AUTONOMIC APPLICATION TOPOLOGY REDISTRIBUTION

ERIC RWEMIGABO

Autonomic Computing

MSc Computing Science (Software Engineering and Distributed Systems)
Computer Science
Faculty of Science and Engineering
University of Groningen

Eric Rwemigabo: *Autonomic application topology redistribution,* MSc Computing Science (Software Engineering and Distributed Systems)

SUPERVISORS:
Vasilios Andrikopoulos
Mircea Lungu

LOCATION:
Netherlands

ABSTRACT

After years of advancements in Cloud Computing, including a significant increase in the number of Cloud service providers within a short period of time, application developers and enterprises have been left with a wide range of choice to fill their need for cloud services. Therefore, with this wide range of choices, one may find themselves making a choice of using a service that is cheaper on paper but ends up costing them more in the long run.
In this project, a system is designed and implemented to cover one of the major concerns to application owners, which is the utilisation of the resources one is paying for. Resource utilisation can be one of the biggest costs to the owner of an application given that it can cost them one of two ways; by the loss of users if the application crashes due to lack of enough resources. And secondly, if the user has to over pay for resources that aren't being used by the application. The system developed uses the MAPE-K automation strategy proposed by IBM. It monitors the user's application and then analyses the application's usage statistics through the provisioning API and thereafter predicts what it's usage is going to be in the next time window. From that prediction, it makes an adaptation plan if necessary by selecting a more suitable topology to handle the predicted load and finally redeploys the application with the more adequate topology. The final system's functionalities are first tested by means of a simple voting application and then evaluated using a larger web shop application. Both of these applications use the microservices architecture style and the results of the testing and evaluation are presented.

ACKNOWLEDGMENTS

I would like to thank my supervisor Dr. Vasilios Andrikopoulos for allowing me to take part in this research and for the valuable support, advice and general help he has provided throughout the duration of this Thesis.

CONTENTS

1 INTRODUCTION 1
- 1.1 Problem Statement 1
- 1.2 Project Details 2
- 1.3 Document analysis. 3

2 BACKGROUND AND RELATED WORK 4
- 2.1 Background 4
  - 2.1.1 Topologies 4
  - 2.1.2 Auto-scaling (Autonomous Computing) 5
  - 2.1.3 Control Loops (MAPE-K) 6
  - 2.1.4 Microservices Architecture 6
  - 2.1.5 Fuzzy logic 6
- 2.2 Related work 7

3 SPECIFICATION AND DESIGN 9
- 3.1 Requirements Specification 9
  - 3.1.1 System's Functional requirements 9
  - 3.1.2 System's Non-Functional requirements 10
  - 3.1.3 Component Requirements 10
- 3.2 Design 14
  - 3.2.1 Use Cases 14
- 3.3 System Architecture 20
  - 3.3.1 Activity Diagram 20

4 IMPLEMENTATION 23
- 4.1 Technologies and their utilisation in the system 23
  - 4.1.1 Containerisation Technology (Docker) 23
  - 4.1.2 Programming Language (Java) 24
  - 4.1.3 Database (Relational) 24
  - 4.1.4 Fuzzy Logic (jFuzzyLogic) 24
  - 4.1.5 Locust IO 25
- 4.2 Application Components 25
  - 4.2.1 Sensor 25
  - 4.2.2 Monitor 26
  - 4.2.3 Analyse 27
  - 4.2.4 Plan 29
  - 4.2.5 Execute 30

5 TESTING 32
- 5.1 Test Suite 32
  - 5.1.1 Test Application 32
  - 5.1.2 Load simulator 33
- 5.2 Sensor, Monitor and Analysis Component Testing. 34
- 5.3 Full System Testing 36
  - 5.3.1 Test Case 1: Upscaling and No system reaction Testing on the lowest topology 37

5.3.2 Test Case 2: Downscaling and Testing with a different topology 38
6 EVALUATION 40
6.1 System Requirements 40
6.2 Case Study 41
6.2.1 Cost analysis discussion. 46
6.3 System limitations. 49
7 CONCLUSION 51
7.1 Summary and Discussion 51
7.2 Future Work 52

BIBLIOGRAPHY 53
A APPENDIX 57
A.1 Documents 57

LIST OF FIGURES

Figure 3.1 System architectural layout view 20
Figure 3.2 Activity Diagram 22
Figure 4.1 Sensor Component Class Diagram 26
Figure 4.2 Monitor Component Class Diagram 27
Figure 4.3 Analyse Component Class Diagram 28
Figure 4.4 Plan Component Class Diagram 30
Figure 4.5 Execute Component Class Diagram 31
Figure 5.1 Architecture of the test application 33
Figure 5.2 Plots of the monitored CPU statistics from the containers of the example voting application. 35
Figure 5.3 Plots of the predicted CPU statistics from the containers of the example voting application. 36
Figure 5.4 Container status before the Load is applied using Locust 37
Figure 5.5 Recommended Topology output for Test Case 1 38
Figure 5.6 Recommended Topology output for Test Case 2 39
Figure 6.1 Weave Sock shop application before load is applied to the application. 43
Figure 6.2 Analysis done in test T1 on the Cases Study application. 44
Figure 6.3 The best options among the topology options available 45
Figure 6.4 The topology selected by a combination of points and a low price 46
Figure 6.5 All available topologies labelled as bad topologies hence, no options. 47
Figure 6.6 Locust charts showing the change of number of users and average response times. 48
Figure 6.7 Table of Locust statistics while running a stress test on the Case study application. 48
Figure A.1 Full system components 57

LIST OF TABLES

Table 3.1 Use Case 1 16
Table 3.2 Use Case 2 19
Table 5.1 Topology options for the testing of the application 37
Table 5.2 Test case 1 run on the application. 38
Table 5.3 Test case 2 run on the application. 39
Table 6.1 System Functional Requirements evaluation 40
Table 6.2 System Non-Functional Requirements evaluation 40
Table 6.3 Monitor Component Functional Requirements evaluation 41
Table 6.4 Analysis Component Functional Requirements evaluation 41
Table 6.5 Plan Component Functional Requirements evaluation 41
Table 6.6 Execution Component Functional Requirements evaluation 41
Table 6.7 Test case 2 run on the application. 43

LISTINGS

Listing 5.1 Example Voting App testing Script 33
Listing 5.2 Locust open ports Script 34
Listing 6.1 Viable Topology definition code 42

1 INTRODUCTION

After years of advancement in Cloud Computing, including a significant increase in the number of Cloud service providers within a short period of time, application developers and enterprises have been left with a wide range of choice to fill their need for cloud services. Therefore, with this wide range of choices, one may find themselves making a choice of using a service that is cheaper on paper but ends up costing them more in the long run.

Optimisation is the minimisation of allocated resources conditional to keeping the quality of a service at an acceptable level [33]. If the use of the resources one is paying cheaply for isn't optimised, it will end up costing them in the long run mainly in one of two ways, that is either by affecting the performance of their web application or over provisioning certain services in the application that may not require or ever make use of the resources. In the end, this unsustainable misuse of resources could cost the company or the owner of the application dearly.

1.1 PROBLEM STATEMENT

One argument for the use of Cloud Computing(CC) from an enterprise perspective is it makes it easier for enterprises to scale their services, which are increasingly reliant on accurate information according to client demand [7]. Given this aspect, we can then go further and conclude that the optimal scaling of their services would be important to them because they would want to get the best utilisation of these services in order to get the most value from their money especially, for example; A start up with a limited budget that would go with the cloud computing option as the cheaper and more flexible option. In order to achieve the optimal utilisation of resources provided by CC services, the enterprise has to have a way to monitor and analyse how their services use the allocated resources and then make changes if necessary.

In the work by Andrikopoulos [4], a CBA Lifecycle is proposed, which can be viewed as a set of MAPE-K loops [23] shifting between the defined architectural models (alpha-topologies). These shifts are caused by controllers that provide coordination across the different stages of the lifecycle. The lifecycle defined opens up an opportunity to develop a system with respect to of the proposed application lifecycle. Such a system can first seek out an optimal set of topologies that best suit certain usage scenarios and then switch among the topolo-

gies created during these periods in order to optimise the cost of the application on the cloud.

1.2 PROJECT DETAILS

Projects like [15, 16, 22] to mention but a few have been taken on to work on the optimisation of system resources with different architectural structures and using different approaches to implement their automation systems. However, many of these mentioned projects tackle one or two areas. The system developed in this project tries to build upon some of these solutions developed in order to fulfil the functionality of the system to be developed.

In this project, I design and develop a system, which uses MAPE-K loops introduced and explained in Chapter 2 to perform the optimisation of a cloud-based application with a microservice style architecture [27] that it is managing. This strategy ensures the coverage of some of the less talked about areas like the implementation of a plan component that can be an important decision in the redeployment of an application. This project will also cover the aspect of the cost of the redeployment of an application in its new topology both monetary and in terms of application performance and hence try and improve decision making in the automation/ optimisation of the application being managed.

The system developed in this project is meant to realise the proposal previously mentioned [4] through the implementation of the back end functionalities by the use of the aforementioned MAPE-K loops. MAPE-K stands for Monitor, Analyse, Plan and Execute by using Knowledge about the system's configuration and/ or including other information like the historical data. For the implementation of this system, these steps in the loop are developed as separate components, which interact with one another in order to complete the loop. This system's loop is run on top of a containerised application and uses the available APIs from the containerisation technology to monitor and record statistics of each of the services deployed in the particular containers. These statistics are the starting point at which the cloud application can be monitored and then optimised by switching between the available topologies provided by the owner of the application (System user) to provision for the services in need or to remove unnecessary resources. A switch occurs if the application topology doesn't meet the service level objectives specified by the owner of the application and it is either under using or over using the resources provided to it hence costing the application owner either financially or in terms having their application have poor performance.

1.3 DOCUMENT ANALYSIS.

In this report, the following chapter Chapter 2 starts off by providing some background knowledge required for the reader to be able to follow the project by introducing some of the fundamental concepts that make up the project and therefore, providing some insight into the research topic. After this, some of the related work in the field is presented in the following sub chapter and therefore concluding Chapter 2.

In Chapter 3, the important design documentation of the system is presented, starting with the system requirements extracted from the functionalities expected from the system. These requirements start with those extracted for the system as a whole and then go ahead and cover the individual components of the system.
After this, some of the use cases are presented and finally, the chapter is concluded by presenting the system's architecture by way of the full view of the system and an activity diagram showing how some of the components typically interact.

Chapter 4, provides the implementation details of the system by first looking at some of the technologies incorporated into the system to help it accomplish the different tasks involved in the MAPE-K process. Finally, the chapter proceeds to provide the details of each of the MAPE-K components by showing their relation to one another and provides descriptions of the interactions and functionalities of these components.

In Chapter 5 we look at the testing phase of the system, and to do so, the application to be deployed in order to test this system on is introduced with reasons behind the choice to use it to test the system. Then, the different tests cases involved in order to confirm the functionality of the components of the system are presented with their results and the particular tests run.

Chapter 6 presents the Evaluation stage of the system this will first take us back to the requirements realised for the system from which what was fulfilled and what wasn't will be presented and finally, a case study (e-commerce web application) for testing the system on a usable application, which could be a possible source of income for a company or individual shall be presented to close the chapter off.

Finally, in Chapter 7, I start off by taking a look back at where I started, review what could have been done differently and present what was unable to be accomplished for this project and why. This chapter is then closed off with a proposal of some of the future work that can be done in terms of both the system as is and in the field of Autonomous Application Topology Redistribution.

2 BACKGROUND AND RELATED WORK

In this chapter, a review of the literature relevant to the development process of this application and other relevant terms to help with further understanding of this project are to be presented. Furthermore, we take a look at some of the related work in the field and close of the chapter on that note.

2.1 BACKGROUND

This section covers some background knowledge into the work done in the research areas related to this work, including terms constantly used throughout this report, which help to drive this project.

2.1.1 *Topologies*

An application topology, as defined in [5] is a labelled graph with a set of nodes, edges, labels and, source and target functions. In terms of the application to be developed, one could look at it as the different options for the deployment of one's application based on its architecture and the resources one requires to deploy the application on. [5] introduces and explains in full detail the concept an application's topology.
A Topology can viewed as a μ-Topology, split into α-Topology, and γ-Topology, concepts explained in [5]. The focus of this section will be on making clear what the α-Topology, γ-Topology and the Viable topologies are as these very relevant concepts for this system's development.

2.1.1.1 *α-Topology and γ-Topology*

As can be seen from an example in [5], a type graph for a viable application topology can be referred to as a μ-Topology and therefore, the α-Topology is the application specific subgraph of the μ-Topology, which refers to the general application architectural setup and therefore making the γ-Topology the reusable non-application specific subgraph.

The target functionality of the system being developed is to automatically switch between viable topologies provided by the user. Therefore, with the knowledge of the α-Topology and the γ-Topology, it is possible to come up with various topologies for the application managed by the system. And finally, the knowledge of the α-Topology and γ-Topology is also vital in order to understand what the de-

veloped system should be able to expect and use in terms of input (topologies) from the user, and therefore know what kind of results the system should be to come up with to complete that topology selection and switching functionality.

2.1.1.2 *Viable Topologies*

Knowing about the α-Topology and γ-Topology, it becomes clearer as to what the term viable topologies refers to. In the context of this project, they are the different suitable topology options that are available to the automation system being developed in order for it to be able to select the best and cheapest topology option, which follows the set Service Level agreements and therefore is the best option for the particular application usage scenario. The concept of viable topologies is important for the development process of this automation project because the application that the system will be run on should have a number of viable topologies defined, which will be the topologies the system switches between to optimise the resource usage of the application. The creation of the viable topologies for the distribution of an application is out of the scope of this project however, there are a number of works available to help with the process. [8] for example provides a solution to help developers ensure portability of their applications and [16] presents a solution, which enables the automatic derivation of provisioning plans from the needs of the user among other papers in the field.

2.1.2 *Auto-scaling (Autonomous Computing)*

Different researchers have delved into a number of projects using different strategies to try and solve a variety of problems in the field of automation and autoscaling of application resources in particular. These different projects range from: *The monitoring of different metrics of the applications in question*, for example, looking into whether it's more advisable to monitor the lower level metrics like the memory, CPU usage or even the network statistics or the higher level metrics like the response times of the system being optimised. *To the type of analysis strategies used to determine whether to scale up or down* for example the use of a predictive methods (see [26]), reactive or rule based approaches (see [1]) or hybrid methods using both reactive and predictive approaches. These different projects and automation strategies are discussed in a survey [32], which helped provide an insight into the different options available to help with the completion of the particular components of the developed system. The strategies selected for this project will be looked at in a later chapter.

