Predictive Pod Auto-scaling in the Kubernetes Container Cluster Manager

by

Matt McNaughton

Professor Jeannie Albrecht, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

December 10, 2015

DRAFT

Contents

Abstract

Acknowledgments

Chapter 1

Introduction

Over the past few decades, an explosion in the need for computing resources, and the existence of inexpensive, interconnected computers, has driven a significant increase in the feasibility and benefits of distributed systems [21].

First, we consider the origin of distributed systems as a field of computer science. Before the availability of cheap, powerful microprocessors and reliable, efficient local-area networks (LANs), computational tasks could only be performed on a singular computer [21]. If a task was too computationally expensive for a commodity PC, the only solution was to run it on a larger, more powerful supercomputer. However, as cheap microprocessors increased the availability of affordable computers, and LANs fostered quick inter-computer communication, a new model of performing resource intensive computation arose. In this distributed systems model, a collection of individual computers function as a single mass of computing resources to solve a given computational task [21].

Second, we consider the ever-growing interest in unlocking and implementing the benefits of distributed systems. A number of forces drove, and continue to drive, increased interest in distributed systems over the past decade. The first, and most obvious, factor is the Internet and its substantial impact on the role of computers in everyday life. As more people connected to the Internet, through computers, mobile phones, and tablets, an increasing number of interactions became computerized. Consumption, communication, research, and more all became possible on the Internet. Subsequently, large amounts of computing resources were needed to store the data, and perform the computational tasks, related to these interactions. Closely coupled with this trend is the rise of "Big Data". In 2013, the digital universe contained 4.4 zettabytes of data [13].[1] Without multiple computers working together it would be impossible to store and process this incredible volume of data.

Today, it is nearly impossible to do anything in modern society without interacting with a distributed system and creating new digital data. Driving a car, trading a stock, visiting a doctor, checking an email, and even playing a simple video game, are all activities that distributed systems facilitate and improve [14]. As life becomes more computerized, and as the volume of data humans generate and hope to process grows, distributed systems will only increase in importance. Furthermore, research into distributed systems makes it possible to continue to unlock, and make

[1] A zettabyte equals 10^{21} bytes, which equals 1 billion terabytes.

available to the general public, the incredible power of networked, cooperating computers. As the distributed systems supplying massive computational power become more accessible, both because of decreased cost, increased ease of use and improved reliability, we can computationally address an ever increasing number of challenging, important problems.

There are a number of different models for computing tasks requiring high levels of computing resources, including supercomputing, cluster computing, and grid computing. In this thesis, we focus on cluster computing. Cluster computing groups together similar commodity PCs on the same LAN to offer a singular mass of computing resources. Specifically, we focus on the cluster manager, an integral component of cluster computing. Cluster managers are responsible for abstracting all of the management details of the distinct nodes in the cluster, and instead presenting a single mass of computing resources on which the user can run jobs or applications. In other words, a cluster manager "admits, schedules, starts, restarts, and monitors the full range of applications" on the cluster [24]. Overall, a cluster manager can be thought of like an operating system for a cluster of computers. There are a variety of different cluster managers, the most important of which will be discussed in the next chapter, each pursuing different objectives. This thesis will ultimately focus on Kubernetes, an open-source cluster manager from Google [8].

Cluster managers seek to accomplish a number of different goals, and as a result, multiple metrics measure success. For example, Microsoft's Autopilot is predominantly concerned with application up-time, and thus success is measured with respect to reliability and downtime [17]. Alternatively, a number of cluster managers measure themselves based on efficient resource utilization (ERU) [24]. Essentially, efficient resource utilization relates to the percent of cluster resources which are actually being used to perform computation/store data. One such measurement of this metric, cluster compaction, examines how many machines could be removed from the cluster, while still comfortably running the cluster's current application load [23]. This metric is particularly important, because the more efficient the cluster manager is at utilizing resources, the less the cluster costs, and the more accessible cluster computing becomes to the general public. A final important cluster management metric is quality of service (QOS). Quality of service measures the ability of an application to function at a specified performance level, despite ever-changing external factors.[2] Again, this metric is particularly important because increasing the robustness of applications run on cluster managers means these applications can be trusted with increasingly important tasks. Attempts to maximize efficient resource utilization and quality of service often lead to the cluster manager implementing auto-scaling, a behavior we will examine in great depth throughout this thesis. Cluster managers predominantly differ with respect to which metrics they optimize for, and the process by which this optimization occurs.

