define the following concepts that are sometimes used when referring to elasticity and elastic applications. These concepts are:

- Elasticity requirements - Elasticity requirements are defined as requirements that a stakeholder, such as the user, developer, or cloud provider, may have for elastic applications [23].
- Elasticity capabilities - Elasticity capabilities is defined as any change or reconfiguration that can be made to the application at run time in order to enforce the elasticity requirements [23].

2.1.2 *Characterization of elastic solutions*

Galante et al. [29] present a classification for elastic solutions, in which elasticity is defined based on four characteristics: scope, policy, purpose and method. For a diagram illustrating these classifications, refer to Figure 1.

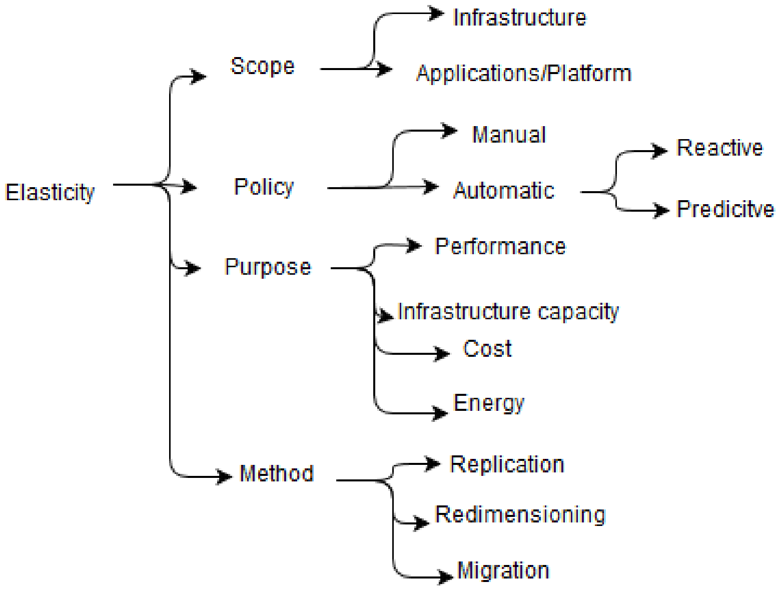

Figure 1: Classification of elastic solutions. Extracted from [29].

Scope describes where the elasticity actions are controlled. Possible scopes are located on the infrastructure management of systems, as commonly provided in infrastructures as a service clouds. Alternatively, the scope may be on the application or platform. In this approach the elastic control is performed in the application itself or the execution platform. In the latter, the application must be able to interact with an infrastructures as a service in order to obtain or release resources.

Elastic solutions with different scopes often behave very differently as the scope usually affects what metrics the elastic controller has access to, as well as the type of operations performed by the elastic controller. IaaS clouds usually possess an infrastructure level elasticity controller, which will receive monitored metrics, based on these metrics the controller has to decide whether to scale the resources or not. This type of elastic solutions are generally well suited for client-server applications that can take advantage of the additional virtual machines running server replicas [29].

Applications that are not aimed toward multiple clients or parallel processing will not be able to take advantage of these actions. Alternatively, application and platform scope elastic solutions will generally be favored for approaches that require a closer view on the application. However, the application must be developed in a supporting platform in case of platform scope, or designed with an elasticity controller in consideration in case of application scope.

The **policy** characteristic classifies elastic systems based on the interactions needed for the execution of elasticity actions. In the analyzed solutions two common policies were found: manual and automatic. In a manual policy, the user or cloud provider must monitor and perform all elasticity actions, while in automatic policies these actions are performed by the system or application. The authors sub-classify automatic policies as either reactive or predictive, based on whether elasticity actions are performed according to a set of predefined conditions or whether there is an use of heuristics and analytical behaviour to decide when and how to scale resources preemptively.

Cases of mixed policies can also be found in the literature, these policies are combinations of reactive and proactive elements. These methods are generally aimed at handling prediction inaccuracies in the proactive policies and avoid instabilities in the provisioned capacity due to oscillations in the workload that are present in reactive approaches [29].

Another characteristic defined was **purpose**. By the definitions presented earlier, the general purpose of elasticity is to be able to adapt to changes in workload and demand. However there might be specific requirements that the elastic solution focuses on. Galante et al. [29] identify that most elastic solutions have the purpose of avoiding inadequate provision of resources and avoidance of degradation of the system's performance, while others had different purposes, such as increasing local resources capacity, cost reduction and energy savings.

Finally, the last characteristic refers to the **methods** employed in the implementation of elasticity solutions. These were identified as replication methods, resizing methods and virtual machine migration.

Replication methods consist in the addition or removal of resources. These resources can be virtual machine instances, containers, or even application modules in the case of software as a service. Replication methods are currently the most popular elastic policies. Replication methods sometimes require the addition of load balancing techniques, and currently many popular cloud providers such as Amazon EC2 offer load balancing mechanisms features [29].

Redimentioning methods consist of the addition/removal of processing, memory, or storage resources from previously created virtual machines.

Virtual machine migration methods transfer virtual machines running in a physical server to another, more adequately fit to the demand. Galante et al. [29] note that this type of method is generally used in environments where resizing is not possible.

For an in depth explanation of these characteristics and the classifications of existing or proposed elastic solutions refer to [29].

2.1.3 *Workloads and benchmarks*

Most works in the area of elasticity use a tool or program that generates workloads in order to test or evaluate the application as well as to obtain comparisons between different approaches. An adequate benchmark tool should simulate the usual workload that faces a cloud computing application, including possible alterations in the intensity of request and the proportion of the request types[26].

Coutinho et al. [26] identify the most used benchmarks and workloads in previous elasticity works. The two most used were TPC-W[20], and SPECweb2005[18], however, many others were present such as TPC-C[19] and even workload generating tools for applications such as Httperf[13] and JMeter[1].

Li et al. [33] categorizes these benchmarks in three different categories: application benchmarks, synthetic benchmarks, and microbenchmarks.

There are some open issues identified in the area of benchmarks suited for cloud applications. Traditional benchmark tools for cloud systems are are not well suited to fully evaluate these systems. Benchmarks adequate for elastic environments should be able to simulate the different workloads including the change of intensity of requests, such as periodic increases and decreased of demand and rapid load surges experienced by real life applications [26].

2.1.4 *General metrics*

The definition of metrics is an important decision in the design of elastic systems. These metrics reflect how the system is performing. Some form of general metrics is used in all of the elastic cloud applications analysed. Some metrics are general use metrics, commonly used in general application benchmarks, while some other metrics are specific to evaluating elasticity [26]. In this section we will focus on the former and the next section will be focused on the latter.

Jain et al. [31] present the following categories for metrics: response time, utilization, throughput, reliability, availability, and acquisition. The most commonly used metrics of the different categories presented by the authors were:

- Response time - Latency, allocation, access, idleness.
- Utilization - Resource utilization, and percentage of CPU utilization.
- Throughput - Requisitions per second, and Megabytes per second.
- Reliability - Number of violations.
- Availability - Downtime, and uptime.
- Acquisition - Total cost, and cost to performance ratio. [31].

In a survey by Coutinho [26], the authors reported that the most common metric used in elastic application was percentage of CPU utilization. Moreover the authors have also reported the use of statistical techniques for interpretation and exposition of the previously acquired metrics, with standard deviation, variance and error the most commonly used methods.

Another important fact noted by Coutinho et al. is that cost and pricing aspects often being associated with public clouds and there are metrics that expose the total cost or infrastructure cost were identified in Coutinho's survey. However, no standardized definition of how to price resources the resources used in cloud computing scenarios were found by the authors. As these costs may vary across providers, there is no way to universally translate the resource usage of an application to its cost in public clouds. In some public clouds the price of different resources may vary, leading to elastic solutions not being optimal across clouds [26].

2.1.5 *Evaluating elasticity and elasticity metrics*

In addition to evaluating the application's performance, methods of evaluating elasticity are also be required in order to evaluate whether the application is behaving elastically. An application is behaving elastically if the application adapts itself to changes in the workload according to it's requirements. In this section, we will describe one proposed solution for evaluating elasticity and review it.

Herbst et al. [30] approach the problem of benchmarking elasticity in applications by evaluating elasticity at every 'dimension'. For the authors, dimensions are defined by the type of resource that can be scaled out or in such as CPU cores or amount of RAM. At every dimension, the objective is to determine the speed of adaptation and precision. Speed of adaptation scaling up and down is defined as the amount of time it takes to achieve an optimal state from an under provisioned or over provisioned state respectively. Precision is defined as the absolute deviation between the allocated resources and the actual resource demand.

In order to measure these two concepts, for each type of scalable resource, the actual elasticity and optimal elasticity are compared. In this publication, optimal elasticity is defined as the hypothetical case where the system is scalable with respect to all considered elasticity dimensions without any upper bounds on the amount of resources that can be provisioned, and in which resources are provisioned and deprovisioned immediately as they are needed matching exactly the actual demand at any point in time. In this definition however, optimal elasticity can still limited by the quantification of resource units, for example, as we cannot add half virtual machine in real scenarios, optimal elasticity also cannot bypass the discrete nature of resource units.

In order to capture the optimal amount of provisioned resources, function 1 returns the minimal amount of resources, r, to guarantee the system's minimum performance requirement facing workload intensity, w. The workload can be defined in many ways such as number of requests per unit of time or number of workload units present in the system. It is noteworthy that there should be two distinct functions for both directions of scaling, as they cannot be assumed to be identical when scaling up and down [30].

$$m(w) = r \tag{1}$$

In order to obtain function 1 values the authors propose the usage of experimental methods. The authors' suggested procedure consists of benchmarking the application at increasingly higher workload levels step by step. The resource consumption should then be documented in order to create a table with matching values of workload and optimal resources. This process is then repeated for a increasingly lower workload. During this process, it is necessary to allow as much time as the application requires to scale up and down to its optimal state.

After the matching tables are created, the elastic application should be evaluated with a benchmark tool capable of varying its workload between the various workload levels present in the previously constructed matching table. In this evaluation, time is a concern and therefore under or over provisioning of resources will allow for the calculation of the core elasticity aspects referenced earlier: precision and speed. The authors propose formulas to calculate these aspects using the optimal elasticity table and the application benchmark values. Furthermore, it is recommended that for the comparison of different elastic systems, the resource demand curves for the systems should be as identical as possible. For example, if two systems can handle a specific workload with very different resource curves then distinct workloads should be applied to each system in order to correct large differences between these resource curves. In order to obtain two distinct workloads the author suggests increasing/decreasing the original workload for one of the systems [30].

While this work takes steps towards benchmarking elasticity, we argue that there are unresolved issues. The first issue arises in the unidimensional approach to the evaluation of elasticity. Elasticity capabilities in a system are often used simultaneously, and different systems may have different elasticity capabilities. One example of this is load balancing offered by cloud providers such as Amazon, in which this resource might be inherently different from other load balancing solutions. This issue may result the comparison being incomplete. A possible solution would be to compare multiple dimensions simultaneously. For example, by the adoption of a general resource allocation metric, which takes into account all allocated resources in the system, and their respective cost and produces a value equivalent to total resource utilization. Still, as there is no generalized resource pricing system among cloud providers, it might be difficult to compare the costs of the different resources provided.

Considering the multi-dimension elasticity definition [28], we argue that this evaluation method is only capable of measuring resource elasticity. This method considers the elastic system's desired performance level to be the only requirements, not considering cost elasticity and quality elasticity. In some cases, changes in cost may influence the elastic system's desired performance level. For example, there might be a requirement of relaxing the desired performance level when a certain system cost threshold is passed. Taking these into consideration, it might make sense not only to vary the workload, but also to vary resource costs and the desired accuracy or quality of results produced.

2.1.6 *Expressing elasticity requirements*

In cloud computing services, developers and public cloud users always have requirements for the service provided. This section focuses on ways of expressing quality of service requirements as elasticity requirements.

Elasticity requirements may be represented in many ways, such as a service level agreement, or simple metric thresholds. Google Compute Engine, offers the possibility to define utilization level targets. These utilization level targets are the level at which the user decides to keep leased virtual machine instances' resources. For example, the user can define a target utilization level based on CPU utilization level at 75% percent and the autoscaler will attempt to maintain the CPU utilization of the specified group of instances at or as close as possible to 75%. Possible metrics to use in utilization targets are CPU utilization, HTTP load balancing serving capacity, and exported metrics from their Cloud Monitoring service. [3]

Copil et al [24] present SYBL, a language for controlling elasticity in cloud applications. SYBL proposes programming directives for definition of metrics to monitor, constrains to enforce and definition of action strategies. The authors follow the definition of elasticity composed of three dimensions referred earlier, in which elasticity is divided by resource elasticity, cost elasticity and quality elasticity. The authors consider that elastic application's users should be able to specify constraints and relations between the aforementioned dimensions.

According to the authors, SYBL adopts a multi-level specification levels, allowing the user to control elasticity in the following levels: application level, component level and programming level. The application level provides a high level description of the user's preferences and constraints to the application deployed in the cloud infrastructure. In the application level, data collected from the different components must be aggregated in order to be used. Component level refers to the components in the application, and thus gives a higher level of detail and lower level of abstraction. Finally, programming level elasticity requirements define requirements at code level, and thus affect only a part of the application's code, requiring the application to be designed with elasticity in mind [24].

SYBL's language constructs are composed of predefined functions, monitoring directives, constraint directives and strategy directives. Predefined functions give information on the current environment and elasticity specifications, the environment varies according to the present level and therefore are required to be called at the correct levels [24]. For examples of predefined functions and their description see Table 1. From this table we can obtain a basic idea of the role of predefined functions in the language. In the table we can see one function that gives the information on the environment, GetEnv and three others that give details about specifications, Violated, Enabled and Priority.

Function	Description
GetEnv	Obtains the current cloud infrastructure environment, depends of the level
Violated	Checks whether a constraint sent as a parameter is violated
Enabled	Returns whether an elasticity specification is enabled or not
Priority	Returns the priority of an elasticity specification

Table 1: Exaple of SYBL's predefined functions. Extracted from the original publication[24].

Monitoring directives assign a new variable information to be monitored. This information can be either pre-existing cloud information or formulas combining several types of pre-existing cloud information. Constraint directives define constrains or limits to the application. Finally, strategy directives define a trigger specifying that when a condition becomes true, a recipe containing one or multiple actions is to be executed [24].