2.1.3 *Control Loops (MAPE-K)*

Throughout the various papers discussed in the survey [32] in the previous section, it is noticeable that the MAPE control loop strategy is currently a commonly used automation strategy, which involves the use four main procedures that make up this strategy. These are Monitor, Analyse, Plan and Execute. The MAPE automation strategy [23] used for this project is complimented with a Knowledge base, which all the MAPE components interact with, in order for the system to make informed optimisation decisions. These MAPE control loops and partly the knowledge base are the core driver of the system being developed as they are what makes up the functionality of the system, as will seen in a later chapter.

2.1.4 *Microservices Architecture*

Microservices as discussed by Martin Fowler. [27] describes an architecture style of building systems into a suite of smaller services each running on their own. These may be written in different programming languages and use different data storages. The microservice architecture is the architecture style focus for the applications this project's system is developed for and will be able to perform its optimisation services on. The isolated services in this architecture style make it possible for the developed system to be able to individually run its loops on each service and therefore in the end perform the full assessment of the whole system, therefore performing optimisation more efficiently and its because of this characteristic that the microservice style architecture was selected for this project. Additionally, the microservice style architecture is also well supported by most of the containerisation engines. This is an advantage in the case of this project since the popularity of containers among cloud developers recently has increased and therefore we were able to easily find a number of test applications that have the microservice architecture style and were deployed in a containerisation technology most significantly, the one selected for this project (Docker), which is introduced and shown later in a following chapter.

2.1.5 *Fuzzy logic*

Fuzzy logic is a concept that has been used to help machines loosely translate human terms like high and low into concrete values that they can use to assess a certain situation. The paper [39] points out that the fuzzy logic concept stems from the Computing with Words methodology, which all started from [38]. Fuzzy logic helps in simplifying the decision process of complex systems and therefore, it presented as a compelling option for the implementation of part of

the analysis phase of the MAPE loop, where it would help decide whether the application being managed by the autonomous system to be developed required a switch in topology or not. The decision to use fuzzy logic was further enforced by [15], where they used fuzzy logic to help optimise the scaling mechanism of a cloud service.

2.2 RELATED WORK

As mentioned previously, a number of projects have been undertaken in order to tackle various problems in the field of self-adaptive systems. A lot of the projects in the field of automation have more specifically covered particular domains similar to robotics and smart house systems. Even though these also have helped provide me some understanding from the work done there, my focus is in the cloud computing domain where there are a number of techniques that have been employed to perform the task of automation among the various projects that have been undertaken. Many of these works look to tackle, solve or answer particular questions in the field of cloud computing. These different solutions for the particular problems proved to be an important resource because by looking through and combining these works, I was able to tailor a solution for the various components in the MAPE-K loop.

Some projects looked into the different ways to most effectively estimate the required resources to be made available. For example, [1, 2, 10] all implement Rule-based approaches, which is an approach of resource estimation where; if, and else rules are used in order to trigger a particular resource provisioning function. Additionally, work done in [14, 17, 37] provides predictive solutions to the resource estimation problem done in the analysis component of the system to be developed. They perform the predictions by monitoring and using either lower or higher level metrics of the managed application of which the lower level metrics would represent the CPU, network memory and so on whereas the higher level metrics would represent a metric like the response time of the application [32]. This range of solutions, which includes others helped with the decision to use a predictive (*regression*) solution for the analysis component of the system with a combination of a fuzzy logic solution introduced in the previous section and used by [15].

One of the major problems associated with the auto scaling of an application is the problem of oscillation, where the auto scaler in this case would switch to a new viable topology and within a short time switch that topology again [32]. The solution of the use of dynamic parameters as seen for example in [24] helped provide inspiration to use a solution for the planner where the switch of a topology is only authorised when the time after the last switch is double the time it takes for the change (redeployment) to be executed and hence miti-

gating the oscillation problem. Finally, the work done in [5] provides insight into the selection of a topology among the available alternative topologies provided and with this information among others, the implementation of the planning component was made possible.

3 SPECIFICATION AND DESIGN

In this chapter, I present the requirements specification and design of the project in two sections. In the first section, some of the most important functional requirements of the full system are presented with a few non-functional requirements and then, the functional requirements of the individual components of the system are presented. After this, some of the use cases of the system as a whole are presented in the following section including some design patterns. Finally, the logical view of the system is presented under two view points, one showing the activities performed by the components and the other providing a higher level view of the whole system.

3.1 REQUIREMENTS SPECIFICATION

Some of the requirements presented in [19] provided a template upon which to start writing my system's main functional requirements, as they cover the functional and non-functional aspects to enable the dynamic (re-)distribution of applications in the cloud, which is the aim of my system. However, during the development of the system, a few other functional requirements were realised and added to the list for both the system and its individual components.

3.1.1 *System's Functional requirements*

FR1 The System should be able to connect to or interact with a containerisation engine.

FR2 The System should have access to the containerisation engine's API.

FR3 The System should sort the services of the application being managed into a list.

FR4 The System should have access to the alternative viable topologies.

FR5 The System should be able to Monitor data from the managed Application.

FR6 The System should be able to Analyse the data from the managed Application

FR7 The System should be able to make a Plan using the data from the analysis of the managed Application.

FR8 The System should be able to Execute the plan made for the managed Application

FR9 The System should create Monitors for each of the services in the managed Application.

FR10 The System should create Analysers for each of the services of the managed Application.

FR11 The System should have access to the Service Level Agreements (SLAs) or Service Level Objectives (SLOs) set by the user.

3.1.2 *System's Non-Functional requirements*

NFR1 The System shall monitor statistics in real time.

NFR2 The System shall be able to make a topology change decision before the time window is finished.

NFR3 The System shall have the capacity to store data for at least a day's recorded metrics.

NFR4 The System shall conform to the security parameters set by the application it is managing.

NFR5 The System shall be keep the data recorded about other application usage private.

3.1.3 *Component Requirements*

3.1.3.1 ***Monitor Component***

FR1.1 The Monitor component should create sensors for each of the containers of the service it is monitoring.

FR1.2 The Monitor component should log statistics for each of the containers of the service being monitored.

FR1.3 The Monitor component should set the sensors to monitor different application metrics.

FR1.4 The Sensor(s) should check that the metric values recorded are within the SLOs.

FR1.5 The Monitor component should notify the analysis whenever a new statistic is recorded.

FR1.6 The Monitor component should notify the analysis whenever a metric out of scope of the set SLA/ SLO is recorded by the sensor(s).

FR1.7 The Monitor component should save the monitored statistics to the knowledge base.

FR1.8 The Monitor component should continuously monitor the service until the point that it is no longer available.

3.1.3.2 *Analysis Component*

FR2.1 The Analysis Component should receive notifications from the Monitor component of new statistics.

FR2.2 The Analysis Component should perform data analysis in time windows defined by the user.

FR2.3 The Analysis Component should retrieve a batch of statistics in a given time window for analysis from the knowledge base.

FR2.4 The Analysis Component should perform a set of analysis tactics on the statistics gathered from the knowledge base.

FR2.5 The Analysis Component should provide a visual representation of the analysed data.

FR2.6 The Analysis Component should perform a prediction of data points for the next time window.

FR2.7 The Analysis Component should make an estimation of the resources that need to be added, removed or kept as is for each of the services.

FR2.8 The Analysis Component should make an aggregation of the results from the various analysers.

FR2.9 The Analysis Component should create a topology suggestion using the estimated resources based on the topology running and the aggregated results.

FR2.10 The Analysis Component should notify the plan component of the new topology suggestion.

FR2.11 The Analysis Component should save the results from the analysis to the knowledge base.

FR2.12 The Analysis Component should save the topology suggestion to the knowledge base.

3.1.3.3 *Plan Component*

FR3.1 The Plan Component should be able to receive notifications from the Analysis Component.

FR3.2 The Plan Component should have access to the list of alternative viable application topologies.

FR3.3 The Plan Component should have access to the recommendation(Adaptation request) from the Analysis Component.

FR3.4 The Plan Component should ensure enough time (set by user) has passed since last topology (re-)distribution.

FR3.5 The Plan Component should retrieve the latest suggested topology by the Analysis component.

FR3.6 The Plan Component should make a comparison between the suggested topology from the Analysis Component to the alternative viable application topologies.

FR3.7 The Plan Component should retrieve the alternative viable topology closest in similarity to the suggested topology.

FR3.8 The Plan Component should have access to a record of the currently running or deployed topology.

FR3.9 The Plan Component should notify the Execution component when a change is confirmed.

FR3.10 The Plan Component should store the new topology for the (re-)distribution to the Knowledge Base

3.1.3.4 *Execution Component*

FR4.1 The Execution component should be able to receive notifications from the Plan component.

FR4.2 The Execution component should create an Effector, whose job it is to perform the execution actions.

FR4.3 The Execution component should have access to the alternative topologies available.

FR4.4 The Execution component should have access to the containerisation API.

FR4.5 The Execution component should be able to run the commands necessary to make the topology change requested by the plan component.

FR4.6 The Execution component should confirm when the change has been made successfully

FR4.7 The Execution component should record or save the time of the latest change of topology.

FR4.8 The Execution component should record the time it took to make the change and update it in the knowledge base

FR4.9 The Execution component should be able to reset the MAPE-K loop to start working on the newly redistributed topology.

FR4.10 The Execution component should update the currently running topology.

FR4.11 The Execution component should save the new topology change to the knowledge base.

3.1.3.5 *Knowledge Base Component*

FR5.1 The Knowledge Base Component should provide an access point for the various components to create the necessary data.

FR5.2 The Knowledge Base Component should provide an access point for the various components to update the necessary data.

FR5.3 The Knowledge Base Component should provide an access point for the various components to retrieve the necessary data.

FR5.4 The Knowledge Base Component should provide an access point for the various components to delete data.

FR5.5 The Knowledge Base Component should contain a record of the alternative viable topologies.

FR5.6 The Knowledge Base Component should contain a record of the statistics recorded by the monitor component.

FR5.7 The Knowledge Base Component should contain a record of the data output by the analysis component.

FR5.8 The Knowledge Base Component should contain a record of the topology picked by the plan component to be deployed.

FR5.9 The Knowledge Base Component should contain a record of the successful topology changes made by the execution component including the time.

FR5.10 The Knowledge Base Component should contain a record of the service level objectives set by the system user.

FR5.11 The Knowledge Base Component should contain a record of the other user preferences like the time windows for the analysis e.t.c.

FR5.12 The Knowledge Base Component should contain a record of the history of the performance of the various topologies that have been run before.

3.2 DESIGN

In this Section, I present 2 use cases, which are cases under which the system is expected to behave differently. In the first case, the system records normal workloads under which the managed application should not have any problems dealing with and therefore, there is no need for a change. In the second use case, the system predicts stress on some of the services of the managed application or under use of the resources for the next time window and therefore requests an appropriate change to be made in order for the application to utilise it's available resources optimally.

3.2.1 *Use Cases*

Use Case 1	**Normal Managed Application load**
Goal:	This use case depicts a situation where the managed application load is predicted to be within the Service Level Objectives defined by the application owner. With this case, the system is expected to keep reporting the normal application usage and not make any changes to the topology
Pre-Condition:	The managed application is running, the system is deployed on top of it and is monitoring the usage statistics of the application
Post-Condition:	The System has run an analysis of the statistics and detected that no changes are required and hence, there are no changes made and the monitoring and analysis processes continue.
Primary Actor:	Managed application

Assumptions:

- Alternative Viable topologies are made available by the application owner.
- The Service Level Objectives are defined by the application owner.
- The application services are running normally
- The application traffic is predicted to be on par with the resources made available to the application.
- The User has set other parameters like the preferred time windows for analysis.

Main Success Scenario:

1. The system deploys its MAPE-K control loop on the application services.
2. The system gains access to the containerisation API.
3. The monitor functionality records the statistics received from the containerisation API and reports to the Analysis.
4. The analysis component makes a record of the time of the first notification/ statistic from the monitor component.
5. The analysis component compares the current system time with the recorded time whenever it is notified.
6. After the correct (set by the user) time window has passed, the analysis component makes a record of that last timestamp and performs an analysis action on the data of the time window.
7. The analysis component makes a prediction of the expected workload in the next time window and suggests the number of containers required to be added or removed
8. The number to be added or removed is 0 and therefore no further actions are required.
9. The analysis component continues to compare the current time to the last recorded timestamp for the next analysis window

---

Extensions (Temporary Spike):

3a The Monitor functionality reports high metrics and shortens the time window to the analysis.

6a Analysis is performed on the shorter time window and predicts that increased use was a temporary spike so, no additional resources should be provisioned.

Table 3.1: Use Case 1

Use Case 2	**High/ Low Managed Application load**
Goal:	This use case depicts a situation where the managed application load is predicted to be outside the Service Level Objectives defined by the application owner. With this case, the system is expected to make a change in the topology being used by the managed application and therefore get the application back within the Service Level Objectives set by the user.
Pre-Condition:	The managed application is running, the system is deployed on top of it and is monitoring the usage statistics of the application.
Post-Condition:	The System has run an analysis of the statistics and predicted values outside the SLOs and therefore a change in topology is performed and a new topology is being used by the managed application.
Primary Actor:	Managed application
Assumptions:	• Alternative Viable topologies are made available by the application owner. • The Service Level Objectives are defined by the application owner. • The application services are running normally • The application traffic is predicted to be outside the set SLOs within the next time window. • The User has set other parameters like the preferred time windows for analysis.

Main Success Scenario:

1. The system deploys its MAPE-K control loop on the application services.
2. The system gains access to the containerisation API.
3. The monitor functionality records the statistics received from the containerisation API and reports to the Analysis.
4. The analysis component makes a record of the time of the first notification/ statistic from the monitor component.
5. The analysis component compares the current system time with the recorded time every time it is notified.
6. After the correct (set by the user) time window has passed, the analysis component makes a record of that last timestamp and performs an analysis action on the data of the time window.
7. The analysis component makes a prediction of the expected workload in the next time window and suggests the number of containers required to be added or removed
8. The number to be added or removed is greater or less than 0 and therefore a change in topology is required.
9. The Analysis component makes an addition and/ or subtraction to the resources available in the current topology therefore coming up with a recommendation of a topology like structure for the required resources
10. The recommendation is saved and the Plan component is notified.
11. The Plan component checks the time of the last Topology change.
12. If enough time has passed since the last Topology change, the Plan component accesses the alternative topologies in the knowledge base and selects the one closest to the suggested topology
13. The Plan component notifies the execution component of the required change and saves the suggestion.
14. The Execution component retrieves the selected topology and runs the required commands to redeploy the application in the new topology.
15. The Execution component updates the time to redeploy the application.
16. The execution component updates the last successful redeployment time and resets the system to run on the new topology.
17. The loop restarts

Extensions (Reactive):

3a The Monitor functionality reports high metrics and shortens the time window to the analysis.

6a Analysis is performed on the shorter more urgent time window.

