Feedback Driven Development of Cloud Applications

Feedback Driven Development of Cloud Applications
Master-Thesis von Harini Gunabalan aus Indisch
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini, Dr. -Ing. Guido Salvaneschi [TU Darmstadt]
2. Gutachten: Dr. Gerald Junkermann, Aryan Dadashi [SAP Research, Darmstadt]

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Computer Science
Software Technology Group

Feedback Driven Development of Cloud Applications
Feedback Driven Development of Cloud Applications

Vorgelegte Master-Thesis von Harini Gunabalan aus Indisch

1. Gutachten: Prof. Dr.-Ing. Mira Mezini, Dr. -Ing. Guido Salvaneschi [TU Darmstadt]
2. Gutachten: Dr. Gerald Junkermann, Aryan Dadashi [SAP Research, Darmstadt]

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 8th August 2016

_____

(Harini Gunabalan)

Acknowledgment

"My sincere thanks to Guido and Prof. Mezini, for encouraging me to work on the topic of my interest and also supporting my efforts. I received an extraordinary support from Gerald, Aryan, and Anne from SAP Research, helping with all my doubts, and providing me a possibility to work on my thesis as a part of Cloudwave research project. I was lucky to get support from Juergen and all members of the Cloudwave project, receiving valuable ideas and tips at the right time. I am also grateful for the support from my family, friends - Hari and others! All of you play a major role in supporting my Master studies"

-Harini

Abstract

Over the past few years, Cloud Computing paradigm has gained a lot of importance in both academia and industry. Cloud computing has influenced software development to a great extent by introducing new concepts such as continuous delivery and elasticity [1]. It has given rise to the term 'DevOps', by blending the working of developers and operators due to the requirement of frequent code deployments. The increasing adoption of DevOps has led to the removal of boundaries between development and operations, thereby providing a production-like staging environment right from the start of the development phase [2]. Applications running on Cloud environment generate run-time production data. Making this run-time feedback available to DevOps in an accessible format to adapt accordingly, is known as Feedback Driven Development (FDD) [3]

Among the three layers of Cloud Computing stack namely Infrastructure-as-a-Service (IaaS), Platform-as-a-Service(PaaS) and Software-as-a-Service(SaaS), 'developers' are mainly concerned with the PaaS, which allows them to focus on application development without worrying about the infrastructure details such as hardware, network, operating systems and the software environment. 'Operators' are more concerned with the infrastructure layer to monitor and provide sufficient resource provisioning and capacity planning.

Leveraging the fact that applications are hosted in the cloud, there are several metrics that can be monitored. There are also several cloud monitoring tools that can be used to monitor the performance, app usage, resource usage, and other aspects of the cloud application. Though these tools exist, the information provided by them are mostly in the form of cumbersome log messages or graphs showing a wide variety of metrics. It is not usually preferred by the DevOps nor is it efficient to manually go through and analyze this information to utilize them. Nonetheless, these logs and graphs can be useful to capture production issues that occur at scale which normally could not be captured by profilers. Continuous monitoring of metrics from these logs can be utilized to improve scalability in cloud. It can be realized by monitoring the log data and deriving scaling decisions whenever necessary.

This Thesis involves research about cloud monitoring to design and implement a platform level auto-scaler. Developers can continue to deploy new features and operators do not have to manually provision more application instances as the usage goes higher. The auto-scaler can also be configured to work with customizable metrics or a combination of metrics that is flexible to be configured. The monitoring information and scaling decisions are used to derive a correlation model between the metrics collected.

Contents

List of Figures

List of Tables

1 Introduction

Cloud computing has gained rapid growth and importance in the recent years. The fact that servers are hosted remotely rather than locally has led to innumerable small scale businesses. Start-ups no longer require huge server infrastructure to be set-up at their locations, instead they can use infrastructure provided by cloud providers such as AWS [7]. They just need to pay for the amount of resources that are actually used and this is known as the pay-per-use model [8]. This significantly reduces the initial monetary setup costs. DevOps is a very popular term in cloud computing. It refers to the reduction in the gap between developers and operators [2]. Owing to increased amount of run-time information available immediately in the cloud, DevOps benefit a lot. Developers can make use of this information to optimize the code by visualizing run-time behavior, whereas operators can utilize the information to automate resource provisioning and capacity planning.

This chapter provides us the motivation behind this topic, gives an insight into the problem statement that this thesis aims to solve, summarizes the contribution of the thesis, and finally describes how the following chapters are structured.

1.1 Motivation

Cloud computing has caused a change in software engineering. This can be attributed to the flexibility, scalability, and elasticity offered by cloud. Cloud applications are dynamically allocated resources and infrastructure based on the usage and demand of the application. When the app usage is less, resources consumed are also reduced and hence cloud consumers pay less depending on the usage. This proves not only to be elastic but also cost-effective. The elasticity is achieved by continuous monitoring of several metrics that indicate the demand at the moment, and provisioning the necessary resources to meet the monitored demand. Other cloud offerings include automatic scaling, load balancing, and integration with other services such as email services, authentication, etc. Distributed, scalable enterprise-wide applications also mandate the monitoring of metrics that aid the developers and business analysts to reason the effectiveness of their applications [9].

The metrics that are monitored vary depending on the type of the application and the level of the cloud computing stack as shown in Figure 1. For instance, the metrics such as memory consumption, CPU utilization, network bandwidth utilization are considered to be at the Infrastructure level, whereas some other metrics such as response times of methods/procedures, the number of users accessing the application, maximum number of users who can use the application simultaneously, etc. are considered to be at the platform level. Sometimes, the application level metrics depend on other primitive metrics at the infrastructure level and vice-versa.

As cloud application development has become more common, the run time monitoring metrics of the applications are easily available through several application performance monitoring (APM) tools such as Amazon cloudwatch, New Relic, etc. Though a vast amount of information is made available through APM tools to the DevOps, how often the information is utilized is still an open question. APM tools do not provide any direct accessible feedback to the developers or operators, and hence most of the DevOps do not use it. However, this run-time monitoring data could be used to provide useful analytic information such as performance hotspots that take a lot of execution time, and predictive information such as methods or loops that may become critical, even before the deployment takes place. This type of analytic and predictive feedback can be provided to the developers in their IDEs which otherwise may not be explored by the developers. This is because the information collected by APM tools are huge amount of log data that are quite tedious for manual interpretation. At the same time, on the cloud operations side, this information can be leveraged to provide automation of infrastructure provisioning and capacity planning. This technique of providing useful information from the monitoring data is known as Feedback driven development [3]. FDD that provides useful tools to the cloud DevOps is the focus of this Thesis.

1.2 Problem Statement

The cloud computing stack consists of the three layers as shown in Figure 1. The three layers of the stack coordinate and work with each other. Each of the layers represent a cloud service. The top layer of the stack can be composed of the services provided by the bottom layer [10]. The top layer, SaaS, represent cloud applications that can be consumed by end users, whereas the middle layer, PaaS, represent the software environment/platform from which the cloud applications can be developed. The cloud infrastructure layer at the bottom provide the fundamental resources and hardware (computational resources, storage etc.) necessary to create the software environment and cloud applications [10].

From the software engineering perspective, it is important to note how cloud computing impacts the DevOps (developers and operators). Based on a research conducted, there are two important issues [11].

- **Impact of cloud computing on DevOps:** Development and operations are brought much closer than the traditional software methods. Sometimes the same person acts as both the developer and operator. The cloud DevOps are forced to look into the huge amount of run time log data of the cloud applications. However, performing this manually proves to be extremely tedious. This imposes a need for automation tools for DevOps.

- **Data and tools utilized by DevOps:** The data produced by the logs include business metrics, system information, usage metrics etc. and this information can be mined to predict performance/monetary implications of the cloud application in advance. Hence it can be argued that when the operational metrics are brought closer to the DevOps, they would be able to improve the performance and cost-effectiveness of the application.
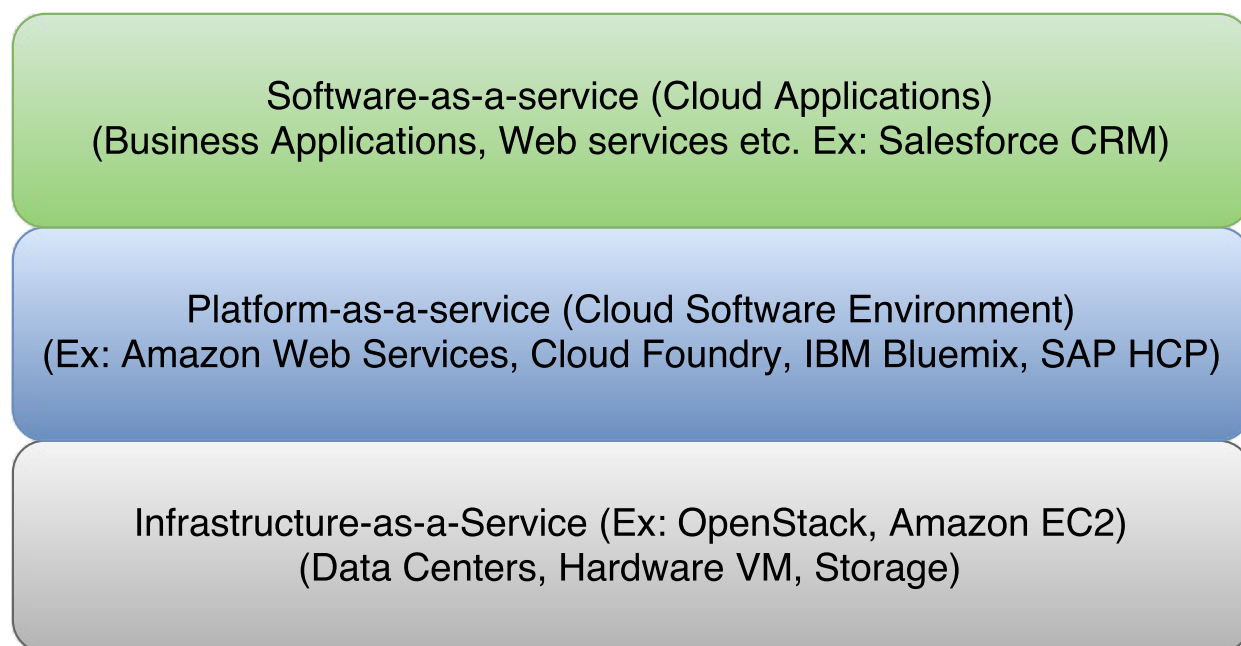
Figure 1: Cloud computing stack

1.3 Contribution

From the above mentioned issues, it is therefore important to monitor relevant information and provide efficient automated tools to the DevOps for improving cloud application performance. In this thesis, a new monitoring-scaling framework is designed and implemented. It leverages monitoring metrics such as response time of requests, throughput, CPU utilization etc. Bruneo et al. [12] define a framework that monitors the cloud defined by a 3-D cloud monitoring model, where each of the dimension correspond to the different layers of the cloud computing stack. Considering these issues, it is certainly important how these cloud monitoring information can be leveraged to make a useful impact for the cloud developers and operators.

The thesis contributes a new auto-scaling system that can be adapted to any cloud platform. It proposes a new monitoring system that collects relevant metrics to make important scaling decisions. Using the collected metrics, the design aims to provide an efficient feedback to the cloud DevOps. The collected feedback is integrated into the development environment (IDEs) so that developers are able to utilize the feedback to make their applications better scalable and highly available [3]. On the other hand, this feedback can also be leveraged to automate configuration management and to facilitate dynamic infrastructure provisioning. In this thesis, the monitoring metrics are aggregated for a sufficient period to make scaling decision. Scaling decisions can depend on a single metric or even a combination of metrics that can be customized by the cloud application developer.

The thesis also models the monitored information collected by the auto-scaler to identify a suitable correlation model. The model provides a better understanding of the relation between the collected metrics.

1.4 Structure of the thesis

The rest of the thesis is structured into five chapters.

Chapter 2 includes background information and presents the state of the art of topics related to this thesis: cloud monitoring, auto-scaling, data modeling, and feedback driven development.

Chapter 3 provides an overview of the high-level system design made in this research work. It proposes a new cloud monitoring-scaling framework for making the cloud applications scalable. It also provides a design for deriving a correlation model for the monitored metrics.

Chapter 4 explains the implementation of the proposed design on a lower fine-grained level. Interesting implementation details, the tools and technologies used, and the system footprint are also provided.

Chapter 5 evaluates the implemented system and illustrates the monitoring and scaling of a sample application. The metrics monitored are modeled and a correlation model is derived as well.

Finally, chapter 6 concludes the thesis and outlines future work ideas.

2 State of the Art

This chapter presents the state of art of the topics relevant for this thesis. We explain the properties of cloud computing such as continuous delivery, scalability, reliability and availability. We also discuss the state of the art concerning the following topics in detail: FDD, cloud monitoring, auto-scaling, data modeling, and performance analysis using source code history in evolving software.

2.1 Cloud computing basic concepts

In this section, we explain software release cycles in cloud applications. We also brief about aspects such as scalability, reliability and availability in cloud.

2.1.1 Continuous delivery in cloud development

If we compare classic software development with that of software development in cloud, we observe that there has been a huge change in the frequency of software version releases. Deployment cycles have been reduced from months to days and sometimes even within a few hours the next version is released. This process is referred to continuous delivery(CD) in the cloud computing terms [1]. This is shown in Figure 2.

Most companies make use of CD to roll out new features and evaluate their new ideas in a controlled manner [13]. CD has become a huge success and companies such as Google,Facebook etc. adopt CD of varying degrees for some of their services. When a feature is delayed for the current roll-out, it gets delayed by months in the case of traditional software development. The feature needs to be delayed until the next release. Whereas in this CD approach the next release could be in the same day or in the same week, leading to small changes of production code. This also leads to a state called perpetual development where the code is always under continuous development and there is no stable release version for a particular product.

Due to this new release paradigm, there are a lot of extra information generated. The live performance of the application, click-streams from the user interface of the app, error and warning logs, infrastructure related data etc. are produced. There are existing APM tools that collect this data and generate information out of it. Nevertheless, how this information can be made effective to the software developers or cloud operators in their daily routine is a topic that is not discussed very often.

2.1.2 Scalability, Reliability and Availability in Cloud

Some of the monitoring metrics collected include the CPU usage, response time of the request, number of instances the application is hosted on, number of requests each instance serves during
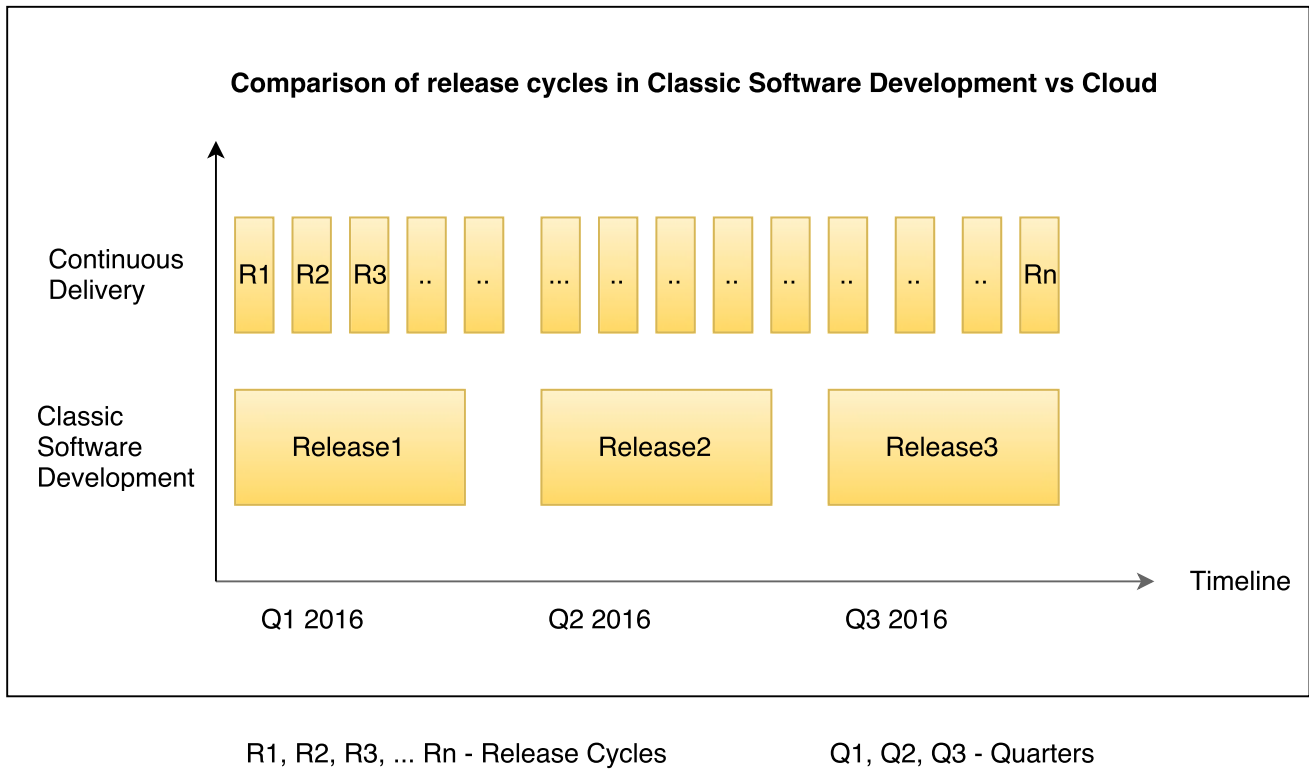
Figure 2: Continuous delivery in cloud computing [1]

a particular time period, error logs of the request etc. While these metrics focus mainly at the infrastructure level, the logs instrumented into the application are also collected. We use these metrics to focus on main challenges of cloud computing: scalability, reliability and availability of cloud applications.

Scalability is the ability to increase or decrease the resources of an instance or the number of instances so that the changing demand of the incoming requests can be met. The platform related log data collected are utilized to perform scaling. Scaling is of two types: horizontal and vertical scaling. While horizontal scaling focuses on increasing the number of instances, vertical scaling implies increasing the resources of each instance.

Scalability is one of the major advantages of cloud computing. Compared to the legacy software applications, cloud computing offers special features of elasticity and scalability. Hence customers can scale their infrastructure/application instances whenever necessary. Scaling can occur at both the platform level and the infrastructure level. While scaling at both levels proves to be important, these two are quite different from each another.

