KTH Information and
Communication Technology

Self-trained Proactive Elasticity Manager for Cloud-based Storage Services

DAVID DAHAREWA GUREYA

Master's Thesis at KTH Information and Communication Technology
Supervisors: Ahmad Al-Shishtawy and Ying Liu
Examiner: Vladimir Vlassov

TRITA xxx yyyy-nn

Abstract

The pay-as-you-go pricing model and the illusion of unlimited resources makes cloud computing a conducive environment for provision of elastic services where different resources are dynamically requested and released in response to changes in their demand. The benefit of elastic resource allocation to cloud systems is to minimize resource provisioning costs while meeting service level objectives (SLOs). With the emergence of elastic services, and more particularly elastic key-value stores, that can scale horizontally by adding/removing servers, organizations perceive potential in being able to reduce cost and complexity of large scale Web 2.0 applications. A well-designed elasticity controller helps reducing the cost of hosting services using dynamic resource provisioning and, in the meantime, does not compromise service quality. An elasticity controller often needs to be trained either online or offline in order to make it intelligent enough to make decisions on spawning or removing extra instances when workload increase or decrease. However, there are two main issues on the process of control model training. A significant amount of recent works train the models offline and apply them to an online system. This approach may lead the elasticity controller to make inaccurate decisions since not all parameters can be considered when building the model offline. The complete training of the model consumes large efforts, including modifying system setups and changing system configurations. Worse, some models can even include several dimensions of system parameters. To overcome these limitations, we present the design and evaluation of a self-trained proactive elasticity manager for cloud-based elastic key-value stores. Our elasticity controller uses online profiling and support vector machines (SVM) to provide a black-box performance model of an application's SLO violation for a given resource demand. The model is dynamically updated to adapt to operating environment changes such as workload pattern variations, data rebalance, changes in data size, etc. We have implemented and evaluated our controller using the Apache Cassandra key-value store in an OpenStack Cloud environment. Our experiments with artificial workload traces shows that our controller guarantees a high level of SLO commitments while keeping the overall resource utilization optimal.

Referat

INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Self-trained Proactive Elasticity Manager for Cloud-based Storage Services

Gureya Daharewa David

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Examination Committee

Chairperson:
Supervisor:
Member of the Committee:

July 2015

Acknowledgements

I would like to thank Professor Vladimir Vlassov and Professor Luís Manuel Antunes Veiga who gave me the honor to work with them during my Master Thesis.

Special thanks to Ahmad Al-Shishtawy for supervising my thesis and giving me the opportunity to work at SICS, Swedish Institute of Computer Science. I owe my greatest gratitude to my co-supervisor, Ying Liu. Ahmad's and Liu's patience and guidance helped me a lot in the time of research and implementation of this thesis. I gained amazing experiences and expertise under their supervision.

To all my Professors who taught us at IST and KTH during my master studies, thank you all.

Last but not the least, I am truly thankful to my family for supporting me throughout my life.

Lisboa, October 13, 2015

David Daharewa Gureya

European Master in Distributed Computing (EMDC)

This thesis is a part of the curricula of the European Master in Distributed Computing, a co-operation between KTH Royal Institute of Technology in Sweden, Instituto Superior Tecnico (IST) in Portugal and Universitat Politecnica de Catalunya (UPC) in Spain. This double degree master program is supported by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Union.

My study track during the master studies of the two years is as follows:

1. First year: Instituto Superior Tecnico, Universidade de Lisboa
2. Third semester: KTH Royal Institute of Technology
3. Fourth semester (Thesis): SICS[1] /KTH Royal Institute of Technology

[1] Swedish ICT
https://www.sics.se/

Dedication

To my parents and my teachers

Resumo

O pay-as-you-go modelo de preços ea ilusão de recursos ilimitados faz computação um ambiente propício para a prestação de serviços de elásticos onde diferentes recursos são dinamicamente solicitadas e liberadas em resposta a mudanças na sua demanda nuvem. O benefício de alocação de recursos em nuvem elástica sistemas é minimizar os custos de aprovisionamento de recursos enquanto atende os objetivos de nível de serviço (SLOs). Com o surgimento de serviços elásticas, e lojas de valores-chave, mais particularmente elásticas, que pode escalar horizontalmente por adição / remoção de servidores, as organizações percebem potencial em ser capaz de reduzir o custo ea complexidade de aplicações em grande escala da Web 2.0. Um controlador de elasticidade bem projetado ajuda a reduzir o custo de serviços de hospedagem usando o provisionamento dinâmico de recursos e, entretanto, não compromete a qualidade do serviço. Um controlador de elasticidade muitas vezes precisa ser treinado on-line ou off-line, a fim de torná-lo inteligente o suficiente para tomar decisões sobre a desova ou removendo instâncias extras quando aumento ou diminuição da carga de trabalho. No entanto, existem duas questões principais sobre o processo de formação modelo de controle. Uma quantidade significativa de obras recentes treinar os modelos off-line e aplicá-los a um sistema online. Esta abordagem pode levar o controlador de elasticidade para tomar decisões imprecisas, já que nem todos os parâmetros podem ser considerados quando a construção do modelo off-line. A formação completa do modelo consome grandes esforços, incluindo modificar configurações do sistema e alterar as configurações do sistema. Pior, alguns modelos podem até mesmo incluir várias dimensões de parâmetros do sistema. Para superar essas limitações, apresentamos o projeto e avaliação de um gerente de elasticidade pró-ativa auto-treinados para lojas de valores-chave elástica baseados em nuvem. Nosso controlador elasticidade usa on-line de criação de perfil e de apoio máquinas de vetores (SVM) para fornecer um modelo de desempenho de caixa-preta de um aplicativo do SLO violação de uma determinada demanda de recursos. O modelo é atualizado dinamicamente para se adaptar às mudanças no ambiente operacional, tais como variações de padrão de carga de trabalho, reequilíbrio de dados, mudanças

no tamanho dos dados, etc. Temos realizadas e avaliadas nosso controlador usando o Apache Cassandra loja de valor-chave em um ambiente OpenStack Cloud. Nossos experimentos com vestígios de carga de trabalho artificiais mostra que nosso controlador garante um elevado nível de autorizações SLO, mantendo o ótimo global de utilização de recursos.

Abstract

The pay-as-you-go pricing model and the illusion of unlimited resources makes cloud computing a conducive environment for provision of elastic services where different resources are dynamically requested and released in response to changes in their demand. The benefit of elastic resource allocation to cloud systems is to minimize resource provisioning costs while meeting service level objectives (SLOs). With the emergence of elastic services, and more particularly elastic key-value stores, that can scale horizontally by adding/removing servers, organizations perceive potential in being able to reduce cost and complexity of large scale Web 2.0 applications. A well-designed elasticity controller helps reducing the cost of hosting services using dynamic resource provisioning and, in the meantime, does not compromise service quality. An elasticity controller often needs to be trained either online or offline in order to make it intelligent enough to make decisions on spawning or removing extra instances when workload increase or decrease. However, there are two main issues on the process of control model training. A significant amount of recent works train the models offline and apply them to an online system. This approach may lead the elasticity controller to make inaccurate decisions since not all parameters can be considered when building the model offline. The complete training of the model consumes large efforts, including modifying system setups and changing system configurations. Worse, some models can even include several dimensions of system parameters. To overcome these limitations, we present the design and evaluation of a self-trained proactive elasticity manager for cloud-based elastic key-value stores. Our elasticity controller uses online profiling and support vector machines (SVM) to provide a black-box performance model of an application's SLO violation for a given resource demand. The model is dynamically updated to adapt to operating environment changes such as workload pattern variations, data rebalance, changes in data size, etc. We have implemented and evaluated our controller using the Apache cassandra key-value store in an OpenStack Cloud environment. Our experiments with artificial workload traces shows that our controller guarantees a high level of SLO commitments while keeping the overall resource utilization optimal.

Palavras Chave
Keywords

Palavras Chave

Computação Em Nuvem

Elasticidade Controlador

Armazenamento Na Núvem

Previsão de carga de trabalho

Objetivo de nível de serviço

Treino Online

Análise de séries temporais

Keywords

Cloud Computing

Elasticity Controller

Cloud Storage

Workload prediction

SLO

Online Training

Time series analysis

Índice

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The Cloud platform provides a set of desired properties, such as low setup cost, professional maintenance and elastic provisioning. As a result, hosting services in the Cloud are becoming more and more popular. Elastically provisioned services in the Cloud are able to use platform resources on demand, thereby reducing hosting costs by appropriate provisioning. Specifically, instances are added when they are needed for handling an increasing workload and removed when the workload drops. Since users only pay for the resources that are used to serve their demand, elastic provisioning saves the cost of hosting services in the Cloud.

A well-designed elasticity controller aids lessen the cost of hosting services using dynamic resource provisioning and, in the meantime, does not compromise service quality. An elasticity controller often needs to be trained either online or offline in order to make it intelligent enough to make decisions on spawning or removing extra instances, when workload increases or decreases. Specifically, the trained model maps a monitored parameter from the runtime system, such as CPU utilization and incoming workload intensity, and a controlled parameter, such as request percentile latency. The model inputs (monitored parameters) and the quality of the model directly affects the quality of the elasticity controller that influences system provision cost and service quality.

1.2 Problem Statement

In general, elasticity in a distributed storage system is achieved in two ways: One approach reacts to real-time system metrics such as workload intensity, I/O operations, CPU usage, etc. It is often called a *reactive control*. The other approach uses historic data of a system to carry out workload prediction and control for future time periods. It is referred to as *proactive control*. Reactive control can scale a system with a good accuracy because scaling is based on observed

workload pattern. However, the system reacts to workload pattern only after it is observed. Therefore, as a result of data/state migration when adding/removing nodes in a distributed storage system, SLO[1] violations (Armbrust et al. 2010) are evident during the initial phase of scaling. On the other hand, proactive control avoids this by preparing the nodes in advance, minimizing the SLO violations. However, workload prediction accuracy is application specific. Furthermore, some workload patterns are not even predictable. Thus, appropriate prediction algorithms need to be applied to minimize workload prediction inaccuracies. Workload prediction determines the scaling accuracy which in turn impacts SLO guarantees.

In this work, we strive to improve the input and the model training process of an elasticity controller. For the model inputs, we investigate different algorithms to predict the pattern of our input metrics, i.e. the intensity of the workload. With different prediction algorithms, we are able to obtain high prediction accuracy even for different input patterns. With accurate inputs, we then focus on the model training of the elasticity controller. A well trained model improves the accuracy of the system through a resizing command issued by the controller.

However, there are two main issues on the process of control model training. A significant amount of recent works train the models offline and apply them to an online system. This approach may lead the elasticity controller to make inaccurate decisions since not all parameters can be considered when building the model offline. For example, the varying of data size and the inferences of VMs are usually not considered in model building. With online training, the model is able to adapt itself to the current workload composition and the operating environment. Another disadvantage of offline training is that control models are usually trained with only representative cases for simplicity. The complete training of the model consumes large efforts, including modifying system setups and changing system configurations. Worse, some models can even include several dimensions of system parameters. Read request intensity, write request intensity, and data rebalance workloads map to request latency are examples of mapping three monitored parameters to a controlled parameter. The effort of changing parameters and system setups to cover a fine-grained three dimensional space is huge.

[1]Service Level Objectives (SLOs) are ways of measuring the performance of a service provider regarding a particular service. It's a key component of a Service Level Agreement (SLA) between a service provider and a customer and are often quantitative and have related measurements

1.3 Contribution

In this thesis we build an "out-of-the-box" elasticity controller that can be easily embedded in any cloud system with certain minimum requirements. The controller is able to adapt to different workload patterns and its control model is able to get trained automatically by only specifying monitored parameters and controlled parameter (target).

The core of our demand prediction module will be supported by a two-level algorithm. The first level is the workload forecasting/prediction which estimates the incoming workload of the system for future time periods. No single elasticity algorithm is suitable for future workload predictions for all workloads because different applications' workloads are dynamic (Ali-Eldin et al. 2013). To support different workload scenarios, more than one prediction algorithm is used. Different workload patterns are collaboratively predicted by several prediction algorithms. Existing techniques can be applied for predicting the traffic incident on a service and a simple weighted majority algorithm can be used to select the best prediction. The second level algorithm estimates the system behavior over the prediction window using an online trained performance model to provision resources based on the prediction. Training once and predicting forever is not suitable for cloud environments' demands prediction due to the dynamic characteristics of input patterns. In order to capture up-to-date characteristics of the system, the prediction and performance models need to be updated periodically based on the new requests history. The elasticity controller should be able to function after being deployed in the platform for a sufficient amount of time in order to get self-trained and it continues improving/evolving the model during runtime.

In summary, the contributions of this work are as follows:

1. The prediction module of the elasticity controller is able to select/adjust prediction algorithms for different workload patterns to achieve better prediction accuracy and thus accurate capacity provisioning decisions.
2. The elasticity controller is able to train itself automatically online, in the warm up phase, and after sufficient amount of time, it should be able to serve the real workload.
3. The online trained model continues improving/evolving itself during runtime.

1.4 Context

This work was carried out under the supervision of Ahmad Al-Shishtawy, Ying Liu and Associate Professor Vladimir Vlassov, who have related publications and ongoing research on self-management and automatic control for cloud-based storage services (Al-Shishtawy & Vlassov 2013) (Liu et al. 2015).

1.5 Thesis Outline

This thesis is organized as follows. The background and related work is reviewed in the next chapter. Chapter 3 presents the controlling framework architecture in detail. In chapter 4, we present the implementation details. In chapter 5, we present the experimental study. Finally, the last chapter concludes this thesis.

2 Background and Related work

2.1 Background

This section introduces important concepts in understanding the use and management of elasticity managers for cloud-based storage services.

2.1.1 Cloud computing features

According to the National Institute of Standards and Technology (NIST), Cloud computing is defined as *"a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* (Mell & Grance 2011).

This model emphasizes on five essential features, three service models (*Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS)*) and four deployment models (*Private cloud, Community cloud, Public cloud and Hybrid cloud*) that together categorize ways to deliver cloud services. (Lorido-Botran et al. 2014) gives a brief description of the service and deployment models.

2.1.1.1 Essential features

- On-demand self-service: The capability to provide computational resources such as service time and network storage automatically whenever needed.
- Broad network access: Capabilities are provided over the network and accessed through standard mechanisms that allow heterogeneous thin or thick client platforms to make use of the computational resources.

- Resource pooling: Cloud subscribers are served by pooling the cloud provider's resources in a multi-tenant model where different physical and virtual resources are dynamically assigned or reassigned to subscribers according to their demand. The resources include storage, processing, memory, network bandwidth among others. Additionally, details such as resource location, specific configurations, failures, etc are abstracted from the subscriber.

- Rapid elasticity: Cloud services can be elastically provisioned and released, in some cases automatically, to quickly scale in and out depending on the demand. The cloud provider provide an illusion of unlimited resources, so that the consumer may request for resources in any quantity at any time.

- Measured service: To provide transparency to the cloud provider and consumer of the utilized service, cloud resource usage could be monitored, controlled and reported. In cloud computing, a metering capability[1] is used to control and optimize resource use. Just like utility companies sell services such as municipality water or electricity to subscribers, cloud services are also charged per usage metrics - **pay as you go**. The more a resource is utilized, the higher the bill. In order to keep consumers happy with a system, it is important to keep real time constraints on its performance without compromising service quality.