7a A prediction is made and the prediction is outside the SLO.

10a Plan is notified with an urgency/ priority recommendation.

11a No time check is performed.

Extensions (Oscillation Mitigation):

11a Not enough time passed triggers wait action (Depending on how close to breaking point system is predicted to be).

Table 3.2: Use Case 2

3.3 SYSTEM ARCHITECTURE

Figure 3.1 shows the architectural layout of the system.

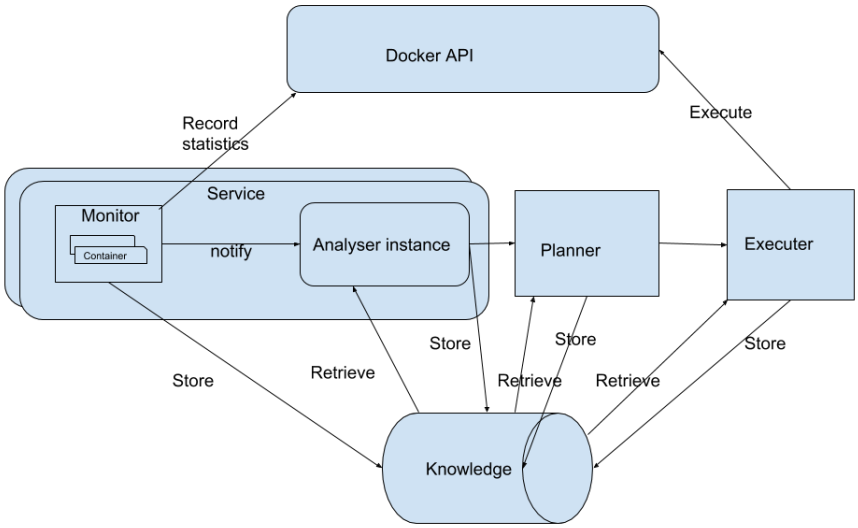


Figure 3.1: System architectural layout view

3.3.1 *Activity Diagram*

In the Figure 3.2, the activities required to perform the functionalities in the particular component instances are presented. The monitor and analysis components have multiple instances and are managed by the monitor manager or the Analyser manager and therefore these managers ensure the aggregation of data for the entire application topology being managed. The details of the manager and individual components shall be discussed in Chapter 4. At the start of the system, each of these components are created by their managers and their activities after that are as follows.

3.3.1.1 *Monitor*

A monitor component is created for each container in the particular service being monitored. This component starts off by creating sensors for each of the metrics it is set to monitor from a container and the monitoring task begins. Every 2 seconds, the Sensors send their registered metric to the monitor component and these are metrics are collected to form a statistic. A sensor can also report a metric which is outside the SLOs and this will cause the component to create a new more urgent statistic. After the statistic is created it is stored to the

Knowledge base and the Analyser is notified. However, this functionality is to be turned off in the current state of the system given that the focus is currently on a predictive approach rather than a reactive one.

3.3.1.2 *Analyse*

An analyser is also created for each of the containers of a service therefore, the analysers and monitor components have a one to one relationship. When an analyser is created, it waits for its first notification from the monitor it has a relation with and when it receives it, the analyser takes note of the time in the first statistic the monitor saved. After this point, the time window is checked every time a notification is received and when enough time has passed, the components retrieves the statistics for that window, sets the last timestamp as the latest time and analyses the data. On the other hand, an analysis is also performed on double time windows to get a clearer analysis with more data. After the analysis is done, the manager aggregates the data and notifies the plan if necessary of a necessary change.

3.3.1.3 *Plan*

The job of the plan component is to select a topology. Once it is notified by the analysis component, it uses the request from the analysis to select the topologies better suited to that suggested request and thereafter, it performs a cost/ price comparison on the relevant viable topologies and selects the cheapest option. Therefore, ensuring the solution doesn't under/ over provision and the solution is at a good cost. It then saves its choice and notifies the execution component

3.3.1.4 *Execute*

The execution component possesses the simplest job and that is to get the selected solution/ topology and run its configuration script in order to redeploy the application in the selected topology. After that, it stores the time taken to redeploy and finally restarts the MAPE loop on the new topology.

Figure 3.2 shows the flow of events describe in the above sections.

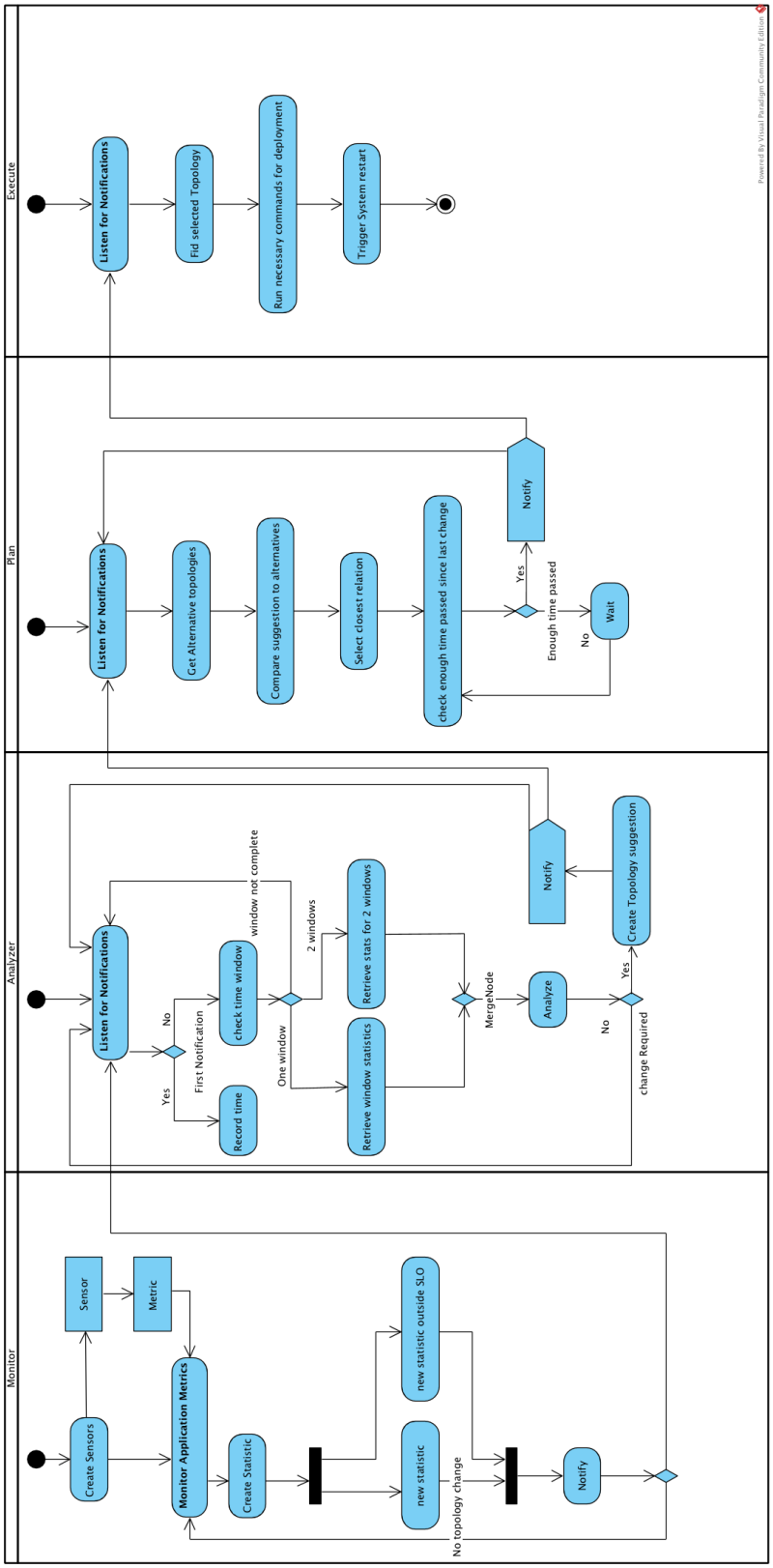


Figure 3.2: Activity Diagram

4 IMPLEMENTATION

In this chapter, the details of the implementation of the system are presented including a description of the individual components of the system and how they interact, create and share data. Some of the technologies that were helpful and necessary in the implementation of the system's components including their use to the system are also presented below.
The code for this project's implementation can be found in the Github repository [18]

4.1 TECHNOLOGIES AND THEIR UTILISATION IN THE SYSTEM

4.1.1 *Containerisation Technology (Docker)*

Containerisation[31] is a lightweight virtualisation technique, and virtualisation is necessary for the deployment of applications on the cloud. Therefore, when starting the development of this system, I had to select a containerisation engine where I could access the necessary metrics and upon which to deploy the test cloud applications given that the aim of the autonomous system would be to manage the resources of an application on the cloud so, I had to simulate this environment. There are a few containerisation technologies like rkt[34] developed by CoreOS, Solaris Containers[29] developed by Oracle and so on. However, given that I have some past experience with the Docker Engine, and additionally, the significant number of applications developed for and deployed with the docker engine, the decision of what containerisation technology to use went to Docker Engine 1.13.1.

4.1.1.1 *Docker API*

While I had used the Docker engine before for the deployment of a simple application, I had not used it used it extensively whereby I would require knowledge about the Application Programming Interface (API). Therefore, I took a look into this and with the aid of [11], I found that it has a Software Development Kit(SDK) for Python and Go, and number of unofficial libraries for a number of programming languages, which opened up my options on the programming language I could use and enforced my confidence in the containerisation selection that had been made.

4.1.1.2 *Docker Compose*

One additional advantage of the docker engine is its easy compatibility with the microservice architecture[27]. Docker Compose[12], a tool for running applications with multiple containers provides this feature when defining the different components in the docker-compose.yml file during the setup of the application. Given that, a number of microservice applications have been developed and deployed on docker, which provided me with a number of options both simple and complex for the testing phase of my system's components.

4.1.2 *Programming Language (Java)*

After deciding on the containerisation technology to use, it was time to move on to the programming language and my extensive knowledge in the Java as compared to other languages was a starting point. Next on the checklist was the compatibility with Docker and as seen from [11], there is an official library for Java, which added to my confidence in the choice. Finally, the extensive number of options available to me in terms of what database to use also pushed my decision towards the java programming language.

4.1.3 *Database (Relational)*

In terms of a database, the first options to consider was whether to use a Relational database or a NoSQL database and I was drawn towards the use of a relational database because of more previous experience with relational databases and they would properly structure and accommodate most of the data that would be passed through and processed by the system. After this, I had to select an option among the various relational databases. Given that the relational databases are quite similar, I selected one I had most recently used, which is the PostgreSQL database for data storage. PostgreSQL 9.4.19 [30] is an open source relational database with good performance and is easy to use.

4.1.4 *Fuzzy Logic (jFuzzyLogic)*

jFuzzyLogic is an open source Java Library for Fuzzy Logic, which with the help of the standard for fuzzy control programming in part 7 of the IEC 6113 published by International Electrotechnical Commission, I was able to learn some the basics of the language to use with jFuzzyLogic [9]. The Library was created to aid in the programming of Fuzzy Logic control systems using the standard Fuzzy Control language defined in the IEC 61131 and using [9], I was able to get some understanding of the library and some useful insight on how to use

it in my project's development and specifically for the analysis component.

4.1.5 *Locust IO*

[25] is a load testing tool that I selected to simulate loads on the applications that were to be managed by the developed system. This testing tool is easy to use and all I had to do was to write a short test script in python to connect to the web application and run the necessary tests. The tool provides its API documentation, which was helpful to use when learning to write the scripts I used to test the applications. Additionally, the case study application I chose to use had load tests written for it already, which meant I had 1 less script to write when performing my testing. Locust also provided a web interface that shows different metrics for the application being tested and with this information being made available, I was confident I would be able to run the appropriate tests on the test applications, which sealed the choice to use Locust IO.

4.2 APPLICATION COMPONENTS

In this section, I discuss the individual system components of the system. Using [21] as a resource during the planning development phase of this system, I was able to implement the components even if some of the behaviour templates were not applicable with this system's domain.
Figure A.1 combines all the components discussed below and others classes excluded this section that help provide the functionality of the system.

4.2.1 *Sensor*

The Sensor is one of the most important parts of the system. The Monitor component depends on it for the metrics that are used throughout the system. Through the sensor manager, the monitor instance creates sensor threads for each of the metrics that are to be monitored. Sensors are connected to only one container and they report on the metric that they are assigned by the monitor component. The sensors are observables in the observer pattern [20], which is also used by most of the other components in the system. They notify the monitor component every 2 seconds with a new recorded metric for that time.
Figure 4.1 shows the relations between the sensor manager, which is a singleton, the sensors and the monitor component.

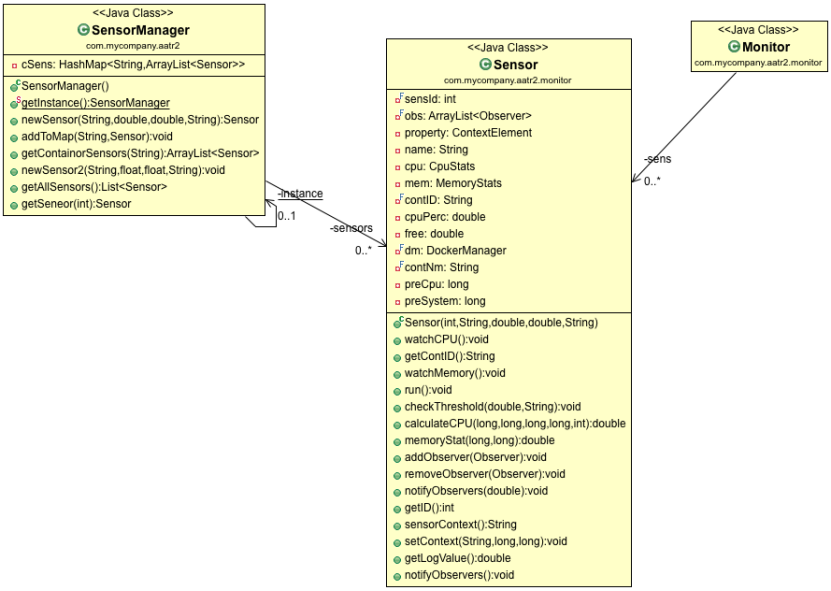


Figure 4.1: Sensor Component Class Diagram

4.2.2 *Monitor*

A monitor component instance is created through its Singleton Manager (Monitor Manager) and thereafter, the monitor instance creates its sensors. There can be multiple monitor instances each monitoring a service (Cluster class), which they are assigned to upon creation. The Docker Manager Singleton is what connects to the docker API, makes note of the running containers and creates services by the use of the names of the images used to create these containers. Once these are sorted then the individual components are launched. The Monitor class instance is both an observer and an observable. It observes the sensors on each of the containers, creates a statistic using the metrics recorded for the container.
Each container in the service being monitored by a monitor instance is assigned a statistic log upon the instance's creation and thereafter, the statistics recorded for that container are saved in that particular log. Every after the Monitor instance finishes creating a statistic, it notifies the Analyser that is registered to the same service it is monitoring.