Infrastructure scaling involves adding or removing virtual machines or server nodes whereas platform scaling involves adding or removing additional instances of the application itself. It is certainly important to have both infrastructure and application scaling when a PaaS is considered. For enterprise IT services, it becomes quite important to evaluate if the application can also scale seamlessly. It may not be sufficient if the infrastructure scales while the application does not [14].

Application Instances

Platform-as-a-service

Infrastructure-as-a-Service

VM   VM   VM   VM

- - - - - - - - -   Platform scaling - addition/removal of application instances

──────►   Infrastructure scaling - addition/removal of physical/virtual machines
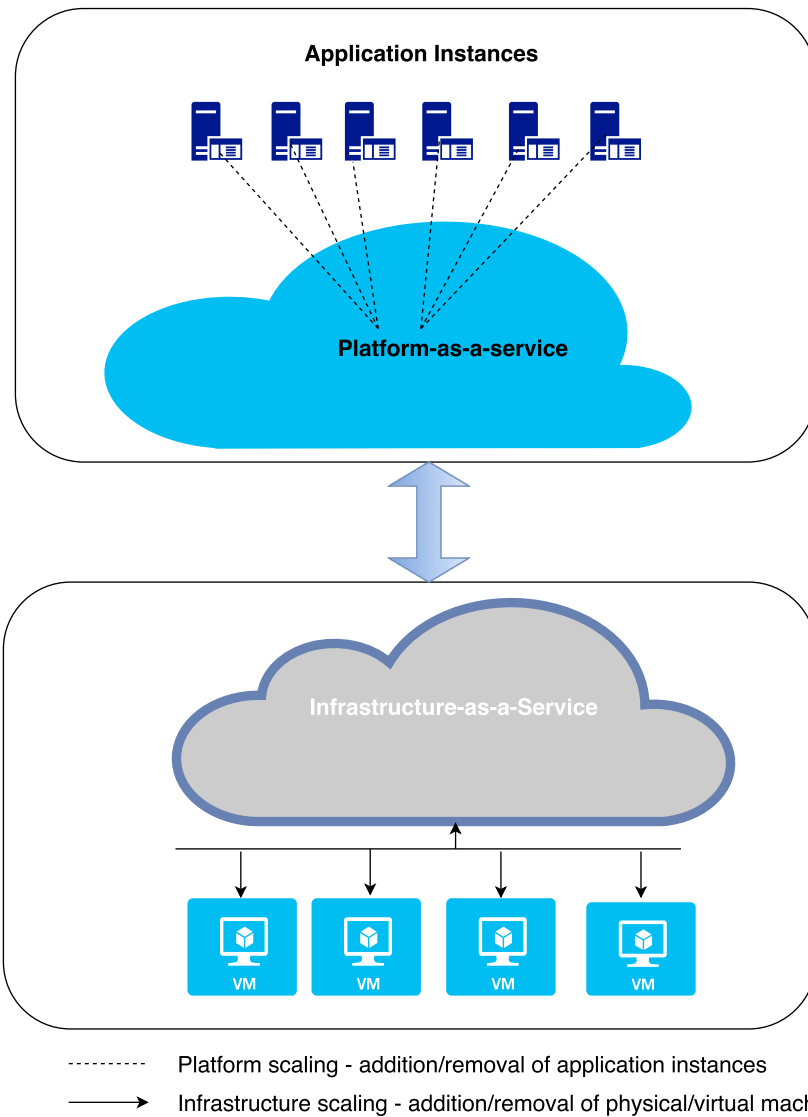
Figure 3: Scaling at two different levels: Platform versus Infrastructure

Figure 3 depicts this distinction between platform and infrastructure scaling. It displays multiple virtual machines at the infrastructure layer and multiple application instances at the platform layer. Depending on implementations, sometimes application scaling could demand scaling infrastructure as well [15]. This needs to be handled by the PaaS.

Availability is one of the major goals of cloud computing. It means that the cloud services need to be available and accessible at anytime from anywhere. For business to happen continuously, it is necessary for the services to be highly available. The definition of availability is specified by the different cloud vendors in their SLA. For example, google search is known for its high availability.

2.2 Feedback Driven Development

By analyzing any cloud application's logs, a huge amount of information is gathered. The logs can be broadly classified into application level logs and infrastructure level logs. This data

can be useful to both the developers and operators. Infrastructure logs provide details such as number of instances, memory, CPU, disk utilization, and which instance serves a particular request etc. By making this kind of data visible to the developers, they can tweak the application development process, as they have access to the run time information. At the same time, cloud operators also benefit by having access to relevant business metrics. They can use this to manage the instances more efficiently.

Collecting these run-time data, aggregating them into useful feedback, and feeding them back into the development process of an application can create a useful impact in the future deployment of the application. This process is known as Feedback Driven Development. FDD can be classified into 2 types: Analytic FDD and Predictive FDD [3].

- *Analytic Feedback Driven Development:* Analytic FDD is the run time data from previous deployments, which is brought directly into the developer environment. It provides a mapping between the log data collected and the source code artifacts. This helps the developers to understand how run-time metrics directly impact the source code. Developers can utilize this to alter and optimize the code based on real time user behavior. In practice, Analytic FDD deals with visualizing run time operations data and how it is being mapped to code artifacts.

- *Predictive Feedback Driven Development:* Predictive FDD utilizes run-time feedback to warn the developers about current code changes even before the updated source code is deployed. Predictive FDD is combined with static code analysis to give better predictions regarding a code change.
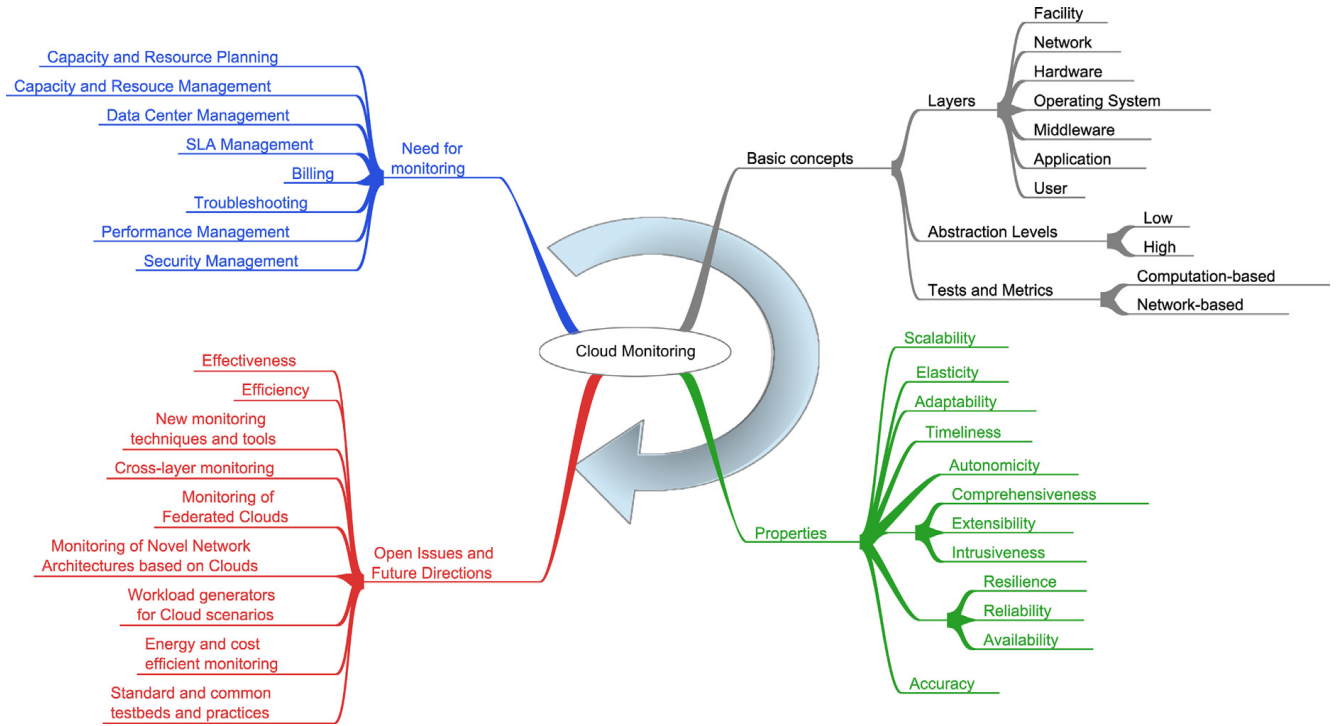
2.3 Cloud Monitoring

As cloud computing is gaining popularity, the need for cloud monitoring is becoming increasingly important to both the cloud providers and the cloud consumers. At the cloud provider side, cloud monitoring is the key principle behind which the actual controlling of hardware takes place. It enables them to scale the infrastructure, if necessary. Cloud consumers are the users of the cloud. Cloud monitoring enables consumers to check the availability, QoS etc. of the applications. The consumers can verify any SLA violations by comparing the Key Performance Indicator(KPI) parameters provided by cloud monitoring.

Aceto et al. [4] explain in detail about the need for cloud monitoring listing the basic concepts involved in monitoring, the properties that need to be maintained for monitoring, and finally also lists down the open issues with respect to cloud monitoring. These are summarized in Figure 4. The basic concepts include the layers such as network hardware etc., and some of the properties were discussed in section 2.1. The need for monitoring mostly focuses on capacity planning, infrastructure adaptation and metering for billing purposes.

Cloud monitoring platforms are those tools which are provided by the cloud provider. There are also third party vendors providing web services that can be used to monitor cloud applications and these are called cloud monitoring services. Some examples of cloud monitoring platforms and services are CloudWatch [16] [17], AzureWatch [18] , NewRelic [19] etc. Table 1 provides a list of cloud monitoring platforms and services. Amazon CloudWatch provides users

Figure 4: Cloud monitoring [4]

the monitored information for 2 weeks. Users are allowed to plot these information, set thresholds, alerts etc and these alerts can be used to perform any substantial action such as sending an email or even in auto-scaling [20]. Auto-scaling is explored in detail in the next section.

Table 1: Cloud monitoring platforms and services

Cloud Monitoring Platforms	Cloud Monitoring Services
CloudWatch [16] [17]	New Relic [19]
Nimsoft [21]	Cloudyn [22]
AzureWatch [18]	Up.time [23]
Nagios [24]	CloudSleuth [25]
Nimbus [26]	Cloudstone [27]
GroundWork [28]	Boundary [29]
LogicMonitor [30]	Cloudfloor [31]
CloudKick [32]	CloudClimate [33]
Monitis [34]	CloudHarmony [35]

Some other cloud monitoring platforms such as Nimsoft monitoring solution [21] provide a unified monitoring dashboard to view infrastructures provided by Salesforce, Rackspace, Google or Amazon. Nagios [24] is a popular open source cloud monitoring platform that provides monitoring of virtual machines and storage (Amazon EC2 and S3). It also supports OpenStack [36], an open source cloud IaaS. New Relic [19] is a web-based monitoring service that helps to monitor the application infrastructure and performance, adhering to timeliness, resilience, availability and accuracy. Bruneo et al. propose a 3D-Cloud Monitoring framework called the Ceiloesper framework, which combines monitoring in multiple layers with real time data and it

also performs the data analysis for multiple management actions [12]. It is based on CEP and uses the Esper CEP engine.

2.4 Auto-scaling

Scaling of cloud infrastructure means adapting the current infrastructure depending on the demand and usage. It is of two types: horizontal and vertical. Horizontal scaling is a methodology of adding or removing machines whereas vertical scaling is increasing or decreasing the resources such as CPU/Memory/Disk to existing machines.

Auto-scaling is a process where the cloud platform adapts itself by increasing or shutting down the number of instances on which the application is currently deployed depending on the current load. For enterprises running their applications in cloud, auto-scaling could lead to saving costs due to the pay-per-use model of the cloud [8]. Auto-scaling also improves the efficieny of applications. Bunch et al. mention that auto-scaling improves the instance utilization of the open source AppScale PaaS by 91% and it also brings down the average time taken to serve the requests [37]. Auto-scalers can be broadly classified as the following:

- **Reactive Auto-scaling:** Auto-scaling as provided by most of the cloud providers such as AWS, Microsoft Azure, IBM Bluemix etc. are reactive. It is achieved by monitoring relevant metrics. Whenever a certain metric increases or decreases beyond a particular predefined threshold, additional instances are added or removed. This method which is more of a rule-based mechanism is a reactive auto-scaling method. This is easier to be implemented as it involves monitoring metrics, and framing rules and policies for scaling.

  Seelam et al. explain the rule based reactive autoscaler of IBM's Bluemix PaaS, which is known as the Polyglot application [38]. Polyglot autoscaler allows application developers to set thresholds based on which instances need to be added (scale-out) or removed (scale-in). These threshold values can be parameters such as CPU Utilization, memory and heap usage. Polyglot consists of the four components: agents which collect the performance information, a monitoring service which continuously monitors the health of the cloud application, a scaling service which makes the decision of whether scaling needs to be performed or not, and a persistence service to keep track of the enactment points (points where the application is scaled in time). While reactive scaling serves in most scenarios, the question arises whether it is capable to handle bursty traffic.

- **Predictive Auto-scaling:** Predictive auto-scaling comes handy to handle bursty workloads. Bursty traffic is a situation where a sudden unexpected number of users access the application, which maybe triggered by a social media campaign. By analyzing the historic time series data, it may be possible to predict the workload at a future time, thereby enabling predictive auto-scaling. The effectiveness of this method depends on the efficiency of the workload prediction.

  Biswas et al. [39] introduce a predictive auto scaling technique that uses a machine learning engine to make predictions based on a deadline driven algorithm for predicting the future state of the system. Netflix [40, 41] describes a predictive auto-scaling tool, Scyer, used by Netflix to provision the correct number of AWS [7] instances. This is different

from the AAS [20], which is a reactive one. Scryer's prediction engine is able to provision the resources based on two prediction algorithms to predict the workload. The prediction algorithms implemented are augmented linear regression based algorithm and fast fourier transformation based algorithm.

- **Hybrid Auto-scaling:** Hybrid auto-scaling is a combination of both the reactive and predictive approaches. As explained by Netflix [40], Scryer tool works in co-ordination with the AAS for more efficient auto-scaling. Moore et al. [42] describe the architecture and implementation of platform insights, which is another hybrid auto-scaler that employs a reactive rule-based and a predictive model-based approach in a coordinated manner.

Design and implementation of auto-scalers face challenges as well. Lorido-Botran et al. explain the following problems that auto-scalers face and how they can be solved [43]:

Under Provisioning: The application is hosted on lesser infrastructure than that is necessary to process all the incoming requests. Due to SLAs, it takes a while for it to reach up to the required amount of infrastructure. This can also lead to SLA violations.

Over Provisioning: There is no SLA violations in this scenario. However the actual amount of resources is greater than the required amount of resources and hence the customer could be paying extra cost that his actual usage.

Oscillation: When there is an oscillation between under provisioning and over provisioning it causes an undesirable and unstable state.

MAPE loop is a solution proposed to solve the problems listed above [43]. MAPE stands for Monitor, Analyze, Plan and Execute. The necessary monitoring metrics are collected and analyzed to decide on the type of auto-scaling: reactive/predictive/hybrid. The planning phase is done on how to actually perform the scaling: horizontal/vertical. Finally the actual scaling is performed based on SLA configurations.

2.5 Data Modeling

Correlation and covariance indicate how closely two variables are related to each other. Correlation may be either positive or negative as shown in Figure 5 and Figure 6 respectively. If variable Y increases proportionately when variable X is increased by a unit, it is known as positive correlation. On the other hand if the variable Y decreases proportionately when variable X is increased it is a negative correlation. If all the points are centered around the straight line: Y = X, then X and Y are said to be positively correlated. Whereas if all the points are centered around a line Y = -X, then X and Y are said to be inversely correlated. If all the points are scattered throughout then there is no correlation between variables X and Y. This can be explained graphically as shown in Figure 7. If X_i and Y_i are sample data for the two variables under consideration then correlation can be calculated as: $Correlation, r_{xy} = S_{xy}/S_x S_y$ where S_x = sample standard deviation of variable X, S_y = sample standard deviation of variable Y and S_{xy} is the sample covariance of the variables X and Y [44].
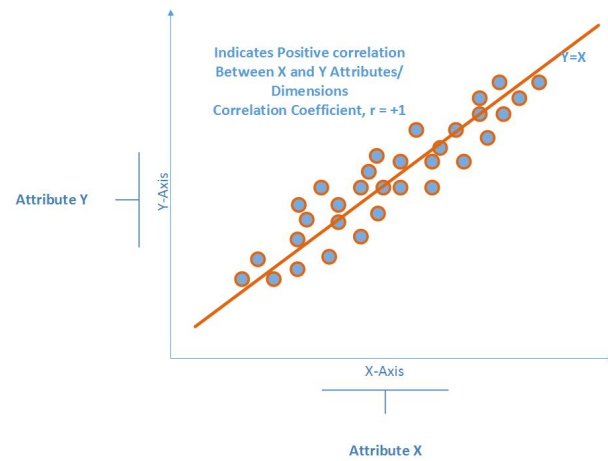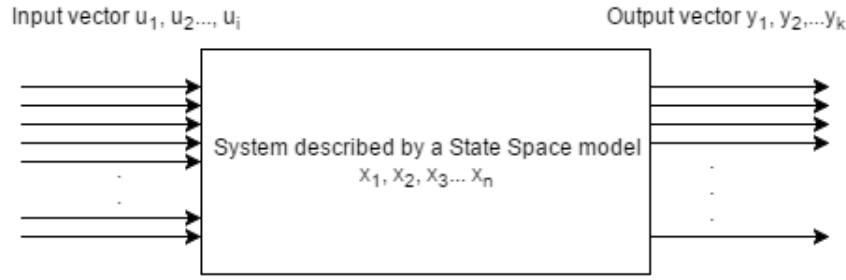
Figure 5: Positive correlation
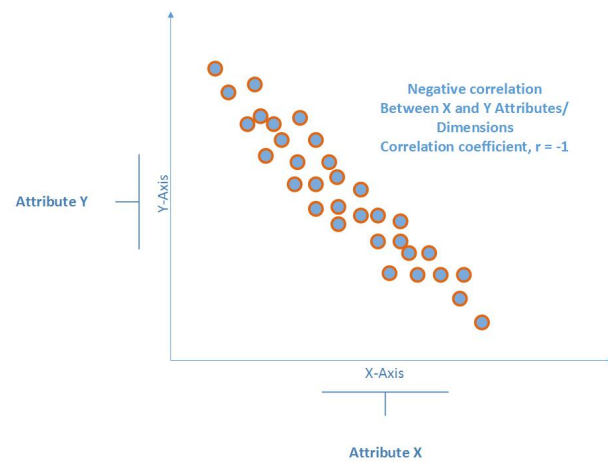
Figure 6: Negative correlation
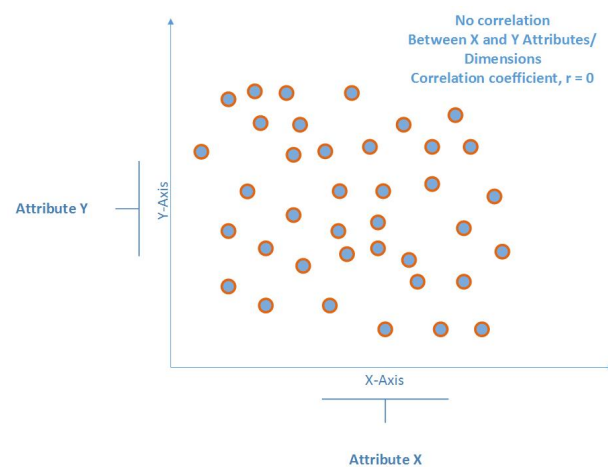
Figure 7: No correlation

Figure 8: System represented by State Space Model

So far only single input variable, X and a single output variable, Y, was considered. However in reality most of the systems tend to be multiple-input multiple-output (MIMO) systems rather than the single-input single-output (SISO) system. The data that is dealt in real world does not contain just 2 attributes. Most of the real world scenarios involve a minimum of 5 to 6 dimensions and depending on the applications this may go as high as 20 or even more [45, 46]. Hence it is important to explore multivariate correlation models: state space models, and polynomial models [47–49].