The pay-as-you-go pricing model and the illusion of unlimited resources makes cloud computing a conducive environment for provision of elastic services where different resources are dynamically requested and released in response to changes in their demand. The benefit of elastic resource allocation to cloud systems is to minimize resource provisioning costs while meeting SLOs. With the emergence of elastic services, and more particularly elastic key-value stores, that can scale horizontally by adding/removing servers, organizations perceive potential in being able to reduce cost and complexity of large scale Web 2.0 applications.

2.1.1.2 Cloud services

Cloud services can be broadly characterized into two categories: state-based and stateless. Scaling stateless services is straightforward because no overhead of state migration is involved.

[1] Typically this is done on a pay-per-use or charge-per-use basis

But, scaling state-based services requires state-transfer and/or replication, which adds some overhead during the scaling. In this work, we study the elastic scaling of state-based distributed storage systems. Service latency is one of the most commonly defined SLOs in a distributed storage system (Liu et al. 2015). Satisfying latency SLOs in back-end distributed storage systems that serve interactive, latency sensitive web 2.0 applications is desirable.

2.1.2 Importance of Elasticity

According to (Herbst et al. 2013), *"Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible"*. The emergence of large scale Web 2.0 applications impose new challenges to the underlying provisioning infrastructure such as scalability, highly dynamic load, partial failures, etc. These web applications often experience dynamic workload patterns and in order to respond to changes in workload, an elastic service is needed to meet SLOs at a reduced cost. Specifically, instances are spawned when they are needed for handling an increasing workload and removed when the workload drops. Therefore, enabling elastic provisioning saves the cost of hosting services in the cloud in that users only pay for the resources that are a classification of elasticity mechanisms based along four characteristics (Galante & de Bona 2012), while Figure 2.2 depicts a high-level view of an elastic software (Jamshidi et al. 2014). For more on what is elasticity and what it is not, see (Herbst et al. 2013). As shown in Figure 2.2, the architecture of an elasticity controller generally follows the idea of MAPE-K (Monitor, Analysis, Plan, Execute - Knowledge) control loop.

2.1.3 Workload Characteristics/Classification

Characterizing/classifying a workload plays a vital role in designing systems such as elastic controllers where, for instance, resources are allocated according to the changing workload. It helps one to understand the current state of the system (Arlitt & Jin 2000). Cloud providers such as Amazon[2] and Rackspace[3] host various applications with different workload patterns.

[2] Amazon Elastic Compute Cloud (amazon ec2), https://aws.amazon.com/solutions/case-studies accessed June 2015

[3] The Rackspace Cloud, https://www.rackspace.com/cloud accessed June 2015

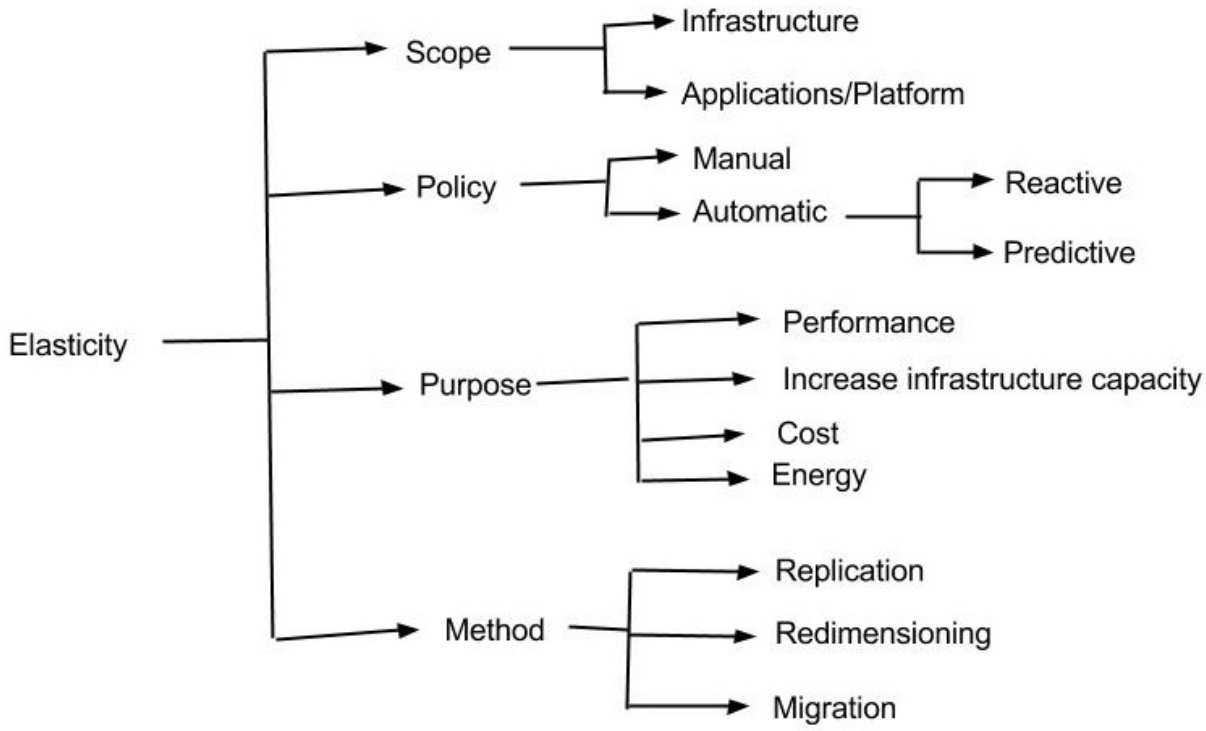

Figure 2.1: Classification of Elasticity Mechanisms. Adopted from Figure 1 of (Galante & de Bona 2012).

Since there are many parameters that can be used to characterise a workload, workload characterization is not an easy task (Gusella 1991). Even for a single application, different users access it with different usage patterns.

Although generic representative workload patterns have emerged for web applications, it is important to consider applications' workload case-by-case. Some workloads have diurnal patterns (repetitive/cyclic pattern). For example, the daytime usage of a resource is regularly greater than its usage at night. On the other hand, some workloads have seasonal patterns. For instance, an online store may experience a drastic increase of its workload before a particular season such as christmas. Due to unusual events such as market campaigns or special offers, some applications may also experience exponential growth in their workloads over a short period of time. This phenomenon is usually referred to as the **"Slashdot effect"** or **"Flash crowds"**[4]. It typically occurs when a smaller website is linked to a popular website, triggering a drastic increase in traffic which causes the smaller website to slow down or even become temporarily unavailable. Unfortunately, some workloads have no patterns at all or have some

[4]Flash crowds: viral popularity growth

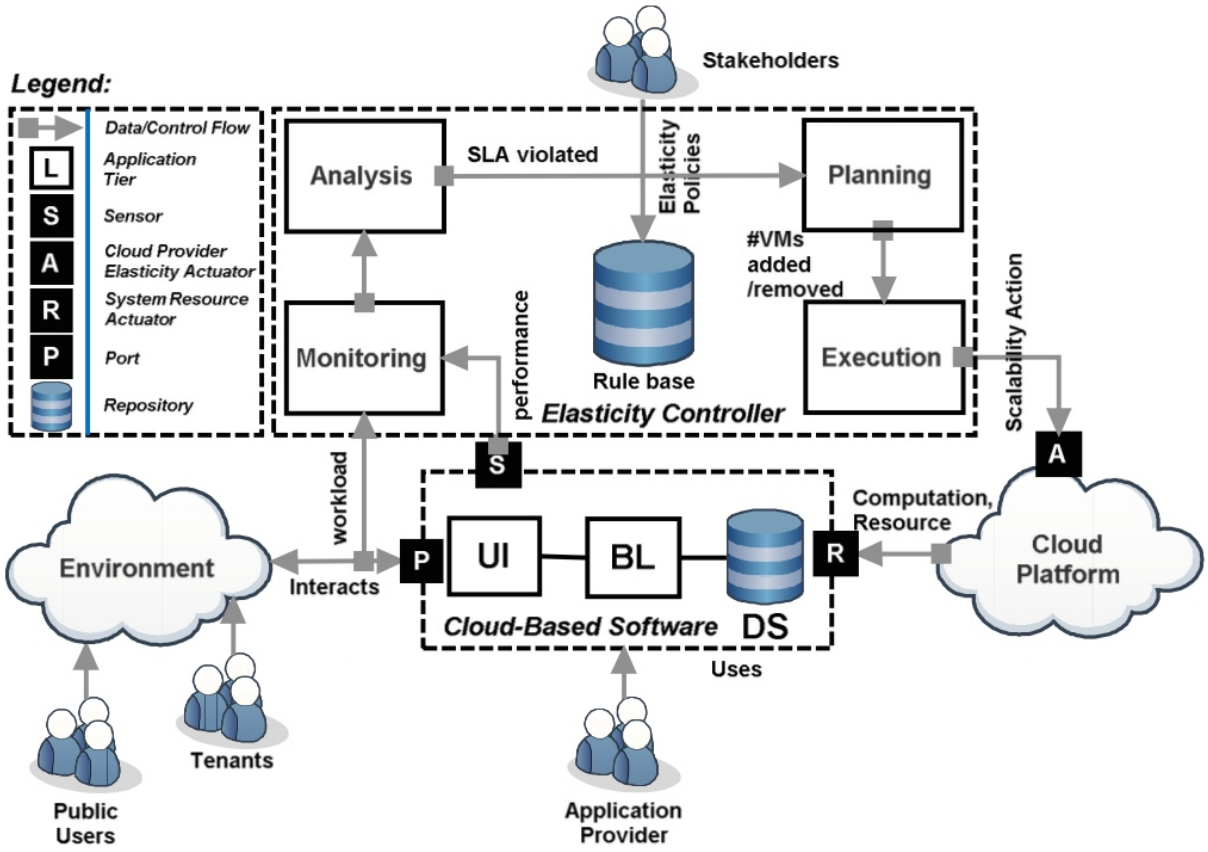

Figure 2.2: High level view of elastic software. Adopted from Figure 1 of (Jamshidi et al. 2014).

weak patterns which makes them difficult to analyze. It is possible to make predictions for workloads with patterns and adjust provisioning based on the expected demands.

2.1.4 Auto-scaling techniques for elastic applications in cloud environments

The goal of an auto-scaling system is to automatically fine-tune acquired resources of a system to minimize resource provisioning costs while meeting SLOs. An auto-scaling technique automatically scales resources according to demand. Different techniques exist in the literature that addresses the problem of auto-scaling. As a result of the wide diversity of these techniques, that are sometimes combination of two or more methods, it is a challenge to find a proper classification of auto-scaling techniques (Lorido-Botran et al. 2014). However, these techniques could be divided into two categories: reactive and proactive. In outline, reactive approach reacts to real time system changes such as incoming workload while a proactive approach relies on historical access patterns of a system to anticipate future needs so as to acquire or release resources in ad-

vance. Each of these approaches have its own merits and demerits (Liu et al. 2015). Under the proactive and reactive categories, the following are some of the widely used auto-scaling techniques: threshold-based policies, reinforcement learning, queuing theory, control theory and time series analysis. Time series analysis is purely a proactive approach, whereas threshold-based rules (used in Amazon and RightScale[5]) is a reactive approach. Contrary, reinforcement learning, queuing theory and control theory could be used with both proactive and reactive approaches, But they also exhibit the following demerits (Lorido-Botran et al. 2014):

- Reinforcement Learning - In addition to the long time required during the learning step, this technique adapts only to slowly changing conditions. Therefore, it cannot be applied to real applications that usually suffer from sudden traffic bursts.
- Queuing theory - Impose hard assumptions that may not be valid for real, complex systems. They are intended for stationary scenarios, thus models need to be recalculated when conditions of the application change.
- Control theory - Setting the gain parameters can be a difficult task.

2.1.5 Performance metrics or variables for auto-scaling

Any auto-scaling technique requires a good monitoring component that gathers various and updated metrics about system current state at an appropriate granularity (e.g per second, per minute, per hour). It is important to review which metrics can be obtained from the target system. For example, the use of percentile as the SLO metric by Amazon's Dynamo is driven by the desire to provide a quality service to almost all customers. The following shows a list of performance metrics or variables for scaling purposes proposed by H Ghanbari et al. (Ghanbari et al. 2011).

- General OS Process: CPU-time, pagefaults, real-memory (resident set) size;
- Hardware: CPU utilization, disk access, network interface access, memory usage;
- Load balancer: request queue length, session rate, number of current sessions, transmitted bytes, num of denied requests, num of errors;

[5]Right Scale, http://www.rightscale.com, accessed June 2015

- Web server: transmitted bytes and requests, number of connections in specific states (e.g. closing, sending, waiting, starting,...;
- Application server: total threads count, active threads count, used memory, session count;
- Database server: number of active threads, number of transactions in (write, commit, roll-back, ...) state;
- Message Queue: average number of jobs in the queue, average job's queuing time.

2.2 Related work

In this section we review prior systems addressing the autonomic control of elastic cloud storage services. We focus on the published systems, because their ideas and limitations provide the motivation for our work. In particular, we study the approach taken by these systems to workload prediction, monitoring and their model training procedure.

2.2.1 The SCADS Director, Elastman and ProRenaTa

The SCADS Director's (Trushkowsky et al. 2011) solution targets storage systems such as key-value stores intended for horizontal scalability that serve interactive web applications. This paper highlights that using percentile based response time as a measured input in control is not suitable because of its high variance. Therefore, it presents a more effective approach, called model-based control (Model Predictive Control). In model-based approach, the controller uses different input patterns/dimensions than the one it is trying to control. A significant amount of recent works use this approach (e.g. Elastman (Al-Shishtawy & Vlassov 2013), ProRenaTa (Liu et al. 2015), (Gong et al. 2010), (Lim et al. 2010), ...). However, their models are trained offline and applied to online systems and that they are usually trained with only representative cases for simplicity.

Offline training is done using data from performance of application on a small scale benchmark test, from historical performance data, or from application performance under a particular workload. Performance models based on model-based control, which are trained offline, are not convenient in real world settings for several reasons. First, experiments on benchmark

tests may not reflect the capacity of applications in production. Second, because of how an application is used, changes in the operating environment and changes in the application itself, the performance of web 2.0 applications changes frequently. These challenges can be avoided by an online trained model, i.e, the model is retrained continuously based on the latest performance metrics from the production system.

ProRenaTa is an elasticity controller that combines both reactive and proactive approaches to leverage on their respective advantages. It also implements a data migration model for handling the scaling overhead. The data migration model provides ProRenaTa with the time that is needed to conduct a scaling plan. The SCADS director also handles data migration by copying as little as possible. It monitors the demand for particular file parts to identify popular parts in order to increase their replication or move them to empty servers. On the other hand, ElastMan combines both feedforward and feedback control to build a scale-independent model of a service for a cloud-based elastic key-value stores. Elastman uses feedforward control to respond to spikes in the workload and feedback control to handle diurnal workload and correct modeling errors. In Elastman, the controller is disabled during the data rebalance operation. As earlier stated, the models of these systems are trained offline, which is one of the motivation for our proposed system.

In summary, key concepts from these works were the use of a performance model to avoid measurement noise and data migration handling during the scaling process.