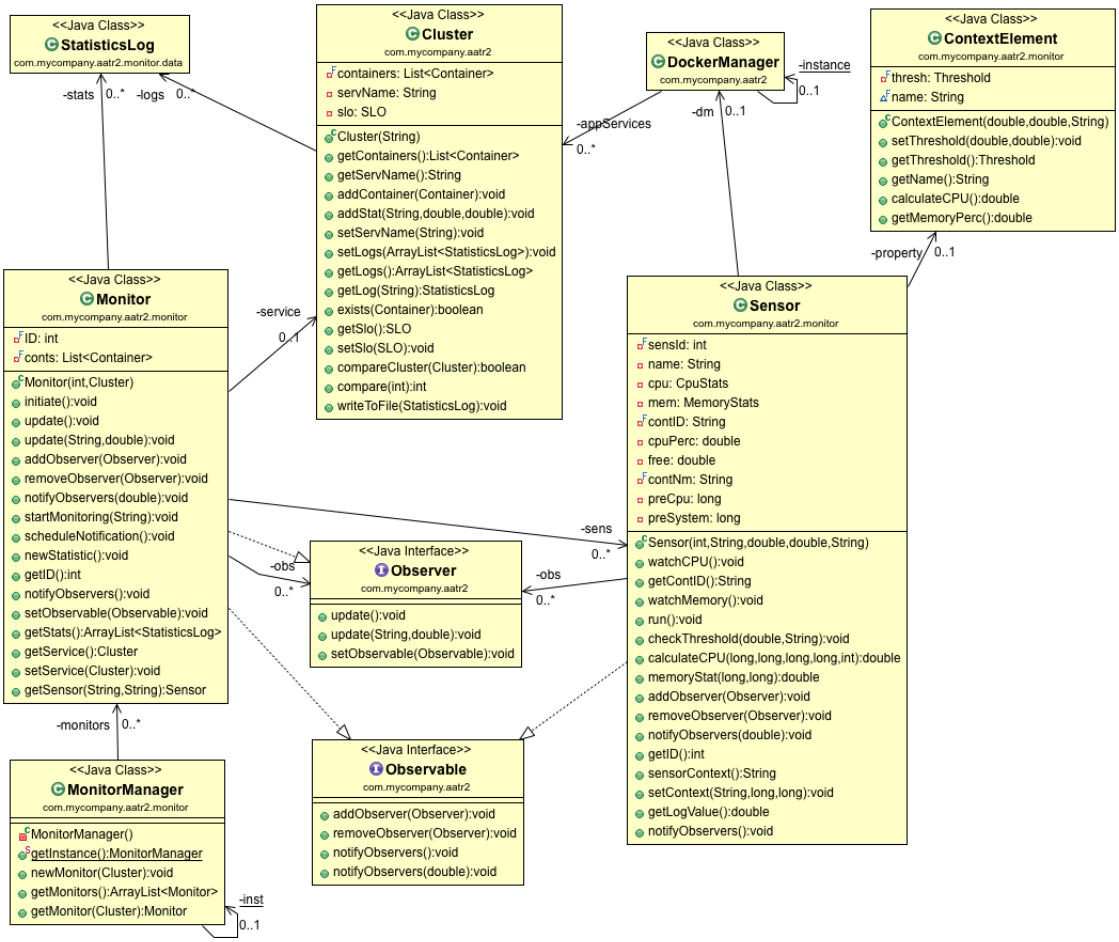


Figure 4.2: Monitor Component Class Diagram

4.2.3 *Analyse*

The Analyser component is also assigned to analyse the statistics of a single service. The Analyser component instances are created through the Analyse Manager Singleton, which keeps a record of each of the analysers. Once an analyser instance receives a notification from the monitor instance, it retrieves the latest statistic log and then checks the time on that statistic in comparison with the system time. If this is the first analysis, then it starts the analysis but if not, then it has to wait for enough time to pass. Once this is true, the component collects the data for the last time window, and calls the *runFullDataAnalysis()* method. This collects all the data received, plots the data from the previous window and then calls the *makePrediction()* method. This also then plots the predicted usage points for every 5 - 10 seconds for the whole of the next time window. Additionally, it collects the data points in a list and gets their average and returns that value.
This value is then passed to the *diagnose()* method, which uses the fuzzy logic component (jFuzzy Logic) to perform an estimation of the resources required to fulfil this usage requirement and returns

this and uses it to create a symptom. A symptom in this case is simply a store of the number of containers required to fulfil the next window's usage requirements. If there is more than one container in the service, the result of this analysis is stored in a list and when all the symptoms of every container are available, these results are averaged and therefore a single symptom is produced.
The analyser component is also both an observer and an observable but however, its observer is the Analyse Manager. The Analyse Manager collects the symptoms of each of the Analyser instances and creates a new system state, which represents the number of containers to be added or removed in order to optimise the managed application. These relationships can be seen in Figure 4.3. The Analyse Manager is the observable of the Plan manager and so, when a new system state is created, it sends the notification.

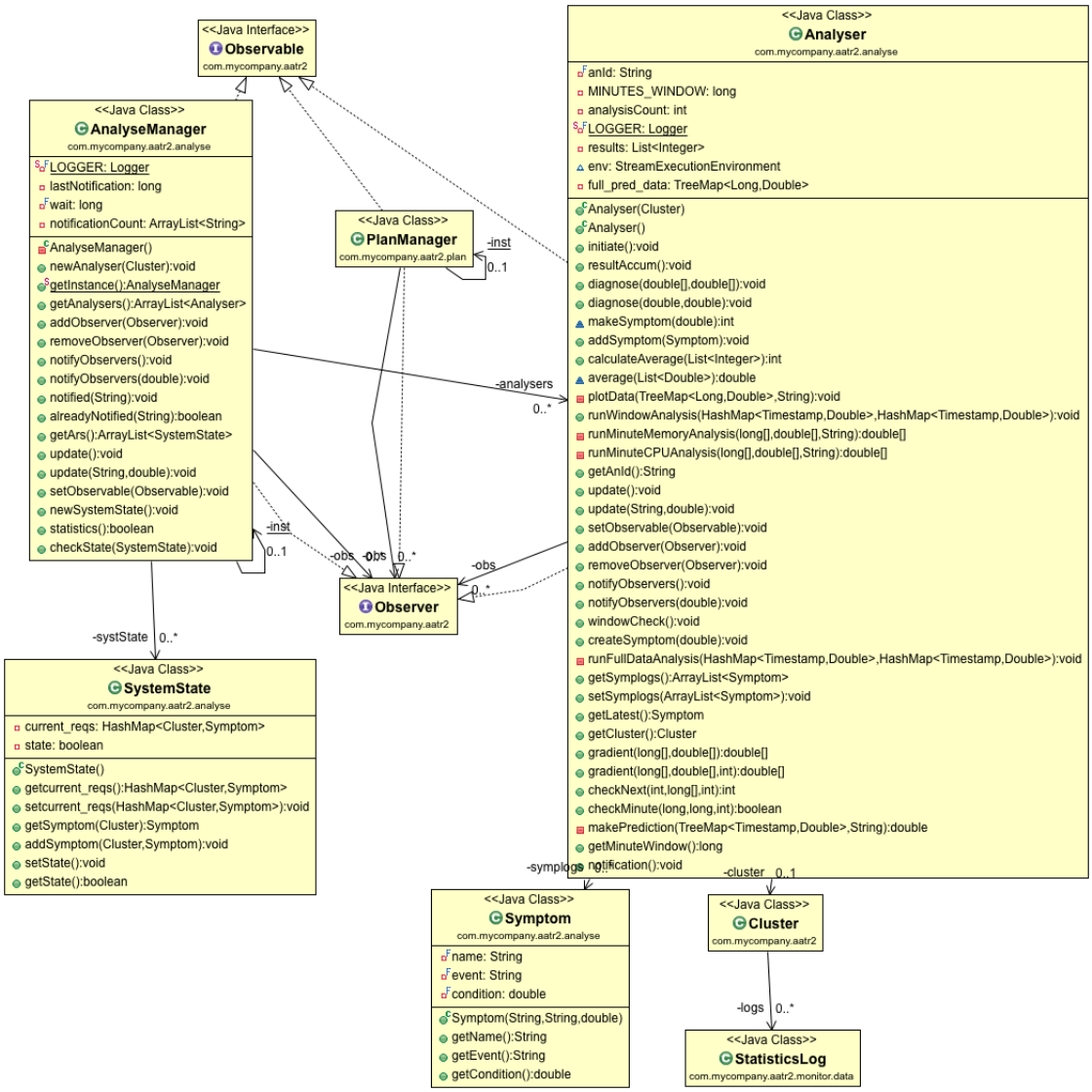


Figure 4.3: Analyse Component Class Diagram

4.2.4 *Plan*

The Plan is just one singleton component created at the start of the system. It is responsible for making the final decision of the topology to be selected for the application's redeployment. The process starts off by the receipt of a notification from the Analyse manager. Once this happens, the plan manager gets the system state and through uses it to create a topology recommendation. From this ideal topology, the plan is able to compare it to the viable topologies that it has access to.
While comparing the viable topologies to the ideal topology it created, it awards points to the topologies according to how close the relation in terms of the number of containers they have. Additionally, while adding these points to the topologies, the method also eliminates the viable topologies where the number of containers is less than the ones in the ideal topology. Once this is done, the topologies with scores are left in the list and using this and their prices, the plan manager performs a price comparison among these remaining topologies and finally selects the most suitable and cheap option. After that, the plan manager notifies the Execution component, which is its observer.

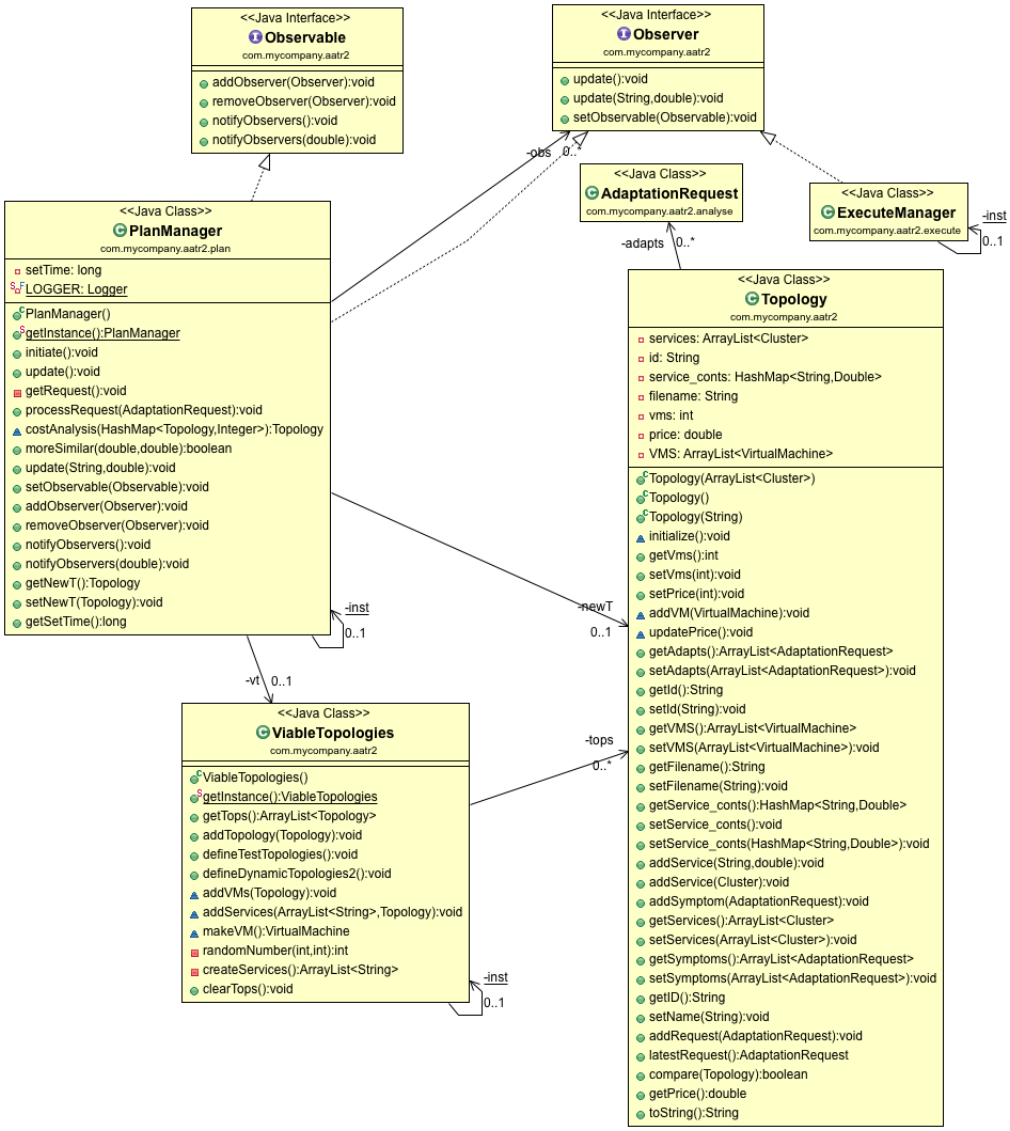


Figure 4.4: Plan Component Class Diagram

4.2.5 *Execute*

The Execution component receives a notification from the plan manager after which it retrieves the viable topology to be executed. Once it has the topology's file information, it used this to find the config files of the topology and runs the scripts for the redeployment of the application. Once the redeployment is completed, it updates the time of redeployment and the loop starts again.