Listing 2.1, exemplifies a possible elasticity requirements document created by a cloud services user using SYBL. In this listing it is possible to see the separation of the application, component and programming level, the use of monitoring directives (Mon1), the definition of several constraints at different levels (Cons1, Cons2, Cons3, Cons4, Cons5, and Cons6), and one definition of priority for two constraints previously defined and the definition of an application level strategy (Str1).

```
#SYBL.ApplicationLevel
Mon1: MONITORING rt = Quality.responseTime
Cons1: CONSTRAINT rt < 2 ms. when nbOfUsers < 1000
Cons2: CONSTRAINT rt < 4 ms. when nbOfUsers < 10000
Cons3: CONSTRAINT totalCost < 800 Euro
Str1: STRATEGY CASE Violated(Cons1) OR Violated(Cons2): ScaleOut
 Priority(Cons1)=3, Priority(Cons2)=5
#SYBL.ComponentLevel
ComponentID = Component3; ComponentName= DataEngine
Cons4: CONSTRAINT totalCost < 600 Euro
#SYBL.ComponentLevel
ComponentID = Component2 ComponentName= Computing
 Engine
Cons5: CONSTRAINT cpuUsage < 80%
#SYBL.ProgrammingLevel
Cons6: CONSTRAINT dataAccuracy>90% AND cost<400
```

Listing 2.1: SYBL elasticity requirements. Extracted from [24].

We believe that the means to express elasticity requirements is an important concern for any elastic application or cloud service. We also think that a language must achieve a balance between versatility and expressiveness, and when possible hide the complexity of the underlying algorithms used by the controller to enforce imposed constraints. Of course, only elasticity requirements that are enforceable should be allowed. For example, SYBL allows the definition of possibly conflicting requirements present in different tiers. The corresponding elastic controller should be able to evaluate appropriate elastic actions to take when defined requirements might conflict, otherwise the constraint might not be followed, resulting in incorrect behavior.

2.1.7 *Solutions for control of elasticity and enforcement of elasticity capabilities*

In this section we discuss works that deal with controlling elasticity in applications and the implementation and enforcement of elastic capabilities.

Automated control of elastic storage.

Lim et. al. [34] addresses elastic control for multi-tier applications services that allocate and release resources in discrete units. Given that the allocated resources are discrete, it must take into account that it is not possible to reach the exact resource usage level defined by the service level agreement.

Lim et al. [34] proposes an Horizontal Scale controller with a new approach on the control policy. The desired number of active server instances is obtained by adding previous number of server instances to the multiplication of the integral gain parameter and difference between the desired target for the metric in use (provided by the SLA) and the current metric value. Since, this approach assumes that the actuator value is continuous, the desired metric value was changed into two values representing the high and low desired sensor measurements. This allows for the creation of a range in which the sensor measurements can vary without requiring the application to scale.

In the proposed equation for calculating the amount of resources required, depicted in equation 2, u_k represents the current actuator level, K_i the integral gain parameter and y_h, y_l, y_{ref} represent the high, low and reference desired sensor value, respectively.

$$u_{k+1} = \begin{cases} u_k + K_i \times (y_h - y_k) & if\ y_h < y_k \\ u_k + K_i \times (y_l - y_k) & if\ y_l > y_k \\ u_k & otherwise \end{cases} \tag{2}$$

With the alterations proposed by the authors to the integral method, the horizontal scale controller will only react if the current metric value is lower than the low reference sensor value or higher than the high reference sensor value. This approach also requires the values of y_h and y_k to be set dynamically, according to the number of existing nodes in order to avoid instability and resource inefficiency [34].

This work contains some assumptions for their data source. These assumptions are the following:

- The distribution of data across nodes in a way that balances load effectively for reasonable access patterns.
- The capacity of performing node join and leave operations, in response to these operations a data redistribution must be performed.
- The data must be replicated for availability, such that the leave operation must not cause interruption of service even if a departing server is released before the leave operation is complete.
- The storage tier must provide scale roughly linearly in terms of storage and I/O capabilities with the size of the active server set.

Some challenges encountered by the authors such as: the data rebalancing was not instantaneous, and thus the performance does improve immediately and will most likely decrease for a short period of time due to data rebalancing. Data rebalancing operations may be allocated a certain amount of maximum bandwidth. High maximum bandwidth values affect the performance decrease during the data rebalance stage but have the advantage of providing a faster data rebalancing. Regardless of bandwidth allocated, there will always be delays before performance improvements can be observed. This delay must be taken into account by the controller.

In this work, the metric that has been chosen to be monitored by the sensor is CPU utilization. This metric satisfies several desired properties, but other metrics could be used in addition, or instead, of CPU utilization [34].

An interesting note about this publication is that it assumes that data rebalance will be reasonable and focuses on the creation of methods to properly identify when an addition or removal of resources are required. In the next section, we will showcase another solution to provide elasticity in a NoSQL cluster, but applies other methods to prevent the unbalance of data access patterns.

MET

Cruz et. al have proposed MET [27]. MET is a framework that aims at automatically managing, configuring, and reconfiguring a NoSQL cluster while providing elasticity. The MET framework implements elasticity capabilities such as the addition and removal of specifically configured nodes, as well as rebalancing and partitioning nodes according to their data access patterns [27].

MeT's is a reactive elastic controller that periodically extracts metrics from the components. The actions taken are calculated from these metrics, statistical methods, and thresholds that may be altered by the user.

MeT's elastic controller is composed of three main components: monitor, decision maker and actuator. The monitor gathers information and maintains information about the application. Several metrics are monitored at two different levels: System metrics and specific NoSQL database metrics. The system metrics collected are CPU utilization, I/O wait and memory usage through Ganglia. The NoSQL database used in MeT is HBase. In HBase, there are RegionServers that serve data and DataNodes that keep data. The NoSQL metrics collected consist of total number of read, write and scan operations, number of requests per second and data locality of the blocks co-located in the DataNode. These metrics should be sufficient to determine the data's access pattern of the workload. The previously referred NoSQL metrics are obtained with each Region Server's Java Management Extensions. Finally, exponential smoothing is used in order to account for short term load spikes, and only the metrics collected after each actuator's actions are considered in order to prevent the collected metrics to be influenced by temporary changes caused by the enforcement of elasticity capabilities [27].

The second component is the decision maker, which is responsible for receiving metrics from the monitor and deciding what actions must be performed in case the state of the cluster is considered sub-optimal. This process consists of several stages, namely the determination of the current state of the cluster, decision for adding and removing nodes, distribution of data partitions into groups, assignment of data to be transferred to other clusters nodes, and output the computation to the actuator. The decision of whether the cluster is in an optimal state is made comparing the metrics obtained with a threshold. The authors suggest the use of a threshold in which the amount of nodes in a sub optimal state must be maintained to less than half of the total amount.

If the first stage considers the cluster state sub-optimal, then the second stage will decide node addition or removal. This stage defines the amount of nodes that are to be added or removed from the cluster, following a quadratic strategy for additions and linear strategy for removal.

The third stage is the distribution decision stage, this stage has the objective of clustering data partitions with similar data access. This process is composed of three phases: classification, grouping and assignment. In the classification phase, data partitions are divided into groups according to data access patters, these patterns are acquired with the usage of read, write and scan metrics of each data node, and each data node is assigned to one of four groups: Read, write, read/write, scan. In the grouping phase the number of nodes that are going to be attributed to each group is computed. Finally the assignment phase attempts to balance the load and number of data partitions in each node with a greedy algorithm, finishing the distribution decision stage.

This information is then passed to the fourth and final stage of decision making, the output computation. This component is in charge of analysing the current data distribution and matching it with the new suggested distribution as best as it can. This process has the objective of minimizing node reconfiguration and data partitioning moves. Finally all decisions are delivered to the actuator, that enforces the cluster changes. A flow chart of this process is present in Figure 2.

We argue that rebalancing and partitioning nodes according to data access patters is itself an elasticity capability. Following the definition of elasticity as the runtime adaptation of an application as a result of changes [28], we can argue that the rebalancing and partitioning is an adaptation of the application in response to changes of the workload's access patterns. The publication's authors report considerable improvements in throughput of a properly balanced and partitioned HBase cluster in relation to the regular behaviour of HBase with an homogeneous configuration of nodes using the standard randomized data placement component that evenly distributes data across all cluster nodes. If the data access patterns change, then unless rebalancing occurs, these gains will be lost.

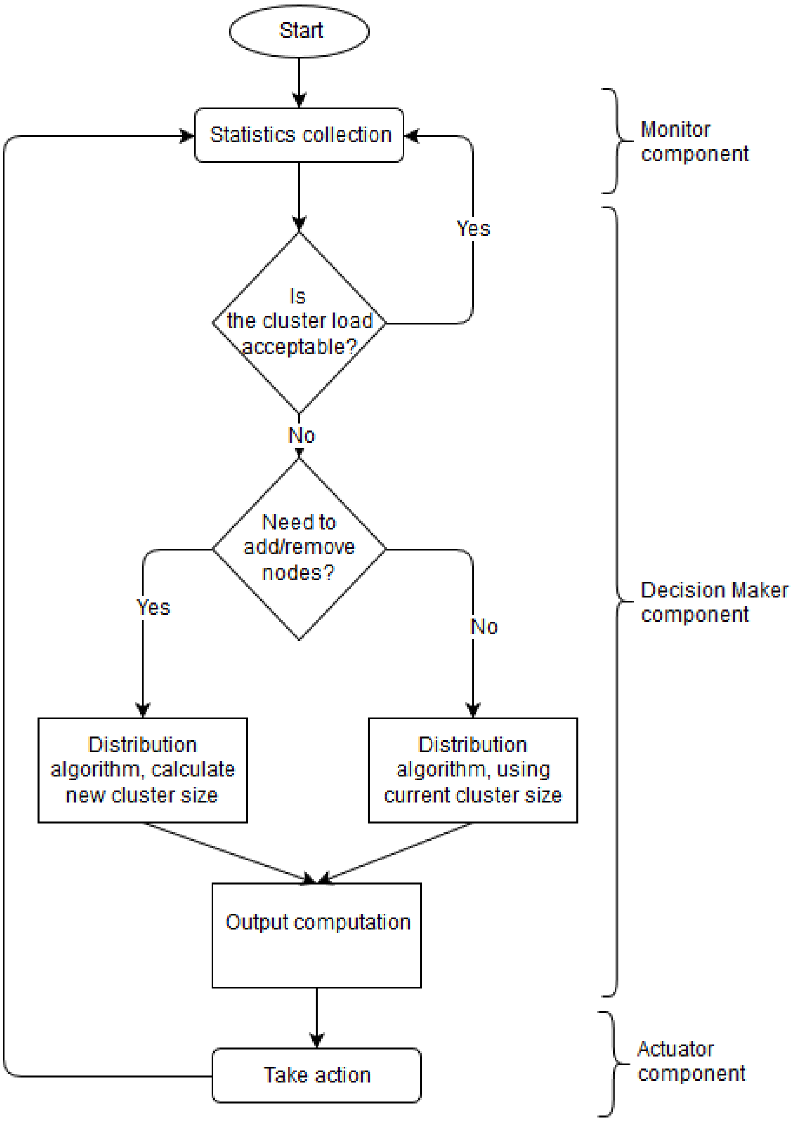
Start
Statistics collection
Is the cluster load acceptable?
Yes
No
Need to add/remove nodes?
Yes
No
Distribution algorithm, calculate new cluster size
Distribution algorithm, using current cluster size
Output computation
Take action
Monitor component
Decision Maker component
Actuator component

Figure 2: MeT's flow chart. Adapted from [27].

DBalancer

In the literature there are other publications that approach the topic of runtime configurations of NoSQL clusters. Konstantinou et al. [32] present DBalancer, a distributed module for load balancing for NoSQL datastores.

The authors consider that unanticipated load spikes or unbalanced data access patterns affect negatively the performance in data service applications and cites it as a typical problem of NoSQL data-stores. It is argued by the authors that the current data balancing mechanisms in NoSQL data stores, performed by consistent hashing, cause two main problems: the first is that hashing process destroy locality of the data, so that if two pieces of data are close and would benefit from a data allocation algorithm that preserves order, this possible performance gain is no longer possible. The second problem is that when a small portion of the data is very popular and causes the disproportionally high load on the node where it is located [22].

The authors then present DBalancer, a generic distributed module that performs load balancing on distributed NoSQL datastores. Its proposed novel features are: datastore and algorithm abstraction, in which DBalancer is independent of the underlying NoSQL datastore and allows new load balancing algorithms to be easily defined, distributed operation, in which there is no central coordinator and works on a per-node basis, and a small impact on the running node. Additionally, DBalancer allows users to launch NoSQL clusters through a GUI and apply different balancing algorithms and workloads. For more detailed information, including architecture, currently implemented balancing algorithms and a demonstration description please refer to the original publication [32].

While the authors do not define DBalancer as providing elasticity, we consider that it solves the problem of skewed data access patterns at run time to be a change in the datastore according to changes in the context it resides and therefore fits in the definition of elasticity presented by Dustdar et al. [28]. We argue that since it achieves increased application performance it could be a useful elasticity capability present in a controller.

Principles of software-defined elastic systems for big data analytics

In this publication [41], the authors propose principles of elasticity and argue that elastic principles should be investigated as fundamental to the development of data analytics platforms in order to tackle big data issues. Some of the issues mentioned are different requirements in terms of cost and quality that customers may have, and the need of using computing resources and even data in different manners for the same types of analytics, obtaining different outputs [41].

The authors state that popular forms of big data analytics deal with large amounts of data from different sources aggregated for analysis in places such as cloud data centers. In this scenario, the number of data analytics is limited because there is prior knowledge of the data models at development. However the addition of more data types at runtime implies the need of different data analytics and data models. The authors suggest that in this case, there is need for provisioning of not only resources, but of new analytics and computational models at runtime.

In a third scenario, the authors consider that different quality and cost requirements of the customers will have an influence on several parts of the analytics, such as: which types of data will be used by the analytics, which types of analytics are to be used, and the amounts of computing units to be used.

The authors also propose a conceptualization for software-defined elastic systems for data analytics and presents a case study in smart city management, urban mobility and energy systems.

This publication approaches the topic of big data applications, identifies cases in which these applications could benefit from implementing principles of elasticity, as well as methods to implement these principles. While our work does not encompass such types of applications, we find that the provided examples of how elasticity could be implemented at software level in order to solve specific issues is relevant for this dissertation.

2.2 CLOUD TECHNOLOGIES

In this section, we will present commercial and open source products that are relevant for this work. One of the objectives of the dissertation is to implement and evaluate elasticity and elastic capabilities. Because of this, we have analyzed current products to use or extend in the course of this project. In this section we will present two of these products, and their features. We will present Cloudify, an orchestration software designed to launch cloud applications, which we used in the project.