2.2.2 AGILE

AGILE (Nguyen et al. 2013) provides online, wavelet-based medium-term (up to 2 minutes) resource demand prediction with adequate upfront time to start new application servers before performance degrades i.e. before application SLO is affected by the changing workload pattern. In addition, AGILE uses online profiling to obtain a resource pressure model for each application it controls. This model calculates the amount of resources required to maintain an application's SLO violation rate at a minimal level. It does not require white-box application modelling or prior application profiling.

Our proposed performance model considers several dimensions of system parameters, unlike our model, AGILE derives resource pressure models for just CPU without considering other resources such as memory, network bandwidth, disk I/O, applications workload inten-

sity etc. A multi-resource model can be built in two ways. Each resource can have a separate resource pressure model or a single resource pressure model can represent all the resources. In this thesis, we adopt the latter approach.

2.2.3 Zoolander

Zoolander (Chakrabarti et al. 2012) provides an efficient latency management in Key-Value stores. Research shows that NoSQL stores such as Apache Cassandra, Zookeeper, and Memcached can attain 10^{10} accesses per day even in cases of software failures, workload changes and performance bugs (Chakrabarti et al. 2012). However, achieving low latency for every access still remains a challenge. This is because unlike metrics such as throughput, latency exhibits diminishing returns under scale out approaches. Factors such as DNS timeouts, garbage collection and other unusual events can hold system resources occassionally. As as result, the latency of some accesses can increase drastically, although these events hardly have effect on throughput.

Zoolander uses a set of analytical models to provide the expected SLO under a workload and replication strategy. This paper emphasizes that interactive web applications require NoSQL stores that provide low latency all the time. In this work, we use the 99^{th} percentile of read latency as the controlled parameter to our Key-Value store elasticity controller to maintain the latency at the desired level.

2.2.4 Assessment of existing prediction algorithms

A significant amount of literature exists that can be applied for predicting the traffic incident on a service. In most cases, to support different workload scenarios, more than one prediction algorithms are used. Figure 2.3 presents a few of this prediction algorithms that are relevant to our work (Ref: 1 (Trushkowsky et al. 2011); 2 (Gong et al. 2010); 3 (Liu et al. 2015); 4 (Danny Yuan, Neeraj Joshi, Daniel Jacobson, Puneet Oberai); 5 (Roy et al. 2011)). The general conclusion extracted from this study is the need to provide an efficient auto-scaling techniques that are able to adapt to the changing conditions of applications workloads. In this thesis, we propose using a predictive auto-scaling technique based on time series forecasting algorithms.

The key concept from these works is that in order to support different workload scenarios,

at least more than one prediction algorithm is used. In most cases the pattern of the workload to be predicted is defined or known, which is not in our case. The most important aspect is how switching is carried out among the prediction algorithms which is not clear in most of these works. We therefore propose a simple weighted majority algorithm to handle this.

Ref	Auto-scaling Technique	Metric	Typical Workload	Comments
1	Control theory - Model Predictive Control(MPC)	system workload	Effective for both Hotspot and gradual variations	MPC requires an accurate model of the system. Better than classical closed loop techniques.
2	Time series analysis - Signal processing (FFT) for cyclic && Discrete Markov chains for non-cyclic	CPU load (focus), memory, I/O and network	Cyclic and and non-cyclic workloads	Compared with auto-regression, auto-correlation, histogram, mean, max
3	Time series analysis - Signal processing (wiener filter for short time forecast) && auto-correlation for long time forecast	system workload	stable load and cyclic behavior, periodic peaks, Random peaks and background noise.	Compared with feedback controller and ARIMA (prediction-based approach). ARIMA and this approach achieve same SLA commitments except ARIMA doesn't utilize resources.
4	Time series analysis - Signal processing (FFT) for regular traffic && linear regression for spikes	user traffic	Rapid spike, outages, variable traffic patterns	Not open-source. Also employs outlier detection algorithms to remove invalid points.
5	Time series analysis - autoregressive moving average method (ARMA) - second order	number of users in the system		No comparison with other methods

Figure 2.3: Existing prediction algorithms

2.3 Summary

In this chapter we introduced the key concepts of cloud computing and discussed its features. We explained the importance of elasticity in the cloud and mentioned the general architecture of an elastic software. An autonomic controller is necessary to add or remove resources in an automatic way. Finally we presented some important related works.

3 Solution Architecture

In this section, we present the design of our self-trained proactive elasticity manager, which is an elasticity manager for distributed storage systems that provides online training and proactive control in order to achieve high system utilization and less SLO violation, and its prototype implementation for controlling the Apache Cassandra key-value store. Firstly, we introduce Cassandra, then describe our elasticity manager in terms of data collection, workload prediction, online training, control decisions, and actuation.

Figure 3.1 outlines the architecture of our system. Conceptually, the Data collector, Workload prediction, Online training and Controller operate concurrently and communicate by message passing. Based on the workload prediction result and updated system model, the controller invokes the cloud storage API to add or remove servers.

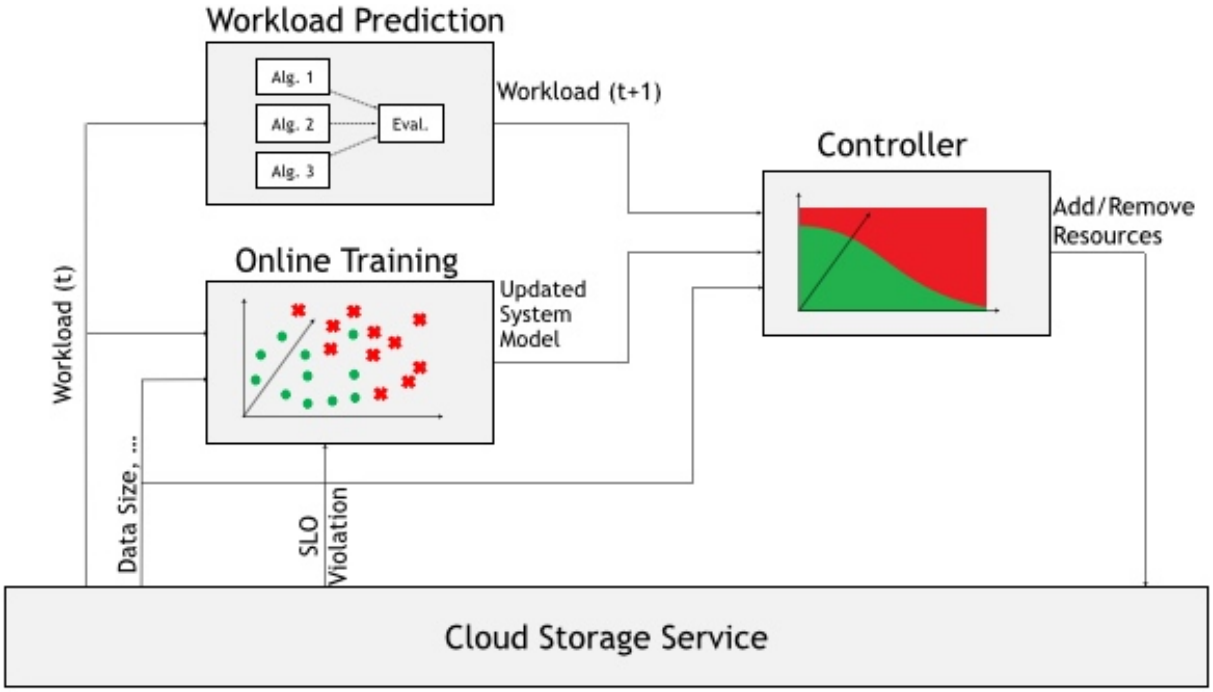

Figure 3.1: Self-trained proactive elasticity manager architecture

Table 3.1: Cassandra data model w.r.t relational data model

Relational Model	**Cassandra Model**
Database	Keyspace
Table	Column Family
Primary Key	Row Key
Column name	Column name/key
Column value	Column value

3.1 Storage service: Apache Cassandra key-value store

Cassandra (Lakshman & Malik 2010), a top level Apache project, is a decentralized structured storage system born at Facebook and built on Amazon's Dynamo (DeCandia et al. 2007) and Google's BigTable (Chang et al. 2008).
Features such as cluster management, replication and fault tolerance are adopted from Dynamo, while columnar data model and storage architecture features are adopted from BigTable. Table 3.1 shows Cassandra's data model w.r.t to the relational database model. Cassandra provides the capability of relational data model on top of key value storage by extending the basic key value model with two level of nesting. Cassandra Query Language (CQL) is the primary language for communicating with the Cassandra database.

Cassandra is the ideal database for today's modern applications, as it supports an infrastructure of hundreds of nodes that may be spread across different data centers. It uses consistent hashing to partition data across the cluster, hence the departure and arrival of a node only affect its immediate neighbors. It also ensures scalability and high availability without compromising the overall performance of a system, by allowing replication even across multiple data centers as well as allowing for synchronous and asynchronous replication for each update. Furthermore, Cassandra was designed to manage large amounts of data spread across multiple machines while ensuring highly available service without single point of failure (SPOF). In addition, its throughput increase linearly as new machines are added.

In practice, read or write requests can be sent to any node in the cluster because all nodes are peers. When a node receives a read or write request from a client, it becomes the coordinator for that particular client operation. The coordinator acts as a proxy between the nodes (replicas) that have the data being requested and the client application.

The idea of how Cassandra handles read and write operations is important because these

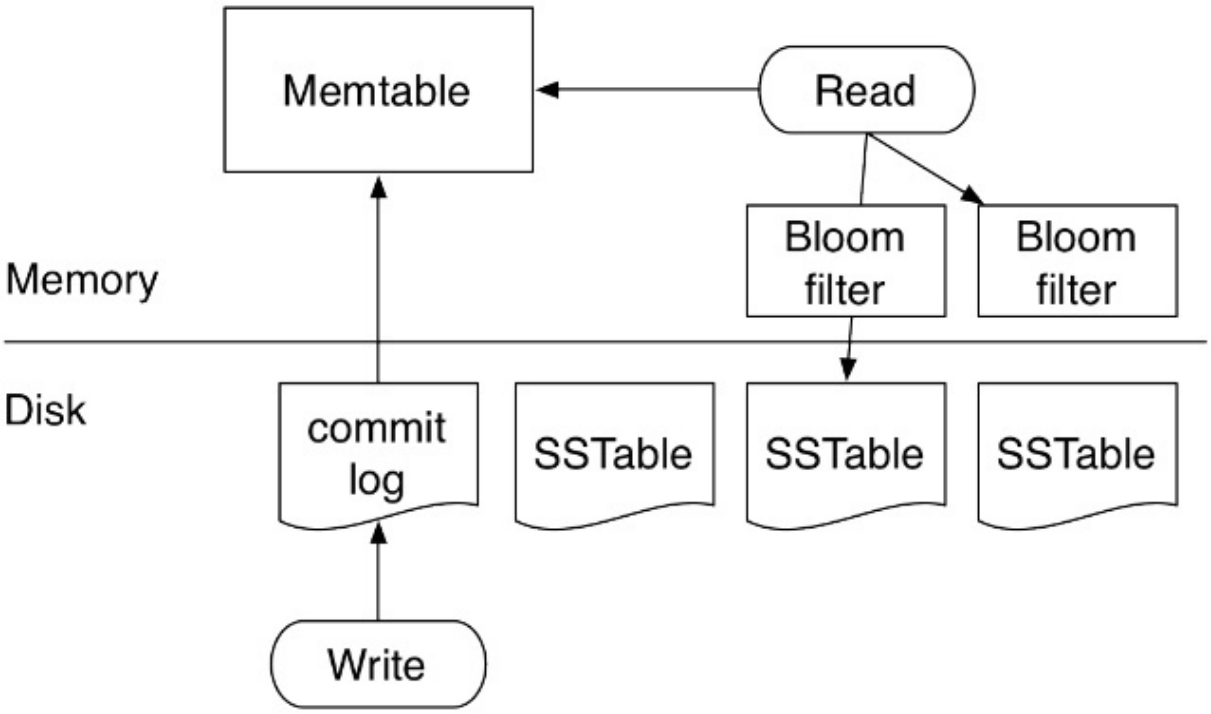

Figure 3.2: Cassandra read and write

operations impact the overall behaviour of a system. As shown in Figure 3.2, when a write operation arrives at a coordinator, it's first written to a commit log for recoverability and durability, then it is written to an in-memory data structure. The in-memory data structure is then dumped into the disk as SSTable once it exceeds a tunable limit. All the writes to the commit log are sequential to maximize the disk throughput, hence Cassandra achieves a higher write throughput than read throughput. On the other hand, when a read operation arrives, the in-memory data structure is first queried before looking into the file (SSTables) on disk. The bloom filter summarizes the keys in the file and prevents the lookups into files that do not contain the key.

Cassandra may issue read/write queries for unexpected reasons such as consistency maintenance or speculative operations which may bias the results. So disabling features such as *read_repair_chance, speculative_retry, and dc_local_read_repair_chance* for all queries may improve system performance, but results may not be consistent.

Since the topic of this work is not specific to this one storage system, instead of presenting it in detail we refer the interested reader to the initial Cassandra paper (Lakshman & Malik

2010), and the project website[1]. For more information about partitioning, replication, tune-able consistency levels, membership, failure handling and scaling refer to those articles.

Our choice of Cassandra as a prototype component puts important constraints on the design of our data collector. The issues we consider most significant are discussed below.

3.1.1 Sensing: measuring system performance

In order to capture the dynamic behavior of the target system as it experiences changes in workload, a data collector component which act as a monitor is necessary. Monitoring is important in capturing the performance of virtual servers during runtime as they come across different workload traffic pattern. Any auto-scaling system requires a good monitoring component that gathers various and updated metrics about system current state at an appropriate granularity (e.g per second, per minute, per hour). The data collector component basically polls monitored performance metrics from the target system, receiving a histogram of monitored parameters since the last pull request.

In our thesis, we describe how Apache Cassandra storage system was modified to obtain sensor input for our controller.

3.1.2 Monitoring a Cassandra Cluster

In order to diagnose and plan capacity of our Cassandra cluster, understanding its performance features was critical. Cassandra uses Java Management Extensions (JMX) to expose various statistics and management operations. The JMX[2] technology provides tools for managing and monitoring Java applications and services. Cassandra exposes statistics and operations that can be monitored during its normal operation using JMX-compliant tools such as:

1. The Cassandra nodetool utility - command-line interface included in the Cassandra distribution for monitoring and executing established routine operations. It provides commands for observing particular metrics for tables, compaction statistics and server metrics such as;

[1] Apache Cassandra, http://cassandra.apache.org, accessed June 2015

[2] JMX technology provides the tools for building distributed, Web-based, modular and dynamic solutions for managing and monitoring devices, applications, and service-driven networks.

(a) nodetool cfstats - provides statistics for each table and keyspace

(b) nodetool cfhistograms - displays information about a table such as number of SSTables, read/write latency, column count and row size.

(c) nodetool netstats - displays statistics about network connections and operations (streaming information).

(d) nodetool tpstats - provides usage statistics of thread pools such as completed, pending and active tasks.

2. DataStax OpsCenter management console - provides a centralized graphical user interface for monitoring and managing Cassandra cluster. OpsCenter provides three general categories of metrics: Operating system metrics, cluster metrics and table metrics. The information provided can either be cluster-wide or per-node information.

3. JConsole - tool for monitoring Java applications such as Cassandra that complies to the JMX specifications. It uses the instrumentation of the Java VM to render statistics about the performance and resource consumption of applications running on the Java platform.