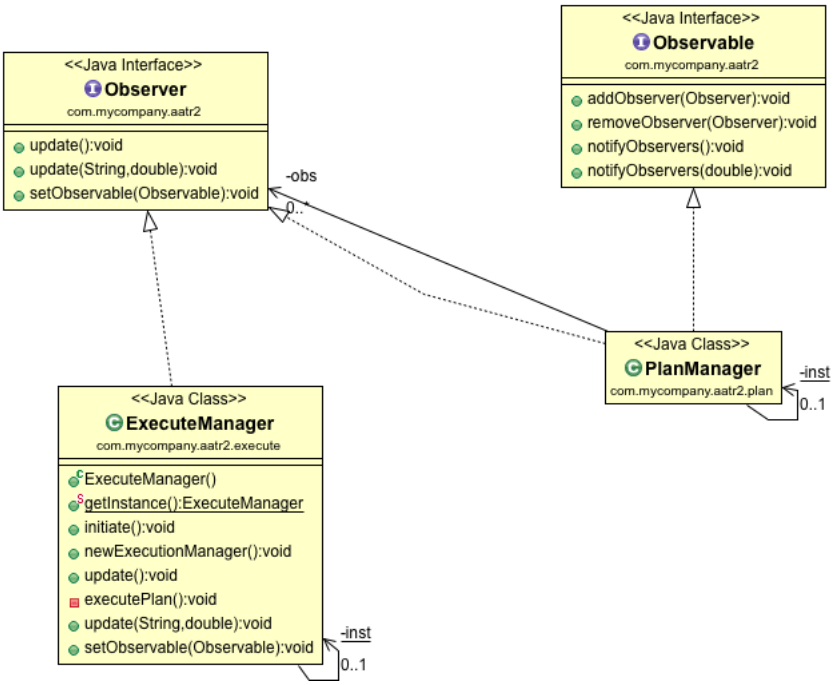


Figure 4.5: Execute Component Class Diagram

5 TESTING

During the development of the system, there were a battery of unit tests run on each of the individual components to ensure their functionality and furthermore their interaction with each other when integrated in order to ensure that they work together. However, this chapter presents the more important tests run on the more complete versions of the system. After implementing three of the major components of the system, that is the Sensor, Monitor and the Analysis components, the more important phase of testing begun and in the following sections, I present the results of most of these tests and finally, an evaluation of the results of the System is done in the following chapter.

5.1 TEST SUITE

5.1.1 *Test Application*

The testing of the system was done using the following application assembled from Github.

example-voting-app:
After the initial testing of my system's Monitor and Analysis components and confirming that they performed the basic functionality, I needed to test the more complex features of the Analysis component, which were the prediction and recommendation functionalities. To test these, I needed a simpler more basic application where I would be able to quickly write a testing script for the load simulator application I was going to use so, I decided to use the Example Voting App [13]. It is a simple voting application where a user either votes for cats or dogs.
The services in this Application, as shown in Figure 5.1, include the *voting-app* service, where the users cast their votes, *result-app* service, which is where the user views the vote tally percentage, which is retrieved from the database, *redis* service, a queue to handle the votes coming it, *worker* service to process the voted and send them to the final database service *db* service.

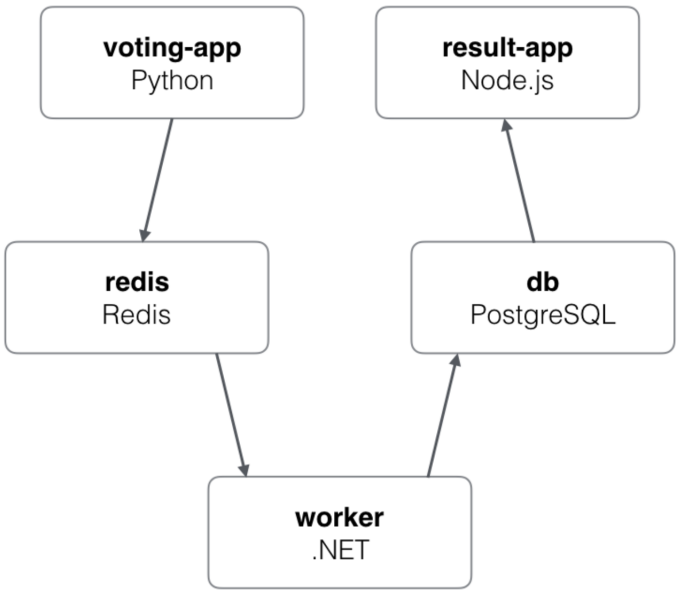


Figure 5.1: Architecture of the test application

5.1.2 *Load simulator*

Locust IO, the Load testing application introduced in Chapter 4 was used for to load test the test applications. I wrote the test scripts for the test applications for example the short test script from the load testing done on the Example Voting App is presented in Listing 5.1. By running this script, the simulated user initially randomly performs a vote and then randomly changes the vote every time Locust sends a request by the simulated user. Listing 5.2 provides the ports where the simulated users can connect.

```
from locust import TaskSet, task
import random
class MyTasks(TaskSet):
    # vote = null
    # vote function
    def vote(self, vt):
        self.client.post("/", {'vote': vt})

    # initial random vote between cats or dogs
    @task(2)
    def votecat(self):
        self.vote("a")

    # change vote task
    @task(3)
    def votedg(self):
        self.vote("b")
```

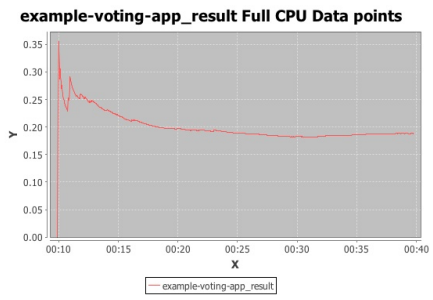
Listing 5.1: Example Voting App testing Script

```
from locust import HttpLocust
from MyTaskSet import MyTasks
# from MyTaskSet import MyServicesTasks


class MyLocust(HttpLocust):
    task_set = MyTasks
    min_wait = 10
    max_wait = 100
    host = 'http://localhost:5000'


# class MyServicesLocust(HttpLocust):
#     task_set = MyServicesTasks
#     min_wait = 10
#     max_wait = 100
#     host = 'http://localhost:5001'
```

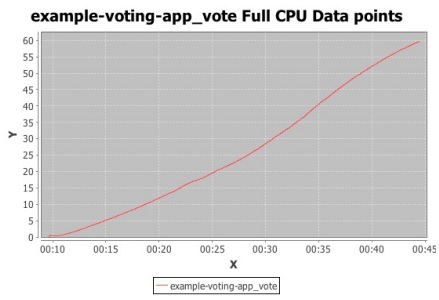
Listing 5.2: Locust open ports Script

5.2 SENSOR, MONITOR AND ANALYSIS COMPONENT TESTING.

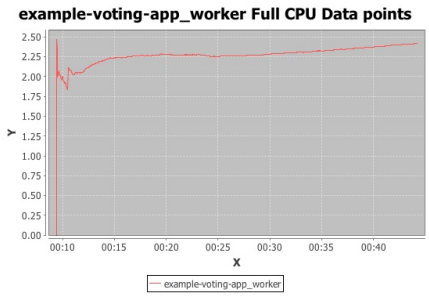
As seen in the Component diagrams in Chapter 4, these components work closely together and in order to fully test one of them, I had to have all of them working. For the testing phase of these components, I started out by testing the Sensor connection to the Docker API to see whether the statistics metrics were being accurately monitored by the sensor and to confirm, I compared them to the container statistics produced when running the command *docker stats* in the terminal. After confirmation of the accuracy of the sensor statistics, I moved on to the monitoring component. Since there is a monitor instance for each of the services being monitored, I decided to plot graphs for each of the statistics for each of the metrics. Therefore creating 2 plots per container per service. Figure 5.2 shows a sample of the statistics recorded from testing the example voting application.



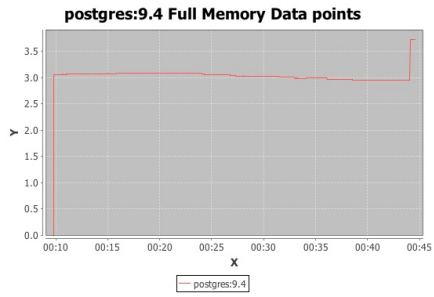
(a) Result service container CPU stats



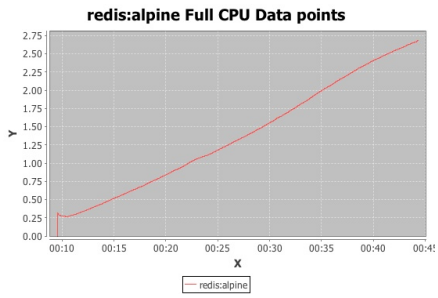
(b) Vote service container CPU stats



(c) Worker service container CPU stats



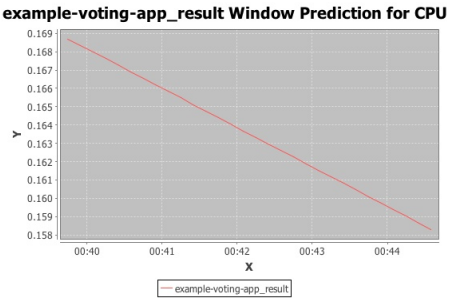
(d) db service container CPU stats



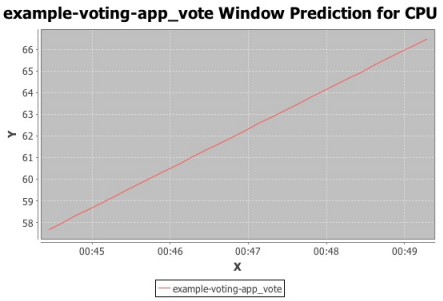
(e) Redis service container CPU stats

Figure 5.2: Plots of the monitored CPU statistics from the containers of the example voting application.

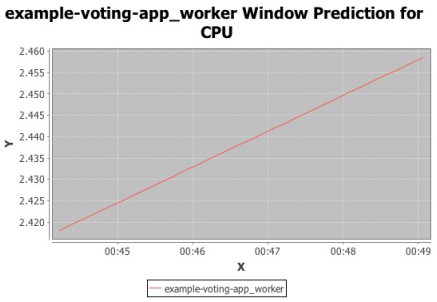
These figures show a small sample of the testing done on the application and after the confirmation of the basic communications between the Monitor and sensor components, I then moved to the Analysis Component. From this component, my aim was to see it perform a prediction of the usage for the next time window (5 minutes) given the analysis of the already recorded data. To visualise the results of this functionality, I used the same plotting mechanism implemented for the statistics and the prediction data can be seen in Figure 5.3, where the prediction of the CPU data for the next time window is performed based on the data from the results in Figure 5.2.



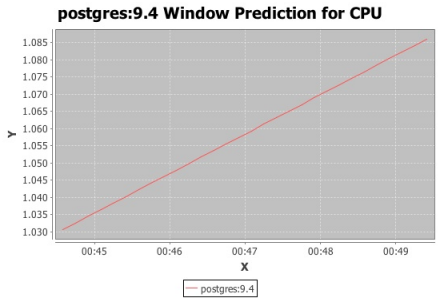
(a) Result service container CPU prediction stats



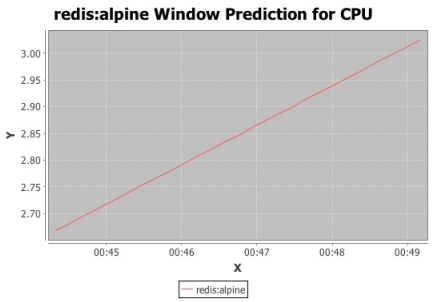
(b) Vote service container CPU prediction stats



(c) Worker service container CPU prediction stats



(d) db service container CPU prediction stats



(e) Redis service container CPU prediction stats

Figure 5.3: Plots of the predicted CPU statistics from the containers of the example voting application.

5.3 FULL SYSTEM TESTING

After the confirmation of the Monitor and analysis components, the next step of testing brought us to the version of the system after the implementation of the fuzzy logic functionality in the Analysis component, which is meant to use the predicted data and decide how many containers need to be added to or removed from the service, and the Plan component, which works with that data/ recommendation from the Analysis component and proposes the topology that best suits the recommendation before notifying the execution component to change the topology.

In order to confirm the above, I had to run varying loads through the application so that the system would respond accordingly. So, the Locust load simulation tool was used for this purpose and the workloads to be tested with are presented in Table 5.2 below. Table 5.1 shows the simulated topology structures of the test application Example Voting app introduced in the Test suite section. The table shows the services in the top row and every other row shows the number of containers for that service in that topology. These topologies were used for the purpose of providing the planning component a simplified example of different topology structures that it can switch between. The last 2 columns are based on pricing for the deployment of Amazon's General purpose dedicated host virtual machines [3] in August 2018.

In Table 5.1 and the developed system, the prices used were represented as; *small:* m4.10xlarge = 2.42 USD per hour, *medium:* m5.24xlarge = 5.069 USD per hour and *large:* m5d.24xlarge = 5.966 USD per hour. These are the prices for deploying an instance of that size on a single dedicated host on Amazon EC2 services in the US East (Ohio) region in August 2018. For all the tests run below, the window between the data analysis performed was set to 2 minutes.

Topology	Worker	Vote	db	Redis	result	Virtual Machines	total Price (USD)
T1	1	1	1	1	1	1 Small	2.42
T2	2	1	1	1	1	2 Small	4.84
T3	3	2	1	1	1	1 Medium	5.069
T4	4	3	2	1	1	1 Medium	5.069
T5	4	4	3	2	1	2 Medium	10.138
T6	4	4	4	3	2	2 Medium	10.138
T7	4	4	4	4	3	1 Large	5.966
T8	4	4	4	4	4	2 Large	11.932

Table 5.1: Topology options for the testing of the application

5.3.1 *Test Case 1: Upscaling and No system reaction Testing on the lowest topology*

```
CONTAINER ID        NAME                          CPU %               MEM USAGE / LIMIT     MEM %
2985da6d8fe9        example-voting-app_worker_1   2.25%               27.39MiB / 1.952GiB   1.37%
e42ed53b1e81        example-voting-app_vote_1     0.45%               30.7MiB / 1.952GiB    1.54%
fe9f47182e5a        redis                         0.46%               1.301MiB / 1.952GiB   0.07%
5b34023a2fbc        example-voting-app_result_1   0.24%               35.37MiB / 1.952GiB   1.77%
ccdba1d85da4        db                            0.38%               15.09MiB / 1.952GiB   0.76%
```

Figure 5.4: Container status before the Load is applied using Locust

For the first test case, I ran the system to test for its ability to request for a topology with an increased number of resources (containers) available to the application for the next time window as seen in Table 5.2. The starting Topology for this test case was Topology 1 (T1) as

seen in Table 5.1 above and can be seen in Figure 5.4 just before applying the simulated load on it. The results of this testing are presented in the table below.

Test	Users | Requests (/s)	average response time	Result	Topology
1	3 | 45	16(ms)	No Adaptation	T1
2	8 | 102	23(ms)	Adaptation	T3
3	15 | 140	51(ms)	Adaptation	T4
4	30 | 149	149(ms)	Adaptation	T4

Table 5.2: Test case 1 run on the application.