2.2.1 *Cloudify*

Cloudify is a software orchestration tool developed by Gigaspaces[12]. Cloudify is installed on top of an Infrastructure as a Service, and allows for the orchestration and maintenance of cloud applications. Possible platforms in which Cloudify can be installed include, among others, Openstack, Amazon WebServices, or Docker [21].

When using Cloudify, the developer creates a blueprint file containing information on the application's topology, defining the virtual machines, networks, connections, as well as script files. Given the blueprint and script files Cloudify can then create deployments, requesting needed computing resources from the Infrastructure as a Service.

Cloudify allows for the definition of workflows. Workflows define tasks that are executed on one or more nodes. In workflows, the executed task may be arbitary code such a Python script [11]. Cloudify contains some built-in workflows such as install, auto-heal and scale. The scale workflow in particular, scales in or out a determined node, passed as parameter, applying the install or uninstall actions in addition to performing relationship operations such as establishing connections. Workflows can be executed using Cloudify's command line interface, REST API or with Cloudify policies.

Cloudify's architecture is composed of the following main parts: The Cloudify Manager, Agents, and Web UI (for the premium version). The manager is composed of Cloudify's code and a set of other open-source applications, such as Elasticsearch, a search server, RabbitMQ a message broker software, and Riemman, a stream processor language.

Cloudify policies enable the creation of triggers based on events. These events are originated in the nodes and go through RabbitMQ[16], a message broker contained in the Cloudify manager. These events can then be processed as a stream using a Riemann[17] event processor defined in the policy. The Riemann program has the responsibility of triggering the policy's actions based on the stream of events received. Finally, policies will trigger the execution of a task, generally in the form of a workflow.

[14]. The Agents are entities designed to execute tasks on hosts. They are installed by the manager and are able to listen to task queues and execute tasks when required.

Finally, there is a WebUI for the commercial version, the WebUI interacts with the manager and presents an alternative to using the Cloudify command line interface. Figure 3 presents a diagram of Cloudify's components, with focus on the manager and agent virtual machines illustrating the parts and flows mentioned earlier.

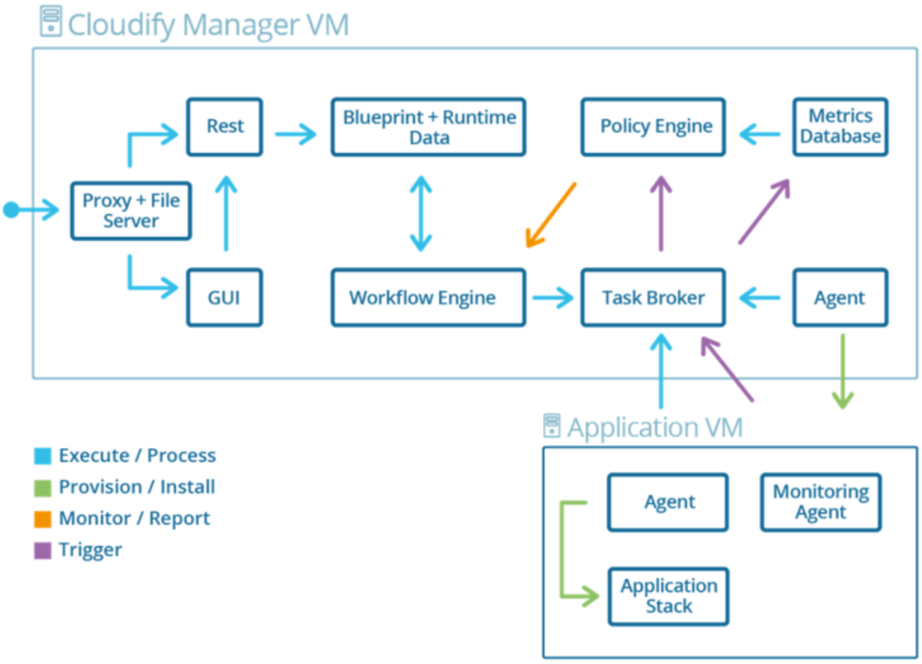

Figure 3: Cloudify's architecture. Taken from [14].

Using the previously mentioned features, it is possible to create an elasticity controller for applications orchestrated by Cloudify. An elasticity controller created with these features should generally have the following general structure:

- Several policies are to be defined. Each policy triggering one elastic capability, for example, one scale-in and scale-out workflow trigger for each node type, as well as one policy and trigger for each desired reconfiguration workflow.

- One Riemann event processor per policy. The processor containing information about the metrics to use, how to aggregate them and which conditions to activate the trigger.

3

ELASTICITY CONTROLLER FOR CLOUDIFY

This section presents the major contribution of this work. It describes the design and implementation of an elasticity controller for Cloudify. The controller will to provide elasticity to applications orchestrated by Cloudify.

In the next section we point out some limitations in obtaining elasticity using Cloudify, as well as the requirements and solutions applied in the controller's design. Later, there is a section dedicated to the controller architecture, going into detail about the controller's components. Finally, the last section describes the controller configuration files and the control settings.

3.1 CLOUDIFY'S LIMITATIONS AND OBJECTIVES

As shown in the previous section, it is possible to create an elasticity controller using a set of features currently available in Cloudify. However, we believe that this approach has the following limitations:

- There is no priority system for policies in Cloudify. When using policies, whichever condition is violated first will cause a trigger in the respective policy. This trigger will in turn start a workflow. Meanwhile, other trigger activations may happen, although new workflows will not start while the original trigger's workflow is running. Therefore, the existing policy mechanism does not allow us to prioritize some elasticity actions over others, and places constraints on the creation of elasticity strategies.
- The policies are independent. Each policy will independently collect metrics from the obtained events and process the metrics. One policy trigger activation has no impact on the collection of events from any other policy. This limitation prevents the developer from establishing periods in which to ignore incoming metrics as a result of ongoing enforcement of elasticity capabilities. For example, in the case of the scale out action being applied to a distributed database, we expect that the database management system will require to transfer data from overloaded nodes to the newly created node. During this period, the system's metrics will likely deteriorate instead

of improving, thus causing undesirable additional actions to be taken, hence ignoring these metrics might be the better option.

Consequently, we propose a new dedicated elasticity controller for Cloudify. This new controller, CEController, receives elasticity requirements and server configurations from the developer, metrics and application information from Cloudify, and requests actions from the Cloudify manager. CEController implements various features present in state of the art elasticity controllers and addresses the previously mentioned limitations. Furthermore, CEController's elasticity requirement classes can be extended and provide a basis for our new strategy, called DependenciesStrategy.

Some initial decisions had to be taken concerning the controller during the design phase. One important question was whether CEController should be developed as a part of Cloudify, such as an extension of the source or plugin, or as a separate program. We have decided to develop CEController as a separate program, as this way we can achieve a faster development cycle and keep a large part of CEController's source code independent from the software orchestrator. This way are be able to, in the future, adapt CEController and integrate it with Cloudify or other software orchestrator solutions. Another relevant decision was the choice of language for the project. Due to Cloudify being written in Python, we have opted to develop CEController in the Python language.

In the next sections, we identify the communication channels between CEController and Cloudify. We also explain in detail how CEController handles elasticity requirements and strategies. Later, we explain in detail what information has to be provided to our controller by the user. Finally. we focus on the implementation of our new strategy, called DependenciesStrategy.

3.2 COMMUNICATION CHANNELS

In order to function properly, CEController must be able to connect to the Cloudify Manager to accomplish the following tasks:

- Obtain elasticity metrics.
- Request actions from the manager.
- Receive information about the manager and state of the orchestrated application.

In this section, we will explain these tasks and discuss their implementation.

3.2.1 *Obtaining elasticity metrics.*

Applications orchestrated with Cloudify may use monitoring agents in order to publish metrics into Cloudify[8]. These monitoring agents publish metrics data to a RabbitMQ broker contained in the Cloudify Manager. With CEController, we connect directly to the RabbitMQ broker and subscribe to the desired metrics channel.

```
1  class MetricReader():
2
3    def __init__(self, ...):
4        ...#receive initialization parameters
5
6    def start_collecting(self):
7      credential = pika.PlainCredentials(username=self.userName,
8                                         password=self.password)
9      connection = pika.BlockingConnection(pika.ConnectionParameters(
10                                          host=self.host,  credentials=
     credential))
11     channel = connection.channel()
12
13     channel.exchange_declare(exchange=self.exchange,
14                              type='topic',
15                              durable=False,
16                              auto_delete=True,
17                              internal=False)
18     channel.queue_declare(queue=self.queue_name, auto_delete=True)
19
20     channel.basic_consume(highLevelCallback(self.messageBroker),
21                           queue=self.queue_name,
22                           no_ack=True)
23
24     channel.start_consuming()
```

Listing 3.1: Establishing connection to RabbitMQ.

The process of connecting to the RabbitMQ message broker present in the Cloudify manager is shown in Listing 3.1. The necessary information to subscribe to the message broker, such as user name, password, host IP, is provided during the object initialization.

Firstly the connection is created using the provided credentials (Lines 7-9). Later we declare an exchange in order to read messages from our desired queue (Lines 13-17), we declare the queue we wish to subscribe to, by using it's name (Line 18). Finally, we request that our callback function receives the messages published into that queue (Lines 20-22). Our callback function will parse incoming metrics and send them to another broker, which separates metrics that need to be monitored from those that do not. We will go into detail about this second broker in a later section.

3.2.2 *Requesting actions from Cloudify manager.*

In order to enforce elasticity requirements, CEController requests actions from Cloudify Manager. These tasks will usually involve requesting the start of a workflow, such as the built-in *scale* workflow or a developer created custom worklfow. In order to accomplish this, CEController has a *REST_Client* class, which is responsible for communicating with the Cloudify manager's REST API[9].

```
rest_client.client.executions.start(rest_client.deployment, 'scale', {'node_id':
    self.node_id, 'delta': self.delta})
```

Listing 3.2: Workflow request using REST api .

Listing 3.2 shows a request from CEController to the Cloudify Manager. In this case the action requested is the *scale* workflow, which increases or decreases the number of instances. The first parameter of the *start* method defines the deployment. The second parameter defines the name of the desired workflow. Finally, the third parameter defines the execution parameters. In the *scale* workflow, we specify as parameters the name of the node that requires scaling, and the amount of instances to be added of subtracted.

3.2.3 *Receiving manager and deployment state.*

The controller has to gather some information on the manager state, such as if there are any pending or started workflows, as well as state of the orchestrated application, such as the number of virtual machines instances of each node type. This information is also obtained via Cloudify's REST API.

3.3 ELASTICITY REQUIREMENT CLASSES

CEController's architecture main components are the classes that represent the developer's elasticity requirements, a metric reader and broker, a client for communication, and a controller to oversee the strategies.

In this section, we will go into detail about the elasticity requirement classes and subclasses, how they interact with one another, as well as the sequence of actions performed by the controller and elasticity classes when enforcing requirements.

Figure 4 presents a diagram illustrating the elasticity requirement classes. These classes represent CEController's language constructs and its subclasses. These classes define elasticity conditions, metrics to monitor, actions to take, and when they will be taken. The classes *MonitoredMetric*, *Condition*, *Strategy* and *Capability* are abstract classes and their subclasses define their behaviour. All subclasses shown in Figure 4 are currently implemented. However, more subclasses can be created to suit the needs of developers.

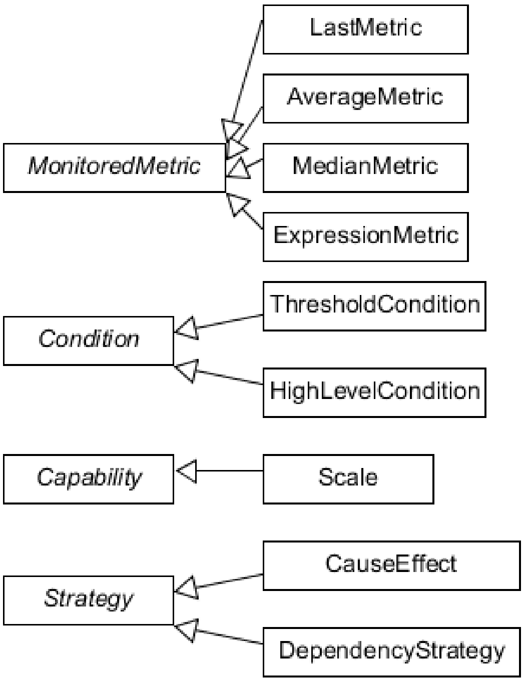

Figure 4: Elasticity requirement classes and subclasses.

Figure 5 presents a domain model of the elasticity requirement classes, as well as the communication channels. In this figure, the relationships between the elasticity requirements classes and the Controller are emphasized. We go into further details about Figure 4 and Figure 5 the next sections.

3.3.1 *MonitoredMetric classes*

The *MonitoredMetrics* subclasses act like monitors to specific metrics. These classes have the responsibility of calculating result values based on the metrics they subscribe to.

As mentioned earlier, all concrete *MonitoredMetric* classes inherit from the asbtract *MonitoredMetric* class and must implement all of its abstract methods. Figure 6 presents a class diagram in which all abstract methods can be seen. Some abstract methods, namely *give_metric_sub*, and *requires_metric*, are related to the subscription and aggregation of metrics. The former method, receives a *Metric* object and must return a *Boolean* that states whether the class subscribes to those kinds of metrics. The later method defines how the class calculates the aggregated value based on subscribed metrics. This value can later be calculated and obtained using another abstract method, the *get_value* method.

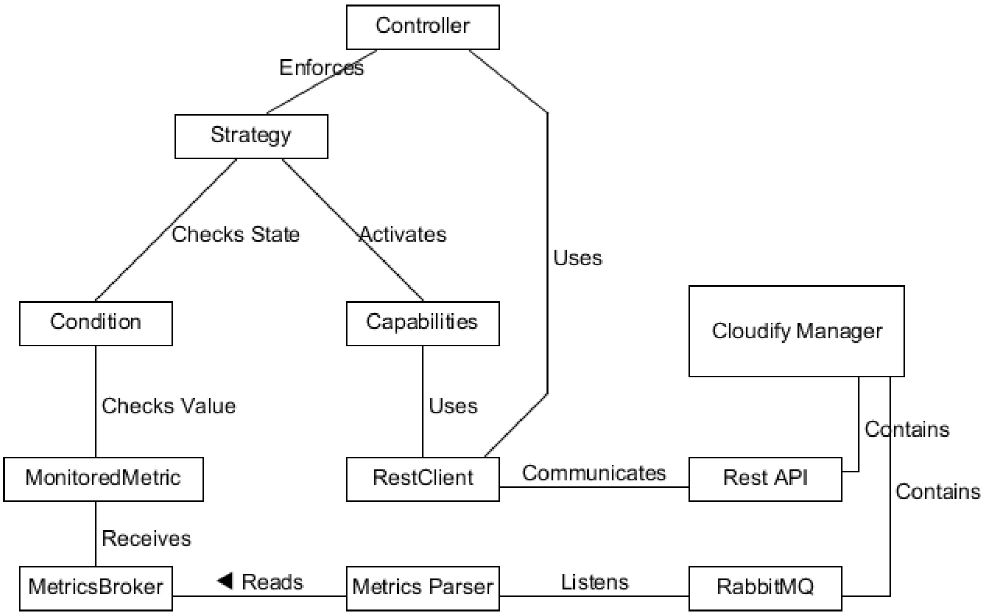

Figure 5: CEController's Domain Model.