2.5.1 State Space Models

State space model represents a system by a set of first order differential equations and state variables. The output Y(t) of a system at time, t, can be predicted for any time t $> t_0$, where t_0 is an initial time, provided that the input and output of the system at time t_0 and a minimum set of variables $x_i(t)$ where i = 1 to n, are known. In this case, n is the order of the state space model [47].

Figure 8 shows a system described by a state space model. The vector u_1, u_2, u_3,...,u_i are the inputs while the output vector is y_1, y_2,....,y_k. By knowing the inputs and outputs at time t_0 the state variables: x_1, x_2, x_3,....,x_n are first measured. Then it becomes possible to predict the output at any future time, t by knowing the inputs at that time and the measured state variables.

In state space modeling, the time derivative of the state variables are represented as a function of the state variable, and inputs, $dx/dt = f(x,u,t)$. Considering a LTI system, the state equation can be written as [47]:

$$dx/dt = Ax + Bu$$

where A and B are matrices with constant coefficients that weigh the system's state variable and inputs respectively. Similarly the output equation can be written as [47]:

$$y = Cx + Du$$

where C and D are matrices with constant coefficients that weight the system's state variables and inputs respectively. There are several physical systems where the D matrix is found to be a null matrix thereby reducing the output equation to $y = Cx$, where the output depends on a weighted combination of the state variables.

2.5.2 Polynomial Models

Some physical systems do not always adhere to linear equations. Hence to model these type of systems, polynomial model are considered. Additionally there could be systems that depend on values of previous inputs and outputs. Based on these, the following four polynomial models are considered [48, 49]:

1. ARX Model: The ARX model to evaluate the output is based on auto-regression. Auto regressive model is a model whose current output depends on the past input and output values. The generic notion to denote auto-regressive model of order p, AR(p) for a variable X is:

$$X_t = c + \sum_{i=1}^{p} \rho_i X_{t-i} + e(t)$$

where c and ρ_i are constants and e(t) is the noise. Considering auto regression and the inputs, ARX model can be mathematically described as:

$$A(z)y(t) = B(z)u(t-n) + e(t)$$

where y(t) is the output, u(t) is the input, and e(t) is the noise/error measured in the output. A(z) and B(z) are polynomials of the specified order with respect to the backward shift operator z^{-1}. For example, $z^{-n}u(k) = u(k-n)$ [49].

2. ARMAX Model: Unlike the ARX model, in ARMAX, the stochastic dynamics are considered. Therefore this model handles a system where there is a domination of noise. ARMAX models are better for systems with more disturbances. In general, the moving average model of order q, MA(q) is represented in the below notation:

$$X_t = e(t) + \sum_{i=1}^{q} \theta_i e(t-i)$$

where θ_i are constants and e(t) and e(t-i) are the noise/errors. The notation for the auto-regressive moving average(ARMA) model is as below:

$$X_t = c + e(t) + \sum_{i=1}^{p} \rho_i X_{t-i} + \sum_{i=1}^{q} \theta_i e(t-i)$$

This model includes both AR(p) and MA(q) models. Based on these the following mathematical equation for the ARMAX model can be written as:

$$A(z)y(t) = B(z)u(t-n) + c(z)e(t)$$

where, y(t) is the output, u(t)is the input, and e(t) is the noise. A(z), B(z) and C(z) are polynomials of specified orders with respect to the backward shift operator z^{-1} [49].

3. Output-Error Model: The notation for the Output Error model is as below:

$$y(t) = [B(z)/F(z)]u(t-n) + e(t)$$

where, y(t) is the output, u(t)is the input, and e(t) is the noise. B(z) and F(z) are polynomials of specified orders with respect to the backward shift operator Z^{-1} [49].

4. Box-Jenkins Model: The notation for the Box Jenkins model is as below:

$$y(t) = [B(z)/F(z)]u(t-n) + [C(z)/D(z)]e(t)$$

where, y(t) is the output, u(t)is the input, and e(t) is the noise. B(z), F(z), C(z) and D(z) are polynomials of specified orders with respect to the backward shift operator Z^{-1} [50].

2.6 Performance analysis using source code history in evolving software

Software evolution is defined as the change of characteristics of a software in time. CD has led to continuously evolving software. This means frequent code changes occur and this could cause performance regressions. Performance of a software is quite important and hence evaluating performance regressions during code changes becomes a necessity. A performance regression can be defined as a state when the application under consideration behaves worse in a new code deployment compared to its previous deployment. In order to identify the root cause of regressions, the source code has to be analyzed, especially those parts that were added/removed/changed in the new deployment. This is called source code mining. In this section, two source code mining tools are discussed: PerfImpact [5] and LITO [6].

1. PerfImpact: PerfImapct identifies the performance regressions and recommends potential code changes that has led to the performance degradation. PerfImpact achieves this as a two step-process:

   - **Identification of inputs that cause the performance regression:**
     PerfImpact defines a fitness function that determines the inputs which cause the delay in execution of a newer code deployment V_{i+1} compared to its previous deployment V_i. The fitness function makes use of genetic algorithms to achieve this.

   - **Mining execution traces to identify code changes that lead to performance regressions:**
     PerfImpact also has a mining function, which identifies those methods that took a longer execution time in V_{i+1} compared to V_i. These methods are tagged as potentially problematic methods. Between the two deployments there could be several code changes/commits. Each code change is ranked based on the number of potentially problematic methods involved. The code changes with higher number of problematic methods are ranked higher and considered as the possible root cause for the performance regression.

   PerfImpact was evaluated on two open source web applications: JPetStore [51] and Agilefant [52]. Figure 9 shows the source code changes in two versions of Agilefant. Agilefant

```
                            V3.2
public String generateTree(){
    Set<Integer> selectedBacklogIds = this.getSelectedBacklogs();
    if(selectedBacklogIds == null || selectedBacklogIds.size() == 0) {



        addActionError("No backlogs selected.");
        return Action.ERROR;
    } ......
    return Action.SUCCESS;
}
```

```
                            V3.5
public String generateTree(){
    Set<Integer> selectedBacklogIds = this.getSelectedBacklogs();
    if(selectedBacklogIds == null || selectedBacklogIds.size() == 0) {
        Collection<Product> products = new ArrayList<Product>();
        productBusiness.storeAllTimeSheets(products);
        for (Product product: products) {
            selectedBacklogIds.add(product.getId());
        }
    } ......
    return Action.SUCCESS;
}
```

```
                            V3.2
public StoryTreeBranchMetrics calculateStoryTreeMetrics(Story story) { ......
    for(Story child : story.getChildren()) {



        StoryTreeBranchMetrics childMetrics = this.calculateStoryTreeMetrics(child);
    } ......
    return metrics;
}
```

```
                            V3.5
public StoryTreeBranchMetrics calculateStoryTreeMetrics(Story story) { ......
    for(Story child : story.getChildren()) {
        if (child.getId() == story.getId()) {
            continue;
        }
        StoryTreeBranchMetrics childMetrics = this.calculateStoryTreeMetrics(child);
    } .....
    return metrics;
}
```

Figure 9: Source code changes of two versions in Agilefant [5]

Table 2: Code Changes that caused maximum performance variations [6]

	Source Code Changes	R	I	R/I	Total
1	Method call additions	23	0	1	24 (29%)
2	Method call swaps	15	9	0	24 (29%)
3	Method call deletion	0	14	0	14 (17%)
4	Complete method change	6	0	3	9 (11%)
5	Loop addition	5	0	0	5 (6%)
6	Change object field value	2	0	0	2 (2%)
7	Conditional block addition	0	2	0	2 (2%)
8	Changing condition expression	0	2	0	2 (2%)
9	Change method call scope	1	0	0	1 (1%)
10	Changing method parameter	0	1	0	1 (1%)
	Total	**52**	**28**	**4**	**84 (100%)**

is a web application that is a lean transformation tool to execute changes faster. The evaluation shows that the inputs which cause performance regressions are identified efficiently. PerfImpact also lists the potentially harmful code changes that can be used further in code inspectors and root cause analysis.

2. LITO, a horizontal profiling technique: Software profiling is a type of program analysis that estimates the space and time complexity of a software. LITO uses horizontal profiling, which is a sampling technique to identify source code versions that cause performance regressions [6]. LITO is a cost model to determine if a code commit has caused performance regressions based on sampling the execution of versions. This approach resolves the following research questions (RQs) as below:

- RQ-1: Is there a set of specific methods which will cause performance variations when the source code of these methods are modified? Sandoval et al. state that this is not really true [6]. This is in contrast to PerfImpact. This approach was tested on 17 open

source projects and the results showed that the methods, which cause performance variations before, not necessarily contributed to the performance variations in the newer versions.

- RQ-2 What are the recurring code changes that affect the performance of an evolving software? The major code changes the caused performance variations are method call addition, method call deletion, method call swap, complete method call change, and loop addition as compared to the other code changes listed in Table 2 [6].

2.7 Background Summary

This chapter briefly described the important topics relevant for this thesis. It briefed about FDD and importance of scalability in cloud and described the differences of scaling in PaaS and IaaS. It also provided details of existing monitoring frameworks. Based on this, a new system is proposed to identify relevant cloud monitoring metrics that can be used to automatically perform scaling at the platform level. The metrics are collected and modeled to find a suitable correlation model. State space models and polynomial models are estimated for the collected metrics.

3 Design and Architecture

In this chapter, we explain the overall architectural design and the three major components of the proposed solution, which are cloud monitoring, auto-scaling, and data modeling. Under cloud monitoring we discuss the choice of the metrics collected and how the metric collection is performed and persisted. Under auto-scaling, we describe about how the auto-scaler leverages the monitored data to make the relevant scaling decisions. We then model the persisted data to find a suitable correlation between the collected metrics and describe this process under data modeling.

First, we deploy a sample application to the cloud and monitor it continuously. We utilize the monitoring metrics of the application for data modeling. Based on the monitoring service, we design an auto-scaler. The auto-scaler aids the applications hosted in the cloud to seamlessly scale-out and scale-in depending on several parameters. The cloud application monitoring has paved way for several adaptations both in terms of application (source code changes) and the infrastructure. As mentioned about FDD in Chapter 2, application level adaptation focuses on root cause identification and source code changes. In this design, we efficiently utilize the monitoring metrics to perform infrastructure adaptation. This design focuses in deriving suitable enactment points where necessary scaling decisions are taken.

3.1 System Architecture

The overall architecture of the proposed system is depicted in Figure 10. The cloud infrastructure layer is present at the bottom layer and the platform layer above the infrastructure. In this thesis, openstack infrastructure and Cloud Foundry(CF) platform are used and an application is hosted in the cloud. The platform provides APIs to collect the application logs, infrastructure logs and the application usage information. This data is accessible in the form of logs or REST APIs. The CF API can be accessed using REST calls [53].

As shown in the Figure 10, two of the components named monitoring service and scaling service are implemented as services that could be provided by the platform. The monitoring service collects the information from logs and API, parses them and persists the information in a database. The scaling service also makes use of periodic monitoring data to make the scaling decisions. The load generator is used to generate load on the application to analyze the behavior of monitoring metrics with varying load. Finally, the data modeling component that retrieves the monitoring metrics from the database and performs data modeling is present.

The three major components of this design: monitoring service, scaling service and data modeling are discussed in detail in the following sections. The component interaction among the three components are depicted in Figure 11. Once the app is deployed to the cloud, the monitoring component continuously collects logs and metrics. It aggregates the metrics and
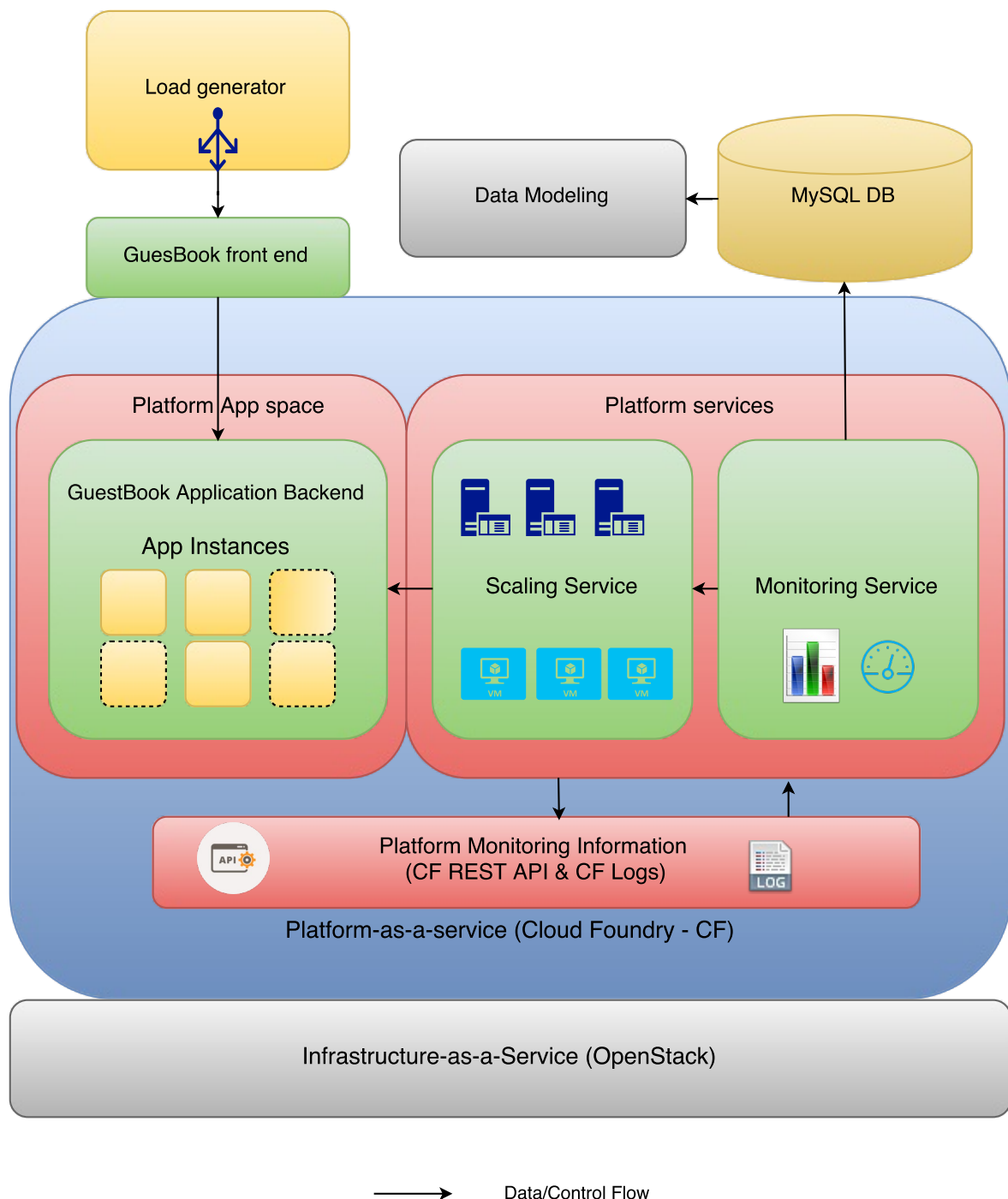
Figure 10: System design
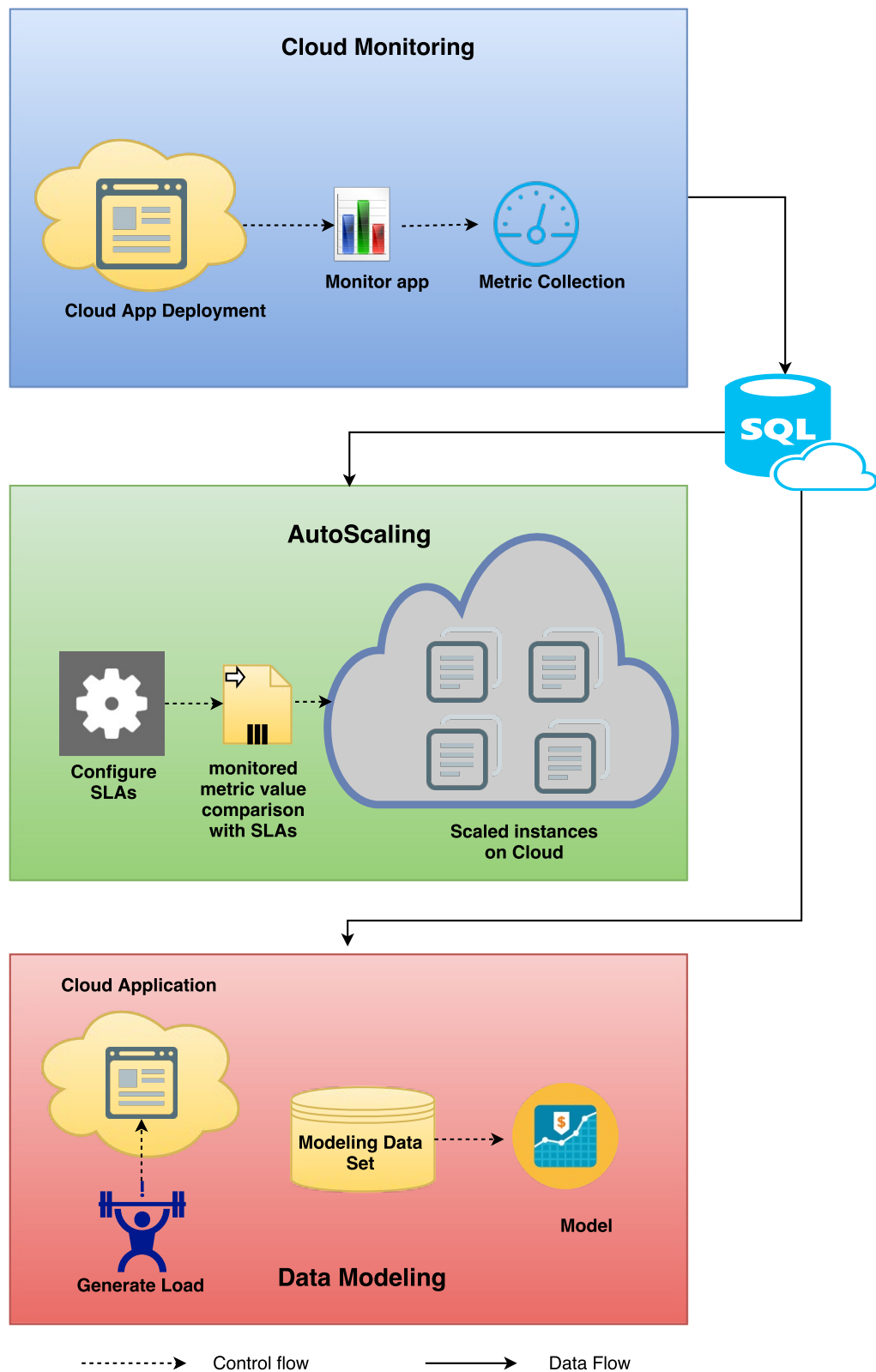
Figure 11: Interaction between components of the proposed design