Monitoring Cassandra using these tools consumes a significant amount of system resources. Furthermore, not all the desired metrics can be obtained from these tools. For instance, considering the read/write latency, they only provide average latency for the entire lifetime of the JVM, without options to reset the metrics. Therefore, pulling periodic fine grained statistics such as 99^{th} percentile latency[3] for our controller is not feasible with these tools. For these reasons, we modified Cassandra in a way that we could easily get measurements from each node. To measure the latency of each request, we used the maths components of Apache Commons project[4], a library of lightweight, self-contained mathematics and statistics components.

A math component (Descriptive statistics) was incorporated on the write/read path of each Cassandra node and the latency of each operation (put, get) was added to a *DescriptiveStatistics* object which maintains the input data in memory and has the ability of producing "rolling" statistics calculated from a "window" comprising of the most recently added values. More precisely, our measurement clients (Cassandra nodes) consist of a thread performing a receive-reply loop to respond to the data collector's pull requests. Periodically, the data collector con-

[3]Get 99^{th} percentile = x, means that 99% of read operations take below x (ms)
[4]Apache commons, http://commons.apache.org/math, accessed June 2015

nects to a Cassandra node via a socket, requesting its current data. Upon receiving the request, the node replies with its current data and resets its metrics, ready for a new pull request. Using the collected *DescriptiveStatistics* object which contains all the desired metrics from a node, we get several statistic results:

- throughput: put and get throughputs;
- minimum latency: put and get minimum latencies;
- maximum latency: put and get maximum latencies;
- average latency: put and get mean latencies;
- 99^{th} percentile latency: put and get 99^{th} percentile latencies;
- 95^{th} percentile latency: put and get 95^{th} percentile latencies;
- total runtime;
- total number of put and get operations.

In Figure 3.3, the StorageProxy (which is the coordinator of a request) contains the put and get methods and also merges all local and distributed operations in Cassandra. It is here that we track the latency of each and every operation. More concretely, we extended the put and get methods in a way that they also measure the latency of each request. These latencies are then added to the respective *DescriptiveStatistics* objects.

In general, the monitoring instrumentation requires only small amount of work, i.e. collect statistics from storage entry points (proxies, load balancers, etc). Thus, our solution is applicable to other storage systems. Monitoring can also be facilitated through the cloud platforms or third party applications. For instance, Amazon CloudWatch[5] provides monitoring for applications running on Amazon's cloud platform.

3.1.3 Monitored and controlled parameters

Since general workload patterns have emerged for web applications which are useful in controller design and evaluation and which can easily be predicted, the read (get) and write (put)

[5] Amazon CloudWatch, https://aws.amazon.com/cloudwatch/, accessed June 2015

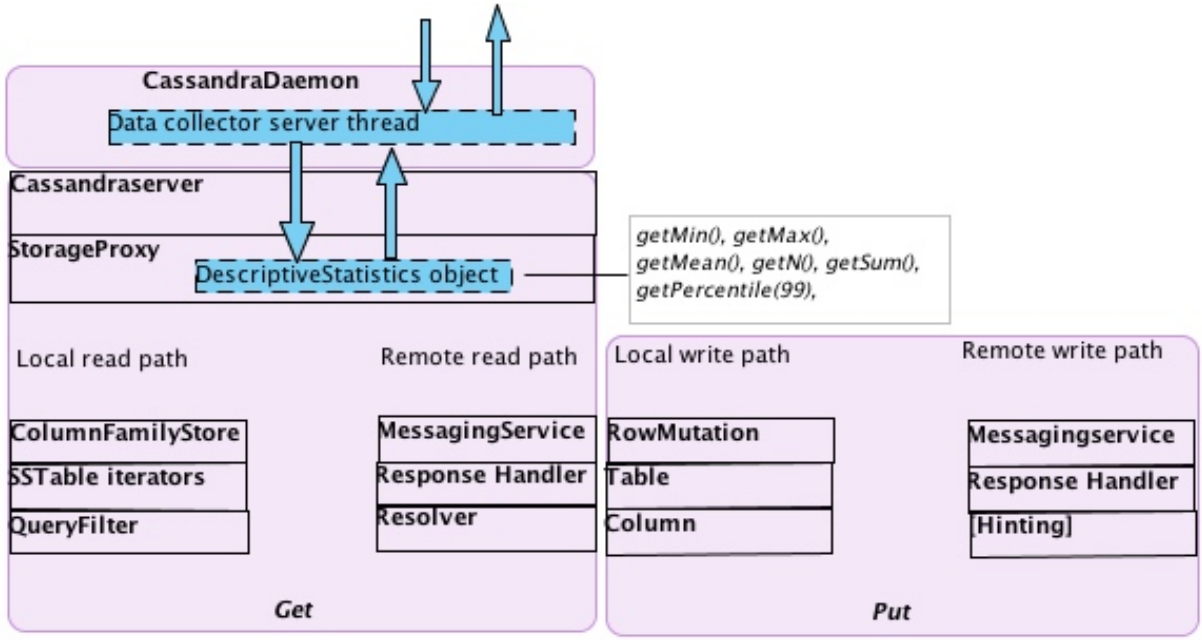

Figure 3.3: Read and write paths in Cassandra

throughputs on each node are used as the **monitored parameters** and defined as input workload in our controller. Since our controller was designed for system parameters with multiple dimensions, data size was used as the 3^{rd} parameter to illustrate this.

In our experiments, we used the get (read) 99^{th} percentile latency as the **controlled parameter**, as this gives more reasonable and stable results (Al-Shishtawy & Vlassov 2013). Therefore, we illustrate that read request intensity, write request intensity and data size mapped to request latency as an example of mapping three monitored parameters to a controlled parameter. This can easily be extended to N dimensions and eases the effort of changing parameters and system setups to cover a fine-grained N dimensional space.

From the data collector, the workload is fed to two modules: workload prediction and online training.

3.2 Workload prediction

The workload from data collector is forwarded to the workload prediction module for forecasting. Workload prediction is needed to estimate the incoming workload of a system for future time periods and it is carried out every prediction window. The workload demand data ac-

quired from periodic monitoring can be considered as a time series data. Hence, predictive models for time series analysis can be used to analyze this workload data so as to make a short term prediction of workload demand. A well-designed predictive model, with an ability to predict the future workload changes accurately is crucial for mitigating the problem of reactive controllers. Considering that there are no perfect predictors, and different applications' workloads are dynamic, no single prediction model is suitable for future predictions for all workloads. Fortunately, several techniques already exist in literature that can be used for predicting the traffic incident on a service.

In this thesis we have studied and analysed several prediction algorithms that are suitable for different workload scenarios. A simple weighted majority algorithm described in section 3.2.7 is then used to select the best prediction at a given time period. The relative accuracy of these algorithms depend on the window size considered and workload pattern. The following algorithms were considered:

3.2.1 Mean

According to this method, also known as the moving averaging method, the predicted workload demand would be the mean value of all the time series data in a given window. Specifically, the prediction is the outcome of averaging the latest t values of the time series. Although this method makes a perfect predictor for steady workloads, it suppresses the peaks leading to underestimation errors. A mathematical representation of this method is provided below (Hansen 1995).

$$X(n+1) = \sum_{i=n+1-t}^{n} \frac{X_i}{t} \tag{3.1}$$

3.2.2 Max and Min

In the Max method, the predicted workload demand would be the maximum value among the values of the time series data in a given window, while in the Min method the prediction value would be the minimum value. These methods try to provide a safe estimation by selecting the minimum or maximum value observed in the recent past. Here, the goal is to estimate the peak values accurately. Although the methods are not efficient, they best prepare the system for the worst case scenario in case of spikes in the workload.

3.2.3 Signature-driven resource demand prediction

This method has been used in PRESS (Gong et al. 2010). PRESS uses a signature derived from historic resource usage pattern to make its prediction. The method have been used for workloads with repeating patterns often caused by iterative computations or repeating requests. Precisely, PRESS uses a Fast Fourier Transform (FFT), a signal processing technique, to discover the presence or absence of a signature. For a detail description of this algorithm refer to the original PRESS paper (Gong et al. 2010). In this thesis, pseudo code 1 was used to implement this algorithm.

3.2.4 Regression Trees model

Regression trees predict responses to data and are basically considered as a variant of decision trees. They specify the form of the relationship between predictors and a response. We first build a tree using the time series data through a process known as recursive partitioning (Algorithm 2) and then fit the leaves values to the input predictors just like Neural Networks. Particularly, to predict a response, we follow the decisions in the tree from the root node all the way to a leaf node which contains the response. Regression trees models are flexible and their ability to do non linear relationships make them good for forecasting.

3.2.5 LIBSVM - A Library for Support Vector Machines

LIBSVM is one of the most widely used Support Vector Machine (SVM) software. SVMs are a popular supervised machine learning method used for regression, classification and other learning tasks (Chang & Lin 2011) (Ovidiu Ivanciuc). Typically, using LIBSVM involves two steps: training a data set to obtain a model and using the trained model to predict information of a given data set. In our work, we used SVM regression for time series prediction. Besides supporting linear regression, SVMs can efficiently accomplish non-linear regression using the "kernel trick"[6], implicitly mapping data into high dimensional feature space. In this thesis, we don't present detail implementation of LIBSVM. For detailed implementation of LIBSVM and Support Vector Regression(SVR), see (Chang & Lin 2011) (Smola & Scholkopf 2004).

[6]In machine learning, a kernel is essentially a mapping function that transforms a given space into some other (usually very high dimensional) space. A kernel function basically represent an infinite dimensional space but still is easy to compute

3.2.6 ARIMA

Autoregressive moving average (ARMA) is one of the most widely used approaches to time series forecasting. ARMA model is convenient for modelling time series data which is stationary. In order to handle non-stationary time series data, ARMA model adopts a differencing component to help deal with both stationary and non-stationary data. This class of models with differencing component is referred to as the autoregressive integrated moving average (ARIMA) model. Specifically, ARIMA model is made up of autoregressive (AR) component of lagged observations, a moving average (MA) of past errors and a differencing component (I) needed to make a time series to be stationary. The MA component is impacted by past and current errors while the AR component shows the recent observations as a function of past observations (Box & Jenkins 1990).

In general, an ARIMA model is represented as "ARIMA(p,d,q)" model where:

- **p** is the number of autoregressive terms (order of AR),
- **d** is the number of differences needed for stationarity, and
- **q** is the number of lagged forecast errors in the prediction equation (order of MA).

It is generally recommended that you stick to models whose at least one of p and q is not greater than one (Mcleod 1993). The following equation represents a time series expressed in terms of AR(n) model:

$$Y^{'}(t) = \mu + \alpha_1 Y(t-1) + \alpha_2 Y(t-2) + \ldots + \alpha_n Y(t-n) \tag{3.2}$$

Equation 3.3 represents a time series expressed in terms of moving averages of white noise and error terms.

$$Y^{'}(t) = \mu + \beta_1 \epsilon(t-1) + \beta_2 \epsilon(t-2) + \ldots + \beta_n \epsilon(t-n) \tag{3.3}$$

where

- $\mu = Mean(1-\alpha)$
- $0 < \alpha \leq 1$

- $0 < \beta \leq 1$
- ϵ is a white noise
- μ is a constant

In this thesis, since we do not know the pattern of our workload, we have chosen some of the types of ARIMA models that are commonly encountered. They include:

- $ARIMA(1,0,0)$ - first-order autoregressive model;
- $ARIMA(0,1,0)$ - random walk;
- $ARIMA(1,1,0)$ - differenced first-order autoregressive model;
- $ARIMA(0,1,1)$ - simple exponential smoothing.
- $ARIMA(2,0,0)$ - second-order autoregressive model

For a time series that is stationary and autocorrelated, a possible model for it is a first-order autoregressive model. On the other hand, if the time series is not stationary, the simplest possible model for it is a random walk model. However, if the errors of a random walk model are autocorrelated, perhaps a differenced first-order autoregressive model may be more suitable. For a detailed explanation of these models, see (Robert Nau).

3.2.7 The Weighted Majority Algorithm

The Weighted Majority Algorithm(WMA) is a machine learning algorithm used to build a combined algorithm from a pool of prediction algorithms (Littlestone & Warmuth 1994). The algorithm assumes that one of some pool of known algorithms will perform well, but no prior knowledge exist about the accuracy of the algorithms. The WMA have different variations suited for different scenarios such as infinite loops, shifting targets and randomized predictions. We present the simple version of WMA in section 3.5, algorithm 3. Generally, the algorithm maintains a list of weights w_1,...,w_n one for each prediction algorithm, and predicts based on a weighted majority vote of the prediction results.

Our workload prediction module is flexible in that any new prediction algorithm can be easily plugged into the system. Figure 3.5 and 3.4 shows a simple arbitrator designed to select

proper prediction algorithm for the incoming workload with respect to prediction accuracy. Our work involved designing and implementing the arbitrator. Figure 3.5 was adopted for our final experiments.

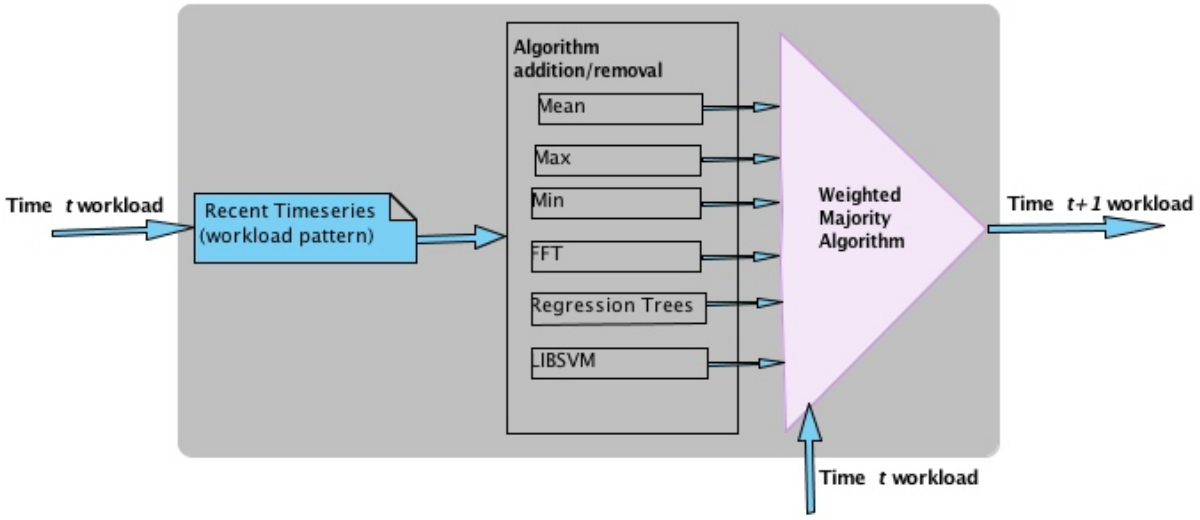

Figure 3.4: Workload prediction module

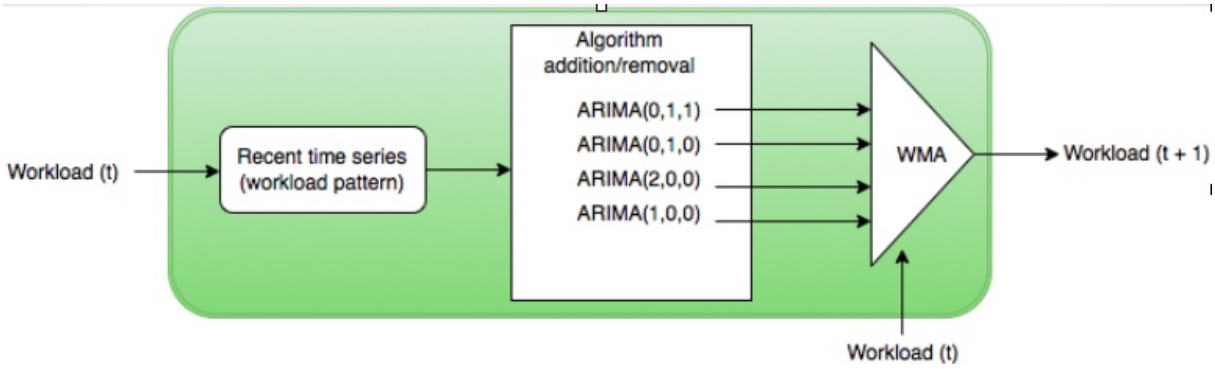

Figure 3.5: Workload prediction module

3.3 Online performance modelling

In order to meet the application's SLO, our controller needs to pick an appropriate resource allocation. One way to do this is to use a performance model of the system to reason about the current status of the target system and make control decisions. Most previous work on performance modelling (e.g.,[(Trushkowsky et al. 2011), (Al-Shishtawy & Vlassov 2013), (Liu et al. 2015), (Gong et al. 2010)]) adapts an offline-trained approach. Our approach uses online profiling and builds a binary classifier using SVM to provide a black-box performance model

of the application's SLO violation for a given resource demand. The model is dynamically updated to adapt to operating environment changes such as workload pattern variations, data rebalance, changes in data size, etc.