Among currently implemented *MonitoredMetrics*, *AverageMetric* aggregates and calculates the average value of a predefined number of metrics . *LastMetric* only concerns itself with the last metric received. *ExpressionMetric* allows the developer to create expressions containing both metrics and *MonitoredMetrics*.

Metric life-cycle

Previously, the communication channels that allow CEController to receive deployment metrics from Cloudify was explained. Here, we focus on how these metrics reach interested monitors though a broker class, *MetricsBroker*.

As shown in Figure 6, during the callback method, the *MetricReader* object parses a metric from RabbitMQ and creates a *Metric* object. This object is then sent to the *MetricsBroker* using the *send_new_metric* method.

The *MetricsBroker* object must then check, for each monitor defined by the developer, if the monitor requires the received metric (Lines 11-12 of listing 3.3). Every interested monitor will receive the metric (Line 13). In this process, the *requires_metric* and *give_metric* methods are used.

Since *MonitoredMetric* concrete classes can not only subscribe to metrics, but also to values of other monitors (such as the *ExpressionMetric* class). Every class that receives new metrics is stored (Line 14), and the process starts once again. This time, monitors will be asked whether they require values from the recently updated monitor. As long as there is at least one new monitor updated the process is repeated (Lines 16-21). There are some consequences to allowing monitors to depend on other monitors. The most important consequence for the developer is the ability to define monitors based on expressions that include constant values, metrics, and other expressions in order to better define elasticity requirements. The developer, however, must be careful not to have monitors depend on themselves or contain cyclical dependencies, as this pattern will cause controller failures through infinite cycles.

```
1  class MetricsBroker():
2    def __init__(...):
3      ...
4  
5    def send_new_metric(self, metric):
6        curr_time = time.time()
7        if time.time() < self.time_to_wait:
8            metric.set_validity(False)
9  
10         updated_metrics = []
11       for name, monitored_metric in self.monitored_metrics_dict.iteritems():
12         if monitored_metric.requires_metric(metric, False):
13             monitored_metric.give_metric(metric)
14             updated_metrics.append(monitored_metric)
15 
16       while len(updated_metrics) > 0:
17         single_updated_metric = updated_metrics.pop()
18         for name, monitored_metric in self.monitored_metrics_dict.iteritems():
19             if monitored_metric.requires_metric(single_updated_metric, True):
20                 monitored_metric.give_metric(single_updated_metric)
21                 updated_metrics.append(monitored_metric)
22 
23   def reset_metrics(self):
24     for monitored_metric in self.monitored_metrics_dict.values():
25         monitored_metric.reset_metric()
26 
27   def ignore_metrics(self, seconds_to_wait):
28     current_time = time.time()
29       self.time_to_wait = current_time + seconds_to_wait
```

Listing 3.3: MetricsBroker class source.

Metric invalidity period

One common issue with elasticity controllers using thresholds is controller instability. In this scenario, actions are triggered by metrics causing the system to scale. However, these actions' effects are not instantaneous and the deployment's performance may not increase

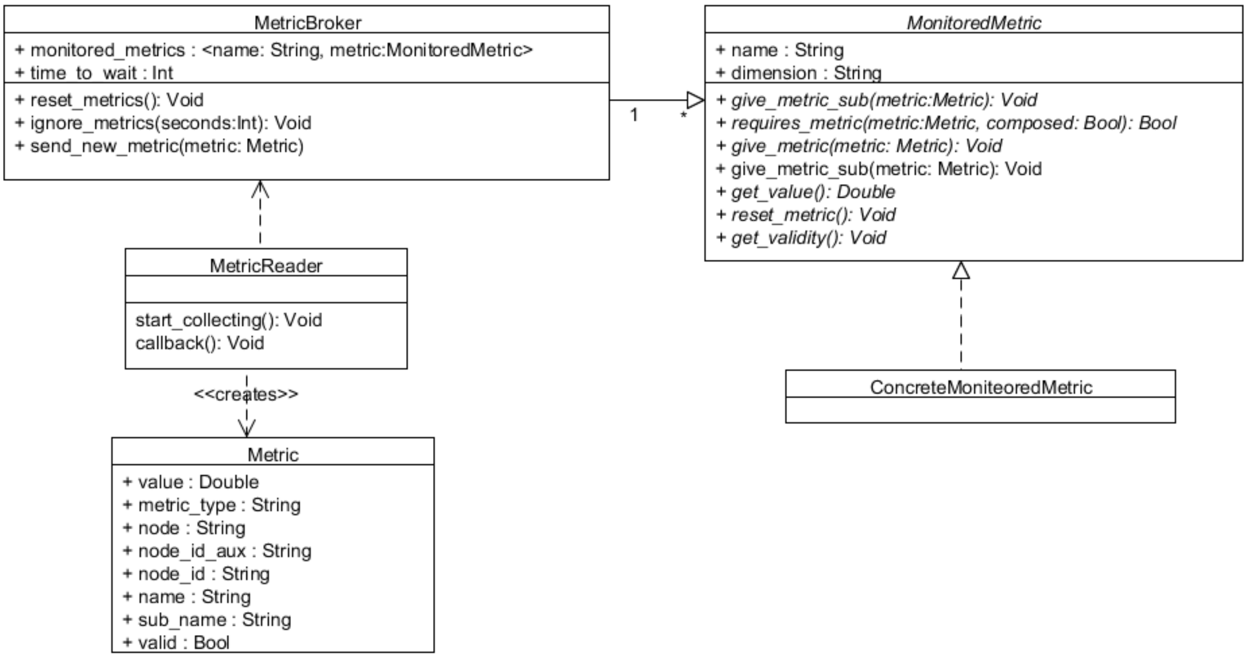

Figure 6: Elasticity requirement classes and subclasses.

right away, and might even temporarily decrease. In these cases a threshold based controller may decide, based on the metrics, perform further unneeded actions. It is advisable that such controllers have an inertia period [35]. According to the authors, during these inertia periods actions are not performed.

Instead of implementing a constant inertia timer that prevents actions to be taken, CE-Controller implements an inertia periods using invalid metrics. In summary, actions may result in an invalidation period. During this invalidation period, the state of all collected metrics is invalid. When monitors are queried on their current state of validity, if there are invalid metric among the aggregated metrics, the monitor must present an invalid state.

This invalidity period is started by calling the *ignore_metrics* method from the *MetricsBroker* class. In order to check whether a monitor has a valid or invalid state the *get_validity* method can be used. In listing 3.3(Lines 6-9) we can see that whenever a *MetricsBroker* object receives a metric, the object determines if it is currently in a invalidity period. If so, then metrics are given to monitors possessing an invalid state.

This method has some benefits over an inertia timer, namely, not allowing metrics from these invalid periods to be used in calculations independently of the monitor aggregation strategy. Considering the following scenario, where an *AverageMetric* monitor is defined with parameters in order to aggregate 100 units of a specific metric. Whenever a inertia scenario ends and actions are once again allowed, there may still be some metrics from the inertia period still aggregated within the monitor. In the same scenario, but involving a

LastMetric monitor, which only aggregates one metric, it is very likely that the monitor will not contain any metric from the inertia period.

3.3.2 *Condition classes*

Condition classes allow the developer to define conditions based on the application's metrics. Although the inner working of the condition varies with each implementation, condition classes generally compute valid or invalid states based on *MonitoredMetrics*.

Concrete *Condition* classes must extend the *Condition* abstract class, shown in Figure 7. All conditions must implement the *get_state* and *get_dimension* method. The *get_state* method must return a valid or invalid state, however each concrete class can define their own algorithm for computing the condition state.

Currently, two types of conditions are implemented. The *ThresholdCondition* sub-class is instantiated with a metric monitor and a developer defined threshold as parameters and calculates whether a certain *MonitoredMetric's* value is under or above the threshold. The *HighLevelCondition* sub-class allows the developer to create conditions using two existing conditions and an operator AND, OR, WHEN. Since *HighLevelCondition* classes can also use other *HighLevelCondition* classes, it is possible to create more complex conditions in order to better suit the developer's needs.

Calculation of dimensions

Dustar et al. [28] has presented a three dimensional model for elasticity, while Moldovan et al. [37] has presented a separation of gathered metrics into these three dimensions. In CEController, we have followed the three dimensional model, and allow the developer to state, in the configuration files, the dimension of each of their defined monitored metrics. The dimension class attribute is shown in Figure 6.

CEController also implements a system in which conditions can be separated into the before mentioned dimensions, based on the monitors used by them. As such, all sub-classes must implement the *get_dimension* method. Whenever a condition is based on monitors that monitor metrics of different dimensions, the condition's dimension returns 'None'. This last scenario often occurs when using *HighLevelCondition* objects.

3.3.3 *Capability classes*

Capability instances define actions that can be taken in order to enforce the elasticity requirements. When capabilities are activated, they receive a REST client object as a parameter and request the desired actions from the Cloudify manager. Currently only the *Scale* capabil-

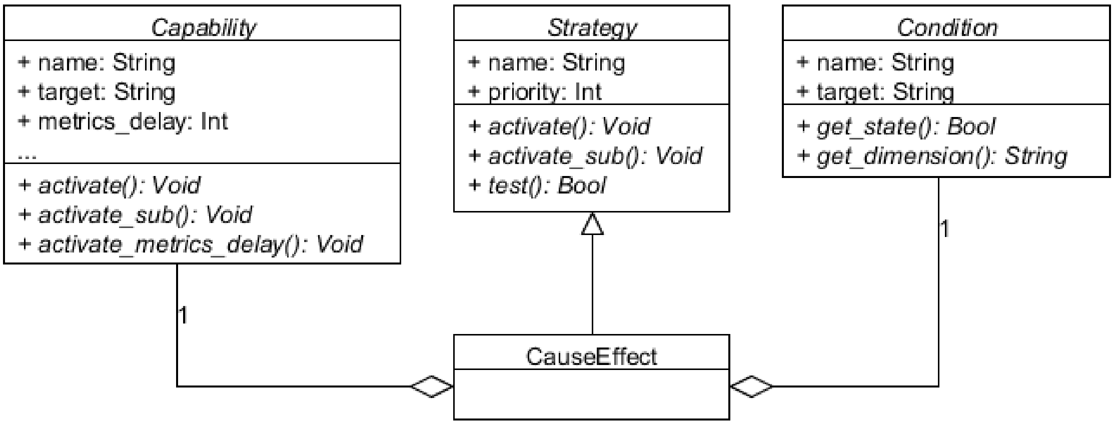

Figure 7: Class diagram illustrating the Capacity, Strategy, and Condition classes. Also included is one concrete Strategy class, CauseEffect.

ity is implemented. The *Scale* capability requests a scale-in or scale-out workflow for a component.

3.3.4 *Strategy classes*

Strategy classes are the bridge between conditions and capabilities. As shown in Figure 7, *Strategy* sub-classes must implement two methods: *test* and *activate_sub*. The *test* method verifies whether the conditions for the use of the strategy are met, while the *activate_sub* method activates elasticity capabilities.

Currently, the simplest implemented strategy is *CauseEffect*. *CauseEffect* has the responsibility of testing one condition in order to activate one capability. Later on, we will go into detail about the *DependenciesStrategy* class.

3.3.5 *Controller*

The central component of CEController is the Controller class. The Controller class has two main responsibilities. First, the Controller class must read the developer's configuration files and instantiate the necessary elasticity requirement classes (Listing 3.4, Lines 2-4). Second, the Controller will continually check all strategies, and activate those that are needed (Lines 7-16).

```
1 class Controller:
2   def __init__(self, config_file_path, server_file_path, topology_file_path):
3     self.options_parser = OptionsParser(...)
4     ...
```

```

  def start(self):
    while True:
      if self.rest_client.check_available():
        for strategy in self.strategies:
          if strategy.test():
            strategy.activate(self.rest_client)
          else:
            print ”Strategy: {} is ok.”.format(strategy.name)
      else:
        print ”Client is busy.”
      sleep(10)
```

Listing 3.4: Controller class source.

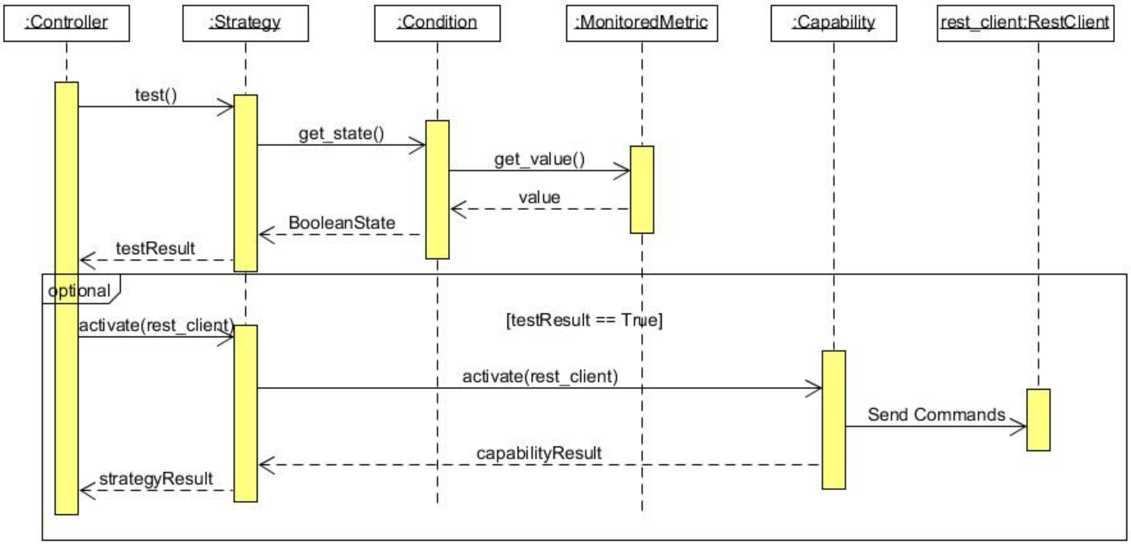

Figure 8: Sequence diagram detailing the test and activation of a strategy.