In the tests where a change in the topology was requested (scale up), for example in Test 4, the recommendation of the new topology structure was triggered by one of the services (Vote) being predicted to require a lot more computing power in the next time window and therefore, the system recommends more containers for the Service. Figure 5.5 shows the recommendation made for the system when tests 2 and 3 are run. These recommendations come from the prediction that the system does in relation to the current and previously recorded statistics, which can also be seen in Figure 5.5.

```
!!!!!!!!!!!!RECOMMENDED TOPOLOGY!!!!!!!!!!!!!!!!!!!!
Service: example-voting-app_vote Containers: 2.0
Service: postgres:9.4 Containers: 1.0
Service: example-voting-app_worker Containers: 1.0
Service: example-voting-app_result Containers: 1.0
Service: redis:alpine Containers: 1.0
```

(a) Test 2

```
!!!!!!!!!!!!RECOMMENDED TOPOLOGY!!!!!!!!!!!!!!!!!!!!
Service: example-voting-app_vote Containers: 3.0
Service: postgres:9.4 Containers: 1.0
Service: example-voting-app_worker Containers: 1.0
Service: example-voting-app_result Containers: 1.0
Service: redis:alpine Containers: 1.0
```

(b) Test 3

```
CONTAINER ID        NAME                          CPU %               MEM USAGE / LIMIT     MEM %
ea1905cebb2d        example-voting-app_worker_1   2.55%               70.22MiB / 1.952GiB   3.51%
2c8a159f2098        example-voting-app_vote_1     63.37%              33.21MiB / 1.952GiB   1.66%
e340db3d1b80        db                            1.21%               12.25MiB / 1.952GiB   0.61%
2187d5ed7da0        example-voting-app_result_1   0.25%               35.09MiB / 1.952GiB   1.76%
c419bef17c8f        redis                         4.77%               2.504MiB / 1.952GiB   0.13%
```

(c) Test 2 load

```
CONTAINER ID        NAME                          CPU %               MEM USAGE / LIMIT     MEM %
ea1905cebb2d        example-voting-app_worker_1   2.87%               70.25MiB / 1.952GiB   3.51%
2c8a159f2098        example-voting-app_vote_1     109.79%             33.95MiB / 1.952GiB   1.70%
e340db3d1b80        db                            1.27%               12.25MiB / 1.952GiB   0.61%
2187d5ed7da0        example-voting-app_result_1   0.23%               37.23MiB / 1.952GiB   1.86%
c419bef17c8f        redis                         6.26%               3.879MiB / 1.952GiB   0.19%
```

(d) Test 3 load

Figure 5.5: Recommended Topology output for Test Case 1

5.3.2 *Test Case 2: Downscaling and Testing with a different topology*

This test case was performed to confirm both the system's *Downscaling functionality* when a topology that isn't the base topology (T1) is used and therefore also verify that it works when the other topology options have been deployed. The tests run in this test case start with Topology 3 (T3) with a number of users just above the number of

users that caused the adaptation switch in the first test case as seen in Figure 5.6 Test 1 load.

Test	Users | Requests(/s)	average response time	Result	Topology
1	10 | 35	90(ms)	Adaptation	T2
2	15 | 50	130(ms)	Adaptation	T3
3	80 | 130	383 (ms)	Adaptation	T4
4	200 | 190	572(ms)	Adaptation	T4

Table 5.3: Test case 2 run on the application.

The selections of the next topology to run from the above tests for the next time window are selected based on the recommended topology by the system as seen in Figure 5.6 below.

```
!!!!!!!!!!!RECOMMENDED TOPOLOGY!!!!!!!!!!!!!!!!!!!
Service: dockersamples/examplevotingapp_result:before@sha256:83b568996e930c292a6ae5187fda84dd6568a19d97cdb933720be15c757b7463 Containers: 1.0
Service: redis:alpine@sha256:c993dd01624249e9c2b68e86dbf0710a97fb67e78a8d212ef0ef9917813bd72 Containers: 1.0
Service: postgres:9.4@sha256:b8391fd5f47258909df96cdc16f52704cea6ed714a80e6f1a244522 6c58cd640 Containers: 2.0
Service: dockersamples/examplevotingapp_worker:latest@sha256:55753a7b7872d3e2eb47f146c53899c41dcbe259d54e24b3da730b9acbff50a1 Containers: 3.0
Service: dockersamples/examplevotingapp_vote:before@sha256:8e64b18b2c87de902f2b72321c89b4af4e2b942d76d0b772532ff27ec4c6ebf6 Containers: 1.0
```

(a) Test 1

```
!!!!!!!!!!!RECOMMENDED TOPOLOGY!!!!!!!!!!!!!!!!!!!
Service: dockersamples/examplevotingapp_result:before@sha256:83b568996e930c292a6ae5187fda84dd6568a19d97cdb933720be15c757b7463 Containers: 1.0
Service: postgres:9.4@sha256:7430585790921d82a56c4cbe62fdf50f03e00b89d39cbf881afa1ef82eefd61c Containers: 2.0
Service: redis:alpine@sha256:43e4d14fcffa05a5967c353dd7061564f130d6021725dd219f0c6fcbcc6b5076 Containers: 1.0
Service: dockersamples/examplevotingapp_worker:latest@sha256:55753a7b7872d3e2eb47f146c53899c41dcbe259d54e24b3da730b9acbff50a1 Containers: 4.0
Service: dockersamples/examplevotingapp_vote:before@sha256:8e64b18b2c87de902f2b72321c89b4af4e2b942d76d0b772532ff27ec4c6ebf6 Containers: 2.0
```

(b) Test 3

```
CONTAINER ID   NAME                                   CPU %     MEM USAGE / LIMIT     MEM %
df842570e3c0   vote_result.1.mb0tf0s2pg4xms9eb3sdnelbv   0.14%     21.12MiB / 1.952GiB   1.00%
e1ac0088cf51   vote_vote.2.5z7ozdk9mt8se4mvngxp3qp0v    3.59%     61.85MiB / 1.952GiB   3.03%
c02a4d9e6eaf   vote_vote.1.esdso7tjsgu8ssdpcl137ncqd    4.35%     61.92MiB / 1.952GiB   3.10%
a769845dbf77   vote_db.1.fbboolnyl3t3fnoz7usz0vghh      39.10%    15.76MiB / 1.952GiB   0.79%
9cbea341eb4d   vote_redis.1.6izkmio9zo44tczf0agctpba5   24.30%    1.742MiB / 1.952GiB   0.09%
cf576c740ff6   vote_visualizer.1.kyx7c8uo4prs92jZo83zb5jyg  0.00%     37.48MiB / 1.952GiB   1.88%
c4257bce0112   vote_worker.1.var6guw6tr48og2jkbvvcksp8  81.19%    81.31MiB / 1.952GiB   4.07%
5ec5c551c3c1   vote_worker.2.i3q2orw4nrni6h2momb33t82m  68.42%    81.04MiB / 1.952GiB   4.05%
fb3f9d674c9d   vote_worker.3.k5fmo4qlbb7lrnvh01v91y9tr  82.31%    81.14MiB / 1.952GiB   4.06%
```

(c) Test 1 load

```
CONTAINER ID   NAME                                   CPU %     MEM USAGE / LIMIT     MEM %
3d91ccd66eef   vote_visualizer.1.jp6w9zl54ih9fcjpw1315vqcm  0.00%     37.55MiB / 1.952GiB   1.88%
51e8b3d6f0e0   vote_worker.3.ykcczqci8uipsu9y1n6rb9dov  68.72%    80.72MiB / 1.952GiB   4.04%
fd56a2847360   vote_worker.1.ebypye7a10j3mtrtgvscue6i3  69.70%    79.84MiB / 1.952GiB   3.99%
ef3ca0ef15c3   vote_worker.2.zp1akl8lpnfvbz70tsxpjqeti  77.85%    81.42MiB / 1.952GiB   4.07%
2fd1145c04c0   vote_result.1.zngyrajv7tk5w2jughuxqiqdk  0.14%     21.79MiB / 1.952GiB   1.09%
97bb3e6ba10b   vote_vote.1.pdjpis61ez8y7smowjqhtv9zu    3.19%     63.09MiB / 1.952GiB   3.16%
72a52b2b9190   vote_vote.2.yhvtudk8m2xvvbmx5ql4l3ct5    0.02%     62.48MiB / 1.952GiB   3.13%
f252a4d08a0b   vote_db.1.urttjsdpc1dshbeyus7q0uv1r      33.58%    15.55MiB / 1.952GiB   0.78%
2937e0ccdc01   vote_redis.1.7osh4ribp0kzd8oh18q4du595   21.41%    1.965MiB / 1.952GiB   0.10%
```

(d) Test 3 load

Figure 5.6: Recommended Topology output for Test Case 2

6 EVALUATION

The objective of this work was to design, develop, and test a system supporting the lifecycle proposed in [4]. To evaluate the system developed, I shall present a checklist of requirements that were realised in Chapter 3 and show the ones fulfilled by the system. Finally, I shall present a case study application used to perform testing on the system with a real world usable application. The application is more significant in terms of the number of services available than in the test application and therefore, it provides better testing value in relation to a real world application.

6.1 SYSTEM REQUIREMENTS

In this section, I review the requirements realised by the developed system using the tables below containing the requirement and whether it was implemented into the system. The tests run in the 2 test cases in Chapter 5 prove most of the functional requirements were fulfilled. However, the requirements in the tables with an ✗represent the requirements that I was unable to implement into the system. Most of these unimplemented requirements relate to the database, which unfortunately was not implemented in the latest state of the system. Some other requirements related to the sensor component were realised for the implementation of a reactive solution. However, since this is a predictive solution, these were not incorporated into the system's functionality.

System Functional Requirements

Requirements	FR1	FR2	FR3	FR4	FR5	FR6	FR7	FR8	FR9	FR10	FR11
Realised	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓

Table 6.1: System Functional Requirements evaluation

System Non-Functional Requirements

Requirements	NFR1	NFR2	NFR3	NFR4	NFR5
Realised	✓	✓	✗	✗	✗

Table 6.2: System Non-Functional Requirements evaluation

Monitor Component Requirements

Requirements	FR1.1	FR1.2	FR1.3	FR1.4	FR1.5	FR1.6	FR1.7	FR1.8
Realised	✓	✓	✓	✗	✓	✗	✗	✓

Table 6.3: Monitor Component Functional Requirements evaluation

Analysis Component Requirements

Requirements	FR2.1	FR2.2	FR2.3	FR2.4	FR2.5	FR2.6	FR2.7	FR2.8	FR2.9	FR2.10	FR2.11	FR2.12
Realised	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗

Table 6.4: Analysis Component Functional Requirements evaluation

Plan Component Requirements

Requirements	FR3.1	FR3.2	FR3.3	FR3.4	FR3.5	FR3.6	FR3.7	FR3.8	FR3.9	FR3.10
Realised	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.5: Plan Component Functional Requirements evaluation

Execution Component Requirements

Requirements	FR4.1	FR4.2	FR4.3	FR4.4	FR4.5	FR4.6	FR4.7	FR4.8	FR4.9	FR4.10	FR4.11
Realised	✓	✗	✓	✓	✓	✓	✓	✗	✗	✓	✗

Table 6.6: Execution Component Functional Requirements evaluation

6.2 CASE STUDY

After the confirmation of the fulfilment of most of the important Functional requirements of the system that make it do what it is meant to, we perform a case study on the system using a larger application with a lot more services than the simple application that was used for testing.

The application used for this case study is a Sock shop application from GitHub [35] by developed by Weaveworks [36]. The application also fulfils the microservice architecture style requirement with services like a shopping cart service, an orders service, a catalogue service to mention but a few and provides the user side for an online shop that sells socks with a total of 14 services working together to complete the full functionality of the application. For this case study, the application is set up with a base topology setup of 1 container pre service and the locust load tester is used like in Chapter 5 to apply the simulated load. The viable topologies are randomly generated by the code in Listing 6.1, which uses the available service names from the currently running topology, creates 6 new topologies, adds a random

number of containers (1-5) to each of the services, assigns a random number of Virtual machines to the topologies (1-3) with random sizes and adds the topologies to the list of viable ones.

```
    public void defineDynamicTopologies2() {

    ArrayList<String> myservs = createServices();
    Topology t1 = DockerManager.getInstance().
   getCurrentTopology();
    t1.setFilename("Current Topology");
    if (t1.getVMS().size() < 1) {
      t1.addVM(new SmallVM());
      t1.addVM(new SmallVM());
      t1.setFilename("Current Topology");
    }
    addTopology(t1);
    Topology vtop2 = new Topology("top1");
    Topology vtop3 = new Topology("top2");
    Topology vtop4 = new Topology("top3");
    Topology vtop5 = new Topology("top4");
    Topology vtop6 = new Topology("top5");
    Topology vtop7 = new Topology("top6");


    addVMs(vtop2);
    addVMs(vtop3);
    addVMs(vtop4);
    addVMs(vtop5);
    addVMs(vtop6);
    addVMs(vtop7);
    addServices(myservs, vtop2);
    addServices(myservs, vtop3);
    addServices(myservs, vtop4);
    addServices(myservs, vtop5);
    addServices(myservs, vtop6);
    addServices(myservs, vtop7);
    addTopology(vtop2);
    addTopology(vtop3);
    addTopology(vtop4);
    addTopology(vtop5);
    addTopology(vtop6);
    addTopology(vtop7);

  }

      void addServices(ArrayList<String> s, Topology t) {
    int toprand = randomNumber(1, 3);
    for (String str : s) {
      if (toprand == 1) {
        t.addService(str, randomNumber(1, 3));
      } else if (toprand == 2) {
        t.addService(str, randomNumber(2, 4));
      } else {
        t.addService(str, randomNumber(3, 5));
      }

    }
```

```
}
```

Listing 6.1: Viable Topology definition code

The results of the tests run on the developed system managing the test application's resources are shown in Table 6.7 below. The system results of the first test run on the application are shown in the images that follow the table and the results of the rest of the tests can be found in the appendix. For all the rests run below, the window between the data analysis performed was set to 2 minutes.

Figure 6.1 bbelow shows the state of the application before any load is applied to it using the locust load tester.

```
CONTAINER ID        NAME                          CPU %               MEM USAGE / LIMIT     MEM %
547e81bc2de8        docker-compose_carts_1        0.39%               297.4MiB / 1.952GiB   14.88%
dd9b46590333        docker-compose_front-end_1    0.00%               41.2MiB / 1.952GiB    2.06%
a3f56c761899        docker-compose_user-db_1      1.08%               17.25MiB / 1.952GiB   0.86%
016992263d37        docker-compose_queue-master_1 0.41%               340.4MiB / 1.952GiB   17.03%
e6a2db0f8085        docker-compose_shipping_1     0.41%               279.3MiB / 1.952GiB   13.98%
596ebaf2e237        docker-compose_rabbitmq_1     0.52%               47.77MiB / 1.952GiB   2.39%
f0d2204688dd        docker-compose_carts-db_1     1.19%               29.52MiB / 1.952GiB   1.48%
de52290a3b81        docker-compose_edge-router_1  0.06%               9.957MiB / 1.952GiB   0.50%
48e84ee51a69        docker-compose_orders-db_1    1.11%               30.64MiB / 1.952GiB   1.53%
9d697e4dac40        docker-compose_orders_1       0.38%               295.4MiB / 1.952GiB   14.78%
9b750f3f126c        docker-compose_catalogue_1    0.00%               2.84MiB / 1.952GiB    0.14%
eaed5a9cbf58        docker-compose_catalogue-db_1 0.13%               164.1MiB / 1.952GiB   8.21%
e2e8335cbbb8        docker-compose_payment_1      0.00%               2.152MiB / 1.952GiB   0.11%
59fe837f93e6        docker-compose_user_1         0.00%               4.785MiB / 1.952GiB   0.24%
```

Figure 6.1: Weave Sock shop application before load is applied to the application.