We used the monitored parameters given in 3.1.3 to build an online trained performance model for a server i.e. we profile a Cassandra instance under three parameters: write intensity, read intensity and data size. However, using the same profiling method, different models can be build for different server flavors. The performance model is application specific, and may change at runtime due to variations in the monitored parameters. For instance, in Cassandra, a workload with more read requests may take more time to execute than the workload with more write requests. For such reasons, it's important to generate the model dynamically at runtime. Considering a given SLO latency constraint, a server can either satisfy SLO or violate SLO. Therefore, at a given time period our performance model is a line that separates the plane into two regions. The SLO is met in the region under the line while it is violated in the region above the line. In the region on the line the SLO is met with the minimum number of servers, which indicates high resource utilization while guaranteeing SLO requirements.

To start building the model, we collect the pairs of monitored parameters (e.g. read request rate, write request rate, and data size) and corresponding percentile latency with respect to SLOs and design a model based on these data. To build a model (identify the system) means finding how the monitored parameters (read request rate, write request rate, and data size) affects the controlled parameter (99^{th} percentile of read latency) of the key-value store. For example, the latency is much shorter in an underloaded store than in an overloaded store. The following parameters were considered when building the online performance model:

1. Data grid scale - Since we cannot map each and every data point[7] of our measurements on the data grid, we maintain a configurable scale which can be selected depending on the memory and granularity demands.

2. Read/Write latency queue - For each data point, we maintain a queue of most recent read-/write 99^{th} percentile latencies. As the model evolves a point may change from satisfying SLO to violating SLO and vice versa.

[7] Data points correspond to a multidimensional array of monitored parameters mapped to a controlled parameter

3. Confidence level - Refers to the percentage of all the Read latency queue samples that can be expected to satisfy the SLO. For example, 95% confidence level implies that 95% of all the Read latency queue samples satisfy the SLO.

If the application's SLO is affected by multiple parameters, the model can easily be extended to cover them. Furthermore, these parameters are continously adjusted to keep the model consistent with the dynamic cloud application. As a result, an up-to-date performance model is always available for users to query and carry out tasks such as auto-scaling and capacity planning.

We now present how the system parameters were modeled using SVM to obtain the SLA border line.

3.3.1 SVM Binary Classifier

SVMs have become popular classification techniques in a wide range of application domains (Gunn 1998). They provide good performance even in cases of high-dimensional data and a small set of training data. Using the "kernel trick", SVMs are also able to find non-linear solutions efficiently (Cristianini & Shawe-Taylor 2000).

Although users do not require to grasp the underlying theory behind SVM, we briefly describe the important basics to explain our performance model. Figure 3.6 shows the flow of a classification task.

Ideally, each instance of the training set contains a class label and several features or observed variables. The goal of SVM is to produce a model based on the training set. More concretely, given a training set of instance-label pairs $(x_i, y_i), i = 1, ..., l$ where $x_i \in R^n$ and $y_i \in \{1, -1\}^l$, the SVM classification solves the following optimization problem:

$$min_{w,b} \qquad \| w \|^2 + C \sum_i \xi_i \tag{3.4}$$

subject to:

$$\begin{aligned} y^{(i)}(w^T x^i + b) \geq 1 - \xi_i, \quad & i = 1, 2, \ldots, m \\ \xi_i \geq 0, \quad & i = 1, 2, \ldots, m \end{aligned} \tag{3.5}$$

After solving, the SVM classifier predicts 1 if $w^T x + b \geq 0$ and -1 otherwise. The decision

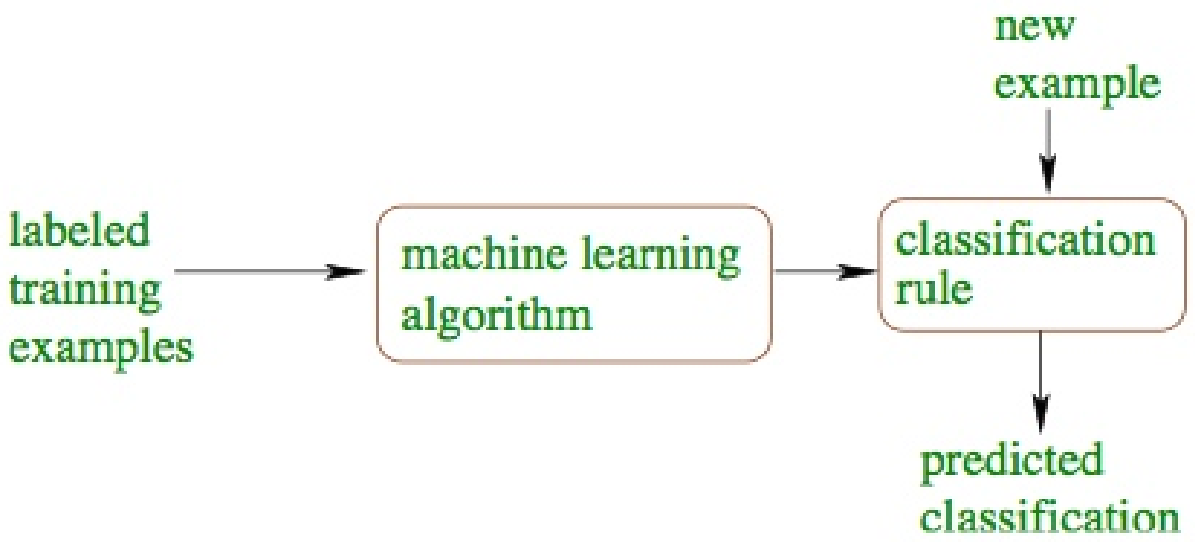

Figure 3.6: The flow of a Classification task

boundary is defined by the following line:

$$w^T x + b = 0 \tag{3.6}$$

Generally, the predicted class can be calculated using the linear discriminant function:

$$f(x) = wx + b \tag{3.7}$$

x refers to a training pattern, **w** as the weight vector and **b** as the bias term. **wx** refers to the dot product, which calculates the sum of the products of vector components $w_i x_i$. For example, in case of training set with three features (e.g. x, y, z), the discriminant function is simply:

$$f(x) = w_1 x + w_2 y + w_3 z + b \tag{3.8}$$

SVM provides the estimates for w_1, w_2, w_3 and b after training.

Our performance model is basically a line (Figure 3.7 and 3.8) given in Equation 3.6. Our controller uses this model to now make control decisions. If the predicted throughput is far above the line, this translates that the system is loaded and servers needs be added and vice versa. When a large change in throughput is observed(predicted), the controller uses the model to determine the new average throughput per server. This is accomplished by calculating the

Figure 3.7: 3D performance model

intersection point between the model line and the line that connects the origin with the point that corresponds to the predicted throughput (Al-Shishtawy & Vlassov 2013). Ideally, to calculate the intersection point between the decision line and the predicted throughput line, we find the equations of the lines and solve for them. More specifically, we can find if they intersect, if there exists values of the parameters in their equations which produce the same point. Chapter one of (Levi 1965) explains how this is done. The slope of the line that connects the origin with the point that corresponds to the predicted throughput is equal to the ratio of the write/read throughput of the predicted workload mix. Since we are only predicting the workload intensity, we assume that the data size will not change in the next prediction window. For example, If the current data size is $5KB$, then the origin of the predicted throughput line would be $(0, 0, 5)$ corresponding to read throughput, write throughput and data size respectively. In a nutshell, the performance model takes a specific workload intensity and data size as the input and outputs the new average throughput per server that is needed to keep the system

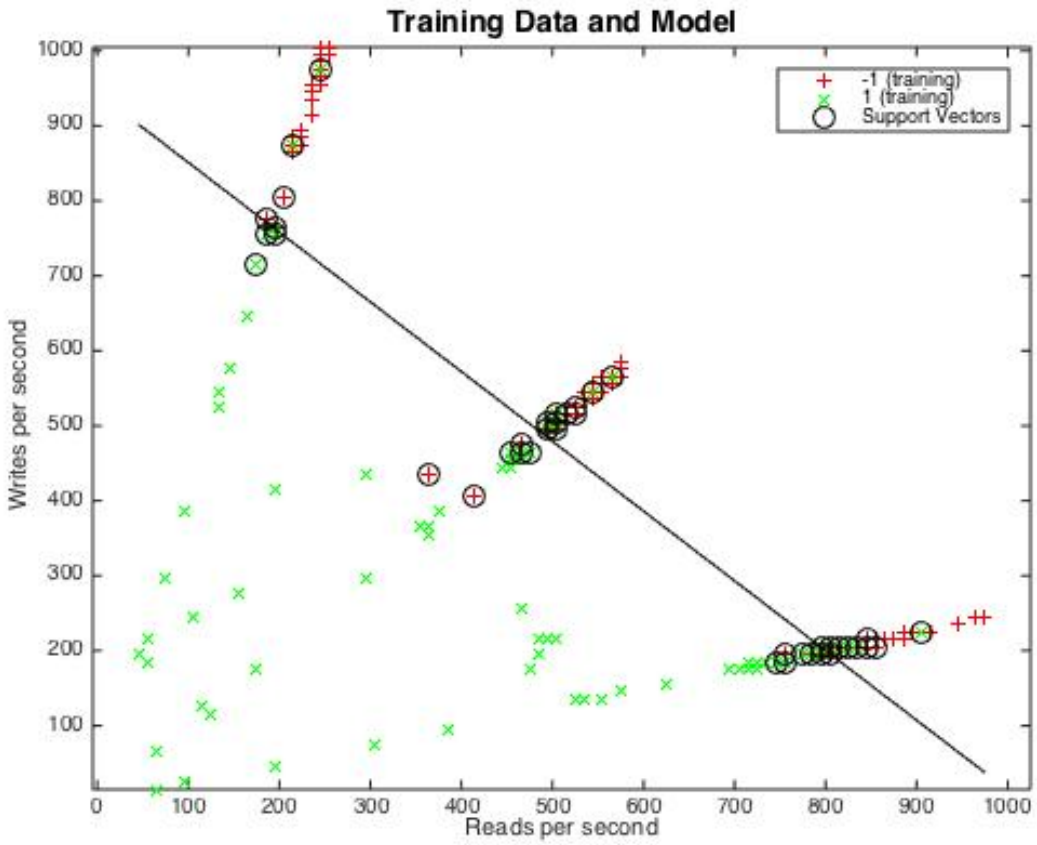

Figure 3.8: SVM Model for System Throughput

at optimal performance. The idea to use the average throughput per server is well motivated by Al-Shishtawy et al. (Al-Shishtawy & Vlassov 2013) i.e. the near linear scalability of elastic key-value stores. This new average throughput per server is then forwarded to the actuator.

3.4 Actuation

The actuator receives the new average throughput per server and calculates the new number of servers using Equation 3.9, that keeps the storage service at optimal performance where the SLO is met with the minimum number of storage servers. From the new number of servers, we then determine the number of servers that should be added or removed and use the Cloud API to request/release resources. Adding or removing new servers will also require a rebalance

API to redistribute the data among servers.

$$NewNumberofServers = \frac{CurrentTotalPredictedThroughput}{NewOptimizedAverageThroughputPerServer} \tag{3.9}$$

subject to:

$$replication\_degree \leq minimum\_servers \leq new\_number\_of\_servers \leq maximum\_servers \tag{3.10}$$

In our experiments adding and removing nodes to/from an existing Cassandra cluster is explained below.

3.4.1 Adding nodes to an existing Cassandra Cluster

Cassandra's virtual nodes (vnodes)[8] significantly simplify the process of adding nodes to an existing cluster:

- Calculating and assigning tokens to each node is no longer required
- Rebalancing is not required when adding or removing nodes from a cluster because a joining node assumes responsibility for an even portion of the data.

Procedure:

1. Cassandra should be installed on the new nodes without starting it.
2. The following properties in the *cassandra.yaml* and *cassandra-topology.properties* configuration files should be set:
   - cluster_name: the cluster name, the new node is joining.
   - listen_address/broadcast_address: the IP address/hostname that other Cassandra nodes use to connect to the new node.
   - seed_provider: A list of nodes the new node should contact to discover about the cluster and start the gossip process.

[8] Apache Cassandra 1.2 DATASTAX Documentation, http://docs.datastax.com/en, accessed June 2015

3. Now start Cassandra on each of the new node allowing two minutes between nodes initializations[9]. The startup and data streaming processes can be monitored using the nodetool's *netstats* command.

3.4.2 Removing a node from an existing Cassandra Cluster

Procedure:

1. Using the nodetool's *status* command, check if the node is down or up. The *status* command gives the status of the node i.e. UN=up, DN=down, UJ=joining, UM=moving.
2. If the node is up, run the nodetool's *decommission* command to remove the node from the cluster. This command will assign the token ranges that the node was responsible to other nodes and also perform replication appropriately. You can use the nodetool's *netstats* command to monitor the progress.
3. If the node is down, run the nodetool's *removenode* command to remove the node from the cluster.

For detailed explanations on Cassandra's elasticity and rebalance API, see Cassandra™ 1.2 documentation[10]

If the new number of servers in the cluster exceed the available maximum number of servers, the actuator would set the new number of servers to the maximum. Similarly, if the new number of servers is less than the minimum number of servers, the actuator would set the new number of servers to the minimum. Refer to Equation 3.10.

The rebalance operation performed when adding or removing servers is expected to add a significant amount of load on the system which leads to an increase in 99^{th} percentile of read latency. This can make the controller to make wrong decisions. Lim et al. (Lim et al. 2010) proposed disabling the controller during the rebalance operation. Al-Shishtawy et al. (Al-Shishtawy & Vlassov 2013) also adapts the same approach. However with our approach, data rebalance can be added as an another dimension to our performance model to handle the behavior added by the extra load during the rebalancing operation.