persists them in a database. The second component of Figure 11 retrieves the information and compares it with SLAs to make a decision and perform scaling, when necessary. The third component, data modeling, collects the data from the database to derive a correlation model. A load generator is used to generate usage on the app and to monitor the app behavior under varying load.

3.2 Monitoring Service

The first component of the architecture is the monitoring service. A sample application is deployed to the cloud and it is monitored for certain metrics related to performance and load such as response times, throughput, CPU utilization etc. We leverage the fact that the application is on the cloud to get the instantaneous real-time metrics of the application.

Having access to the current run-time metrics could be useful in a wide range of scenarios that benefit both the development activities such as proactive production bug identification, bug fixing, and operational activities such as scaling infrastructure. Alerts can also be set-up to notify the involved stakeholders when a specific metric crosses a predefined threshold value. The metrics are monitored and persisted to be utilized by the other components. These metrics include information such as response time of requests, throughput, the CPU % utilization, the memory % utilization, the disk % utilization etc. Once the application is hosted in the cloud, the monitoring service is started. The monitoring service collects the required metrics from the platform log data and the platform APIs. These metrics are aggregated to be consumed by the other components.

Monitoring happens continuously in time to provide its services and capabilities to other components such as scaling and metering. Monitoring is also useful for the cloud providers to meter the usage of application so that Customers are charged accordingly. The second component, scaling service, makes use of the monitoring service's data to make the scaling decisions. This is depicted in Figure 12.
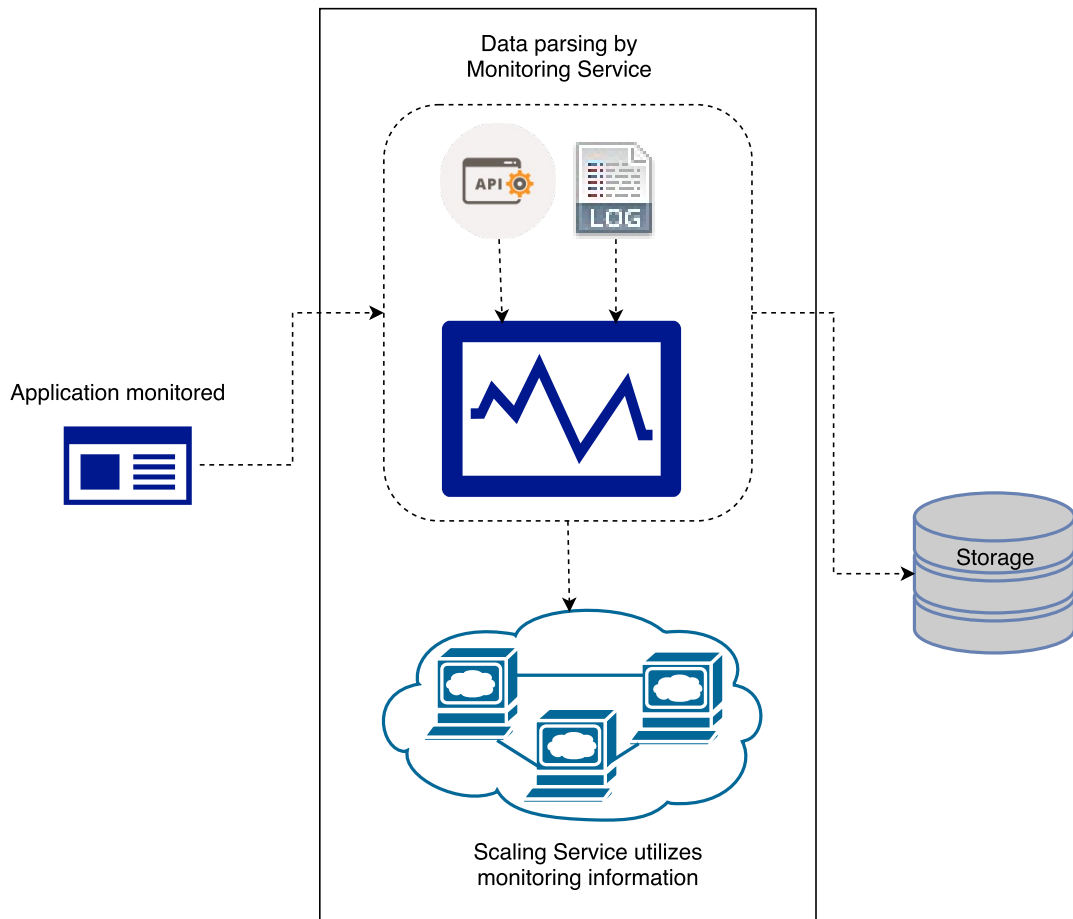
3.3 Scaling Service

The second component of the design is the scaling service. As mentioned in section 2.1, scaling at the infrastructure level is different from the scaling at the platform level. In this thesis, the scaling service performs its actions at the platform level, i.e. it adds or removes application instances. The cloud application developer can set scaling policies and rules. The scaling service takes into consideration these rules. It compares the metric values collected by the monitoring component and makes the scaling decision based on the policies. Monitoring happens continuously and the scaling decision happens once in every specified interval knows as the **cool-down-period**.

The monitoring data within the last cool-down interval is retrieved from the database to make the scaling decision as shown in the Figure 13. The cool-down period is to avoid any oscillations in the metrics due to under-provisioning and over-provisioning of resources. If scaling happens continuously, it may lead to undesirable oscillations. In order to avoid the oscillations, a predefined period called the cool-down-period is defined, which provides some time for the system to stabilize after scaling occurs.

**Figure 12:** Component interaction between monitoring and scaling service

This auto-scaler performs scaling of application instances. The cloud providers have an agreement with the cloud consumers regarding certain values such as the minimum and maximum number of application instances and domain experts can specify minimum and maximum threshold values of scaling metrics. The values are specified in SLA's and policies.

The scaling service further aggregates the collected metrics information by the monitoring service. It utilizes the metrics obtained in the last cool-down period time slot. The final decision is made depending on the aggregated metric values during the latest cool-down period as shown in the Figure 13. These values are compared against the values specified in the SLAs/policies. Both scaling out and scaling in capabilities are performed by this service.

Both the monitoring and scaling components persist the information in a database. This is utilized by the data modeling component and for future referencing of enactment points. Enactment points in cloud computing are those points in time where an adaptation at the infrastructure/platform occurs. The flow chart of the scaling service is shown in Figure 13.

3.4 Data Modeling

The third part of the design focuses on modeling the monitoring data. In this step, load is generated on the application and the collected monitoring data is used to derive a model. The
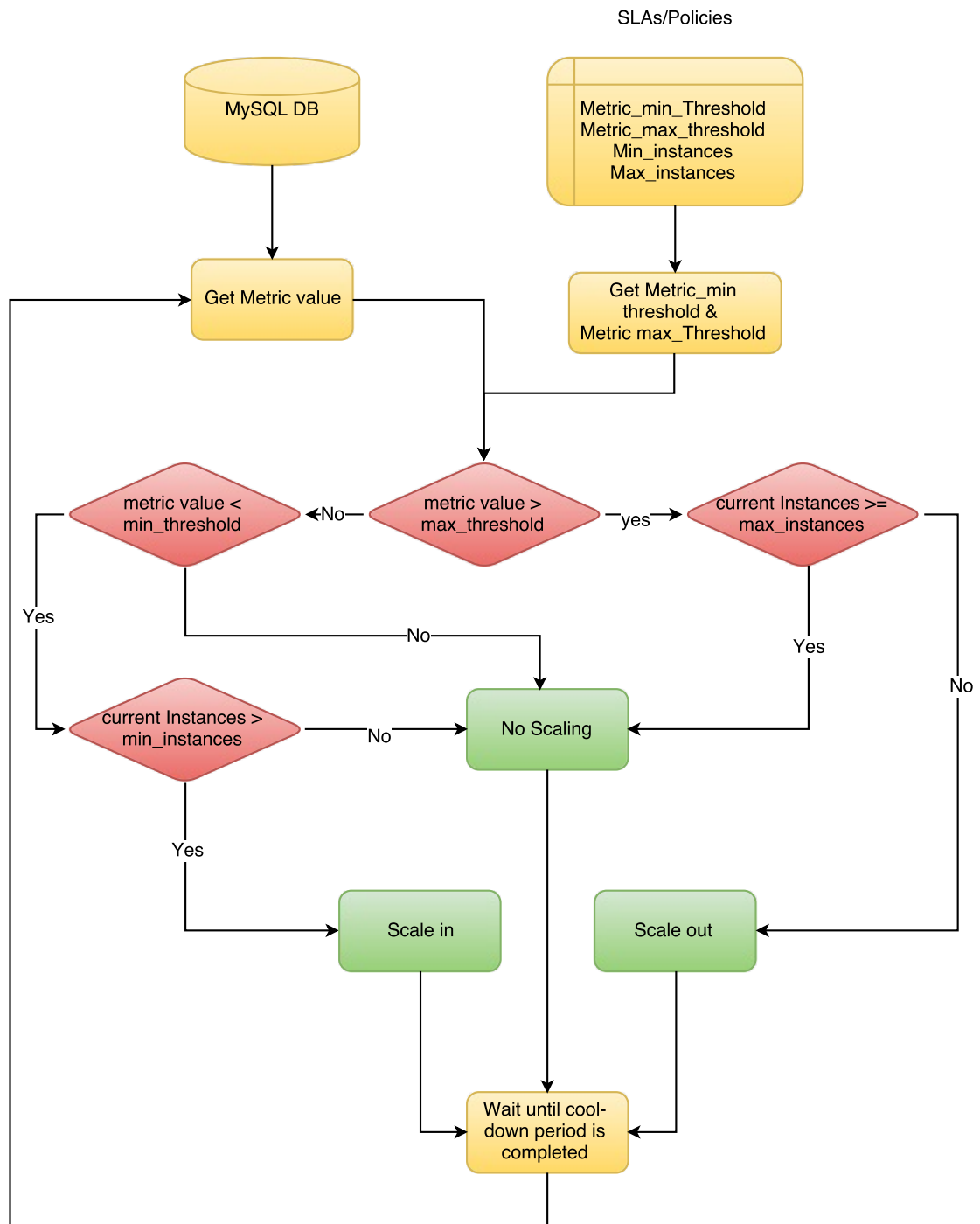
Figure 13: Decision process of the auto-scaler

model identifies a correlation between the metrics collected. In this scenario, multiple data dimensions are available and hence multi-variate modeling is considered. The models that are discussed in section 2.5 are considered for estimation.

Identification of this correlation may be used to predict the future values of some metrics which proves to be extremely useful to adapt the infrastructure based on predictions of the model. It also gives rise to newer methods of root cause identification of production issues using FDD. Mining the large amount of production data is definitely challenging.

The data collected by the monitoring service of the auto-scaler is imported to perform the data modeling. Data modeling is broadly classified into three phases: data pre-processing, estimation phase and the validation phase.

3.4.1 Phase 1: Data pre-processing

Preprocessing involves tasks such as removal of outliers or error data, data conversion steps, choosing specific data range, etc. The dataset is split into estimation data and validation data. As per the 80-20 rule, the data is first roughly split into two parts. The first portion consists of about 80% of the data and this data is used for model estimation. The remaining 20% of the data is used to validate the model. The 80-20 rule is not a standard one and it can be varied depending on the application, but it is a good decision to start with. Sometimes, better model accuracy could be obtained by splitting the dataset in a different manner. The total flow of the data modeling is depicted in Figure 14.

3.4.2 Phase 2: Estimation phase

The second phase of data modeling is known as the estimation phase or the learning phase. During this phase the training data set is available. The inputs and outputs of the training data set is used. The system learns the correlation between inputs and outputs. When more data points are available, better learning is achieved. A modeling tool estimates the best fit of the data points to a model. The estimation data may not always fit into a linear regression model. Sometimes, non-linear models need to be considered. In this thesis, State space and Polynomial models are estimated.

State space model estimation is a simple but powerful technique. The user needs to provide just one parameter: the model order. Choosing the optimal model order is important. The model order determines the size of the state variable vector, x_i. The model estimation determines the values of the state variables using the estimation data inputs and outputs. Once the state variables are determined, the model can estimate the output at anytime later using the inputs at that time.
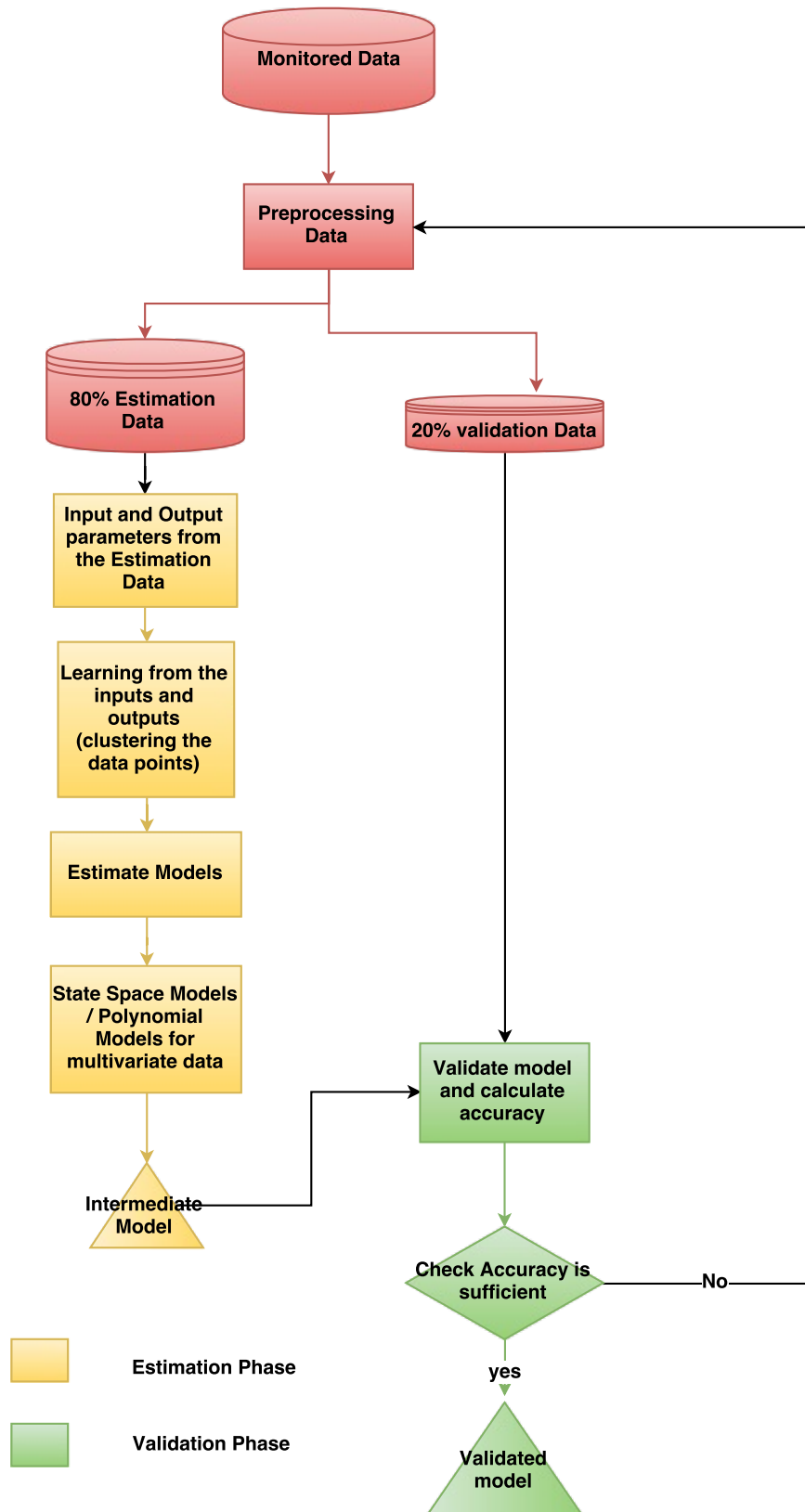
Figure 14: Validating the model

Polynomial model estimation requires to specify the orders of the polynomials. Depending on the type of the polynomial model: ARX, ARMAX, Output-Error or Box-Jenkins, the orders of the corresponding polynomials are specified. During the estimation phase, the coefficients of the corresponding polynomials are estimated.