Test	Users | Requests(/s)	average response time	Result	Topology
1	15 | 70	34(ms)	Adaptation	Topology 2
2	40 | 210	140 (ms)	Adaptation	Topology5, 2Medium VMs
3	300 | 220	1196(ms)	Adaptation	N/A

Table 6.7: Test case 2 run on the application.

After running a number of tests on this application, I present the results of one of the tests below, which was run with the lower load as seen in the table above. Because one of the services required more than one container to function optimally, an adaptation was requested. The number of containers required for the services to function optimally in the next time window using the workload predicted by the system is presented in Figure 6.2.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %
547e81bc2de8	docker-compose_carts_1	13.38%	278.2MiB / 1.952GiB	13.92%
dd9b46590333	docker-compose_front-end_1	41.39%	73.31MiB / 1.952GiB	3.67%
a3f56c761899	docker-compose_user-db_1	5.60%	11.14MiB / 1.952GiB	0.56%
016992263d37	docker-compose_queue-master_1	0.53%	335.5MiB / 1.952GiB	16.78%
e6a2db0f8085	docker-compose_shipping_1	1.90%	285.8MiB / 1.952GiB	14.30%
596ebaf2e237	docker-compose_rabbitmq_1	1.09%	36.45MiB / 1.952GiB	1.82%
f0d2204688dd	docker-compose_carts-db_1	6.67%	34.42MiB / 1.952GiB	1.72%
de52290a3b81	docker-compose_edge-router_1	10.80%	8.629MiB / 1.952GiB	0.43%
48e84ee51a69	docker-compose_orders-db_1	1.14%	33.14MiB / 1.952GiB	1.66%
9d697e4dac40	docker-compose_orders_1	11.41%	309.9MiB / 1.952GiB	15.51%
9b750f3f126c	docker-compose_catalogue_1	3.42%	6.074MiB / 1.952GiB	0.30%
eaed5a9cbf58	docker-compose_catalogue-db_1	2.22%	94.06MiB / 1.952GiB	4.71%
e2e8335cbbb8	docker-compose_payment_1	0.77%	5.512MiB / 1.952GiB	0.28%
59fe837f93e6	docker-compose_user_1	11.16%	7.414MiB / 1.952GiB	0.37%

(a) Load Snapshot

```
!!!!!!!!!!!!RECOMMENDED TOPOLOGY!!!!!!!!!!!!!!!!!!!!!
Service: weaveworksdemos/user-db:0.4.0 Containers: 1.0
Service: weaveworksdemos/edge-router:0.1.1 Containers: 1.0
Service: weaveworksdemos/catalogue-db:0.3.0 Containers: 1.0
Service: weaveworksdemos/user:0.4.4 Containers: 1.0
Service: weaveworksdemos/front-end:0.3.12 Containers: 2.0
Service: weaveworksdemos/shipping:0.4.8 Containers: 1.0
Service: weaveworksdemos/catalogue:0.3.5 Containers: 1.0
Service: weaveworksdemos/orders:0.4.7 Containers: 1.0
Service: weaveworksdemos/carts:0.4.8 Containers: 1.0
Service: weaveworksdemos/queue-master:0.3.1 Containers: 1.0
Service: mongo:3.4 Containers: 1.0
Service: weaveworksdemos/payment:0.4.3 Containers: 1.0
Service: rabbitmq:3.6.8 Containers: 1.0
```

(b) Recommended Topology outlay

Figure 6.2: Analysis done in test T1 on the Cases Study application.

Figure 6.2 shows the topology recommended by the system in terms of service to container count alongside a snapshot of the resource usage at the load mentioned in Table 6.7. These results are derived from the analysis of the monitored data and a prediction of the average load of the system for the next time window, which then is used to estimate the optimal number of containers for the system in the upcoming time window.

```
top3 | 22 | 7.489
Service: weaveworksdemos/user-db:0.4.0 Containers: 2.0

Service: weaveworksdemos/edge-router:0.1.1 Containers: 2.0

Service: weaveworksdemos/catalogue-db:0.3.0 Containers: 3.0

Service: weaveworksdemos/user:0.4.4 Containers: 2.0

Service: weaveworksdemos/front-end:0.3.12 Containers: 2.0

Service: weaveworksdemos/shipping:0.4.8 Containers: 2.0

Service: weaveworksdemos/catalogue:0.3.5 Containers: 3.0

Service: weaveworksdemos/orders:0.4.7 Containers: 3.0

Service: weaveworksdemos/carts:0.4.8 Containers: 2.0

Service: weaveworksdemos/queue-master:0.3.1 Containers: 2.0

Service: mongo:3.4 Containers: 3.0

Service: weaveworksdemos/payment:0.4.3 Containers: 2.0

Service: rabbitmq:3.6.8 Containers: 3.0

top2 | 22 | 5.069
Service: weaveworksdemos/user-db:0.4.0 Containers: 3.0

Service: weaveworksdemos/edge-router:0.1.1 Containers: 2.0

Service: weaveworksdemos/catalogue-db:0.3.0 Containers: 3.0

Service: weaveworksdemos/user:0.4.4 Containers: 3.0

Service: weaveworksdemos/front-end:0.3.12 Containers: 2.0

Service: weaveworksdemos/shipping:0.4.8 Containers: 2.0

Service: weaveworksdemos/catalogue:0.3.5 Containers: 2.0

Service: weaveworksdemos/orders:0.4.7 Containers: 2.0

Service: weaveworksdemos/carts:0.4.8 Containers: 2.0

Service: weaveworksdemos/queue-master:0.3.1 Containers: 2.0

Service: mongo:3.4 Containers: 2.0

Service: weaveworksdemos/payment:0.4.3 Containers: 3.0

Service: rabbitmq:3.6.8 Containers: 3.0
```

Figure 6.3: The best options among the topology options available

After the containers have been suggested, and a current system state is created, the Planner is notified and it makes a list of the topologies which best suit the recommended topology by assigning them scores as seen in Figure 6.3 where the topology is stated and the score next to it. Additionally, the topologies containing a service with a number of instances/ containers less than what is recommended is eliminated from the selection and therefore, no price analysis is done on that topology.

```
INFO: ...............................Topology Selected dNLlj6KH

Service: weaveworksdemos/user-db:0.4.0 | Containers = 3.0

Service: weaveworksdemos/edge-router:0.1.1 | Containers = 2.0

Service: weaveworksdemos/catalogue-db:0.3.0 | Containers = 3.0

Service: weaveworksdemos/user:0.4.4 | Containers = 3.0

Service: weaveworksdemos/front-end:0.3.12 | Containers = 2.0

Service: weaveworksdemos/shipping:0.4.8 | Containers = 2.0

Service: weaveworksdemos/catalogue:0.3.5 | Containers = 2.0

Service: weaveworksdemos/orders:0.4.7 | Containers = 2.0

Service: weaveworksdemos/carts:0.4.8 | Containers = 2.0

Service: weaveworksdemos/queue-master:0.3.1 | Containers = 2.0

Service: mongo:3.4 | Containers = 2.0

Service: weaveworksdemos/payment:0.4.3 | Containers = 3.0

Service: rabbitmq:3.6.8 | Containers = 3.0

Topology to be deployed
Topology top2 | Virtual machines = 1 | Price($) =5.069
```

Figure 6.4: The topology selected by a combination of points and a low price

Some times, there might be only one topology option available however, this time there were 2 with the same score but so, the tie breaker in this case was the cost of the topology. There are other cases where the a number of topologies are available with lower scores but, as usual, the cost of the topology is usually the last barrier for a topology to cross before being selected by the system.

6.2.1 *Cost analysis discussion.*

Scenario 1:
When launching an application, one cannot be certain of the way the application's usage will turn out in terms of how much load it will have to deal with and therefore the application owner(s) can end up paying for extra resources and stick with them until it's absolutely necessary for them to scale up because of growth in traffic. However, the resources constantly being paid for aren't going to be utilised optimally and even if there are autoscalers available, they might only help with the performance of the application and not with how much the application owner pays for these resources.
With the definition of different topologies, and the use of a system like the one developed in this project, an application owner can gain some advantage. By the collection of the data from previous resource usage, the system can make predictions and estimate the resources the application will need at certain times in order to function opti-

mally and therefore, reduce the cost to the owner both in terms of the resources paid for since they won't have to pay for unused resources and in the way of keeping the application performing optimally by adding resources when it is predicted to bye necessary.
Test 3 shown in Table 6.7 shows a situation where no topology suits the one recommended by the system as seen in Figure 6.5.

Figure 6.5: All available topologies labelled as bad topologies hence, no options.

Given that no topology switch was requested, the application service instances started to run out of resources. The simulated rise in the number of users and requests was gradual and not sudden as seen in Figure 6.6 so, the developed system caught this trend and predicted that the current topology was not suitable for the next time window. However, because there was no suitable topology among the viable topology options defined, we ended up in a situation that an application owner would probably be in without the developed system to change to a different topology.
The Sock shop application is meant to be a stand in for an e-commerce website where a user can browse and possibly in the end buy some socks.
However, after applying the load on the system, the site became inaccessible, whereby trying to access the home page took over 2 minutes and given that the tolerable wait time for a Web user is 2 seconds according to [28], the typical user would have already moved on, leading to a loss of revenue by the application owner. Such high usage situations on such a Web application may occur during times of product promotions or even the launch of new products and it would be advantageous to have such a system in place to predict and automatically make the adjustments needed in time to prevent a loss in revenue at such an important time. A case may be argued claiming that one could just scale up at the time of the increased traffic. However, the resources needed to cover this usage increase could also come at a higher price than they would have originally paid before the situation occurred.

Figure 6.6: Locust charts showing the change of number of users and average response times.

In Figure 6.7 below, we notice a significant number of failed requests from some of the more important services like the catalogue, and orders mainly. These are 2 services that the web application would get most of its worth from given that the users have to see the products they are buying and then be able to actually purchase the products. Having these 2 services down would basically mean that the web application built and deployed to make money is not performing its role and therefore making it worthless in this state.

Statistics Charts Failures Exceptions Download Data

Type	Name	# requests	# fails	Median (ms)	Average (ms)	Min (ms)	Max (ms)	Content Size	# reqs/sec
GET	/	6755	0	620	1021	85	35687	8688	20.2
GET	/basket.html	6724	0	690	1063	70	37970	23020	21.3
DELETE	/cart	6715	0	850	1148	79	35529	0	22.6
POST	/cart	6420	295	1200	1426	109	4550	0	21.4
GET	/catalogue	6752	9151	720	748	94	4674	2795	20.9
GET	/category.html	6752	0	640	1015	78	38031	11518	17.7
GET	/detail.html?id=03fef6ac-1896-4ce8-bd69-b798f85c6e0b	758	0	670	1024	104	3860	10364	2.5
GET	/detail.html?id=3395a43e-2d88-40de-b95f-e00e1502085b	739	0	630	998	80	3688	10364	1.9
GET	/detail.html?id=510a0d7e-8e83-4193-b483-e27e09ddc34d	736	0	670	1057	91	3666	10364	2
GET	/detail.html?id=808a2de1-1aaa-4c25-a9b9-6612e8f29a38	778	0	670	1054	97	3675	10364	1.9
GET	/detail.html?id=819e1fbf-8b7e-4f6d-811f-693534916a8b	731	0	680	1059	99	3921	10364	1.8
GET	/detail.html?id=837ab141-399e-4c1f-9abc-bace40296bac	731	0	730	1092	79	3666	10364	2
GET	/detail.html?id=a0a4f044-b040-410d-8ead-4de0446aec7e	764	0	670	1050	114	3896	10364	2.1
GET	/detail.html?id=d3588630-ad8e-49df-bbd7-3167f7efb246	747	0	620	953	79	3586	10364	2.2
GET	/detail.html?id=zzz4f044-b040-410d-8ead-4de0446aec7e	752	0	680	1043	85	4279	10364	2.8
GET	/login	6758	0	1400	1563	149	4800	13	19
POST	/orders	5694	1060	1700	1863	165	38681	655	19.8
	Total	**59306**	**10506**	**720**	**1196**	**70**	**38681**	**6470**	**182.1**

Figure 6.7: Table of Locust statistics while running a stress test on the Case study application.

Scenario 2:

Given that the situation from scenario 1 occurs, one may decide to make use of the more expensive/ safer options in order to make sure

to avoid ending up in that situation. For example, if we take a look at the pricing options of Amazon Web services including; On-demand, Reserved, Spot pricing and Dedicated Hosts. The On-demand option having a price disadvantage over the rest. I.e: spot (90 percent off On-demand) , reserved (up to 75 percent off On-demand) and dedicated hosts (up to 70 percent off On-demand) all having an advantage over On-demand [3]. It is clear that no matter how much that unfortunate scenario may have cost the application owner in a short run, the use of options like reserved instances from amazon web service in a long run without any assurances that the situation will occur again, can end up costing the owner a lot more. That is to say, the web application might have lost a number of the reoccurring users so, the few that remain may not bring the the usage experienced in the first scenario and so, it might be a while before that occurs again. Therefore in conclusion, a system like the one developed in this project would provide the application owner some security knowing that his application is running optimally and he is paying for the optimal amount of resources required for his application.

6.3 system limitations.

This system has been developed to with respect to the solution discussed in [4]. However, in its current state, it faces some limitations, which could impede its functionalities and these include:

1- The current inability to work on multiple Virtual Machines:

The sensors deployed on the different containers for a service may face difficulty in terms of communication if a web application's service is deployed in multiple virtual machine instances. This is because the system in its current state has no way of communication across different virtual machine instances.

2- No viable topology problem.

In its current state, the system has no solution to the lack of a viable topology in case they are all unsuitable for the load coming in the next time window. At the moment, the system will just continue to run and perform another analysis and in essence rerun the MAPE loop without it having previously redistributed the application.

3- Single application running in a virtual machine.

In the current state of the system, there can only be a single application's services running in the virtual machine instances. That is to say, if there is more than one application running on a single virtual

machine, the system will consider all the containers on the virtual machine as part of the same application and therefore lead to errors while performing any analysis.

4- Container upscale restriction.