[9] Apache Cassandra 1.2 DATASTAX Documentation, http://docs.datastax.com/en, accessed June 2015
[10] Apache Cassandra 1.2 DATASTAX Documentation, http://docs.datastax.com/en, accessed June 2015

Equation 3.9 gives continuous values to fully satisfy the controller actuation requests. However, the actuator can only add or remove complete servers in discrete units. For instance, to satisfy the new predicted average throughput per server, the actuator can specify that 1.5 servers need to be removed (or added). We solve this problem by rounding the new number of servers to a discrete value (Al-Shishtawy & Vlassov 2013). But this might cause the controller to continuously add or remove one server (oscillations). When the size of the storage cluster is small, adding or removing a server have a considerable effect on the storage service total capacity. Oscillations occur under such scenarios. Lim et al. (Lim et al. 2010) proposed *the proportional thresholding technique* to avoid oscillations. Ideally, we define a threshold around the model (slightly above and below the model line) called the deadzone, where our controller takes no action.

In summary, the flowchart of our self-trained proactive elasticity manager is shown in Figure 3.9. The final goal is to find a trade-off between meeting the SLO (for example, a maximum response time of 70 milliseconds) and minimizing the cost of renting cloud resources.

3.5 Implementation details: languages and communication protocol

The controller was implemented in Java. The storage service, Cassandra, and the chosen synthetic load generator, YCSB, are Java applications. Hence, instrumentation was consequently done in Java. We used Matlab libraries and functions to implement prototype of all our prediction algorithms. Online training was done using Matlab's LIBSVM as explained in section 3.3. Java to Matlab communication is performed using matlabcontrol[11]. The basic usage pattern with matlabcontrol is first to create a factory, and then to create a proxy. The proxy is used to communicate with MATLAB where you can *eval*, *feval*, get and set variables. MATLAB arrays are always atleast two dimensions, therefore the lowest Java array dimension that can be sent to MATLAB is a $double[][]$. Having the ability to set arrays, get arrays and perform eval, we can now manipulate an array using matlabcontrol. For more on matlabcontrol see (matlabcontrol).

The actuator uses the Cloud elasticity API to add/remove servers, and the key-value store's

[11] A Java API that allows for calling MATLAB from Java

rebalance API to redistribute the data among the servers. In our case, we used Cassandra's nodetool utility to generally manage our Cassandra cluster (see section 3.1.2 and 3.4). More concretely, our controller keeps a list of all available instances with their state (active or inactive). This list is updated upon decommissioning or commissioning operations.

We now present some algorithms as implemented in this thesis:

Algorithm 1: Signature-driven resource demand prediction algorithm

Data: A set of N data points, X_i, i = 1,...,n

Result: A predicted value

Find Dominant Frequency using FFT: $f_d = FFT(X_i)$;

if f_d **then**

  Find Z ;

  $Z = (1/f_d)r$, where r denotes the sampling rate;

  Generate a pattern window size of Z samples ;

  Split the original time series X_i into $Q = N/Z$ pattern windows;

  Find similarity between all pairs of different pattern windows;

  if *all pattern windows are similar* **then**

    $t =$ the average value of the samples in each position of the pattern windows;

    Return t;

  else

    Return null (no repeating behavior found);

else

  Return null (no repeating behavior found);

In algorithm 1, two pattern windows are considered similar if their Pearson correlation[12] value is close to 1 (e.g., greater than 0.85) and their mean values ratio is also close to 1 (e.g.,

[12]The Pearson correlation is obtained by dividing the covariance of two pattern windows (X_i and X_j) by the product of their standard deviations.

within 0.05).

Algorithm 2: Recursive partitioning algorithm

Data: A set of N data points, X_i, i = 1,...,n

Result: A regression tree

if *termination criterion exist* **then**

Generate Leaf Node and allocate it a Given Value;

Return Leaf Node;

else

Identify Best Splitting test $s*$;

Generate node t with $s*$;

Left_branch(t) = $RecursivePartitioningAlgorithm(< x_i, y_i >: x_i = s*)$;

Right_branch(t) = $RecursivePartitioningAlgorithm(< x_i, y_i >: x_i \neq s*)$;

Return Node t;

Algorithm 3: The Weighted Majority Algorithm (simple version)

1. Initialize the weights $w_1, ..., w_n$ of all the prediction algorithms to a positive weight (All weights are initialized to one unless specified otherwise).

2. Given a set of predictions $(x_1, ..., x_n)$ by the prediction algorithms, select the prediction with the highest total weight.

3. When the correct answer is received, penalize each mistaken prediction by multiplying its weight by a fixed β such that $0 < \beta < 1$.

4. Goto 2.

In our implementation, the initial weights of our prediction algorithms was set to 3. We then penalize each mistaken prediction by subtracting its weight by one and rewarding the correct prediction by adding its weight by one. Since our predictions are continuous, we control the weights bound i.e. $0 \leq (w_1,...,w_n) \leq 3$.

3.6 Summary

In this chapter we presented the design and implementation of the controlling framework. We discussed the role of various components of the controlling framework as well as their implementation. Specifically, we discussed the monitoring process, workload prediction and

the online training. We presented how we built our online performance model using SVM. Based on the workload prediction result and updated system model, our controller invokes the cloud storage API to add or remove servers

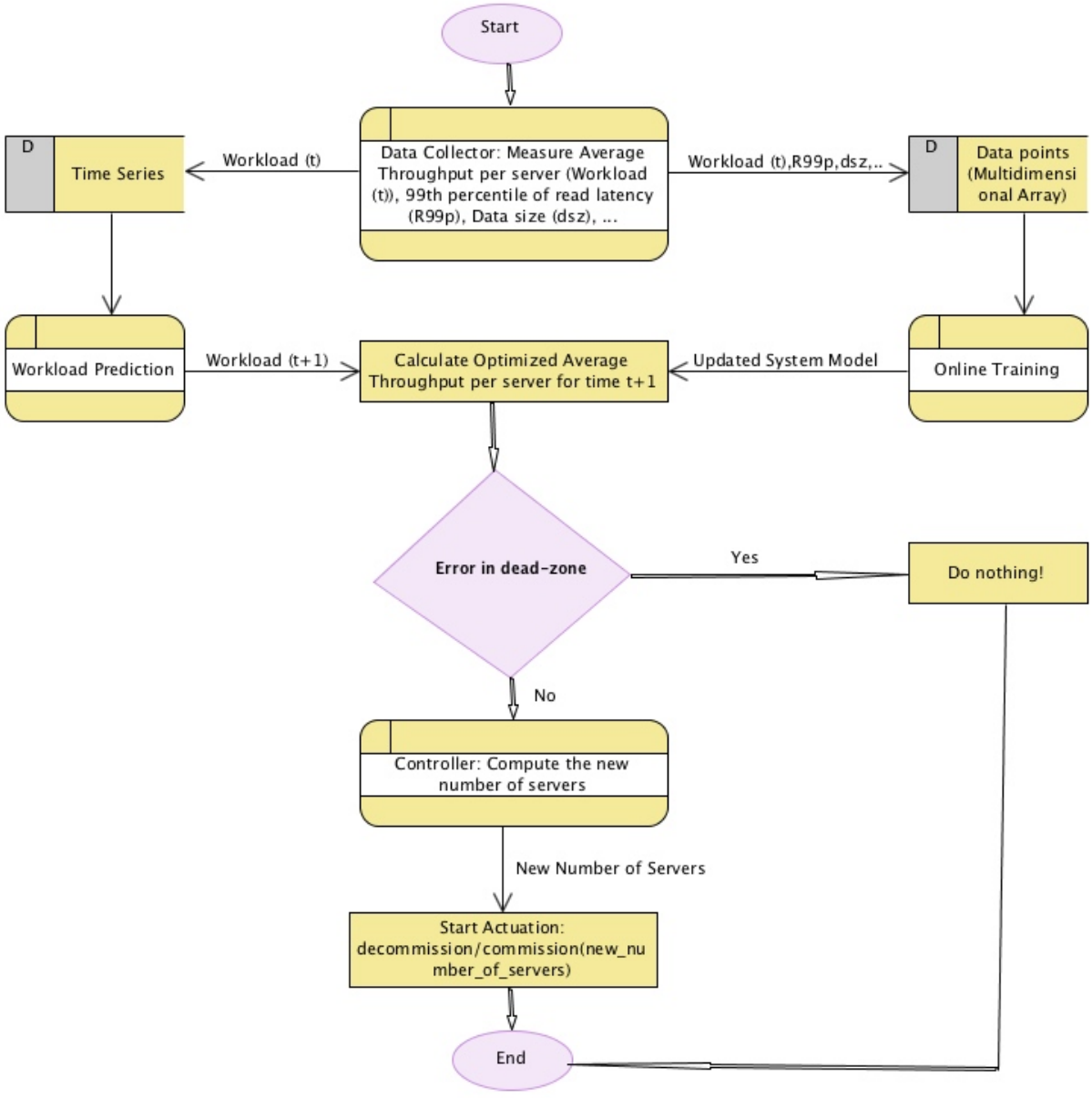

Figure 3.9: Self-trained proactive elasticity manager Flow Chart

4 Evaluation

In this section, we perform a set of experiments to evaluate our self-trained proactive elasticity manager for cloud-based storage services. We evaluate our prediction accuracy, the throughput performance model and finally the performance of Cassandra with our controller.

4.1 Benchmark software

Different benchmarking tools exist for cloud storage systems such as YCSB, Tsung, jMeter, etc. These tools act as synthetic workload generators. In this work we adopt Yahoo! Cloud Serving Benchmark (YCSB) (Cooper et al. 2010) because of its existing integration with Cassandra and prior work by our group with this configuration. Although Cassandra has its own stress tool for benchmarking, we delegate the adoption of alternative measurement software as future work. A brief description of YCSB is given below.

4.1.1 YCSB

YCSB is a standard benchmarking framework used to assist in the evaluation of different cloud systems. Its main goal is to facilitate performance comparisons of the cloud data serving systems. In addition to making it easy to benchmark new systems, the YCSB framework allows easy definition of new workloads, which was one of the main reason it was used in this work. The framework consists of two main components: a workload generating client and a package of standard workloads covering interesting parts of the performance space such as write-heavy workloads, scan workloads, read-heavy workloads, etc.

In this thesis, we describe how the YCSB benchmark was used to report the performance results of our system. The performance tier of the YCSB benchmark focuses on the latency of requests when the cloud data store is under load. In serving systems, which are systems that provide online read or write access to data, latency is very important as people are always im-

patient to wait for a web page to load. However, the latency of each request increases as the amount of system load increases since there is more contention for the system resources such as disk, CPU, network, etc. As a result, the overall throughput of the system is affected. Nevertheless, a system with better performance will achieve the desired throughput and latency even with fewer servers (Vahdat et al. 2002). The workload generator was used to define the dataset and load it into the datastore (Cassandra). It was also used to execute operations against the dataset. The nature of the dataset and operations performed against the data was defined in a set of parameter files. The YCSB clients allow users to report the resulting latency, making it easy to produce latency versus throughput curves. However, in our work we modified the Cassandra storage system to obtain sensor input for our controller, and experimental data for producing graphs.

The YCSB core package defines a set of workloads used to evaluate different aspects of a system's performance. Furthermore, each workload defines a set of parameters such as mix of read/write operations, operation count, record count, request distribution, etc. YCSB users can also implement their own packages by defining a new set of workload parameters or by writing their own code. Since we have to evaluate the performance of a system for different access distributions and data sizes, we developed our own packages by defining a new set of workload parameters. The YCSB client assumes that there is an application such as Cassandra with a table of records (each with N fields). The fields are named $field0$, $field1$ and so on. The records are identified by a primary key and the values of each field of a record are a random string of ASCII characters of length L. For instance, in this work, we constructed 1000 byte records by using $N = 10$ fields, each of $L = 100$ bytes. We varied L for different data sizes. The operations performed against the data store were: insert (insert a new record), update (replace the value of one field) and read (read a record).

YCSB also provides several built-in distributions that assist the workload client to make many decisions when generating load such as which operation to perform, which record to read or write, how many records to scan, etc. These distributions include: uniform, zipfian, latest, multinomial and hotspot distributions. In our evaluation experiments, the distribution of the keys in requests issued by YCSB clients is uniform. With uniform distribution, an item is chosen uniformly at random i.e. all records in the keyspace are equally likely to be chosen.

4.2 Experimental Settings

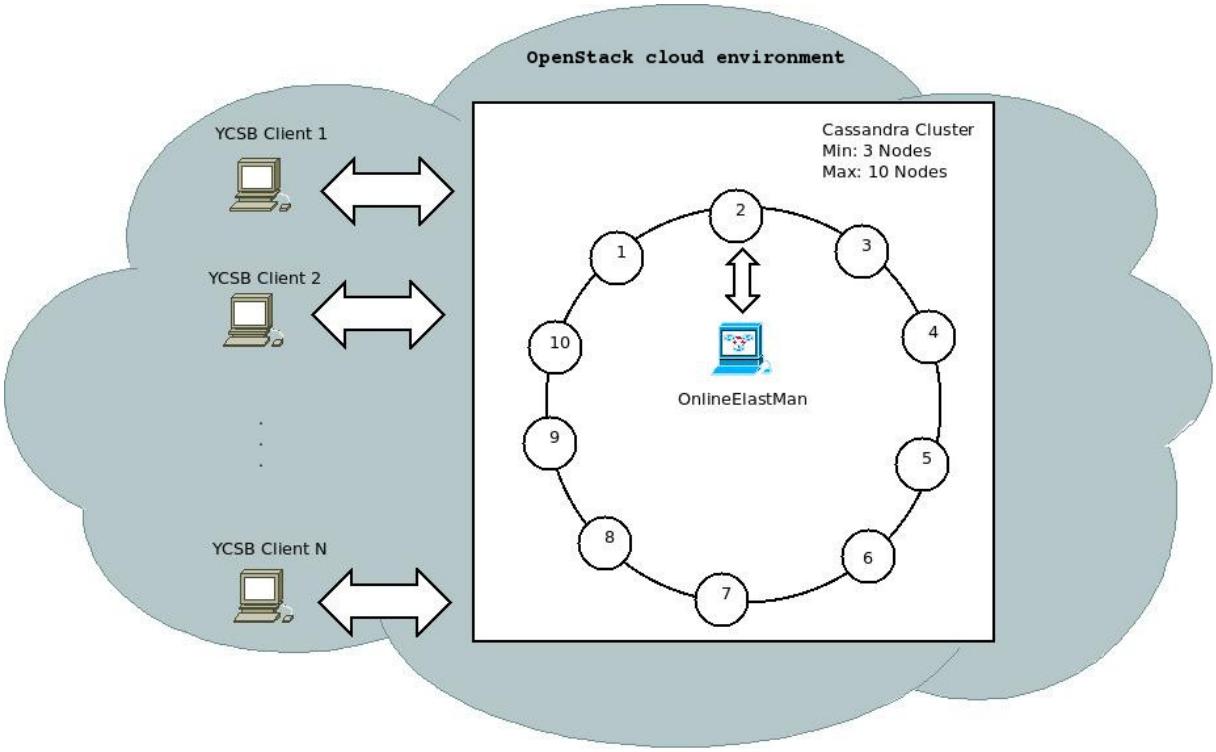

Figure 4.1: Experimental Setup

Figure 4.1 depicts our experimental setup. YCSB clients continuously issue read/write requests to the key-value store. Our controller's data collector measures the throughput, data size and the 99^{th} percentile of read latency (acting the role of sensors). The controller's data collector component periodically (every 5 minutes in our experiments) pulls monitoring data from the Cassandra node and then executes the algorithm given in 3.9. The Cassandra rebalance API is used to distribute data when adding/removing Cassandra instances. We used Cassandra (version 2.0.9) and (YCSB version 0.1.4). We modified the original YCSB version to make it compatible with our Cassandra version. Below we describe the properties of our experiments' components.