3.4.3 Phase 3: Validation phase

The estimated model needs to be validated. It is validated against the validation dataset. The accuracy of the model is calculated by comparing the output values estimated by the model with the actual output values of the validation data set. The data modeling process maybe repeated by using different datasets, by varying parameters such as model orders etc. to derive a model which is accurate enough. The desired accuracy may vary depending on the application.

4 System Implementation

This chapter briefs about the implementation of the proposed design in the previous chapter. The implementation and evaluation is performed on a cloud application hosted in SAP's HANA Cloud Platform. This chapter also briefly explains about deploying the application in HCP, implementing the monitoring-scaling framework, and modeling the collected metric information.

4.1 System Footprint

The implementation of the design is on SAP's HCP [54]. It is very convenient to build new applications or extend existing applications on top of the HCP. HCP is a PaaS that provides unique in-memory databases and application services. The HCP platform is built based on the open source PaaS, CF. CF platform is deployed using the bosh tool on top of OpenStack infrastructure. Bosh is an open source tool for deployment of CF on top of any IaaS provider. It is also useful for reverse engineering and distributed systems monitoring.

The loggregator component of CF is responsible for logging. Loggregator collects all the logs from both the application and the CF system components, which interacts with the application during execution. These CF logs and API's are used to collect and aggregate the metrics.

The monitoring and scaling services are developed as a java service. The monitoring data collected are persisted in the MySQL database. This data is modeled by using MATLAB's System Identification Toolbox. Table 3 summarizes the system footprint of this implementation.

Table 3: System footprint

System Property	Details
Cloud Platform	Hana Cloud Platform - Cloud Foundry
Cloud Infrastructure	OpenStack
Database	MySQL
Data Modeling Tool	MATLAB System Identification Toolbox
Autoscaler (Monitoring and Scaling services)	Java
Sample Cloud App	HTML and Java

4.2 Deployment of Cloud Application

Among the three components of the proposed design, the cloud monitoring is the first component. In order to perform cloud application monitoring, a suitable cloud app needs to be developed and deployed to the cloud. In this implementation, the cloud app is deployed to the HCP based on CF PaaS, which runs on OpenStack [36] infrastructure. In order to deploy the app,

the developer logs into CF using correct credentials and API end point, which is the cloud controller URL of the CF instance. $cf\ login\ [-aAPI-URL]\ [-u\ USERNAME]\ [-p\ PASSWORD]$. The app is then deployed to CF using the following command: $cf\ push\ APP$.

The CF push command: $cf\ push\ [APPNAME]$, reads the manifest.yml file of the application to be deployed. It takes the application name, instances, memory, buildpack, host and several other optional attributes for the initial deployment of the application from this manifest file. The manifest file can also be provided as a command line argument if the file name is different or if it is present in a different location other than the current project directory. This is usually useful for deploying multiple applications together using a single manifest file. The below lines shows the contents of a sample manifest.yml file which deploys two applications named spark and flame with the specified attributes [55].

```
---
// the below manifest deploys two applications
// apps are in flame and spark directories
// flame and spark are in fireplace
// cf push should be run from fireplace
applications:
- name: spark
memory: 1G
instances: 2
host: flint-99
domain: shared-domain.example.com
path: ./spark/
services:
- mysql-flint-99
- name: flame
memory: 1G
instances: 2
host: burnin-77
domain: shared-domain.example.com
path: ./flame/
services:
- redis-burnin-77
```

4.3 Cloud Monitoring Metrics

The deployed cloud app is monitored continuously to collect the various dimensions of data that focuses on infrastructure utilization and application usage. Figure 15 shows an application deployed on cloud being monitored for usage statistics such as number of user requests and the response time of those requests. The users access the cloud app through different browsers. These requests are aggregated to collect the throughput(requests per second) metric. This thesis aims to collect and correlate the following metrics:

1. **Average response time of the requests:**
   The monitoring service collects all the HTTP requests to the application URL during every 10 second interval. This is collected from CF logs. The logs contain the response time
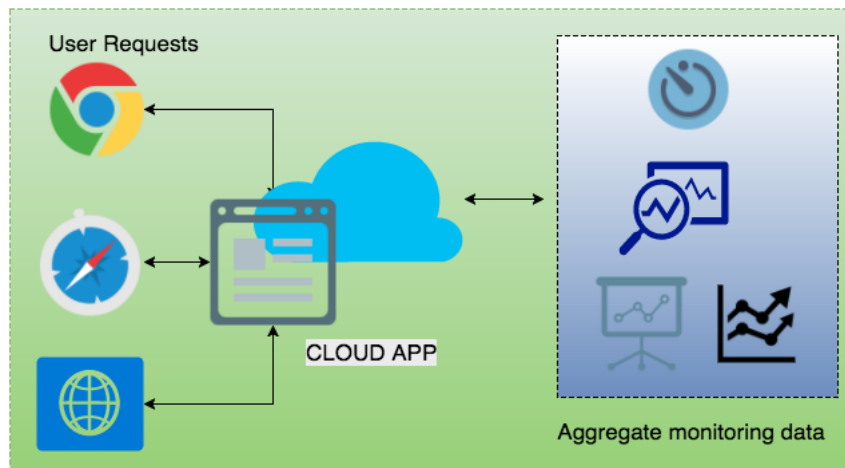
Figure 15: Cloud application monitored for usage statistics

information of each of the HTTP request. We aggregate the response time obtained in that 10 second interval and calculate the average response time. The following lines show a router's log from which the response time needs to be extracted and aggregated.

```
016-04-18T15:13:12.32+0200 [RTR/0] OUT
    masterthesisdemo-d063995.cfapps.sap.hana.ondemand.com -
    [18/04/2016:13:13:12 +0000] "POST /processrequest HTTP/1.1" 200 0 0 "-"
    "Java/1.8.0_74" 192.168.0.107:36027 x_forwarded_for:"172.18.74.8,
    192.168.0.107" x_forwarded_proto:"https"
    vcap_request_id:d70e4b9e-a08f-411d-584e-042e50d737e2
    response_time:0.031885594 app_id:55b4a61c-ee3b-484c-86c9-e656e99b2a96
```

2. **Throughput:**
   From the CF logs, the total number of requests can also be counted. This keeps varying depending on the user load at that time.

3. **Number of instances the application is running on:**
   The number of application instances that are currently running is retrieved by making a GET request to the CF API [56].

```
String url = "https://<cf-target-api>/v2/apps/:guid/stats";
URL obj = new URL(url);
HttpURLConnection con = (HttpURLConnection) obj
        .openConnection(proxy);
con.setRequestMethod("GET");
con.setRequestProperty("Authorization", oAuthToken);
```

In the above code snippet, the authorization token to access the CF API is fetched by making a call to the cf command: $cf\,oauth - token$. The URL is the CF API end point and the :guid in the URL is the GUID of the application in cloud. It is retrieved from the

environment variables of the application. The environmental variables is accessed by the CF command: $cf\ env\ app-name$

4. **CPU utilization, memory utilization, and the disk utilization as percentage** The CPU, memory, and disk utilization are fetched from the CF API the same way as the number of running instances. The URL for this GET request is:

```
String url = "https://<cloud-foundry-target-api-url>/v2/apps/:guid/summary";
```

4.4 Auto-scaler implementation

The auto-scaler is implemented as a java application and it consists of two parts: monitoring service and scaling service. This comprises the first two components of the proposed design. Monitoring service focuses on collecting the metrics discussed in section 4.3 and scaling service performs the scaling decisions and the actual scaling. The SLA values and threshold information are stored in a properties file. Table 4 lists the constants set in the properties file.

Table 4: Properties file of auto-scaler implementation

Property Name	Property Value
app_name	masterthesisdemo_d063995
min_instances	1
max_instances	7
cool_down_interval	60000
min_cpu_threshold	20.0
max_cpu_threshold	80.0
cf_api	cloud_foundry_api_endpoint
cf_username	cloudfoundry_username
cf_password	*********

The autoscaler first fetches this properties file and initializes the values. It consists of the following 2 services: Monitoring Service and the Scaling Service.

4.4.1 Monitoring Service

The monitoring service continuously keeps collecting all the metrics mentioned in section 4.3. During every iteration, it performs the following steps:

- **Initialize the DataModel object:** In this step, the model data object is initialized and the current timestamp is set. This object contains the following fields: timeStamp, *avgResponseTime, requestsPerSecond, private int noInstances, memory_percent, disk_percent, and cpu_percent*. This serves as the base schema of the dataset that is used for data modeling later. The current UTC timestamp is set to the *timeStamp* field.

- **CF Login:** The monitoring service logs in to CF using $cf\,login$ command passing the login credentials as parameters.

- **Get the CF OAuthToken:** After logging in, the authorization token is retrieved to access the CF API information. The $cf\,oauth-token$ command is used to get the authorization token. This token is used in the next steps as the authentication to be sent with the GET requests to access the CF API.

- **Get CPU/memory/disk stats:** The stats of the application are obtained by sending the GET request as described in section 4.3.

- **Stream the CF logs:** In order to get the response time and throughput, the CF Logs are streamed. The monitoring service creates two threads here: one for writing the logs and another one for reading the logs. The write thread streams the logs and writes it for 10 seconds. Meanwhile, the read thread reads the logs written in the previous iteration (previous 10 seconds). The two threads: read thread and write thread run in parallel and once both are completed the already read file is discarded and the new file written in this iteration is renamed to be read by the read thread in the next iteration.

  In the read thread, the logs are parsed to retrieve the response time information. This information is collected in an ArrayList and aggregated to get the average response time and throughput.

- **CF Logout:** The service logs out using the $cf\,logout$ command, similar to the cf login. This does not require any parameters.

- **Persist the data model object in MySQL database.** As a final step, the model data object is stored in the MySQL database. A database is created in MySQL named masterthesis. A table is created under this database called modeldata. The schema of this table is as shown in Figure 16.

  The monitoring service persists the data into this database. A JDBC connection to the database is established and the data is inserted into the table using the statement

```
INSERT INTO modeldata VALUES TimeStamp AvgResponseTime RequestsPerSecond
    NoInstances Memory_percent Disk_percent Cpu_percent
```

4.4.2 Scaling Service

Parallel to the monitoring service, the scaling service also runs continuously comparing monitoring information with threshold values and SLAs. The scaling service is invoked as soon as the monitoring begins. The scaling service has a java.util.Timer and this timer schedules a *TimerTask* to occur for every cool down period. The scaling decision could be based on any of the parameters: CPU/memory/disk depending on the application and the metrics. In this implementation, a CPU intensive application is considered and hence CPU utilization is chosen as the decision metric. We can also use a combination of these metrics. The *TimerTask* thread aggregates the
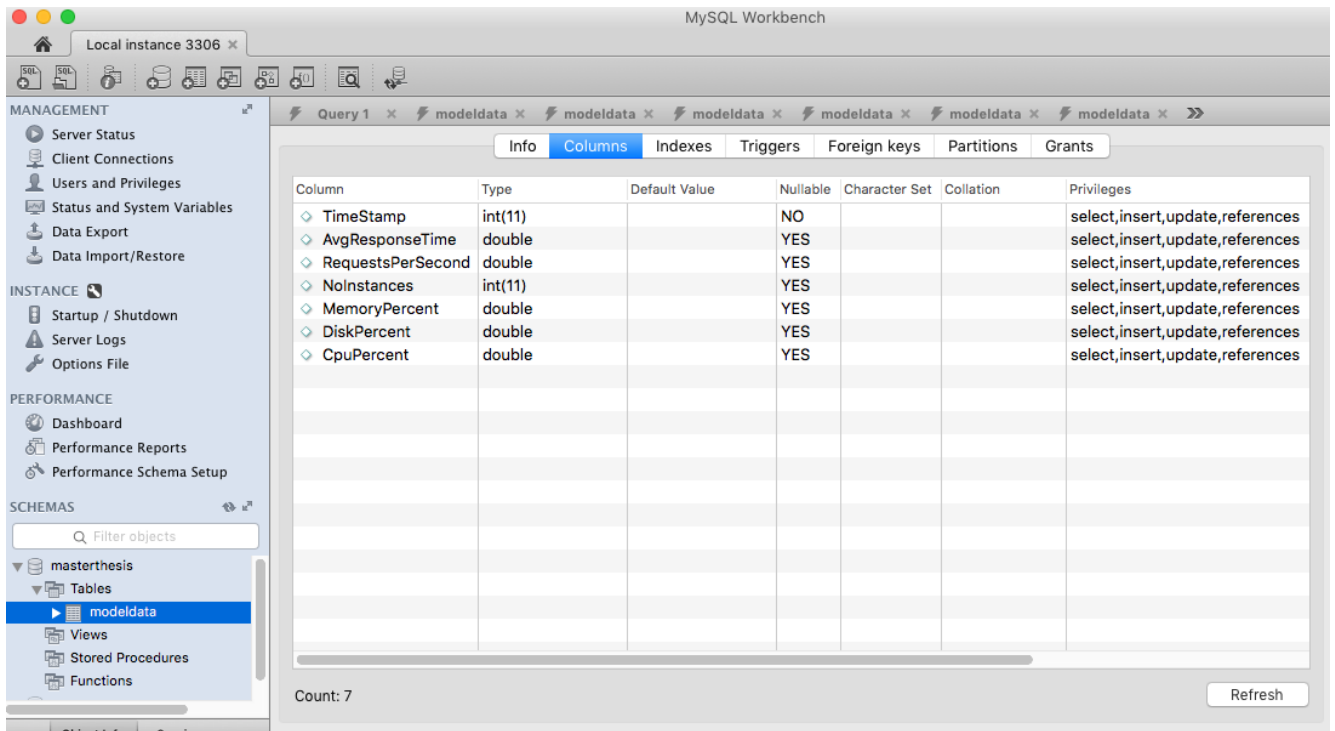
Figure 16: Screen shot of the database table that stores the monitoring data

average CPU utilization values in the last cool down period. These steps are shown in the below code snippet:

```java
static Timer timer_scale = new Timer ();
static TimerTask autoscaler = new TimerTask(){
    @Override
    public void run(){
    calculateAvgCPU();
    }
};
public static void invokeAutoScaler(){
timer_scale.schedule(autoscaler, 60000,
   Constants.cool_down_interval);
}

double cpu_avg = MonitoringService.cpu.parallelStream().mapToDouble
(e -> e.doubleValue()).average().getAsDouble();
CFCalls cf_call = new CFCalls();
if(cpu_avg > Constants.max_cpu_threshold){
   cf_call.cfLogin(Constants.cf_api, Constants.cf_username,
       Constants.cf_password);
   cf_call.getOAuthToken();
   cf_call.cfHorizontalScaling(
       cf_call.getCurrentRunningInstances()+1,
       Constants.app_name);
   cf_call.cfLogout();
}
```

```
else if (cpu_avg < Constants.min_cpu_threshold){
    cf_call.cfLogin(Constants.cf_api, Constants.cf_username,
        Constants.cf_password);
    cf_call.getOAuthToken();
    cf_call.cfHorizontalScaling(
        cf_call.getCurrentRunningInstances()-1,
        Constants.app_name);
    cf_call.cfLogout();
}
```

The actual scaling task involves making calls to the *cf scale* command. The CF scale command takes the number of instances or memory or disk values as parameters depending on the type of scaling (horizontal or vertical). In the below code snippet, we can see the number of instances(-i) passed as a parameter:

```
public void cfHorizontalScaling(int noInstances, String appName){
 if(!(noInstances >= Constants.min_instances &&
        noInstances <= Constants.max_instances)){
    return;
  }
 String[] cf_scale = {"cf" , "scale", appName, "-i" ,
        Integer.toString(noInstances) };
 Process scale = Runtime.getRuntime().exec(cf_scale);

}
```

4.5 Data Modeling

Data modeling is the final component of the proposed design. The MATLAB System Identification Toolbox is used to perform data modeling to find a suitable correlation model. The data from MySQL database table is exported to the MATLAB workspace. The input vectors and output vectors are combined from the datasets as shown below:

```
input = [requestsPerSecond, noInstances, cpu_percent, memory_percent,
            disk_percent];
output = [avgResponseTime];
```