Figure 8 presents a sequence diagram illustrating how the objects interact with each other during the *test* and *activate* methods of *Strategy* classes. *Strategy*, *Condition*, *Capability* and *MonitoredMetric* are abstract classes and their respective methods must be implemented by their subclasses. This model allows for the creation of new subclasses that can perform their tasks differently while still being compatible with the existing model.

Every subclass of MonitoredMetric must implement the *get_value* method, that returns the value of the metrics monitored. In case of the AverageMetric subclass, the get_value method averages the currently stored metrics. Meanwhile, the ExpressionMetric subclass, which allows the developer to use mathematical expressions, will read the value of all MonitoredMetric classes present in the expression and calculate the expression's value. If a developer has the need to read values in a different manner, such as an exponential weighted moving average, the developer could achieve this by creating a new MonitoredMetric subclass that uses an exponential weighted moving average to calculate its value.

3.3.6 *DependenciesStrategy*

When faced with the task of enforcing elasticity requirements, we often encounter cases in which more than one component is not behaving according to the requirements. This would often mean that several conditions would have a violated state and several strategies would require activation. Additionally, some metrics belonging to the quality dimension do not accurately reflect the component's performance when the component has dependencies. For example, in a load-balancer connected to a web server, which in turn is connected to a database topology, defining strategy to act upon the load balancer if a resource metric such as CPU usage is above a certain threshold might be a good option. However, defining a strategy to act upon the same load balancer based on quality dimension metric, such as response time, might not have a positive impact, because the cause might be on its dependencies. Since the load balancer depends on the web server, if the web server takes too long to process requests, the response time metric of he load balancer component will be negatively affected. Additionally, the same pattern might be found on the web server, since it depends on the database.

In the previous example, an overloaded database component may negatively affect quality metrics of all components that depend on it. And the components that depend on the database are not responsible for the suboptimal behaviour of the application, and thus do not require action taken upon them.

Using conventional cause-effect strategies, the priority of each strategy becomes very important as it determines which strategy should be tested first, and thus activated first. However, this approach may be complex for the developer when an application has many complex dependencies.

In order to cover the before mentioned cases, we have designed the DependenciesStrategy. DependenciesStrategy is a strategy designed to be activated when a certain component is not performing according to requirements in ways that we suspect to be caused by its dependencies. In order to explain how the DependenciesStrategy class works, we will first present at its definition parameters and later its test and activate methods.

```
{"type": "dependencies", "name": "dependency_strategy1", "priority":0,
  "parameters":{
        "target": "LoadBalancerGroup",
          "condition_list":["lb_cpu", "lb_respTime", "ws_cpu", "ws_respTime", "
    db_cpu", "db_respTime"],
          "capability_list":["lb_scaleOut", "ws_scaleOut", "db_scaleOut"]
        }
}
```

Listing 3.5: Example DependenciesStrategy definition.

Listing 3.5 contains an example definition of a DependenciesStrategy in a web server application. The *type*, *name* and *priority* parameters are mandatory for all strategies. The inner parameter list contains the parameters *target*, *condition_list* and *capability_list* are parameters specific to the DependenciesStrategy class. The *target* parameter defines the component to be monitored by the strategy. The *condition_list* parameter defines conditions of the monitored component and, optimally, all of its dependencies. For each component, at least one resource dimension condition and one quality dimension condition is required. Finally, the *capability _list* parameter should contain one capability for each component, so that the strategy is able to take action upon each component.

Test method.

The *test* method defines whether the strategy should be activated. The *test* method of *DependeciesStrategy* verifies if the target component's resource condition is valid but contains a violated quality condition.

Activate method.

The *activate* method is responsible for activating the required capabilities. The *activate* method attempts to find out the component that is most likely responsible for the violation of the target's quality dimension condition. This is performed by iterating upon the component's dependencies and finding the components that have a violated resource condition. If the *activation* method finds another dependency that presents the same pattern as the target (violated quality condition, valid resource condition), it will iterate through all dependencies of that component as well. Finally, the *activate* method will select one of the components with the largest distance from the target component and activate that component's capability.

3.3.7 *Logging mechanism*

In order to obtain reliable test results for evaluation, CEController implements a logging mechanism. The logging mechanism's objective is to allow the logging of all events relevant to the developer.

The first step was to identify which events were necessary for evaluating the CEController. During this step, four events were identified as being very important:

- The arrival of metrics that are subscribed by monitors.
- Updates in monitor values.
- Activation of capabilities by strategies.
- Successful capability actions.

The difference between the last two scenarios is that when a strategy activates a capability, the capability must decide whether the activation is possible. A capability activation may not always result in an action due to limiting circumstances, such as reaching the limit of number of instances in a scale out capability.

The logging mechanism is implemented in two parts: A Python module containing global variables for configurations, and through template methods in the abstract elasticity requirement classes. First, a Python module containing global variables is defined, these variables define whether to log each event, the log file paths and log file names. These modules are then imported by all classes that have information to publish to the log, so that they are able to do so if required. As CEController's elasticity requirement classes are designed around having abstract classes that define the purpose of the classes, and concrete classes defining their behavior, we have used the Template Method pattern in order to implement logging. In Figure 6 and Figure 7, two activation methods in the strategy and capacity classes are shown. While one of the methods, *activate_sub*, is abstract, the other method, *activate*, is not. This is due to the *activate* method being a template method. The activate abstract method checks if logging is necessary, performs it when needed, and then calls the *activate_sub* abstract method. The *activate_sub* method defines the activation behavior, and consequently has to be implemented by the subclass. The same pattern occurs in the *MonitoredMetric* class with the abstract *give_metric* method and *give_metric_sub* abstract method.

Listing 3.6 illustrates example contents of each log file.

```
#Strategy activation log file
Time, Strategy Name
20.6800000668; cause_effect_in
30.8580000401; cause_effect_in
41.0309998989; cause_effect_in
91.5339999199; cause_effect_cass_in

#Capability activation log file
Time, Capability Name, Metrics Delay, Could Run
20.6800000668; scale_in_daasweb; 180; False
30.8580000401; scale_in_daasweb; 180; False
41.0299999714; scale_in_daasweb; 180; False
91.5329999924; scale_in_cassnode; 180; False

#Monitor update log file
```

```
Time, Metric Name, Metric Value, Validity
1.4470000267; cpu_system_cassnode; nan; True
1.99399995804; cpu_system_daasweb; nan; True

#Raw metrics log file
Time, Metric Name, Metric Subname, Metric Value, Metric Node
1.4470000267; cpu_system_cassnode; percent; 2.0; 8552d
1.99499988556; cpu_system_daasweb; percent; 1.0; b1793
```

Listing 3.6: Example contents of CEController's log files.

3.4 Control language definition

In this section, we describe the various mandatory and optional developer inputs and setup files present in CEController. In this section, first we discuss the input and explains its importance and utilization scenarios, finally we provide an example illustrating setup possibilities.

There are three main setup files with the purpose of allowing the developer to provide CEController with information about the controlled application, connection information and elasticity requirements. Additionally, there is a project file in which logging configurations can be changed.

Elasticity requirements configuration

The elasticity configuration file is the location in which the developer describes the elasticity requirements for the application. CEController's elasticity control language is derived from SYBL language for controlling elasticity proposed by Copil et al. [24]. Much alike SYBL, CEController's language is based around monitored metrics, conditions, capabilities and strategies. When using our controller.

```
{
  "conditions": [
    {"type": "threshold_condition", "name": "threshold_cpu_daasweb", "target": "DaasWebGroup", "parameters": {"monitored_metric_name":"cpu_system_daasweb", "threshold":80, "higher_lower": 1}},
    {"type": "threshold_condition", "name": "threshold_cpu_low_daasweb", "target": "DaasWebGroup", "parameters": {"monitored_metric_name":"cpu_system_daasweb", "threshold":20, "higher_lower": -1}}
  ],
  "capabilities": [
    {"type": "scale", "name": "scale_out_daasweb", "target":"DaasWebGroup", "metric_delay": 180, "parameters": {"node_id": "daasweb_machine", "delta":1, "minimum_nodes": 1, "maximum_nodes":4}},
    {"type": "scale", "name": "scale_in_daasweb", "target": "DaasWebGroup", "metric_delay": 180, "parameters": {"node_id": "daasweb_machine", "delta":-1, "minimum_nodes": 1, "maximum_nodes":4}}
  ],
  "strategies":[
    {"type": "cause_effect", "name": "cause_effect_out", "priority":0, "parameters":{"condition_name":"threshold_cpu_daasweb", "capability_name":"scale_out_daasweb"}}
    ],
  "metrics":[
    {"name": "cpu_system_daasweb", "type": "average_metric", "dimension":"Resource", "parameters":{"metric_name":"cpu", "metric_sub_name":"percent", "node_name":"daasweb_machine", "number_metrics": 7}}
  ],
```

Listing 3.7: Example elasticity configuration file.

As shown in Listing 3.7, the elasticity requirements configuration class has four sections. These sections are designated for the definition monitors, conditions, strategies and capabilities. The different classes of each group share some parameters. All monitors must have a name, type (defining the monitor type), and elasticity dimension. All conditions are required to have a defined name and type, while a target is optional. Capabilities require a name, type, metric invalidation period and optionally a target. Strategies require a name, type, and the strategy priority as parameters. Each class, however, may have additional parameters, these parameters are defined under the 'parameters' section. For example, when defining a threshold condition, the developer must define which monitor to receive values from, the threshold value, and whether the monitor value should be below or above the threshold.

3.4.1 *Server configurations*

As previously mentioned in the communication channels section, CEController communicates with Cloudify and RabbitMQ. Therefore, CEContoller requires knowledge of the location of these services. The server configurations file allows the developer to provide this information, as well as the queue name and exchange name that RabbitMQ will use for the project's deployment. Since RabbitMQ requires authentication in the default Cloudify configuration, user credentials also have to be provided.

Similar to the definition of elasticity requirements, the server configurations file is also a JSON file. Listing 3.8 contains an example of server configuration file.

```
{
  "metrics_server": {
    "queue_name": "m2m-riemann",
    ″exchange_name″:″m2m−monitoring″,
    ″host″:″192.168.116.26″,
    ″credentials_user″:″user″,
    ″credentials_password″:″password″
  },
  ″rest_client″:{
    ″host″: ″192.168.116.26″,
    ″deployment″: ″m2m″
  }
}
```

Listing 3.8: Example server configuration file.

3.4.2 *Definition of the deployment's topology*

The DependeciesStrategy class requires some knowledge of metric dimensions and application topology. While metric dimensions are defined in the creation of metric monitors, if the developer desires to use this strategy, the application topology has to be provided.The application topology consists of three groups: components, groups, and dependencies.

A component denotes any component type that delivers metrics and can be monitored. In general this means that every machine type will be a component. However, Cloudify considers machines and services running inside that same machine as separate entities. Consider the example of monitoring the CPU utilization and number of open connections in a machine that contains a load balancer. In this example, the CPU utilization metric will come from the machine component, while the number of open connections will come from the load balancer service component. In this example, the developer should define two types of components in the topology, the machine type, and the services running inside the machine instances. It should be noted that components do not represent a single instance, but all instances of the same type. For example, a load balancer component will aggregate metrics from all load balancer machine instances.

A group denotes an arbitrary group of elements defined by the developer. These elements can be components or even other groups. In general groups are used to aggregate metrics originated from various components. While the developer has freedom in defining groups, the groups should represent a set of components that can be viewed as real or logical entity that originates metrics. For example, in the previous load balancer example, it would appropriate to define a load balancer group, containing the load balancer machine type and the load balancer services. In this way, the previously mentioned metrics, CPU utilization and number of open connections, will be aggregated under have the same target: the load balance group.

So far, we have described the main entities in the topology. Now we are going to describe "dependencies", which, as the name suggests, describe dependencies between entities. By defining dependencies, the developer informs CEController of which entities depend on others and, by extension, which entities might be affected if one of their dependencies is in a non-optimal state.

Listing 3.9 shows an example topology definition file. Much alike the other definition files, a JSON syntax is used to define the topology elements and dependencies. In the definition of nodes, the groups must contain an array of any combination of elements or groups. When declaring dependencies, the element that depends on others is listed first, followed by an array of its dependencies, which can be groups or nodes. Figure 9 illustrates the topology defined in this topology file.

```
{
  "Nodes":[
    "CassandraSeed", "CassandraNode", "Haproxy", "DaasWeb", "
    cassandranode_machine", "cassandraseed_machine", "daasweb_machine", "
    haproxy_machine"
  ],
  "Groups":{
    "CassandraSeedGroup": ["CassandraSeed", "cassandraseed_machine"],
    "CassandraNodeGroup": ["CassandraNode", "cassandranode_machine"],
    "HaproxyGroup": ["Haproxy", "haproxy_machine"],
    "DaasWebGroup": ["DaasWeb", "daasweb_machine"],
    "CassandraGroup": ["CassandraSeedGroup", "CassandraNodeGroup"],
    "SystemGroup": ["HaproxyGroup", "DaasWebGroup", "CassandraGroup"]
  },
  "Dependencies":{
    "HaproxyGroup":["DaasWebGroup"],
    "DaasWebGroup":["CassandraGroup"]
  }
}
```

Listing 3.9: Example deployment topology file.