Cassandra Cluster: We run our experiments on a Cassandra cluster of 10 nodes (VMs) each with two Intel T7700 processors (2.40GHz), 4GB RAM and 40GB disk size. The cluster runs ubuntu 14.04 on a private Cloud using OpenStack (Openstack).

YCSB clients: Our YCSB clients have the same properties as the Cassandra instances. Each client runs in its own VM and generates a workload of 1200 operations per second that consists

of varying read-write transactions. We vary the workload intensity by adding and removing YCSB client VMs.

Datasets: We use the synthetic data generated by YCSB clients. The load generation settings per client for all the experiments are as follows:

- Number of threads: 16
- Request distribution: uniform
- Record count: 100000
- Read proportion: varied (0.0 - 1.0)
- Update proportion: varied (0.0 - 1.0)
- Data size: varied (1 - 10) KB records.
- Replication factor: 3
- Consistency level: Default (consistency level ONE for all reads and writes).

We first create a keyspace and a table with 10 fields in the Cassandra cluster. We then load the dataset into the Cassandra cluster using the YCSB clients (load phase). In our experiments, loading the database take longer than any individual experiment. After loading, we execute the workload (transaction phase) for different read/write ratios. In our experiments, the YCSB clients were not a bottleneck. In particular, the CPU never reached 100% utilization as most time was used waiting for the Cassandra system to respond.

4.3 Experiment 1 - Workload Prediction

The prediction model considered in this thesis was tested for prediction accuracy and prediction error. The synthetic load generated by YCSB was used for simulating one-step ahead prediction and a comparative analysis of the prediction model was performed. Figure 4.2 presents the actual workload and the predicted workload intensity by our prediction model. Since the objective of this thesis was to formulate a prediction module that can handle a wide variety of workload patterns, we don't present the performance of each of the algorithms considered.

Although we designed five models to do the forecasting, each with its own advantages and disadvantages, we combined the advantages of those models into one model using the WMA to get the best results. Furthemore, better or new prediction algorithms could easily be added into our prediction module.

Figure 4.2 shows a simulation of actual and predicted workload pattern for the first set of our algorithms (mean, max, min, press, regression trees and libsvm). The first one hour (0-60 minutes) simulates a slowing increasing workload pattern, the next one hour (60-120 minutes) simulates a workload with no definite pattern and finally the last one hour (120-180 minutes) simulates a workload with spikes. It was quite difficult to produce a specific workload pattern(e.g. repeating patterns) from our cloud platform due to the dynamically changing environment caused by interferences from other cloud users.

As shown in Figure 4.2, this prediction model takes some time before switching between the prediction algorithms when the workload pattern changes. Since we reward and penalize algorithms when the actual value for a particular prediction window arrives, it takes at most one prediction window for our prediction module to adjust to the new workload pattern. In worst case scenario, it takes up to three prediction windows (the maximum weight) to adapt to new workload pattern.

The results of the prediction module that comprises of the ARIMA models is shown in Figure 4.3. Figure 4.3 also shows how this prediction module switches between the prediction algorithms. From our experiments, it is clear that the ARIMA models are by far the most consistent and efficient prediction models, hence we adopted this prediction module for our controller. The mean absolute percentage error (MAPE) for the ARIMA models (for Figure 4.3 workload) was: $MAPE = 4.60\%$

4.4 Experiment 2 - Performance Model

Our performance model is trained online by varying the workload intensity, data size and the ratio of read/write requests per server as shown in Figure 4.4. The model gets trained automatically by only specifying the monitored parameters and controlled parameter (target), hence our controller is able to adapt to different input patterns.

In practise, the model needs to train itself automatically online in the warm-up phase and

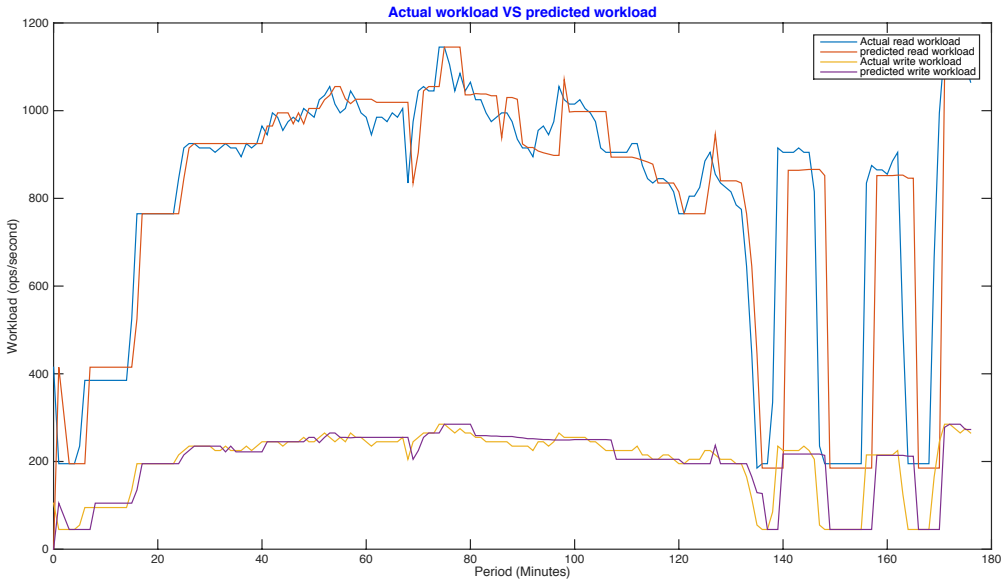

Figure 4.2: Actual workload and predicted workload

after a sufficient amount of time, it should be able to serve the real workload. The controller uses the workload prediction module to predict an application's resource demand, and then uses the model to map the application's SLO violation rate target into a maximum resources needed to keep the target system at an optimal performance. In our experiments, the model is a line that splits the plane into two regions. In the region below and on the line, the SLO is met, while in the region above the line, the SLO is violated. However, with SVM, different kernels can be specified for the decision function to define the decision boundary. The kernel function can be any of the following: linear, polynomial, radial basis function (RBF) and sigmoid. Since we observed that our training data was linearly separable, we adopted the linear kernel. We delegate the evaluation of alternative kernel functions as future work.

Figure 4.4 depicts a model built by specifying three monitored parameters (read request intensity, write request intensity and data size) and 99^{th} percentile read latency as the controlled parameter. The controlled parameter can also be easily switched to 99^{th} percentile write latency. In this experiment the SLO value of the 99^{th} percentile read latency was set to $70ms$. The

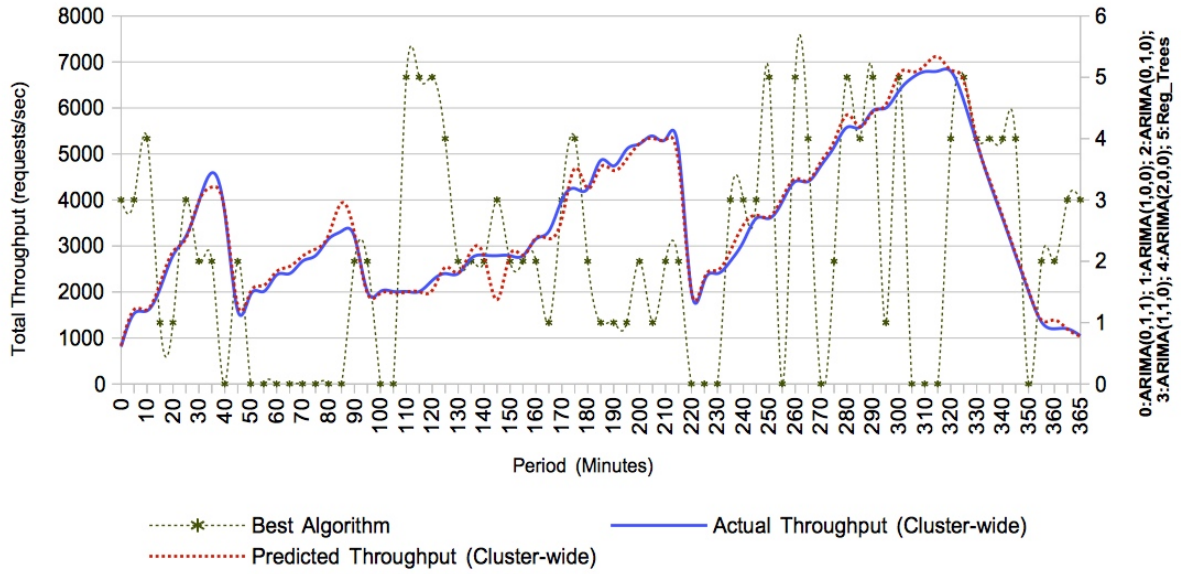

Figure 4.3: Actual workload and predicted workload (ARIMA models)

training requires enough data in both classes (violate SLO/satisfy SLO) in order to produce the model. This is accomplished during the warm-up phase of our controller.

The performance model needs to be updated periodically based on the new requests history, in order to capture up-to-date characteristics of the target system. This is because training once and predicting forever is not suitable for cloud environments' demands prediction due to the dynamic characteristics of input patterns. The model is also application specific and may change at runtime due to variations in the input patterns. Therefore, we generate the model dynamically during the runtime. Figure 4.5 depicts an evolved performance model after an application's data size have changed from $5KB$ to $1KB$. The training takes into account the changes, and produces an up-to-date model of the system.

In our experiments we found out that the rate at which the model evolves affects the accuracy of the decisions made by the controller. The confidence level dictates how fast the model evolves. Ideally we should have enough confidence about the status (violate SLO/satisfy SLO) of a data point before its status changes. Setting the confidence level low may result into the model oscillating (unstable model) while making the confidence level high may cause the model to evolve slowly. In our experiments we set the confidence level as 0.5 i.e. if 50% of all read latency queue samples satisfy SLO then that data point satisfy SLO and vice versa. Since we need to adjust the system resources gradually and wait for the application to be stable

to get a good model, It takes about 20 to 30 minutes for the online training to derive a new performance model from scratch. For applications that have distinct phases of operations, to prevent frequent retraining, one can maintain a set of models and dynamically selects the best model for the current input pattern (Nguyen et al. 2013). A new performance model is only built and added if the decision errors exceeds a predefined threshold.

Figure 4.6 shows 3-D performance model compared to 2-D performance model. The 2-D performance model does not take into account the data size dimension, it's considered as a noise. The effort of changing parameters and system setups to cover a fine-grained three dimensional space is huge.

Figure 4.4: 3D Performance model with fixed data size ($1KB$)

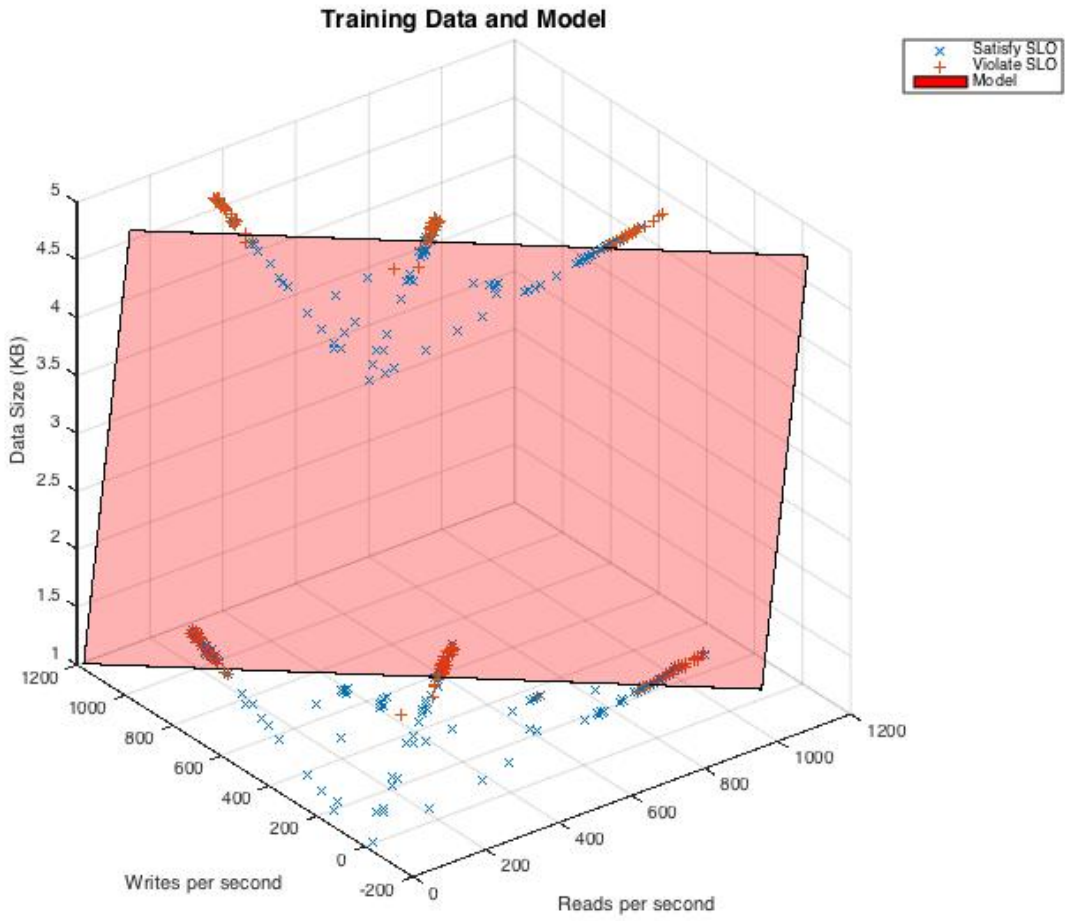

Figure 4.5: 3D Performance model with varying data sizes ($1KB$ & $5KB$)

4.5 Experiment 3 - Performance of Cassandra with OnlineElastMan

Cassandra Key-Value store described in section 3.1.2 was used to evaluate our self-trained proactive elasticity manager. The goal of our elasticity manager is to keep the 99^{th} percentile of read latency at a predefined value as stated in the SLO. In this experiment we choose the value to be 35 ms in five minute period.