The input and output vectors are imported into the Sytem Identification Toolbox as shown in Figure 17. The data is now ready to be split into estimation Data and validation data. To do this, the select range option underneath preprocess combo box has to be clicked as shown in Figure 18. Once the data is split, the datasets are dragged to the corresponding working data area and validation area. Then the model estimation needs to be performed. The model that needs to be estimated for the data needs to be chosen underneath the estimate box. The estimated models include state space models, and polynomial models: ARX and ARMAX models.
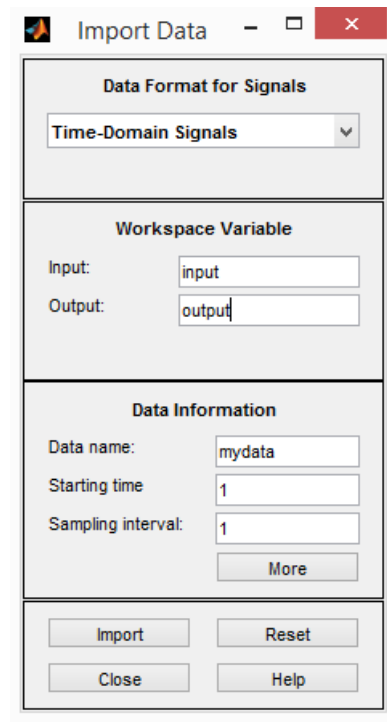
Figure 17: Importing data into System Identification Toolbox
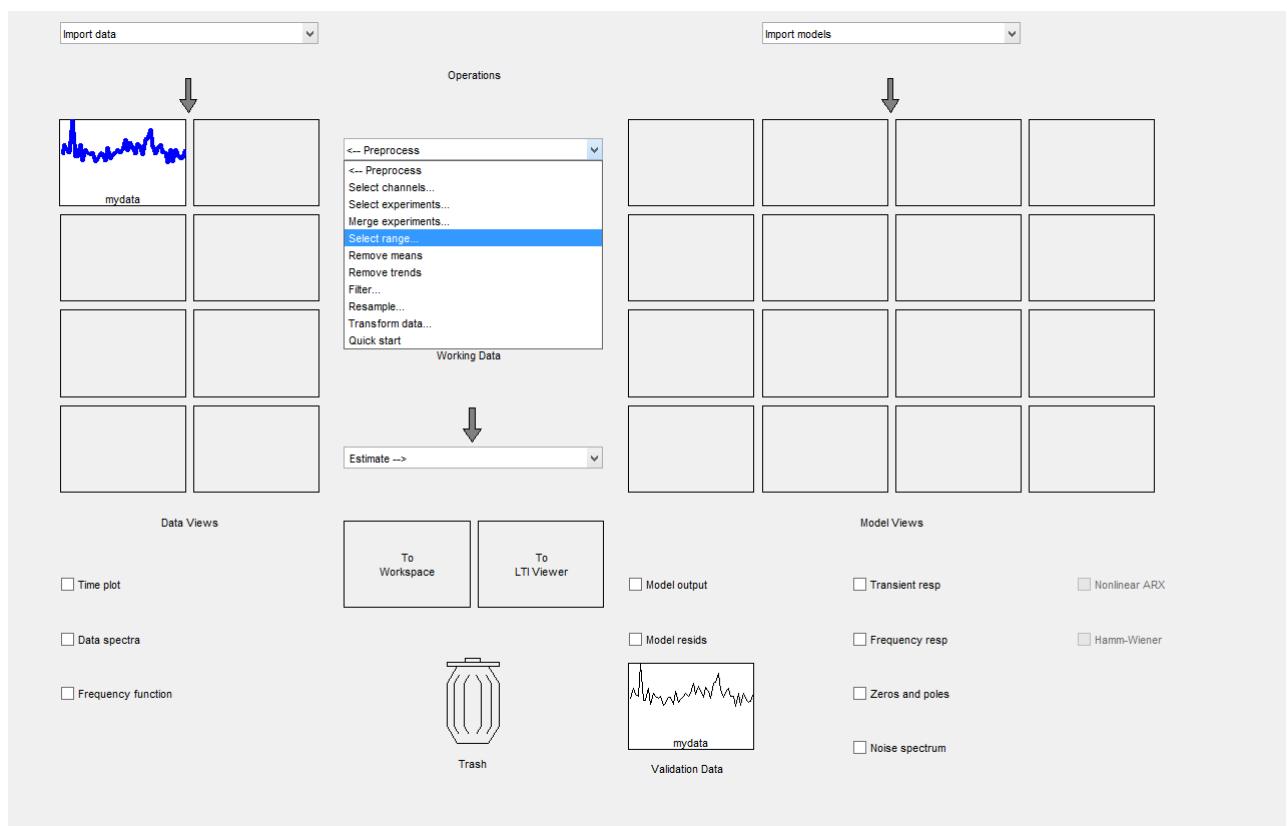
Figure 18: Splitting the data into estimation and validation datasets
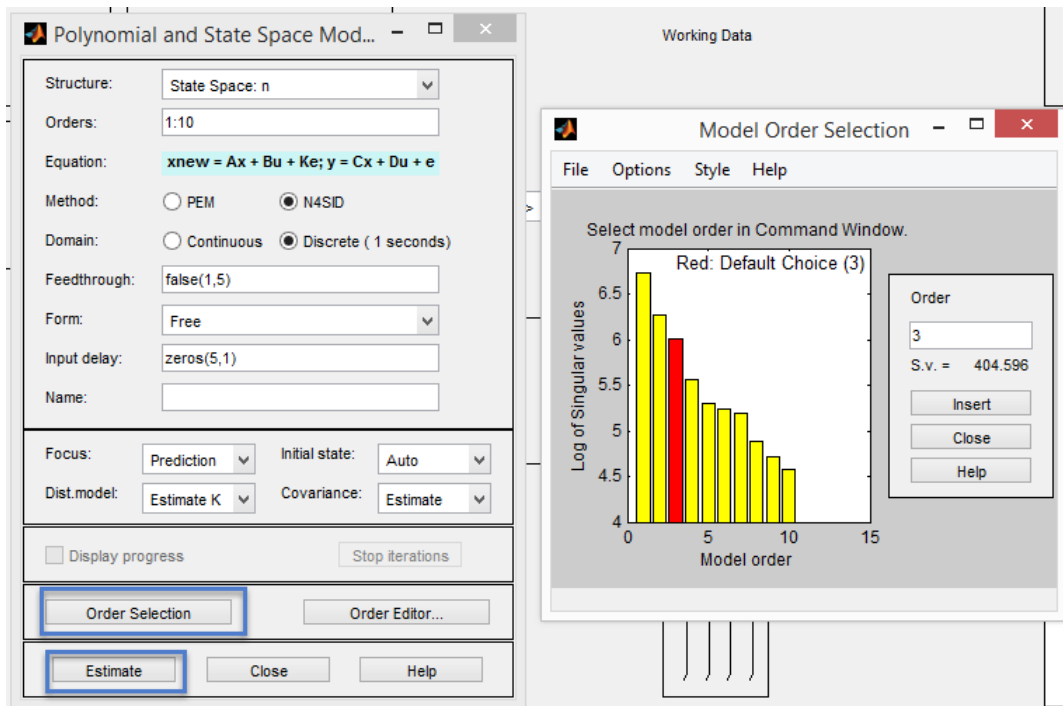
Figure 19: State space modeling
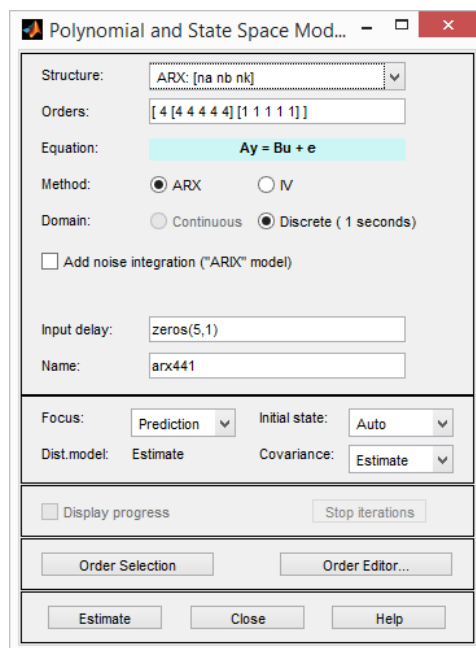
Figure 20: ARX model
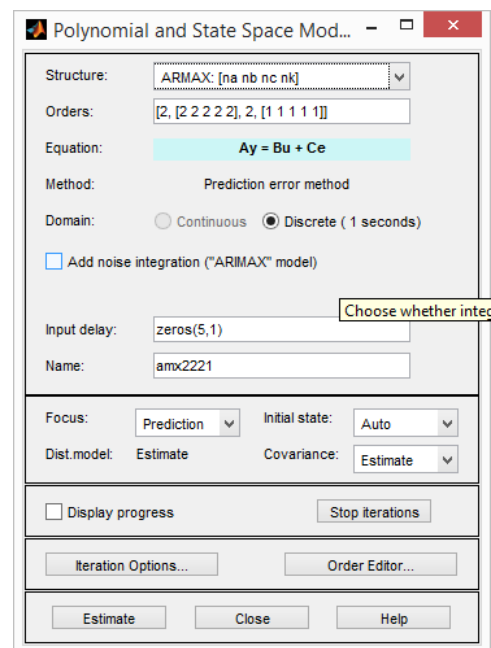
Figure 21: ARMAX model

- **State space model**: The state space mode estimation is shown in Figure 19. It provides options to specify the model order or to specify a range of orders and the system would suggest which model order would suit best for this scenario. The model order will determine the size of the state variable vector which in turn will determine the output at a certain time in the future, given the inputs at that time.
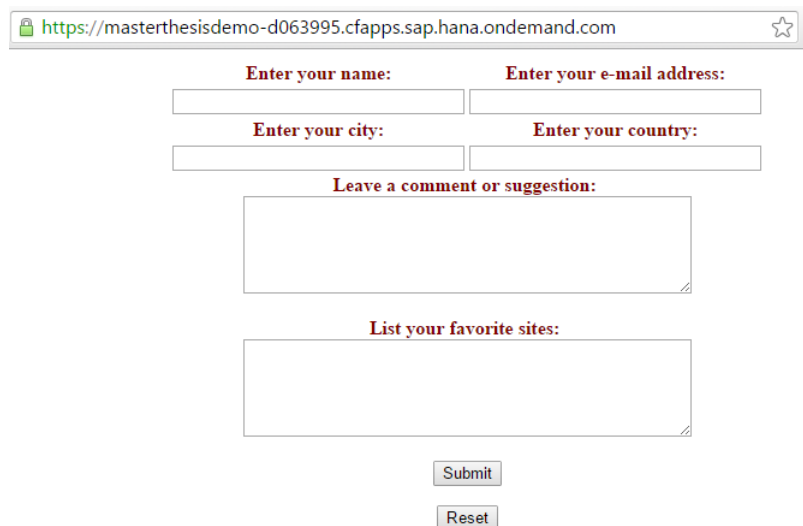
- **ARX and ARMAX models**: Following the state space model estimation polynomial model estimation is done. In this context, ARX and ARMAX models are considered for estimation. In ARX, the orders of the polynomials A and B (na and nb) and the error co-efficients, nk, are specified. This is shown in Figure 20. For ARMAX modeling, the order of the polynomials A, B and C (na, nb and nc) and the error co-efficients are specified. Figure 21 shows this in the System Identification Toolbox.

5 Evaluation

In this chapter, the evaluation of our implementation is discussed. A sample guest book application is deployed to HCP, and load is generated on the app. The metric collection by monitoring service and automatic scaling of instances are tested and verified. This forms the first part of the contribution of the thesis. The second part focuses on deriving a correlation model between the metrics. The collected metric information is used evaluate the correlation model.

5.1 Sample application under consideration

We consider a sample web application hosted in the HCP. This is a guest book application which provides a web interface for the users to enter the data as shown in Figure 22. For every

Figure 22: User interface of the guest book application

user who posts the data, a background calculation of fibonacci series upto any random number between 0 and 38 is performed. The source code of the recursive fibonacci series calculation is shown below. This code runs with an exponential complexity runtime and its time complexity is represented as $O(2^n)$. The fibonacci series is calculated by recursion without memoization to ensure an exponential complexity, thereby ensuring that the application is a CPU intensive one.

```java
public void ComputeRandomFibonacchi(){
    int random_number = (int) (Math.random() * 38);

    long lStartTime = new Date().getTime();
    for(int i=1; i<=random_number; i++){
```
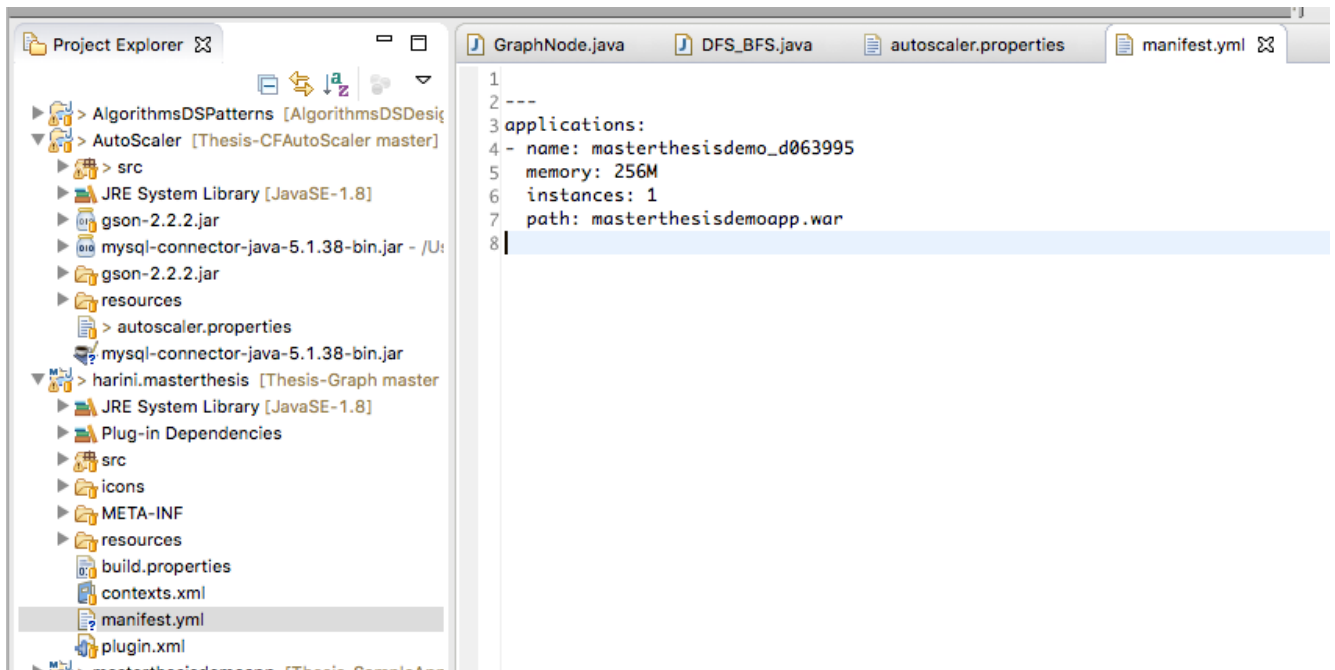
Figure 23: Manifest.yml file of the guest book application

```java
        fibonacciRecusion(i);
    }
    long lEndTime = new Date().getTime();
    long difference = lEndTime - lStartTime;
}

public long fibonacciRecusion(int number){
    if(number == 1 || number == 2){
        return 1;
    }
    return fibonacciRecusion(number-1)
          + fibonacciRecusion(number -2);
}
```

The manifest file of this application is shown in Figure 23. The app is deployed to CF using the following command: $cf\,push\,masterthesisdemo\_d063995$. The cf push command reads the attributes from the manifest.yml file. So, the app is created in the cloud with the name masterthesisdemo_d063995, and the war file specified in the path is deployed. The values specified in the manifest are used during the initial deployment. In this scenario, 1 instance of the application is started with a memory of 256 MB as shown in Figure 23.

5.2 Load generation on the sample application

We use apache JMeter to generete load on our application. A test plan is created in JMeter and it is configured to contain 5 thread groups. Each thread group is set to run one after the other (serial execution). The first thread group generates 100 users simultaneously over a ramp up

period of 50 seconds. Each of the thread groups are set to loop for a loop counter of 3600. In the following thread groups, the number of threads (users) are increased to 200, 400, 600, 800 and 1000 respectively. The JMeter test plan configuration is shown in Figure 24. Before executing the test plan, the auto-scaling framework is started. This means the application is now being monitored by the monitoring service and scaling service is activated. Now the test plan is executed to see if our application performs monitoring and scaling consistently.
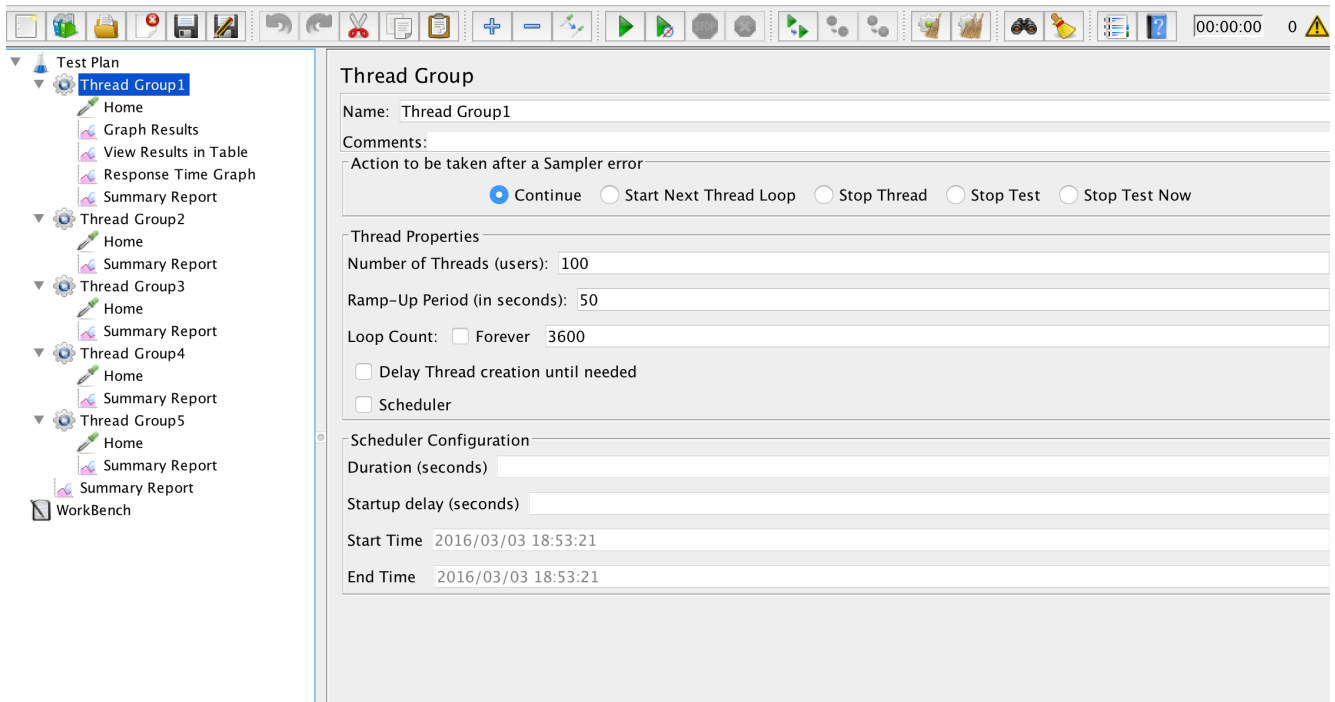
Figure 24: JMeter test plan configuration

5.3 Scaling out and scaling in of the the application Instances

In this section, the results of scaling service are demonstrated. As and when the load increases, it can be noted that the scaling service creates and adds more instances of the application. This is because the CPU utilization of the instances go up as our sample application under consideration is a CPU intensive one. This can be verified in the command line console as well. Figure 25 shows the statistics of application instances in the console. We notice that the CPU column shows a huge spike in the percentage utilization of CPU. The scaling out happens until the values falls below the threshold CPU value. For this evaluation, the maximum CPU threshold was set to 80%, which can be verified in Figure 26.