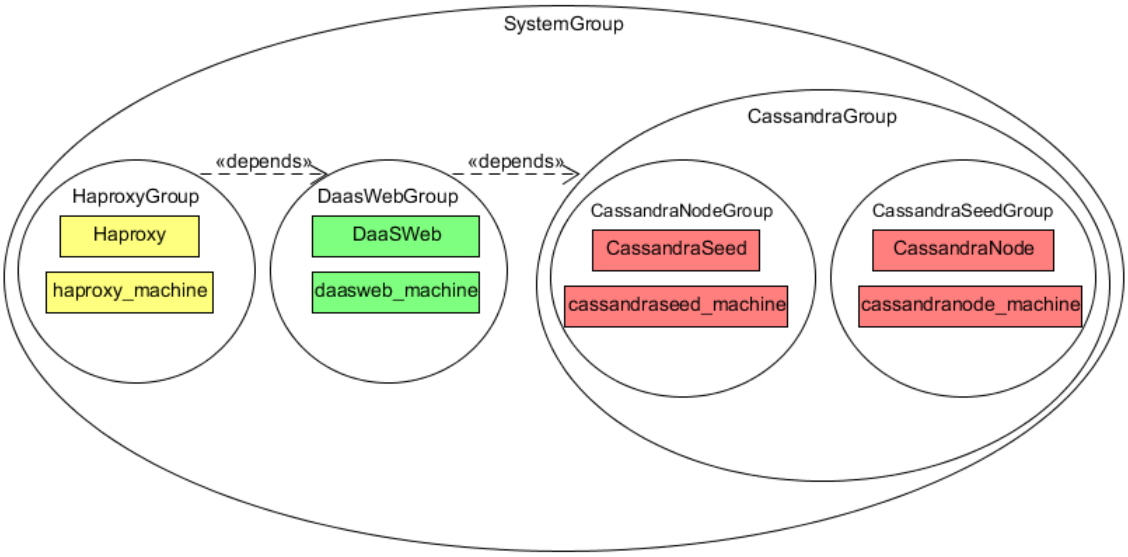

Figure 9: Diagram illustrating the topology defined in Listing 3.9.

4

EVALUATION

After the development of CEController, we had to deploy the controller in live environments in order to evaluate it. For this, we adapted an application that has been previously used in the evaluation of elasticity controllers. In this section, we go into detail about the creation of the Cloudify blueprint and scripts for the orchestration of the evaluation environment. Finally, we present the results obtained during the tests and discuss the evaluation results.

4.1 EVALUATION ENVIRONMENT

In order to test and evaluate CEController, we have adapted the M2M DaaS application, presented by Copil et al. [25], for Cloudify. The M2M DaaS application is a web service that runs on top of a NoSQL cluster. The application consists of four machine types:

- A load balancer machine. This node distributes client requests between the existing web servers.
- A web server machine. The web server node processes the clients requests. The web server machine connects to the data seed machine to perform database requests.
- A Cassandra seed and node machines.

M2M DaaS processes many types of client requests, such as loading a static web page, and database requests encoded in XML. In the last scenario, the web server receives the XML in a HTTP data field, and requests the appropriate database action. There are two valid elasticity actions possible in the M2M DaaS application, adding or removing web server instances, and adding or removing Cassandra node instances.

Figure 10 shows a diagram of the M2M DaaS application topology. Also contained in Figure 10 is the FTP source code distribution machine, later in this section we will go into detail about this machine and it's uses.

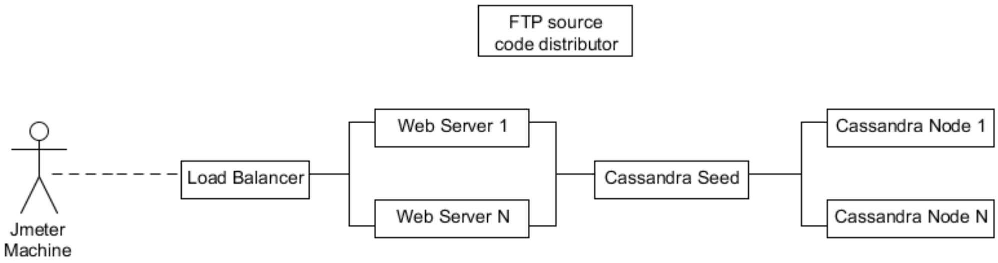

Figure 10: M2M DaaS application topology. Adapted to include a FTP server to provide the application's source code.

4.1.1 *Cloudify blueprint and scripts implementation*

In order to orchestrate the program with Cloudify, we created a Cloudify blueprint with the application's topology and configurations, as well as Bash and Python scripts that allow the machines to start their services.

In Cloudify, the blueprint creator defines nodes. There are two types of nodes, *node_templates* and *node_types*. *Node_templates* represent services or applications that are to be created. *Node_types*[5], on the other hand, are used to define common properties of nodes, and should be extended by *node_templates*[4].

The terminology becomes a somewhat more complex when using nodes with *contained_in* relationships. *Contained_in* relationships are commonly used to define that service nodes that are contained within an host node. For example, in a web service, the blueprint creator may define a host node to represent the virtual machine, and a service node to represent the actual web service application[6]. The blueprint creator will then apply the *connected_in* relationship, using the application node as the source, and the host node as the target. If there are multiple instances of the host node, each host node instance will have a service node contained in it [6].

In Figure 11 presents a diagram of the Cloudify blueprint for the M2M DaaS application, during this section, we will explain in detail the various components applied in the blueprint.

The first step in the blueprint creation is the definition of two *node_types* for the hosts. We use different two types of hosts. A smaller *node_type*, *MonitoredServer*, containing one virtual CPU and has 0.5 Gb RAM, is used for the load balancer and web server. The larger one, *MonitoredServerCassandra*, contains 1 VPU and 1 Gb RAM, and is used for the Cassandra seed and nodes as they require at least 1Gb of RAM. In these two *node_types*, we add several Diamond collectors that collect metric general information, such as CPU utilization percentage, memory usage, disk usage, and network usage.

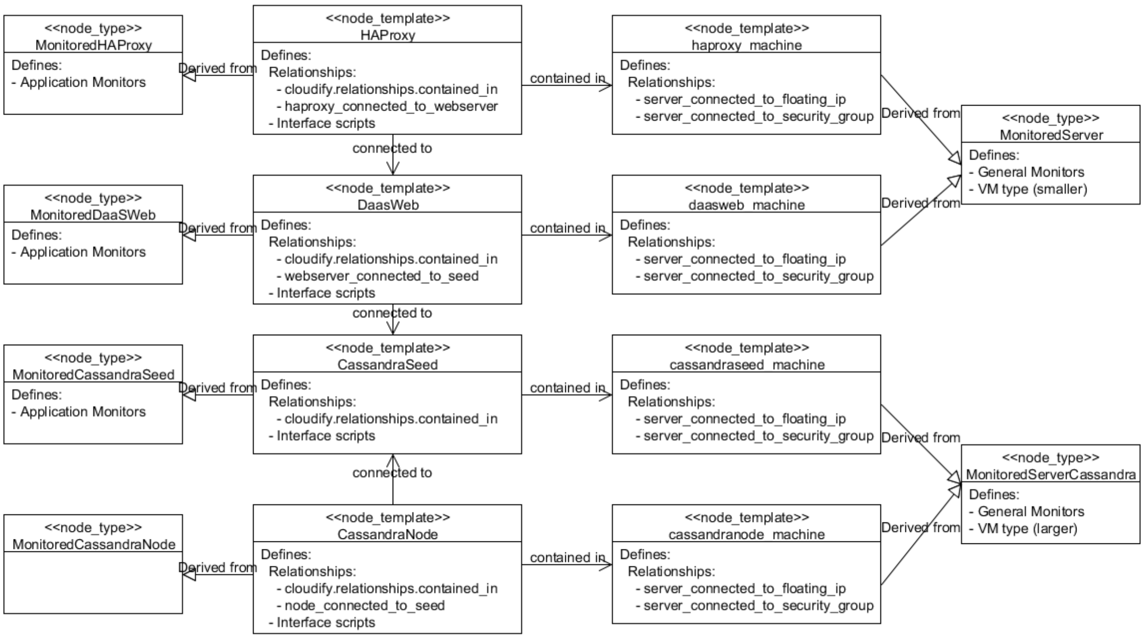

Figure 11: Diagram illustrating nodes and relations in M2M DaaS' Cloudify blueprint.

Later, we extended the two *node_types* into four M2M DaaS *node_templates* that represent the host machines. This is done by extending the previously defined *node_types*. In these *node_templates*, we also define whether the host will have an external IP address, and define the host's security group. The security group has to be defined in the blueprint along with the group's rules. For this project, we have created a security group that allows all IPv4 traffic flow through as security is not the main concern. All four *node_templates* are then connected to the previously defined security group.

Later, we created one *node_type* in order to define each M2M DaaS component. These *node_types* define the service that will run on top of the hosts. First, we declare a *node_type* per service, and define the service specific collectors. These collector are tasked with acquiring metrics that are specific to the server and sending them to the Cloudify manager. These specific metrics measure the performance of the services through application specific metrics. In the web server type, the collectors track the following metrics: average number of pending requests, average throughput, and average response time. In the HAProxy service, the average connection rate is collected. In the Cassandra seed service, the read count, write count, read latency, and write latency are collected. Because Cloudify requires Diamond collectors, the service collectors were adapted from their original form as Ganglia collectors, in the M2M DaaS source[15], into Diamond collectors.

After the creation of *node_types* for the services, *node_templates* are created by extending these types and providing the necessary additional information. The most important part of defining the service is perhaps the definition of the installation life cycle scripts. The install built-in workflow [7] executes the operations necessary for the node installation. In turn, we have to make sure that we define the correct operations to be executed. For this task, we define three operations for each *node_template*: One operation to be executed after the VM creation, one as the installation of the service application, and one to start the service application.

The scripts developed for the M2M DaaS application consist of bash and Python scripts, and their task is to automatize the creation of the services.

Now that one *node_template* was created for each host, and another *node_template* was created for each service. We define a *contained_in* relationship in each service pointed at their respective host. In Cloudify, the developer is allowed to declare relationships by extending existing ones and attributing a custom operator in order to redefine their default implementation[10]. In the M2M DaaS, we extend the *connected_to* relationships defining the following new operations:

- For the HAProxy connection to the web server services, we create a Python script for adding the new web server instance to HAProxy. Whenever there is a new connection, the HAProxy instance retrieves the new web instance's IP address from Cloudify and uses a registration bash script present in the M2M DaaS source to register the new instance. When a web server instance is uninstalled, the HAProxy instance retrieves the web server's IP address and calls a de-registration bash script present in the M2M DaaS source.

- For the Cassandra node service connection to the Cassandra seed, a Python script is created to be executed by the Cassandra seed whenever new Cassandra node is created. The script reads the Cassandra seed's IP and stores it on the Cloudify manager so that the Cassandra node can later access it in order to join the group during its installation.

- For the web service connection to the Cassandra service, we define a similar Python script similar to the above one. During the script's execution, the Cassandra seed stores its IP address to be later used by the web server service during its installation.

These relationships are then used to connect the services with their dependencies, as illustrated in Figure 11.

4.1.2 *FTP server*

Because the M2M DaaS source has to be compiled before its installation and we have made some small changes in the M2M DaaS source, we have created an FTP server virtual machine in order to store the compiled M2M DaaS application. During their installation, the services request the compiled application from the FTP server. The FTP VM has to be launched in separately, during the Cloudify manager's installation and the installation scripts must be provided with the FTP VM address. In order to be quickly launched, we have created a snapshot of the virtual machine in Openstack. In this way, the virtual machine can be created through Openstack very quickly. In Figure 10, we can see the FTP server virtual machine as part of the topology.

4.1.3 *Workload generation*

Our main objective in using a load testing tool was being able to apply a reliable and known workload to the test application. The tool chosen for workload generation was Apache Jmeter[1]. Jmeter can be used to simulate a variable workload on the server and allows us to simulate changes in workloads seen by live applications. This allows us to to analyze the application's performance under each load, and most importantly, analyze CEController's response to the variations in the generated loads.

JMeter allows for the creation of client threads that constantly send requests to the M2M DaaS application. Each thread performs a write request to the server, and waits for the response, each thread then repeats this process until it is terminated. In order to allow the number of threads to ramp up and down, we have used the Ultimate Thread Group JMeter plugin[2]. Figure 12 contains a graph taken from JMeter illustrating the number of client threads over the test period. The test period is 47 minutes and 40 seconds long, and the maximum number of client threads is 30. The test period contains a one minute idle period at the beginning, and a ramp up period lasting 10 minutes. JMeter then holds at a constant number of client threads for 20 minutes, and finally discontinues all client threads in a 16:40 minute period.

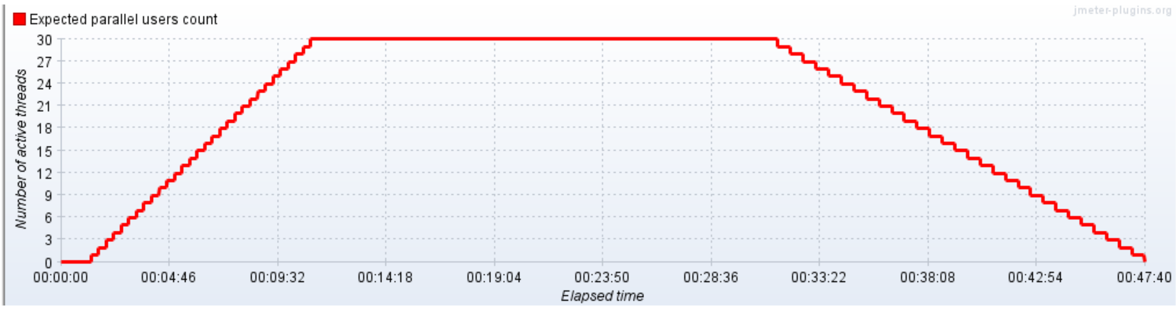

Figure 12: Amount of client threads during the test period. Taken from JMeter.

In order to guarantee that every test is performed under the same conditions, we have taken the following measures: Jmeter is used from a virtual machine created in Openstack, which is connected to the same network as the M2M DaaS virtual machines. The load generator virtual machine has 2Gb of RAM and 2 virtual CPUs. Additionally, before every test, an initialization procedure is requested by Jmeter. The initialization procedure consists of a table deletion request, followed by a table creation request. The initialization is performed in order to ensure that the Cassandra database table is empty before the start of every test.

4.2 CONTROLLER EVALUATION

In this section, we present two tests performed with the objective of evaluating CEController. For each test, we first present its purpose, then we describe its elasticity requirements, and finally we describe the actions taken by CEController and present the results.

4.2.1 *Cause effect strategies*

The objective of this test was to evaluate CEController with commonly used elasticity requirements. The elasticity requirements used in this test were the following:

- We have used *AverageMetrics* to monitor CPU utilization metrics from the web server machines and Cassandra node machines. These monitors were defined to aggregate the last 7 observations collected. Additionally, an *AverageMetric* monitor was created to track each web server's throughput, measured in requests per minute. While the throughput data will not be used in the elasticity requirements, we have used this information in the controller evaluation.

- For the web server and Cassadra node machines, we have defined *ThresholdCondition* instances, with thresholds concerning the CPU utilization being lower than 20% and higher than 80%.