In the current implementation of the system, the maximum number of containers that can be recommended has been set to 5 per service. This restriction was put due to the limitations the computer I was running the testing on given the fact that it struggled to run a topology style where there were more than 20 containers deployed for the application. This restriction can therefore result in the occurrence of limitation 2 above.

5- Pricing model.

The pricing model defined in the system for the Virtual Machines is currently implemented in a static manner and therefore, a change in the prices of the virtual machine services by the service provider would lead to inaccurate results from the system.

6- Execution component.

While the execution component class is defined and has some of its methods implemented, it is currently disconnected from the Plan component. Therefore, the redeployment of the application in a new topology is not currently performed by the system and was done manually during the testing of the system.

7 CONCLUSION

7.1 SUMMARY AND DISCUSSION

At the start of this project, we set out to develop a system in relation to the application topology lifecycle discussed in [4]. The role of this system was to fulfil part of this lifecycle including the discovery of the services of an application managed by the system, retrieve viable topologies for the application, monitor and analyse the application's resource usage in the topology currently deployed and finally to redeploy the application in an alternative topology if necessary by selecting the most reasonable option after considering the topology's suitability for the situation and the cost of the topology compared to the other options. To perform this automation strategy, we set out to use the MAPE-K strategy as presented in the application lifecycle and this involved the monitoring of the statistics through the Docker API of the docker containers forming the application's services. Then an analysis of these statistics are performed where by the use of regression, a prediction is performed and then by the use of fuzzy logic, the number of containers suitable for a service's predicted usage is retrieved. Finally, by the aggregation of these analysis results, a plan is formed for what topology should be used in the next redeployment of the application.
Upon completion of the implementation of the main functionality of the system, we then first tested it on a small microservice style voting application, which consisted of 5 services and presented the results of these tests. Finally, a case study was performed on an e-commerce application called the sock shop application, which was much more significant in size, that is it consisted of 14 services and also made use of the microservice architecture style. The results of the case study were also presented and therefore finalising with a review of the system's limitations.

The system however, also has some unresolved issues that came up during its design and development. One of these issues was reacting to the recorded metric data falling outside the service level agreements. However, the solution selected to be done was a predictive solution due to the fact that not only one part of the managed application is affected by the change in topology so, the time window required for a reactive solution might be too small when compared to the time needed for the redeployment the whole system.

The Implementation of a knowledge base was another part of the system left out. However, this didn't affect the system in its current

state too adversely. This is because the knowledge base would be more of a necessity when the system requires to store hours or days of metric data in order to perform an analysis on that data. Additionally, the knowledge base would also come in handy when there is the need to store more topologies and in the case of this project, this was not a major necessity as the aim was to implement the initial functionality of the system.

7.2 FUTURE WORK

In the field of cloud computing, the proposed life cycle of a cloud-based application [4] in the form of a system could prove to be a very useful tool. Therefore, the extension of the system developed for this project to fulfil the remaining set of MAPE-K loops hence, giving the system the ability to generate and test topologies until it finds the optimum solution for the given situation could prove to be useful both on an enterprise level and research level. Additionally, the complete system would also be able to work in conjunction with an IDE to be developed in the manner discussed by the MODAClouds approach [6] to help in the validation of the proposed lifecycle.

BIBLIOGRAPHY

[1] *AWS Auto Scaling*. https://aws.amazon.com/autoscaling/. Accessed: 2018-08-15.

[2] Fahd Al-Haidari, M Sqalli, and Khaled Salah. "Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources." In: *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*. Vol. 2. IEEE. 2013, pp. 256–261.

[3] *Amazon EC2 Dedicated Hosts Pricing*. https://aws.amazon.com/ec2/dedicated-hosts/pricing/. Accessed: 2018-08-26.

[4] Vasilios Andrikopoulos. "Engineering Cloud-based Applications: Towards an Application Lifecycle." In: *European Conference on Service-Oriented and Cloud Computing*. Springer. 2017, pp. 57–72.

[5] Vasilios Andrikopoulos, Santiago Gómez Sáez, Frank Leymann, and Johannes Wettinger. "Optimal distribution of applications in the cloud." In: *International Conference on Advanced Information Systems Engineering*. Springer. 2014, pp. 75–90.

[6] Danilo Ardagna, Elisabetta Di Nitto, Giuliano Casale, Dana Petcu, Parastoo Mohagheghi, Sébastien Mosser, Peter Matthews, Anke Gericke, Cyril Ballagny, Francesco D'Andria, et al. "Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds." In: *Proceedings of the 4th International Workshop on Modeling in Software Engineering*. IEEE Press. 2012, pp. 50–56.

[7] Maricela-Georgiana Avram. "Advantages and challenges of adopting cloud computing from an enterprise perspective." In: *Procedia Technology* 12 (2014), pp. 529–534.

[8] Tobias Binz, Gerd Breiter, Frank Leyman, and Thomas Spatzier. "Portable cloud services using tosca." In: *IEEE Internet Computing* 16.3 (2012), pp. 80–85.

[9] Pablo Cingolani and Jesús Alcalá-Fdez. "jFuzzyLogic: a java library to design fuzzy logic controllers according to the standard for fuzzy control programming." In: *International Journal of Computational Intelligence Systems* 6.sup1 (2013), pp. 61–75.

[10] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. "Elastic virtual machine for fine-grained cloud resource provisioning." In: *Global Trends in Computing and Communication Systems*. Springer, 2012, pp. 11–25.

[11] *Develop with Docker Engine SDKs and API*. https://docs.docker.com/develop/sdk/. Accessed: 2018-08-23.

[12] *Docker Compose*. https://docs.docker.com/compose/. Accessed: 2018-08-23.

[13] *Example Voting App*. https://github.com/dockersamples/example-voting-app. Accessed: 2018-08-25.

[14] Wei Fang, ZhiHui Lu, Jie Wu, and ZhenYin Cao. "RPPS: a novel resource prediction and provisioning scheme in cloud data center." In: *Services Computing (SCC), 2012 IEEE Ninth International Conference on*. IEEE. 2012, pp. 609–616.

[15] Stefan Frey, Claudia Lüthje, Christoph Reich, and Nathan Clarke. "Cloud QoS scaling by fuzzy logic." In: *Cloud Engineering (IC2E), 2014 IEEE International Conference on*. IEEE. 2014, pp. 343–348.

[16] José María García, Octavio Martín-Díaz, Pablo Fernandez, Antonio Ruiz-Cortés, and Miguel Toro. "Automated analysis of cloud offerings for optimal service provisioning." In: *International Conference on Service-Oriented Computing*. Springer. 2017, pp. 331–339.

[17] Mostafa Ghobaei-Arani, Sam Jabbehdari, and Mohammad Ali Pourmina. "An autonomic resource provisioning approach for service-based cloud applications: A hybrid approach." In: *Future Generation Computer Systems* 78 (2018), pp. 191–210.

[18] *Github thesis project code and files*. https://github.com/Rwemigabo/Thesis. Accessed: 2018-08-24.

[19] Santiago Gómez Sáez, Vasilios Andrikopoulos, Florian Wessling, Clarissa Cassales Marquezan, et al. "Cloud Adaptation & Application (Re-) Distribution: Bridging the two Perspectives." In: *IEEE Computer Society Press* (2014).

[20] John Hunt. "Gang of four design patterns." In: *Scala Design Patterns*. Springer, 2013, pp. 135–136.

[21] Didac Gil De La Iglesia and Danny Weyns. "MAPE-K formal templates to rigorously design behaviors for self-adaptive systems." In: *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 10.3 (2015), p. 15.

[22] Dejun Jiang, Guillaume Pierre, and Chi-Hung Chi. "Autonomous resource provisioning for multi-service web applications." In: *Proceedings of the 19th international conference on World wide web*. ACM. 2010, pp. 471–480.

[23] Jeffrey O Kephart and David M Chess. "The vision of autonomic computing." In: *Computer* 36.1 (2003), pp. 41–50.

[24] Harold C Lim, Shivnath Babu, and Jeffrey S Chase. "Automated control for elastic storage." In: *Proceedings of the 7th international conference on Autonomic computing*. ACM. 2010, pp. 1–10.

[25] *Locust*. https://docs.locust.io/en/stable/index.html. Accessed: 2018-08-25.

[26] Joao Loff and Joao Garcia. "Vadara: Predictive elasticity for cloud applications." In: *Cloud Computing Technology and Science (CloudCom), 2014 IEEE 6th International Conference on*. IEEE. 2014, pp. 541–546.

[27] *Microservices*. https://martinfowler.com/articles/microservices.html. Accessed: 2018-08-04.

[28] Fiona Fui-Hoon Nah. "A study on tolerable waiting time: how long are web users willing to wait?" In: *Behaviour & Information Technology* 23.3 (2004), pp. 153–163.

[29] *Oracle Solaris Containers*. http://www.oracle.com/technetwork/server-storage/solaris/containers-169727.html. Accessed: 2018-08-23.

[30] *POSTGRESQL: THE WORLD'S MOST ADVANCED OPEN SOURCE RELATIONAL DATABASE*. https://www.postgresql.org/. Accessed: 2018-08-26.

[31] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi. "Cloud Container Technologies: a State-of-the-Art Review." In: *IEEE Transactions on Cloud Computing* (2018), pp. 1–1. ISSN: 2168-7161. DOI: 10.1109/TCC.2017.2702586.

[32] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. "Auto-scaling web applications in clouds: A taxonomy and survey." In: *ACM Computing Surveys (CSUR)* 51.4 (2018), p. 73.

[33] Alfonso Quarati, Daniele D'Agostino, Antonella Galizia, Matteo Mangini, and Andrea Clematis. "Delivering cloud services with QoS requirements: an opportunity for ICT SMEs." In: *International Conference on Grid Economics and Business Models*. Springer. 2012, pp. 197–211.

[34] *RKT overview page*. https://coreos.com/rkt/. Accessed: 2018-08-23.

[35] *Sock Shop, A Microservice Demo Application*. https://github.com/microservices-demo/microservices-demo. Accessed: 2018-08-27.

[36] *Weave Cloud, Simplify Containers and Microservices*. https://www.weave.works/. Accessed: 2018-08-27.

[37] Lenar Yazdanov and Christof Fetzer. "Vertical scaling for prioritized vms provisioning." In: *Cloud and Green Computing (CGC), 2012 Second International Conference on*. IEEE. 2012, pp. 118–125.

[38] Lotfi A Zadeh. "Outline of a new approach to the analysis of complex systems and decision processes." In: *IEEE Transactions on systems, Man, and Cybernetics* 1 (1973), pp. 28–44.

[39] Lotfi A Zadeh. "Fuzzy logic= computing with words." In: *IEEE transactions on fuzzy systems* 4.2 (1996), pp. 103–111.

A

APPENDIX

A.1 DOCUMENTS

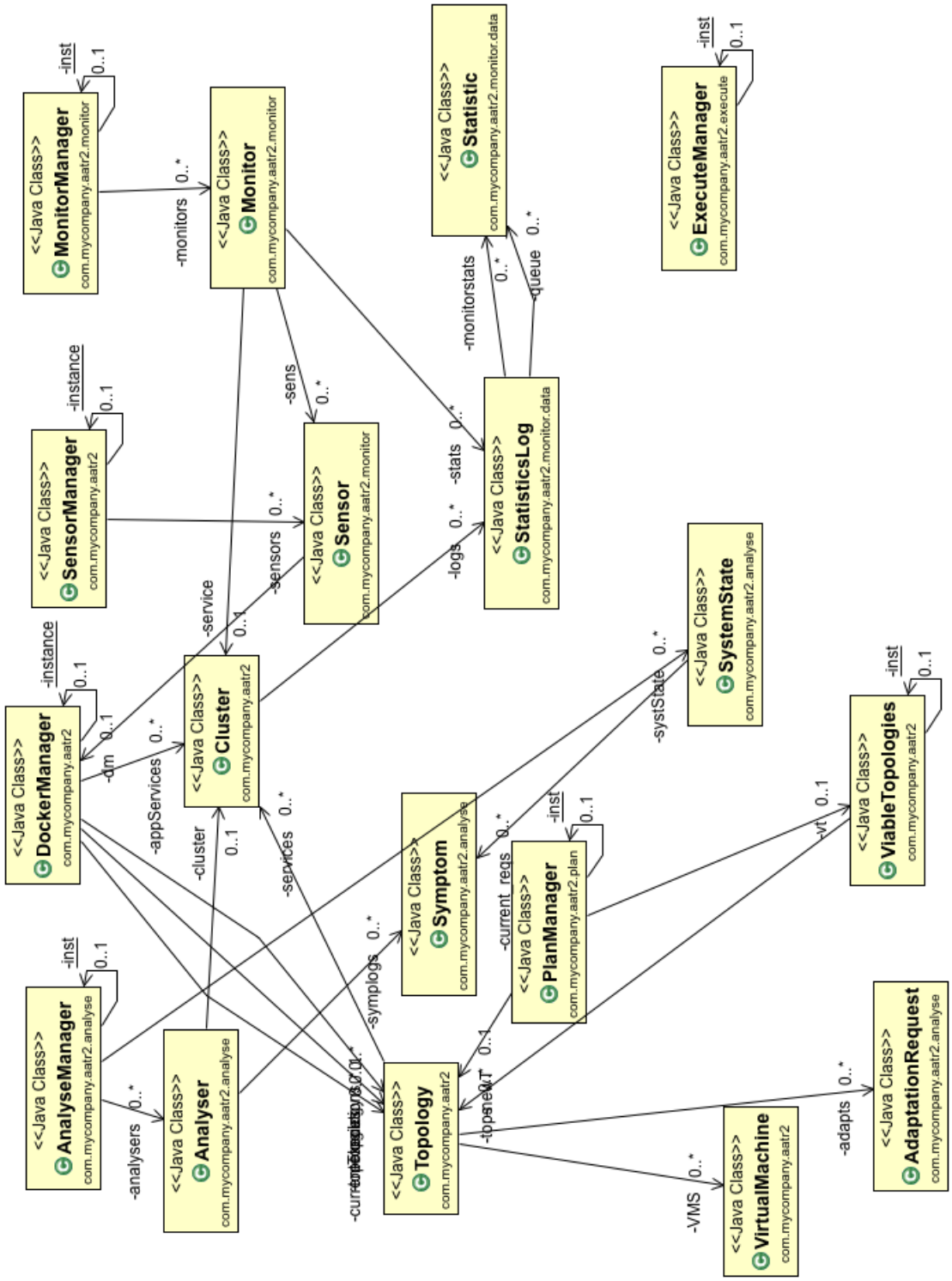


Figure A.1: Full system components

DECLARATION

I confirm that this Master's thesis is my own work and I have documented all sources and material used. This thesis was not previously presented to another examination board and has not been published anywhere else.

Netherlands, August 2018

Eric

Eric Rwemigabo