Figure 27 shows the graph indicating the scaling action performed. The graph on top, y1 denotes the load (requests per second). The graph beneath, u1 shows the number of application instances. The x axis indicates the time. In this scenario, about 2000 records were collected and the x axis indicates these relative values. From this graph, it can be inferred that as the load increases, the number of instances also increase up to 7 and then once the load generation is turned off the number of instances reduces back to 1. For this evaluation, 7 is the maximum

number of instances, and 1 is the minimum number of instances configured as an SLA. The SLA settings screen shot is shown in Figure 26. This depicts the scaling out and scaling in procedures.

Figure 25: Stats of the application instances in the command line interface

Figure 26: Properties file where the default SLAs/policies are stored
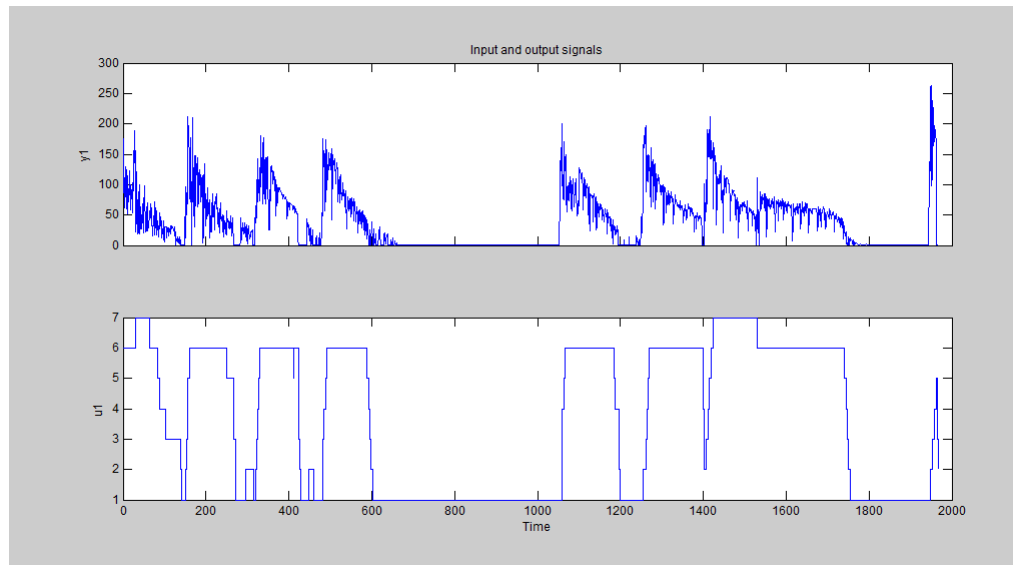
Figure 27: Graph depicting the correlation between throughput and number of application instances

5.4 Persistence of data in MySQL database

In this section, the results of persistence of the monitoring and scaling services are demonstrated. It continuously collects the information and persists it into the MySQL database. Figure 28 shows the data stored in the MySql DB. The dimensions persisted are time stamp of the record, average response time, requests per second, number of instances, memory utilization as percentage, disk utilization as percentage, and CPU % utilization. Depending on the type of scaling and the type of sample application, scaling may be reflected in number of instances (horizontal scaling) or memory/disk (vertical scaling).

As our example application involves horizontal scaling, whenever scaling happens it will be reflected in the number of instances. The screen shot in Figure 28 shows the number of instances gradually increasing from 1 until 7 which is the minimum and maximum instances set during this evaluation. This can be verified in Figure 26, which shows the screen shot of the properties file containing the SLA settings.

**Figure 28:** MySQL workbench showing the top rows of the table modeldata

The dataset from MySql databse is exported into the MATLAB workspace. Average response time is estimated to see the performance improvement of the application using auto-scaler. The input and output datasets are now imported to the MATLAB System Identification Toolbox with the name EvalDataSet. The structure of the input and output datasets are as below:

```
input = [requestsPerSecond, noInstances, cpu_percent, memory_percent,
    disk_percent];
output = [avgResponseTime];
```

Figure 29 depicts the inputs and output graphically. It contains six graphs where the first one represents the output dimension and the following five graphs represent the input dimensions. The X-axis is time on all the graphs and the Y-axis indicates average response time, requests per second, number of instances, CPU % utilization, memory % utilization, and disk % utilization from the top to bottom respectively.

This dataset is now split into estimation and validation datasets. The complete dataSet, EvalDataSet, is dragged into the working data area. Select range option is used to split the entire dataset into the estimation and validation datasets: EvalDataSetEstimation and EvalDataSetValidation.

The range selection process is shown in Figure 30 . The dataset is split into 80% and 20% respectively. The 80% data is shown in Figure 31, which represents the estimation dataset and the 20% data is shown in Figure 32, which represents the validation dataset.
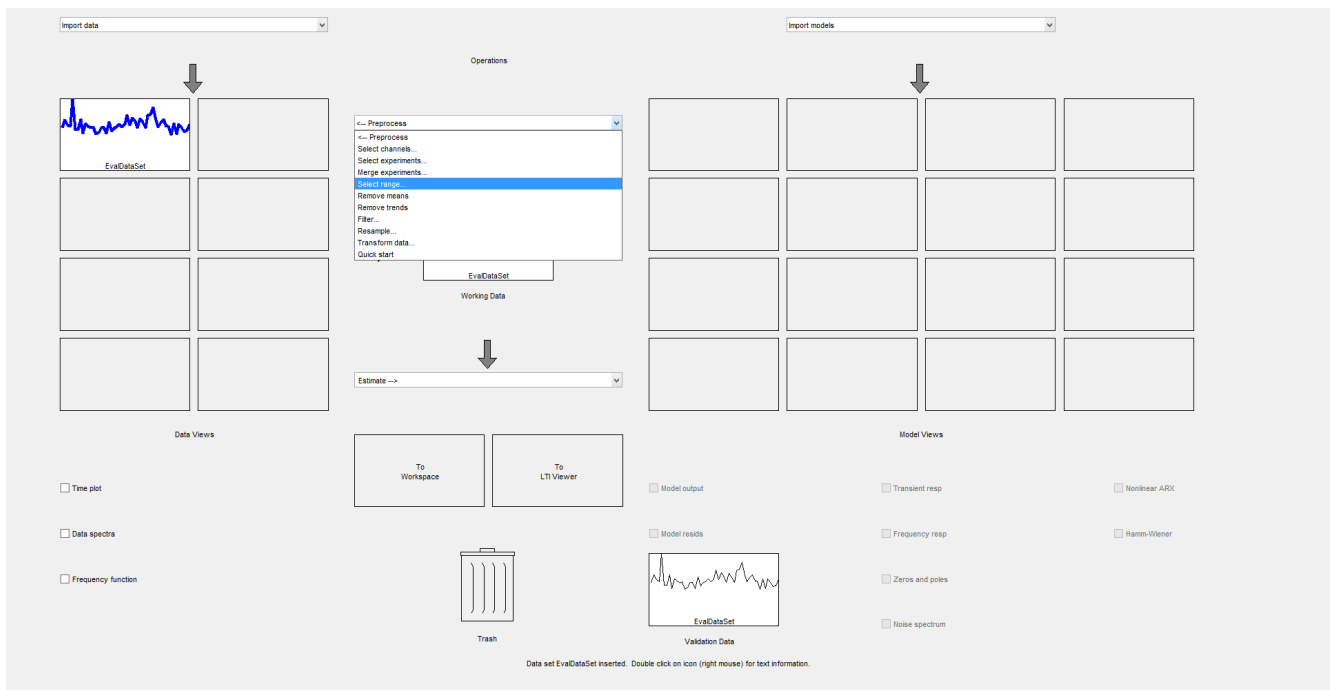
Figure 30: Select ranges from the imported dataset: EvalDataSet
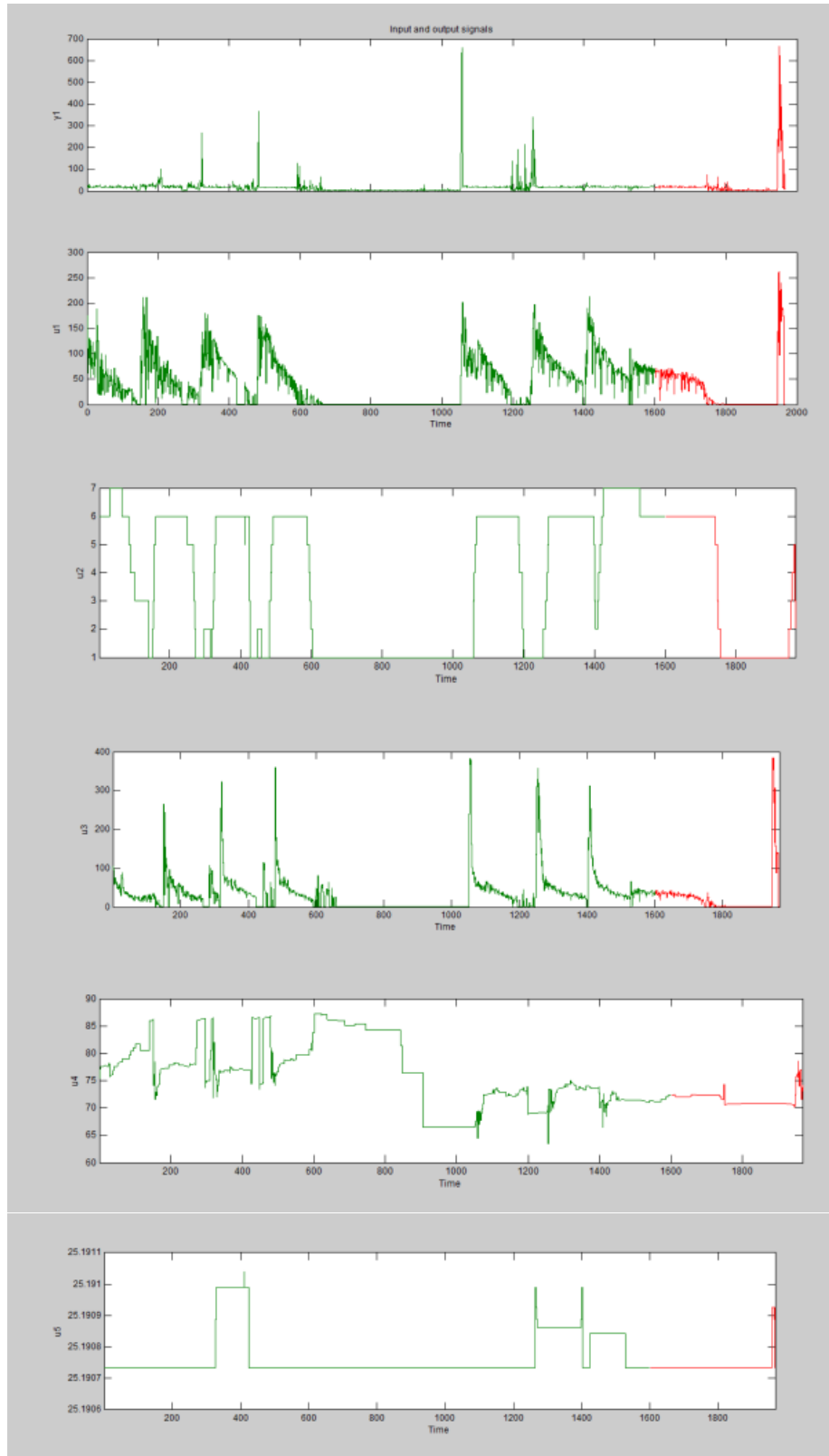
Figure 29: Graph showing the output Y1: average response time and the inputs U1, U2, U3, U4 and U5: requests per second, number of instances, CPU, memory and disk utilization
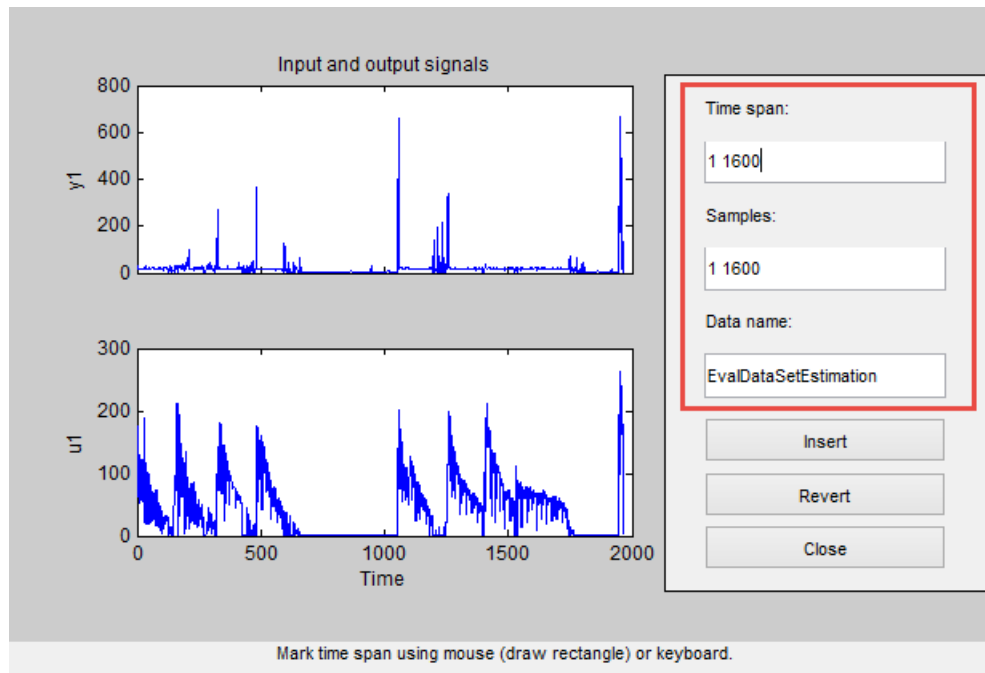
Figure 31: Choosing 80% data as estimation data: EvalDataSetEstimation

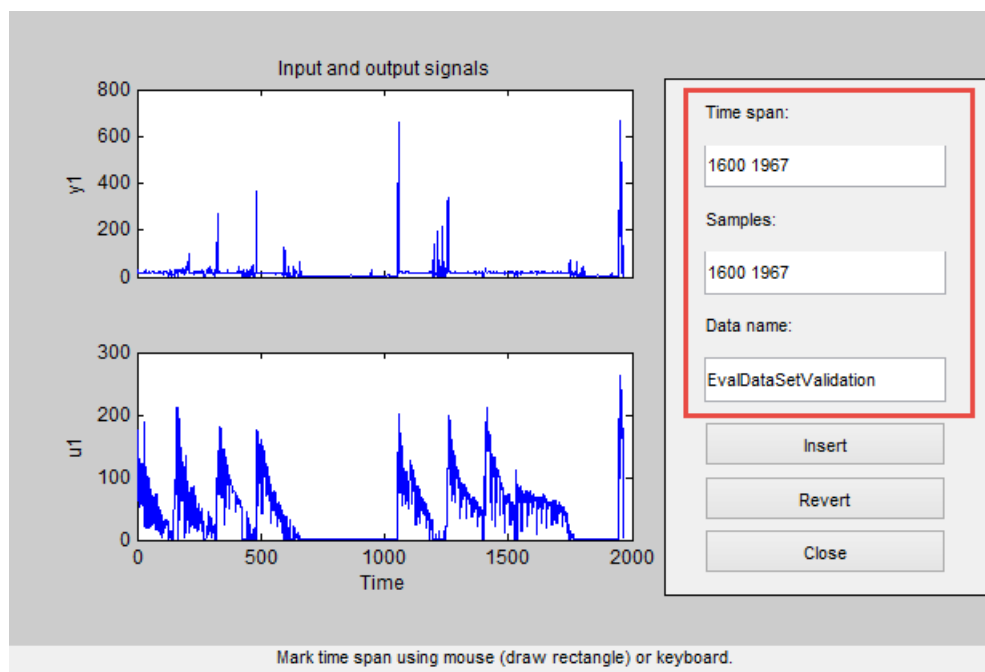**Figure 32:** Choosing 20% data as validation data: EvalDataSetValidation

Now the split data sets need to be dragged into the working data area and validation data area respectively. This is depicted in Figure 33. The green dataset that represents the estimation data is copied to working area box and the red dataset that represents the validation dataset is copied to the validation data box.
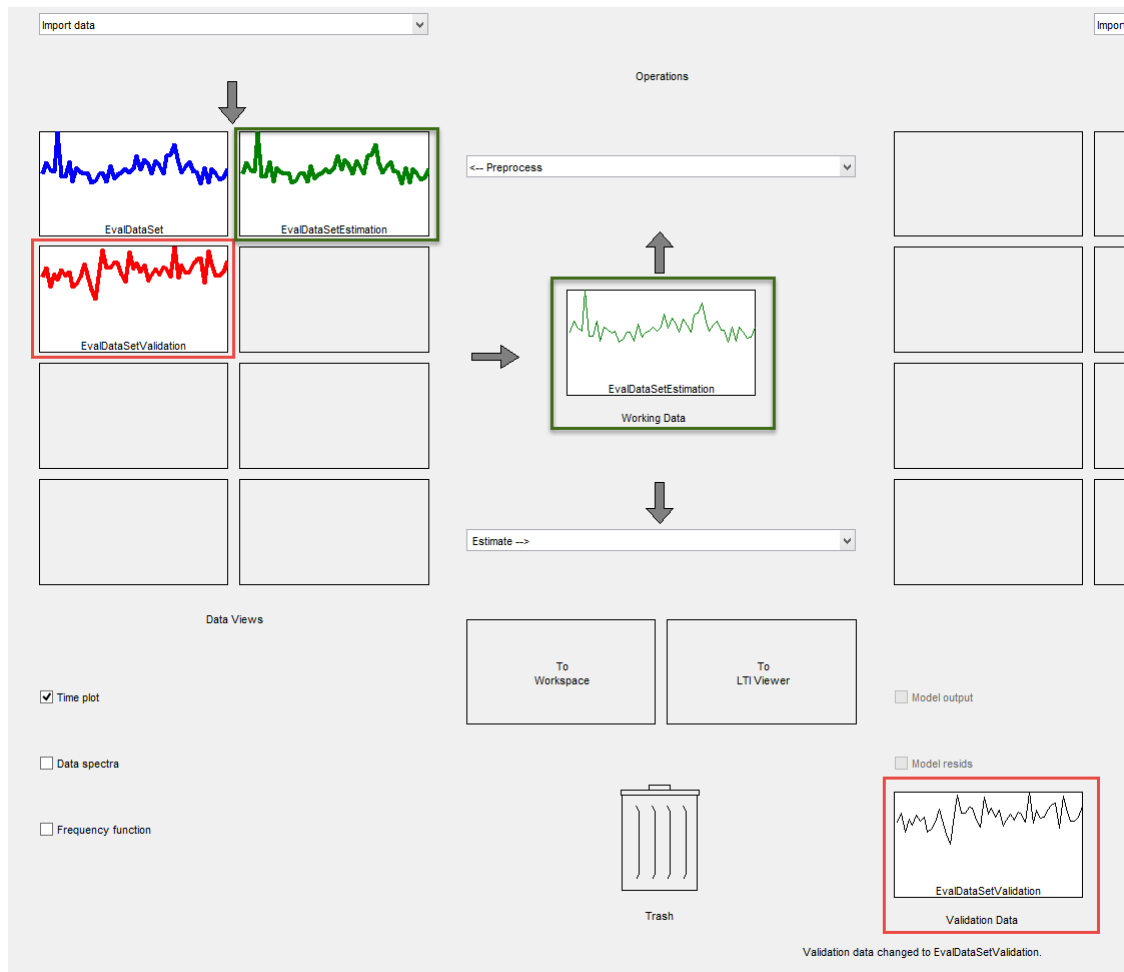
Figure 33: Working data and validation data area

The next step is to estimate the models for above datasets. Estimation takes place as described in section 4.5. The state space model estimation is performed by giving the input model order value as 3. This can be visualized in Figure 19.