- For the web server and Cassandra node, we have defined *Scale* capabilities, one capability to scale out and another to scale in each component. Finally, we use *CauseEffect* strategies to associate the upper bound threshold conditions with the scale out capabilities, and lower bound threshold conditions with scale in capabilities.

- In order to prevent instability, all capabilities have been parametrized to force the controller to ignore incoming metrics for 3 minutes upon capability activation. This prevents instability by not allowing CEController to use metrics from periods when scaling actions may have adverse effects.

Evaluation results

A graph containing the throughput of each web server machine over the test period can be seen in Figure 13. Figure 14 contains the average CPU utilization rate of all instantiated web server machines over the test period. Additionally, Figure 15 contains the average CPU utilization rate of all Cassandra node machines. CEController allows for the definition of metric invalidity periods during capability activations. Consequently, there are time periods in which the CPU utilization monitors have invalid values. In Figures 14 and 15, the graph values are presented in blue during the monitor validity periods, and orange during the monitor invalidity periods.

As we can see from the test results in Figure 14 and Figure 15, at 165 seconds, a few moments after CPU utilization rate exceeded 80%, a strategy was activated and the scale out capability was activated. Consequently, a new web server instance is added to the application. The same process occurs again at 379 seconds, increasing the number of web server instances to 3, which is the maximum due to our resource availability. At 671 seconds, the average CPU utilization from the Cassandra node exceeded 80%. Consequently, a scale out strategy is activated and another Cassandra node instance is added. At 858 seconds, when the number of threads was around 30, the CPU web server monitor once again has only valid metrics. From this point onward, the amount of instances is still not sufficient for the workload, and the CPU utilization sometimes goes above its desired range. However, at this point CEController has already allocated all available resources.

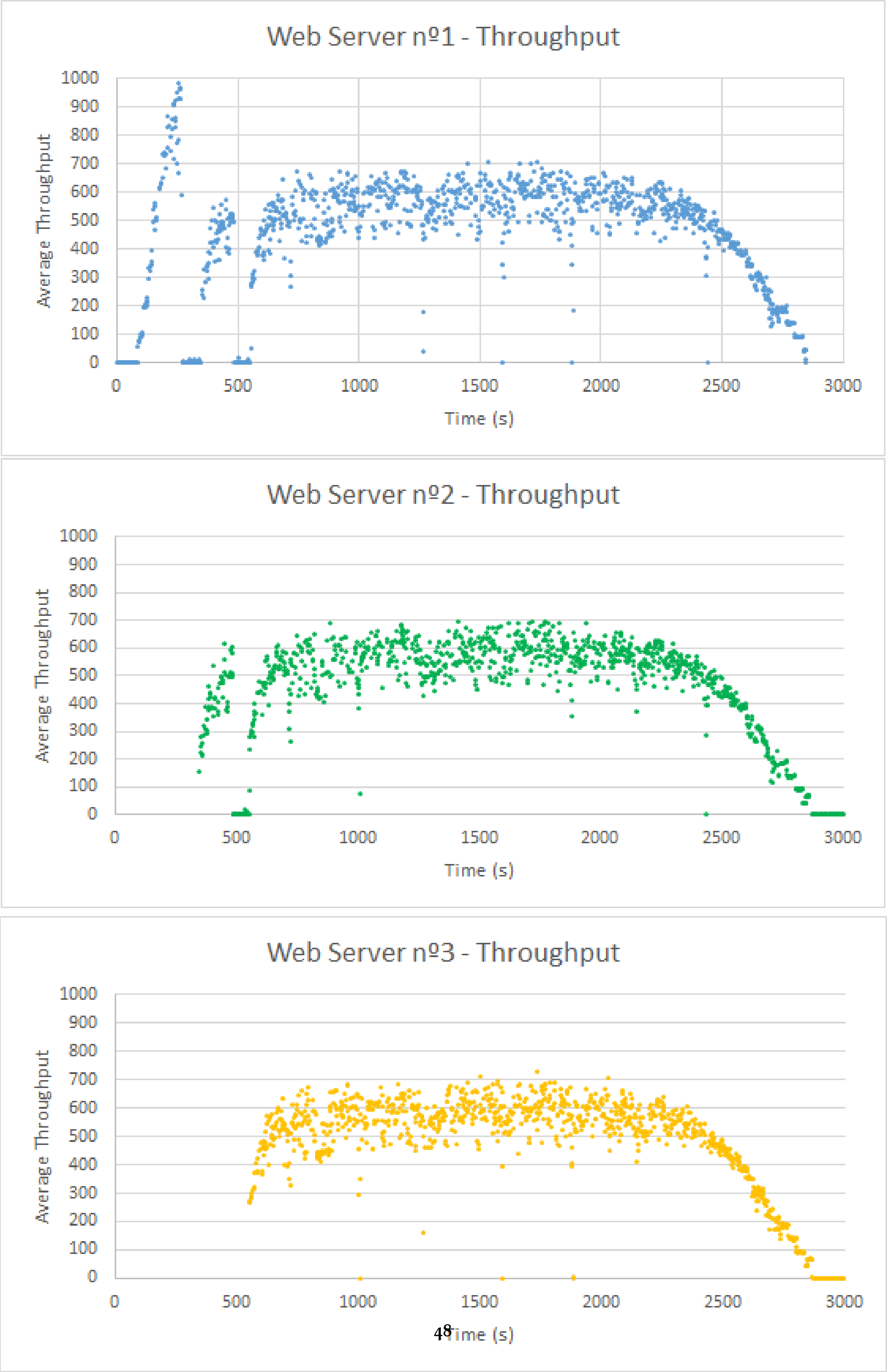

Figure 13: Calculated average throughput for each web server machine during the test period.

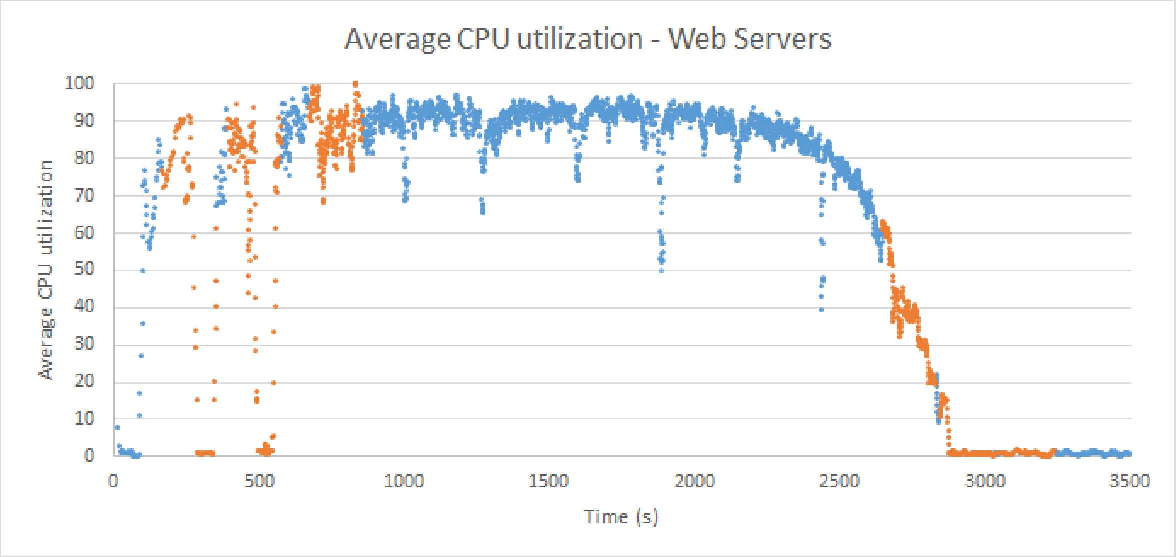

Figure 14: Calculated average CPU utilization percentage of the web server machines during the test period.

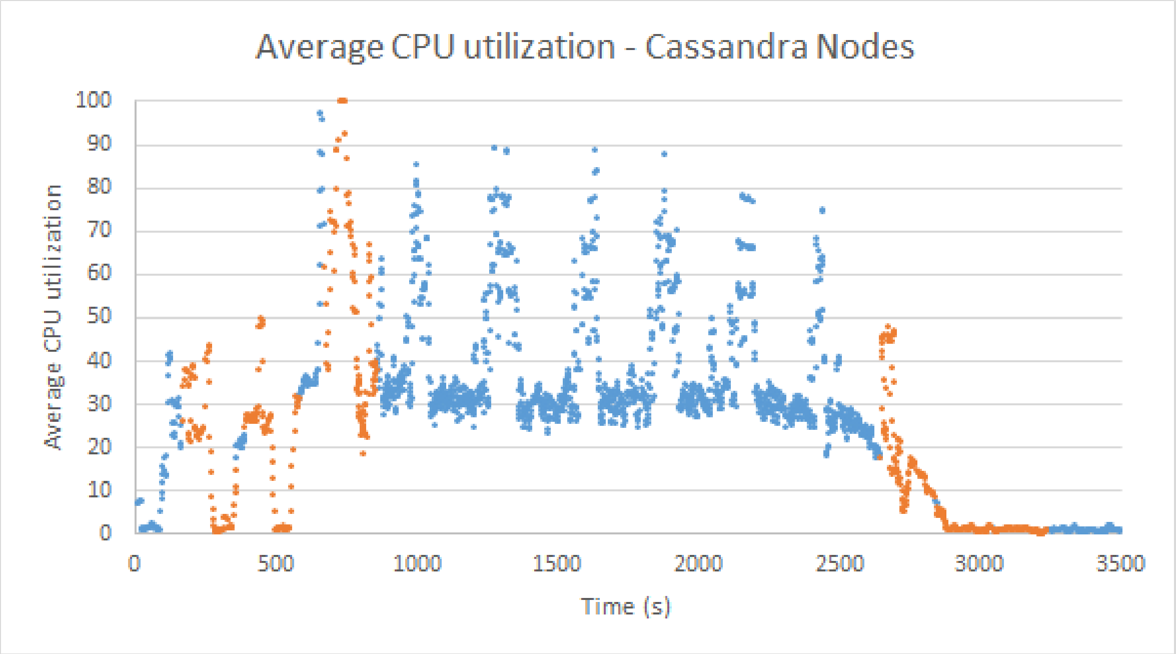

Figure 15: Calculated average CPU utilization percentage of the Cassandra node machines during the test period.

Finally, as the number of client threads decrease, CEController scales in one Cassandra node at 2645 seconds as the CPU utilization rate drops below 20%. This process happens and again for the web server at 2845 seconds and at 3046 seconds for the Cassandra node, leaving only one web server instance and one Cassandra node instance. In the final moments, the workload has already ended and so the CPU usage percentage is very low. However, since the minimum number of instances is one, no further actions are taken.

During the test period with 30 client threads, CEController could not reach a configuration in which the workload was adequate for the number of instances. This was because the JMeter virtual machine using 30 client threads generates more load than our fully scaled application can handle. However, by analyzing the throughput from each web server in Figure 13, we can see that the maximum throughput achieved with only one web server was close to 1000, while with the second instance the total throughput value is increased, and with the third instance the total throughput value was considerably higher, reaching around 1800. So we can conclude from these results that the elasticity actions taken by CEController followed the elasticity requirements as closely as possible and improved the performance of the application.

4.2.2 *DependenciesStrategy test*

In this test, we use only one *DependenciesStrategy* to scale out nodes, and using threshold based cause effect strategies to scale in nodes. With this test we present *DependenciesStrategy* as a possible alternative for other commonly used elasticity strategies. *DependenciesStrategy* requires both resource and quality metric monitor values for each dependency. In this test, the quality Because of this, we will be performing some small changes in the Workflow in order to better capture the latency of the system's components. For this test, in order to better evaluate the latency of requests, we have modified the workload in the following way: writing two new rows instead of only one in each write request. We have also lowered the maximum number of Jmeter threads to 25 so that the load remains similar.

The elasticity requirements defined for this test were:

- The following metrics were monitored: average CPU for the webserver, Cassandra node, and Haproxy machines, average response time per 5 seconds of the web server machines, average write time of the last 1024 requests given to Cassandra node machines, average response time of the Haproxy machine, and average web server throughput. All average CPU metrics are monitored by *AverageMetric* monitors. The response time and write time metrics, however, are monitored by *MedianMetrics*. This is because we have noted in previous tests that response time is a metric very prone a small amount of outliers with large deviations. For example, we have noted that one dropped packet may cause a response time of several seconds due to timeouts, such packets can, even in very small amounts, cause the average latency to become unstable. Because of this, we have decided to aggregate the before mentioned metrics with monitors that provide the median of the aggregated metrics as a result value.

- In order to scale in the machines whenever needed, a scale in capability is defined for the Cassandra node and web server. Two conditions are defined to compute if the calculated value of the CPU utilization rate is below 20%. Additionally, two *CauseEffect* strategies are defined so that when the before mentioned conditions are invalid, the scale in capabilities are to be called.
- Three *ThresholdConditions* are defined in order to set the desirable range of the calculated latency metrics. One condition to verify if the monitored haproxy response time is above 35 milliseconds, another for verifying if the web server response time is above 30 milliseconds, and a final one for verifying if the Cassandra node write time is above 28 milliseconds. These conditions represent our desirable range for the quality metrics of the application.
- Two *Scale* capabilities are defined for scaling out the web server and Cassandra nodes. An *Empty* capability is declared so that haproxy has a valid capability. This capability will perform no actions and will not activate any invalidity periods, but exists so that we can use the *DependenciesStrategy* targeting haproxy.
- One *DependenciesStrategy* is defined. The *DependenciesStrategy* is given all three conditions regarding quality metrics mentioned above, the two scale out capabilities, and the *Empty* capability.

Evaluation results

In this test, the *DependenciesStrategy* is the only strategy that scales out the application. This strategy is activated whenever the latency of the service goes above 35 milliseconds. This means that the main factor for activating actions is the degradation of quality of the service. In order to choose which component of the application needs to be scaled, both quality and resource metrics are used. As the defined *DependenciesStrategy* is only used for scaling out the application, we use a *CauseEffect* strategy to scale in whenever the average CPU utilization of a component drops below 20%.