We start the experiments with three Cassandra servers each running on its own VM and set the maximum number of Cassandra servers to 10. The controller needs sometimes to get trained automatically online before serving the real workload. In our experiments the warm-up phase took approximately 30-40 minutes (Figure 4.7, Period 0-40). The controller is started

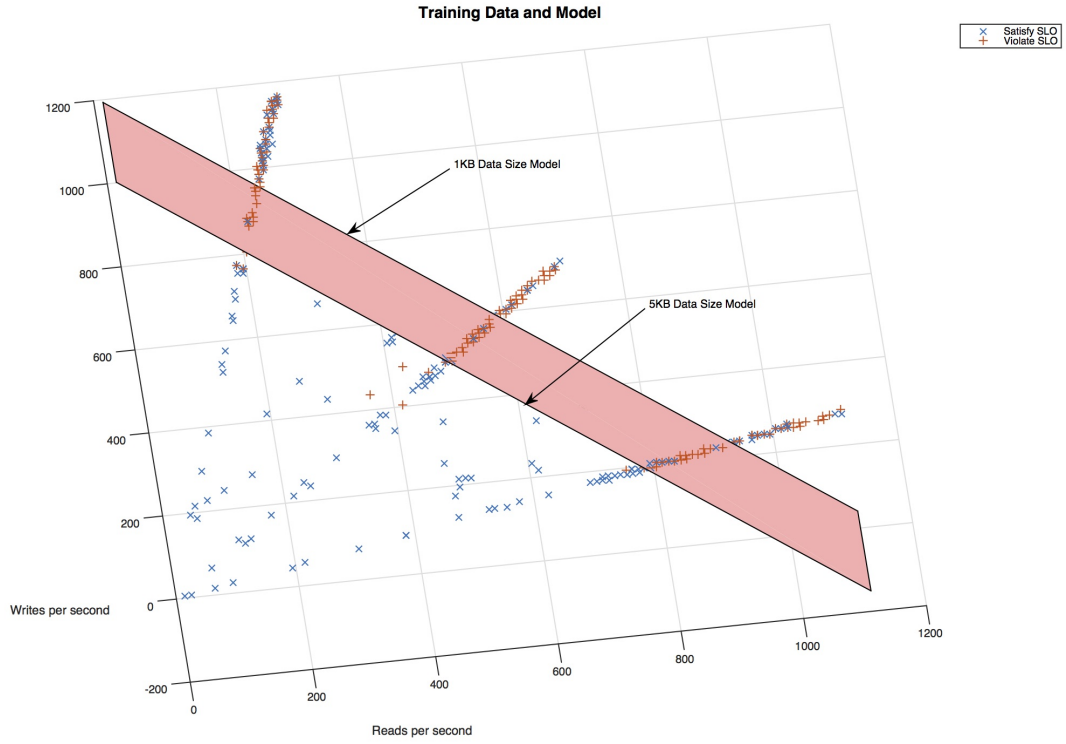

Figure 4.6: 2D Performance model without considering data sizes

after the warm-up period. After the start of the controller, we increased the workload. As the load increases, the 99^{th} percentile of read latency also increases approaching the expected SLO value. Therefore the controller starts adding enough nodes to handle the increasing workload (as predicted).

As shown in Figure 4.7, the controller continues to add nodes respective to the increasing workload. As a result, the 99^{th} percentile of read latency becomes closer to the SLO value as shown in Figure 4.8. Adding new instances decreases the workload per server as the overall workload is now shared among the new servers. We continously varied (increased/decreased) the workload. When controller notices that the 99^{th} percentile of read latency is much below the SLO value, it start removing nodes not to waste system resources. See Figure 4.7 & 4.8.

This experiment shows that our controller is able to keep the 99^{th} percentile of read latency within the desired region most of the time. Since our controller outputs discrete values (new number of servers required to keep the system at optimal performance), sometimes the results were not as expected. For instance, Equation 3.9 gives 1.5 as the extra number of servers

needed. Therefore 1.5 servers should have been added which is impossible in our case.

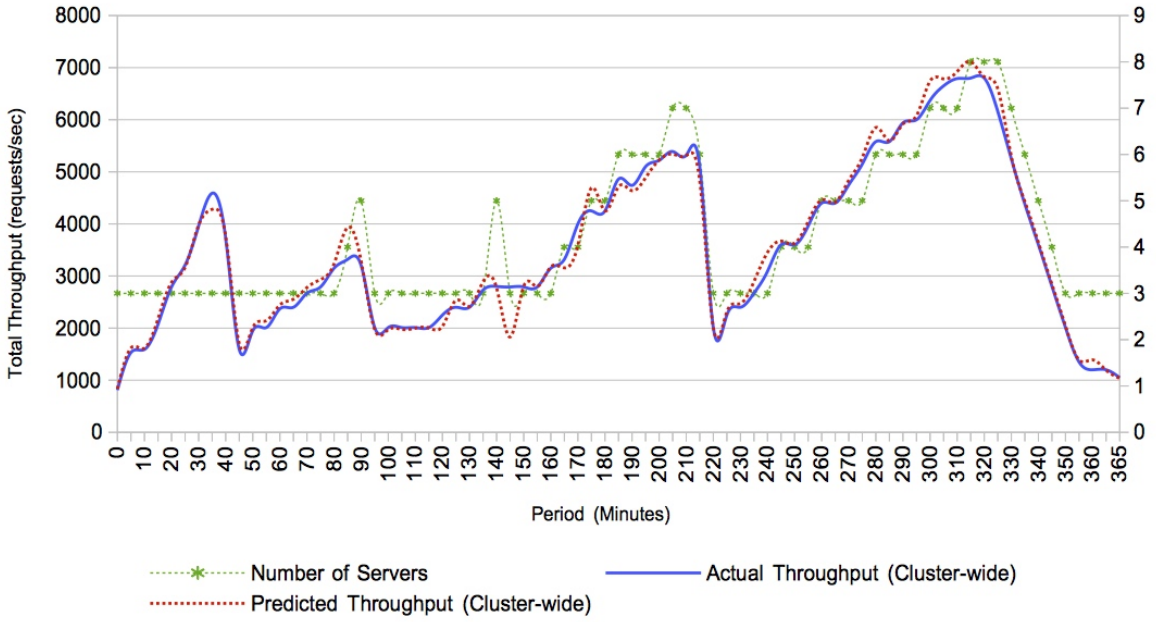

Figure 4.7: Performance of Cassandra with OnlineElastMan

4.6 Summary

In this thesis, we presented a self-trained proactive elasticity manager for cloud-based storage services, with ability to predict future workloads and optimize overall system performance. Our elasticity controller is able to function after being deployed in a cloud platform for a sufficient amount of time in order to get self-trained. Our prediction module is able to select/adjust prediction algorithms for different workload patterns to achieve better prediction accuracy and thus accurate capacity provisioning decisions. In order to capture up-to-date characteristics of the target system, the prediction and performance models are updated periodically based on the new requests history. The online trained model eases the effort of changing parameters and system setups to cover a fine-grained N dimensional space. In addition, our performance model continues improving/evolving itself during runtime.

From our experiments with artificial workload traces it is clear that, given a predefined SLO, our controller guarantees a high level of SLO commitments while keeping the overall resource utilization optimal.

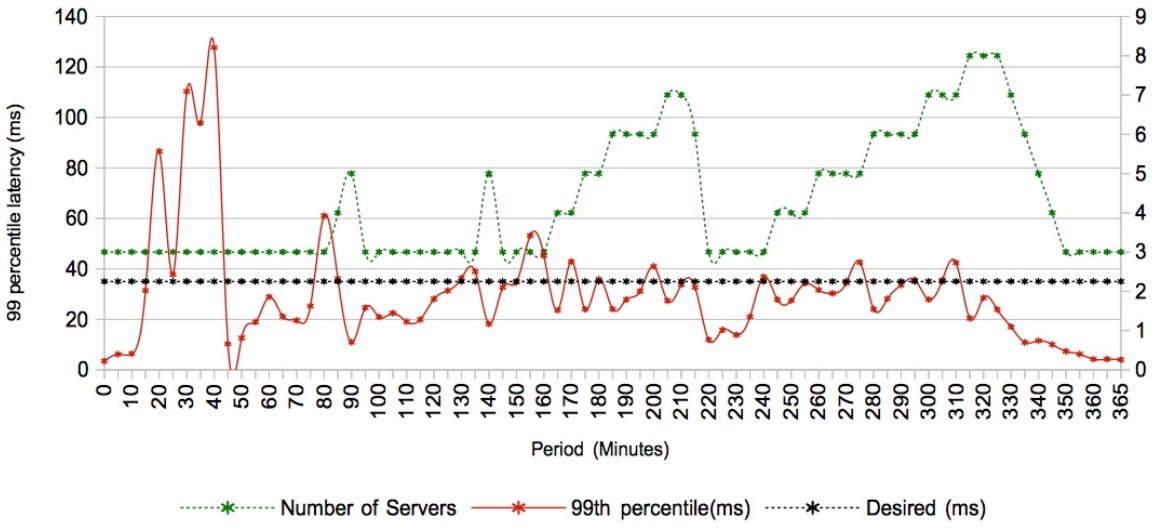

Figure 4.8: Performance of Cassandra with OnlineElastMan

5 Conclusion

5.1 Conclusion

In this thesis, we propose a self-trained proactive elasticity manager for cloud-based storage services for solving the most important issues of auto-scaling in cloud environments. We begin by highlighting the limitations of current approaches to auto-scaling. We then provide the motivation for online trained resource provisioning models and study a few prediction algorithms. A simple weighted majority algorithm is used to select the best prediction among the outputs of the prediction algorithms. The performance of the chosen models is evaluated with artificial workload traces. In particular, YCSB was used in conducting simulations. We showed that our online trained performance model eases the model training process for storage systems. In addition, the trained model continues improving itself during runtime. We have implemented a prototype and evaluated our elasticity manager for the Cassandra storage system in an OpenStack Cloud environment. Our evaluation has shown that our elasticity controller achieves a high level of SLO commitments, thus improving overall resource utilization.

5.2 Future work

In addition to the ongoing practical extensions/improvements to our system, we suggests some ideas for future work.

Designing a distributed controller instead of a central controller is one of the suggestions. Although in our experiments a central controller was enough because we only had 10 nodes, in a big cluster (millions of nodes) a central controller is inefficient. A distributed controller improves performance, is scalable and has no single point of failure. However, it requires more maintenance such as consistency, security and tolerating failures.

Evaluating the relative performance of the chosen models with real workload traces could

also be appropriate.

Another interesting topic is to investigate the advantages of using a hybrid approach to resource provisioning. Combining the reactive and proactive approaches can effectively eliminate the possible failure of a prediction model, in predicting the future workloads.

Bibliography

Al-Shishtawy, A. & V. Vlassov (2013). Elastman: Autonomic elasticity manager for cloud-based key-value stores. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, New York, NY, USA, pp. 115–116. ACM.

Ali-Eldin, A., J. Tordsson, E. Elmroth, & M. Kihl (2013). Workload classification for efficient auto-scaling of cloud resources. May 21, 2013.

Arlitt, M. & T. Jin (2000, May). A workload characterization study of the 1998 world cup web site. *Network, IEEE 14*(3), 30–37.

Armbrust, M., A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, & M. Zaharia (2010, April). A view of cloud computing. *Commun. ACM 53*(4), 50–58.

Box, G. E. P. & G. Jenkins (1990). *Time Series Analysis, Forecasting and Control*. Holden-Day, Incorporated.

Chakrabarti, A., C. Stewart, D. Yang, & R. Griffith (2012). Zoolander: Efficient latency management in nosql stores. In *Proceedings of the Posters and Demo Track*, Middleware '12, New York, NY, USA, pp. 7:1–7:2. ACM.

Chang, C.-C. & C.-J. Lin (2011, May). Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol. 2*(3), 27:1–27:27.

Chang, F., J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, & R. E. Gruber (2008, June). Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst. 26*(2), 4:1–4:26.

Cooper, B. F., A. Silberstein, E. Tam, R. Ramakrishnan, & R. Sears (2010). Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, New York, NY, USA, pp. 143–154. ACM.

Cristianini, N. & J. Shawe-Taylor (2000). *An Introduction to Support Vector Machines: And Other Kernel-based Learning Methods*. New York, NY, USA: Cambridge University Press.

Danny Yuan, Neeraj Joshi, Daniel Jacobson, Puneet Oberai. Scryer: Netflix's Predictive Auto Scaling Engine. http://techblog.netflix.com/2013/12/scryer-netflixs-predictive-auto-scaling.html. [Online; accessed June 2015].

DeCandia, G., D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, & W. Vogels (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, New York, NY, USA, pp. 205–220. ACM.

Galante, G. & L. de Bona (2012, Nov). A survey on cloud computing elasticity. In *Utility and Cloud Computing (UCC), 2012 IEEE Fifth International Conference on*, pp. 263–270.

Ghanbari, H., B. Simmons, M. Litoiu, & G. Iszlai (2011, July). Exploring alternative approaches to implement an elasticity policy. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 716–723.

Gong, Z., X. Gu, & J. Wilkes (2010, Oct). Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pp. 9–16.

Gunn, S. R. (1998). Support vector machines for classification and regression. Technical report, University of Southampton.

Gusella, R. (1991, Feb). Characterizing the variability of arrival processes with indexes of dispersion. *Selected Areas in Communications, IEEE Journal on 9*(2), 203–211.

Hansen, B. (1995). Time series analysis james d. hamilton princeton university press, 1994. *Econometric Theory 11*(03), 625–630.

Herbst, N. R., S. Kounev, & R. Reussner (2013). Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA, pp. 23–27. USENIX.

Jamshidi, P., A. Ahmad, & C. Pahl (2014). Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, New York, NY, USA, pp. 95–104. ACM.

Lakshman, A. & P. Malik (2010, April). Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*(2), 35–40.

Levi, H. (1965). A vector space approach to geometry. melvin hausner. prentice-hall, englewood cliffs, n.j., 1965. xii + 397 pp. illus. 12. $Science$ $149(3689)$, 1225.

Lim, H. C., S. Babu, & J. S. Chase (2010). Automated control for elastic storage. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10, New York, NY, USA, pp. 1–10. ACM.

Littlestone, N. & M. K. Warmuth (1994, February). The weighted majority algorithm. *Inf. Comput. 108*(2), 212–261.

Liu, Y., N. Rameshan, E. Monte, V. Vlassov, & L. Navarro (2015, May). Prorenata: Proactive and reactive tuning to scale a distributed storage system. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pp. 453–464.

Lorido-Botran, T., J. Miguel-Alonso, & J. A. Lozano (2014, December). A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput. 12*(4), 559–592.

matlabcontrol. A Java API to interact with MATLAB. https://code.google.com/p/matlabcontrol/wiki/Walkthrough. [Online; accessed June 2015].

Mcleod, A. I. (1993). Parsimony, model adequacy and periodic correlation in time series forecasting.

Mell, P. M. & T. Grance (2011). Sp 800-145. the nist definition of cloud computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States.

Nguyen, H., Z. Shen, X. Gu, S. Subbiah, & J. Wilkes (2013). Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, San Jose, CA, pp. 69–82. USENIX.

Openstack. Open source software for building private and public clouds. http://openstack.org/. [Online; accessed June 2015].

Ovidiu Ivanciuc. SVM - Support Vector Machines. http://www.support-vector-machines.org. [Online; accessed June 2015].

Robert Nau. Statistical forecasting: notes on regression and time series analysis. http://people.duke.edu/~rnau/411home.htm. [Online; accessed June 2015].

Roy, N., A. Dubey, & A. Gokhale (2011, July). Efficient autoscaling in the cloud using predictive models for workload forecasting. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pp. 500–507.

Smola, A. & B. Scholkopf (2004). A tutorial on support vector regression. *Statistics and Computing 14*(3), 199–222.

Trushkowsky, B., P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, & D. A. Patterson (2011). The scads director: Scaling a distributed storage system under stringent performance requirements. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, Berkeley, CA, USA, pp. 12–12. USENIX Association.

Vahdat, A., K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, & D. Becker (2002, December). Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev. 36*(SI), 271–284.