Following the state space model estimation, polynomial models are estimated. In this context, ARX and ARMAX models are estimated. In ARX, the orders of the polynomials A and B (na and nb) and the error co-efficients, nk are specified with values:

$$[n_a \ n_b \ n_k] = [4, \ [4\,4\,4\,4\,4\,], \ [1\,1\,1\,1\,1]]$$

For ARMAX modeling, the order of the polynomials A, B and C (na, nb and nc) and the error co-efficients with values are specified with values:

$$[n_a \ n_b \ n_c \ n_k] = [4, \ [4\,4\,4\,4\,4], \ 2, \ [1\,1\,1\,1\,1]]$$

ARX and ARMAX estimation are depicted in Figure 20 and Figure 21 respectively.

The models estimated are validated against the validation dataset. The model output check-box is checked to see the the accuracy of each model. It shows the best fitting models listed in order. This is depicted in Figure 34.
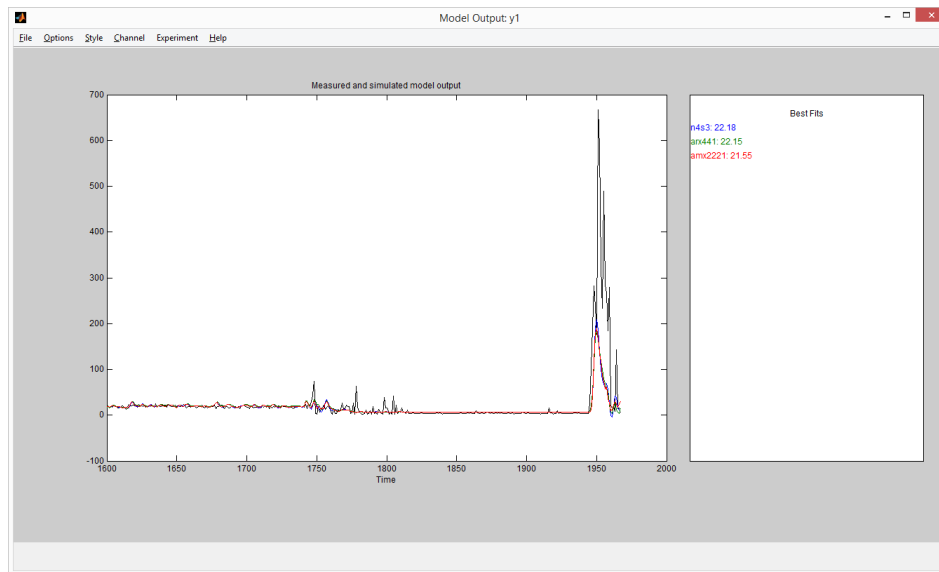
Figure 34: Model output with the best fits listed

5.6 Comparison of the response times with and without the auto-scaler

The previous section shows the models for a dataset which was generated with the auto-scaler. In this section, charts, displaying the response time fluctuations for a small load generation without an auto-scaler and with an auto-scaler, are analyzed. Figure 35 and Figure 36 shows this clearly. In the first graph, it can be inferred that the response time peaks as high as 750 milliseconds whereas in the next graph it can be noticed that the maximum response times are reduced to around 100 milliseconds for almost the same load. In fact, the second case(with the auto-scaler) has slightly additional load compared to the previous scenario.
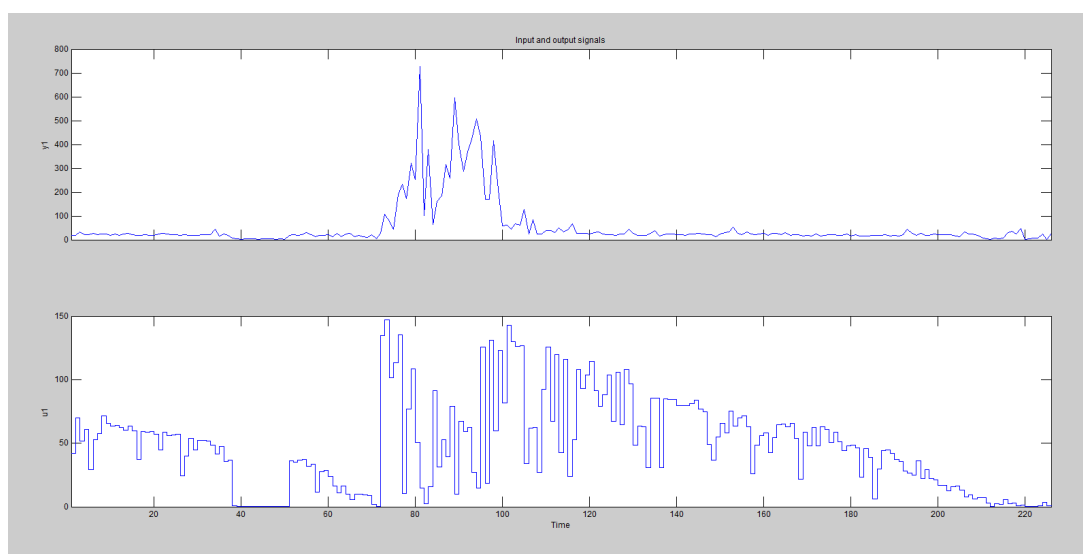
Figure 35: Response time versus load without an auto-scaler
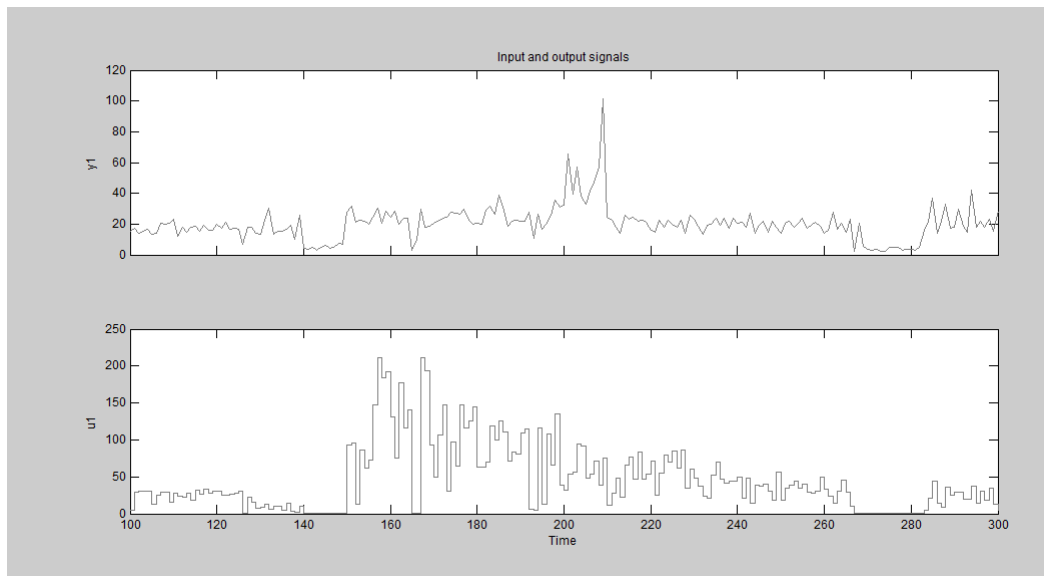
Figure 36: Response time versus load with an auto-scaler

6   Conclusion

This chapter concludes the thesis by providing a short summary of the contributions of this thesis. It also lists some of the challenges and limitations faced during this thesis and provides some ideas for future work in this direction.

6.1 Contribution of the Thesis

With cloud computing and CD on a significant rise, DevOps gained a lot of popularity. This thesis successfully performs monitoring of applications hosted on cloud platforms. Cloud monitoring has proven to be extremely useful especially for cloud developers and cloud operators (DevOps). Developers leverage this information for FDD and root cause identification. Operators depend on this information for provisioning the right amount of infrastructure.

This thesis proceeds in the operations side with the idea of how to run operations with software engineering ideas thereby reducing downtime/outages etc. It navigates from FDD to feedback driven operations. The relevant run-time metrics are chosen to design an auto-scaling system. This is slightly different from the many available infrastructure auto-scaling mechanisms. This is a platform-level auto-scaler which increases or decreases the number of running application instances depending on the threshold parameters. Sometimes, adding infrastructure may not solve the problem whereas what is essential could be the efficient utilization of the existing infrastructure. Cloud platforms need to take this into consideration and provide a Platform Auto-scaling mechanism. This thesis contributes an efficient platform level Auto-scaler.

The monitoring data collected for making scaling decisions are also modeled to identify the correlation of the metrics. In this thesis, the average response time is estimated. It is extremely helpful to forecast the response time which aids in automating the resource provisioning, and configuration management tools.

6.2 Limitations and Challenges

One of the challenges faced during this implementation is the choice of parameters to design the auto-scaler. This proves to be quite important as the decisions of auto-scaler is based on these parameters. The auto-scaler could be designed to work on any metric such as response time or CPU % utilization. In this implementation, it was designed to scale based on the CPU.

The evaluation is performed with a single application. It could possibly be evaluated with several other applications. The app considered in this evaluation is a CPU intensive application. It can be performed on other applications that are CPU, memory, storage, database or data (streaming) intensive applications. Considerations to collect larger datasets for modeling is also important. This may provide different aspects or design considerations to the auto-scaler.

We envision the following research directions. On one side, the rule based auto-scaler can be enhanced to combine the different metrics on which the scaling decision depends. Also the threshold values on which the scaling decisions are made can be verified to find a better threshold if applicable. This will involve testing the existing prototype on different types of applications to choose the right combination of metrics and the right minimum and maximum thresholds.

On the other side, modeling can be improved. A correlation model from the monitored metrics is derived. Modeling can be enhanced to achieve a better accuracy. In order to do this, larger monitoring datasets would be required. Additionally, the script can be automated and used for modeling each application. This way a dynamic model, depending on the application on which the auto-scaler is run, is derived. Using this dynamic model, predictions of the response times customized for each application can be provided. These predictions can in turn be used in making the rule-based auto-scaler a proactive one.

References

[1] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[2] M. Httermann, *DevOps for developers*. Apress, 2012.

[3] J. Cito, P. Leitner, H. C. Gall, A. Dadashi, A. Keller, and A. Roth, "Runtime metric meets developer: building better cloud applications using feedback," in *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 14–27, ACM, 2015.

[4] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[5] Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," in *Proceedings of the 13th International Workshop on Mining Software Repositories*, pp. 25–36, ACM, 2016.

[6] J. P. Sandoval Alcocer, A. Bergel, and M. T. Valente, "Learning from source code history to identify performance failures," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, pp. 37–48, ACM, 2016.

[7] "Amazon web services." https://aws.amazon.com/.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[9] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, "Application-level performance monitoring of cloud services based on the complex event processing paradigm," in *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pp. 1–8, IEEE, 2012.

[10] L. Youseff, M. Butrico, and D. Da Silva, "Toward a unified ontology of cloud computing," in *2008 Grid Computing Environments Workshop*, pp. 1–10, IEEE, 2008.

[11] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: An empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 393–403, ACM, 2015.

[12] D. Bruneo, F. Longo, and C. C. Marquezan, "A framework for the 3-d cloud monitoring based on data stream generation and analysis," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pp. 62–70, IEEE, 2015.

[13] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 959–967, ACM, 2007.

[14] "Application scaling versus infrastructure scaling." http://allthingsplatforms.com/platforms/on-paas-and-scaling/.

[15] "Cloud foundry scaling." http://zone.ni.com/reference/en-XX/help/372458D-01/lvsysidconcepts/modeldefinitionsbj/.

[16] "Cloudwatch: developer's guide to monitor applications." http://awsdocs.s3.amazonaws.com/AmazonCloudWatch/latest/acw-dg.pdf/.

[17] "Cloudwatch." https://aws.amazon.com/cloudwatch/.

[18] "Azurewatch." http://www.paraleap.com/azurewatch.

[19] "New relic." https://newrelic.com/.

[20] "Amazon auto scaling." https://aws.amazon.com/autoscaling/.

[21] "Nimsoft." http://www.nimsoft.com/solutions/nimsoft-monitor/cloud.

[22] "cloudyn." http://www.cloudyn.com/.

[23] "Up.time." http://www.uptimesoftware.com/cloud-monitoring.php.

[24] "Nagios." http://nagios.sourceforge.net/docs/nagioscore-3-en.pdf.

[25] "Cloudsleuth." https://cloudsleuth.net/.

[26] "Nimbus." http://www.nimbusproject.org/.

[27] "Cloudstone." http://radlab.cs.berkeley.edu/wiki/Projects/Cloudstone.

[28] "Groundwork." http://www.gwos.com/features/.

[29] "Boundary." https://boundary.com/.

[30] "Logicmonitor." http://www.logicmonitor.com/.

[31] "cloudfloor." http://cloudfloor.com/.

[32] "Cloudkick." http://www.cloudkick.com/home.

[33] "Cloudclimate." http://www.cloudclimate.com.

[34] "Monitis." http://portal.monitis.com/.

[35] "Cloudharmony." http://cloudharmony.com/.

[36] "Openstack iaas." http://docs.openstack.org/diablo/openstack-compute/admin/oscompute-adminguide-trunk.pdf.

[37] C. Bunch, V. Arora, N. Chohan, C. Krintz, S. Hegde, and A. Srivastava, "A pluggable autoscaling service for open cloud paas systems," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pp. 191–194, IEEE Computer Society, 2012.

[38] S. R. Seelam, P. Dettori, P. Westerink, and B. B. Yang, "Polyglot application auto scaling service for platform as a service cloud," in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pp. 84–91, IEEE, 2015.

[39] A. Biswas, S. Majumdar, B. Nandy, and A. El-Haraki, "Predictive auto-scaling techniques for clouds subjected to requests with service level agreements," in *Services (SERVICES), 2015 IEEE World Congress on*, pp. 311–318, IEEE, 2015.

[40] "Scryer - netflix auto-scaling tool: Part 1." http://techblog.netflix.com/2013/11/scryer-netflixs-predictive-auto-scaling.html.

[41] "Scryer - netflix auto-scaling tool: Part 2." http://techblog.netflix.com/2013/12/scryer-netflixs-predictive-auto-scaling.html.

[42] L. R. Moore, K. Bean, and T. Ellahi, "A coordinated reactive and predictive approach to cloud elasticity," 2013.

[43] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[44] D. A. Freedman, *Statistical models: theory and practice*. cambridge university press, 2009.

[45] M. Friendly, "Corrgrams: Exploratory displays for correlation matrices," *The American Statistician*, vol. 56, no. 4, pp. 316–324, 2002.

[46] J. Li, J.-B. Martens, and J. J. Van Wijk, "Judging correlation from scatterplots and parallel coordinate plots," *Information Visualization*, vol. 9, no. 1, pp. 13–30, 2010.

[47] L. Ljung, "System identification: Theory for the user," *Englewood Cliffs*, 1987.

[48] L. Ljung, *System identification*. Springer, 1998.

[49] L. Ljung, "System identification: Theory for the user, ptr prentice hall information and system sciences series," 1999.

[50] G. E. Box, G. M. Jenkins, G. C. Reinsel, and G. M. Ljung, *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[51] "Jpetstore." http://sourceforge.net/projects/ibatisjpetstore.

[52] "Agilefant." http://agilefant.com/.

[53] "Cloud foundry: Get app summary." https://apidocs.cloudfoundry.org/237/apps/get_detailed_stats_for_a_started_app.html.

[54] "Sap hana cloud platform." https://hcp.sap.com/index.html.

[55] "Cloud foundry: Manifest file." https://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html.

[56] "Cloud foundry: Get detailed stats for a started app." https://apidocs.cloudfoundry.org/237/apps/get_detailed_stats_for_a_started_app.html.

Glossary

AAS Amazon Auto-Scaling. 18

APM Application Performance Monitoring. 9, 12

AWS Amazon Web Services. 8, 17, 18

CD Continuous Delivery. 12, 54

CEP Complex Event Processing. 17

CF Cloud Foundry. 33, 34, 36, 37, 44

FDD Feedback Driven Development. 9, 15, 24, 30, 54

GUID Globally Unique Identifier. 36

HCP Hana Cloud Platform. 33, 34, 43

IaaS Infrastructure-as-a-Service. 17, 33

IDE Integrated Development Environment. 9, 10

JDBC Java DataBase Connectivity. 37

LTI Linear Time Invariant. 19

PaaS Platform-as-a-Service. 9, 13, 14, 17, 33, 34

QoS Quality of Service. 15

SaaS Software-as-a-Service. 9

SLA Service Level Agreements. 14, 15

UTC Coordinated Universal Time. 37