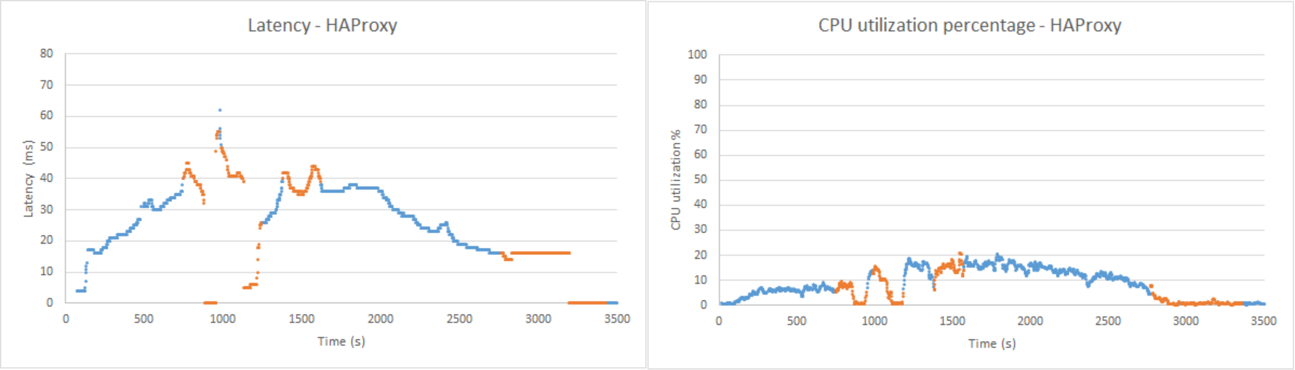

Figure 16: Calculated latency and CPU utilization percentage of the Haproxy during the test period.

Figure 16 contains a graph illustrating the latency and CPU utilization values calculated by the monitors. It can be seen that the CPU utilization never surpasses 80% during the test period. Consequently, the *DependenciesStrategy* will only act upon haproxy if all of its dependencies are in acceptable states, and since there is an *Empty* capability associated with it, no actions will be taken. Also in Figure 17, we can see that, in some periods, the latency exceeds 35 milliseconds. During these periods, *DependencyStrategy* is activated.

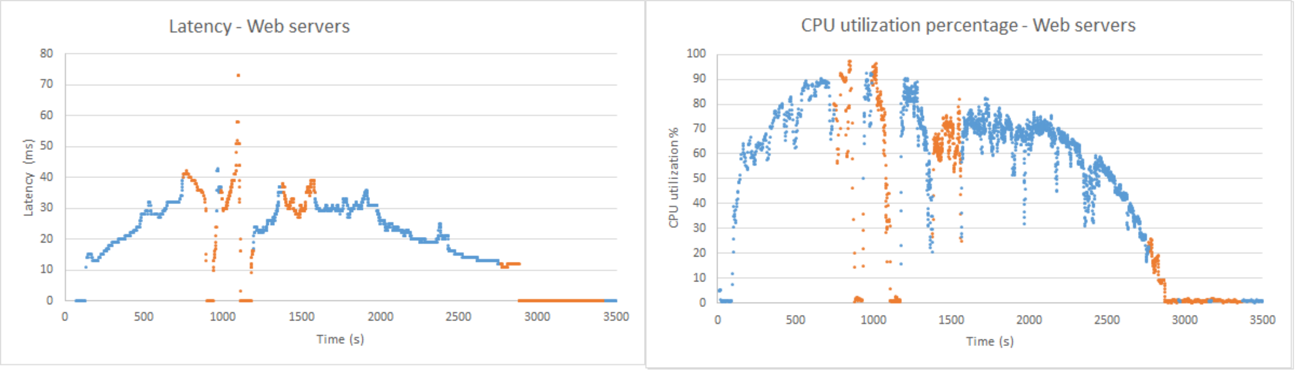

Figure 17: Calculated latency and CPU utilization percentage of the web server machines during the test period.

Figure 17 contains the values calculated by the monitors for the latency and CPU utilization of the application's web server machines. From Figure 17, it can be seen that in some periods, both the CPU utilization and the request latency exceeds the desirable values of 80% and 32 milliseconds respectively. Figure 18 contain the values calculated by the monitors for the write time and CPU utilization of the Cassandra node machines. As we can see in the figure, the write time is much lower than the desired maximum of 28 milliseconds at all times. Further analysis of the application can likely lead to the definition of a better threshold for this metric.

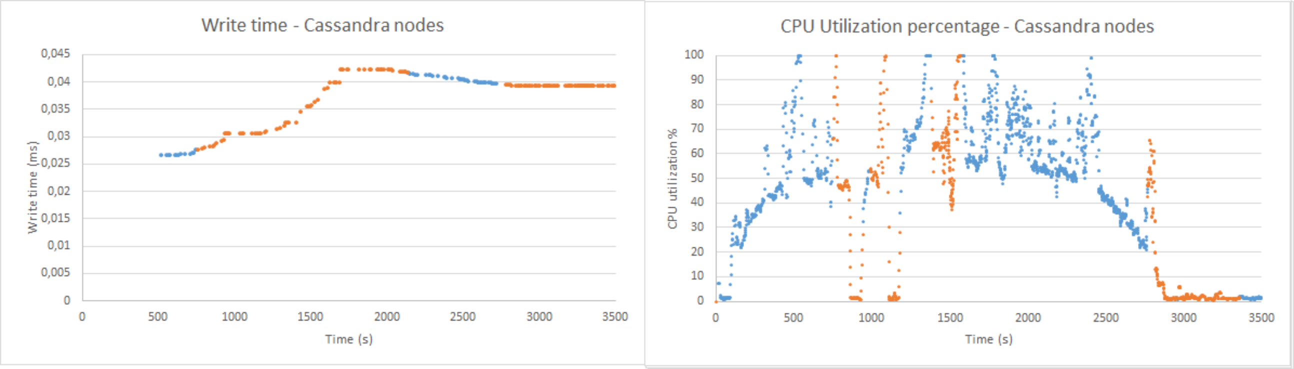

Figure 18: Calculated write time and CPU utilization percentage of the Cassandra node machines during the test period.

During the test, at 745 seconds, the Haproxy and web server latency was above their maximum desirable values, however, the web server's CPU utilization was still inside the desirable range. Consequently, the CPU utilization and the write time of the Cassandra nodes were checked. Since all monitor values for the Cassandra nodes were within the desired ranges, the web servers were scaled out, creating a second web server instance. The same situation occurs again at 978 seconds. However, this CPU utilization value for the web servers were above its maximum desired value. Therefore, another web server instance was created, reaching the maximum of 3 instances. At 1380, a similar situation happens. This time however, the web server CPU utilization was below the maximum desired value of 80%. While the write time of the Cassandra nodes was much lower than its maximum desired value, the CPU utilization of the Cassandra nodes was not. Consequently, another Cassandra node instance is created, reaching the maximum number of instances of 2. From this point onward, as the number of resources had reached its maximum, no other scale out actions would be performed.

At 1860, the number of Jmeter threads running started to decrease. Because of the reduced load, the defined *CauseEffect* strategies caused the web servers to be scaled in twice at 2766 and 3168 seconds, and the Cassandra nodes were scaled in at 49:24, going down to only one instance of each.

From this test we can conclude that the *DependencyStrategy* can be used in applications with components that depend upon one another. We also conclude that using these elasticity requirements we can obtain similar results to those achieved with thresholds and cause effect elasticity requirements in web applications. While we believe that the use of *DependencyStrategy* would possibly be easier than *CauseEffect* in applications with many, complicated dependencies, we are not able to evaluate this with our currently application topology.

5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

The objective of this work was the study of elasticity and current elasticity solutions, as well as the creation of an elasticity controller for the orchestration software Cloudify, while also exploring new strategies for elasticity control.

In this document, we have presented a study into the state of the art of cloud computing, and most specifically, elasticity. We have analyzed the conventional definition of elasticity and the three dimension elasticity definition, as well as current elasticity solutions.

Later we have introduced our elasticity controller, CEController. CEController maintains: communication channels with Cloudify in order to obtain the necessary information about the orchestrated application, setup files to allow the developer to specify in detail elasticity requirements, and elasticity requirement classes to enforce the developer's requirements. CEController also allows its strategies to take into account the application topology, and the dimension of collected metrics. The controller was designed to allow the development of new elasticity classes. CEController also contains a new elasticity strategy, *DependenciesStrategy*, that takes dependencies into account and aims at finding problematic nodes that may cause issues to other nodes. A logging mechanism was also implemented so that the developer may review and evaluate the controller's actions.

CEController was evaluated by creating a test environment with a web server test application and a load generator. During our tests, we observe the controller's actions when enforcing elasticity requirements and the impact of those actions on the orchestrated application. We also test the use of the *DependenciesStrategy* in elasticity requirements. The results confirmed the controller implementation and the benefits obtained by the developed elasticity control strategies, and provided results obtained from the use of the *DependenciesStrategy* instead of the conventional methods.

5.2 FUTURE WORK

There are three approaches for future work regarding CEController's development:

- Integration with Cloudify.
- Further evaluation of *DependenciesStrategy*.
- Expansion of CEController.

The integration with Cloudify approach consists of adapting CEController into becoming a part of Cloudify. In the current approach, CEController connects to a currently deployed Cloudify manager and needs to be deployed separately. Under Cloudify integration, some of CEController's setup information could be directly obtained from the applications blueprint, such as the topology.

The second direction consists of further evaluation of the new elasticity control strategy, *DependenciesStrategy*. This would be done by expanding the test application, increasing the complexity and number of dependencies, in order to evaluate the *DependenciesStrategy* class compared to other elasticity control strategies. This approach would allow us to better understand which application types benefit most from the strategy and how to best employ it. Additionally, other algorithms for calculation of the most problematic dependencies could be developed and tested.

The third proposed approach is the expansion of CEController's features. As CEController's architecture allows for the extension of the elasticity requirement classes, it is possible to create of new ways to calculate monitor values and to implement new condition types into our controller. New capabilities, involving the reconfiguration of resources, instead of scaling, could also be implemented and tested.

BIBLIOGRAPHY

[1] Apache jmeter. http://jmeter.apache.org/.

[2] Apache jmeter ultimate thread group. https://jmeter-plugins.org/wiki/UltimateThreadGroup/.

[3] Autoscaling groups of instances. https://cloud.google.com/compute/docs/autoscaler/.

[4] Cloudify 3.3.1 documentation, node templates. http://docs.getcloudify.org/3.3.1/blueprints/spec-node-templates/.

[5] Cloudify 3.3.1 documentation, node types. http://docs.getcloudify.org/3.3.1/blueprints/spec-node-types/.

[6] Cloudify 3.3.1 documentation, relationships. http://docs.getcloudify.org/3.3.1/blueprints/spec-relationships/.

[7] Cloudify, built-in workflows. http://docs.getcloudify.org/3.3.1/workflows/built-in-workflows/.

[8] Cloudify diamond plugin. http://docs.getcloudify.org/3.3.1/plugins/diamond/.

[9] Cloudify rest client documentation. http://cloudify-rest-client.readthedocs.io/en/3.3/.

[10] Cloudify terminology. http://getcloudify.org/guide/3.1/reference-terminology.html#relationship-type.

[11] Cloudify workflows. http://docs.getcloudify.org/3.3.1/blueprints/spec-workflows/.

[12] Gigaspaces website. http://www.gigaspaces.com/.

[13] httperf homepage. http://www.labs.hpe.com/research/linux/httperf/.

[14] Introducing cloudify pure-play orchestration. http://getcloudify.org/cloud_orchestration_cloud_automation.html.

[15] M2m daas github page. https://github.com/tuwiendsg/DaaSM2M.

[16] Rabbitmq website. https://www.rabbitmq.com/.

[17] Riemann website. http://riemann.io/.

[18] Specweb2005 main page. https://www.spec.org/web2005/.

[19] Tpc-c main page. http://www.tpc.org/tpcc/.

[20] Tpc-w main page. http://www.tpc.org/tpcw/.

[21] What is cloudify? http://getcloudify.org/guide/3.2/what-why-cloudify.html.

[22] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: coping with skewed content popularity in mapreduce clusters. In *Proceedings of the sixth conference on Computer systems*, pages 287–300. ACM, 2011.

[23] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar. Multi-level elasticity control of cloud services. In *Service-Oriented Computing*, pages 429–436. Springer, 2013.

[24] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar. Sybl: An extensible language for controlling elasticity in cloud applications. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 112–119. IEEE, 2013.

[25] G. Copil, D. Trihinas, H.-L. Truong, D. Moldovan, G. Pallis, S. Dustdar, and M. Dikaiakos. Advise–a framework for evaluating cloud service elasticity behavior. In *Service-Oriented Computing*, pages 275–290. Springer, 2014.

[26] E. F. Coutinho, F. R. de Carvalho Sousa, P. A. L. Rego, D. G. Gomes, and J. N. de Souza. Elasticity in cloud computing: a survey. *Annales des Télécommunications*, 70(7-8):289–309, 2015.

[27] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaça. Met: workload aware elasticity for nosql. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 183–196. ACM, 2013.

[28] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong. Principles of elastic processes. *IEEE Internet Computing*, (5):66–71, 2011.

[29] G. Galante and L. C. E. D. Bona. A survey on cloud computing elasticity. In *IEEE Fifth International Conference on Utility and Cloud Computing, UCC 2012, Chicago, IL, USA, November 5-8, 2012*, pages 263–270, 2012.

[30] N. R. Herbst, S. Kounev, and R. Reussner. Elasticity in cloud computing: What it is, and what it is not.

[31] R. Jain, D. Menasce, L. W. Dowdy, V. A. Almeida, C. U. Smith, and L. G. Williams. The art of computer systems performance analysis: Techniques. 2010.

[32] I. Konstantinou, D. Tsoumakos, I. Mytilinis, and N. Koziris. Dbalancer: distributed load balancing for nosql data-stores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1037–1040. ACM, 2013.

[33] M. Li, F. Ye, M. Kim, H. Chen, and H. Lei. A scalable and elastic publish/subscribe service. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1254–1265. IEEE, 2011.

[34] H. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing, ICAC 2010, Washington, DC, USA, June 7-11, 2010*, pages 1–10, 2010.

[35] T. Lorido-Botrán, J. Miguel-Alonso, and J. A. Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12*, 2012.

[36] P. Mell and T. Grance. The nist definition of cloud computing. 2011.

[37] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar. Mela: Monitoring and analyzing elasticity of cloud services. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 80–87. IEEE, 2013.

[38] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar. On analyzing elasticity relationships of cloud services. In *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*, pages 447–454. IEEE, 2014.

[39] C. Sheridan. Elasticity in the cloud part 1: What it means. `http://www.celarcloud.eu/newsroom/elasticity-in-the-cloud-what-it-means`, 2013.

[40] D. M. Smith. Hype cycle for cloud computing, 2011. *Gartner Inc., Stamford*, page 71, 2011.

[41] H. L. Truong and S. Dustdar. Principles of software-defined elastic systems for big data analytics. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 562–567. IEEE, 2014.