[2] The most prominent varying external factor is changes in the load on the application. For example, for an application serving a website, changes in the number of people requesting web pages from the website would vary the application load.

1.1 Goals

This thesis is most concerned with maximizing the efficient resource utilization (ERU) and quality of service (QOS) metrics with respect to the Kubernetes cluster manager. As such, this thesis pursues three goals:

1. Given an application running on a Kubernetes cluster, we seek to determine a method which ensures quality of service stays consistently high regardless of variation in certain external factors. While it is difficult to make guarantees regarding quality of service, because application performance is dependent on a number of uncontrollable, varying external factors, it is possible to ensure each application has, and is utilizing, the resources it needs to function efficiently.

2. A simplistic solution to the first goal of ensuring a high application quality of service is to just give each application many more resources than it requests.[3] Yet, this over-provisioning is inefficient and costly. Thus, our methods for ensuring a high quality of service must also ensure the maintenance, or improvement, of the efficient resource utilization metric. As such, we add an additional goal: given a certain number of applications running on a Kubernetes cluster, we seek to determine a method which ensures the cluster is as small as possible, while still comfortably supporting the applications' current, and future, resource needs.

3. Given that Kubernetes is an open-source project, we seek to implement, test, and evaluate a proposed enactment of the previous two goals. The methods we pursue will in part be dictated by the current structure and implementation of Kubernetes. Tests will be conducted using the Google Compute Engine [4] on both simulated and real Kubernetes user data. The eventual goal is for this thesis' improvements to be merged into the production version of Kubernetes.

1.2 Contributions

This thesis presents our given contributions to Kubernetes. Kubernetes seeks to ensure high application quality of service and efficient resource utilization, and our contributions look to further its ability to accomplish these goals. As such, we present not only new methodology, but also new, working implementations with the accompanying evaluation. We demonstrate the effectiveness of our modifications in comparison to the non-modified Kubernetes using both simulated and real-world data sets. Finally, we discuss the experiences of making these modifications to Kubernetes, as well as avenues for future improvements with respect to Kubernetes and cluster managers in general.

1.3 Contents

@TODO - This section cannot be written until the thesis is completed.

[3] Assigning an application more resources than it needs to function is defined as over-provisioning.

Chapter 2

Background

2.1 Resource Intensive Computing Paradigms

As was briefly mentioned in the introduction, a number of different paradigms exist for undertaking computing tasks too resource intensive for a single commodity computer. These paradigms are discussed in detail below:

1. Supercomputing: The supercomputing model responds to increased demands for computing resources by increasing the technical specifications of the computer far beyond the range of the traditional commodity PC. While supercomputers are able to avoid the majority of the complications resulting from the introduction of networks, most prominently reliability and security, there are naturally limits on the power of supercomputers. Importantly, constructing supercomputers is extremely expensive, and thus their computing power is not available to the general public. Furthermore, it is difficult to scale a supercomputer should the need arise. Finally, supercomputers offer a single point of failure, meaning they are not particularly robust to error [11]. These limitations have decreased the usage of supercomputers to provide the mass of computing power needed in the "Big Data" era.

2. Cluster Computing: Cluster computing is defined as utilizing "a collection of similar workstations of PCs, loosely connected by means of a high-speed local-area network [where] each node runs the same operating system." [21, pg. 17-18] Cluster computing can provide a mass of computing power similar to that contained in a supercomputer. Cluster computing also offers many advantages over the single supercomputer. First, and perhaps most importantly, cluster of commodity computers are more cost-efficient, and thus considerably more accessible. Second, clusters are easily scaled by simply adding new commodity PCs. Finally, cluster computing provides greater fault tolerance, as a single failing node will simply be removed from the cluster [15]. Cluster computing is used in the implementation of what is colloquially referred to as Cloud computing, in which large amounts of computing resources are offered on a per-usage basis [14]. Cloud computing, as implemented by Amazon Web Services [2], Microsoft Azure [9], and Google Compute Engine [4], continue to revolutionize the develop-

ment and deployment of computing applications, as developers gain access to cheap, easily accessible, quickly scalable, fault-tolerant computing power.

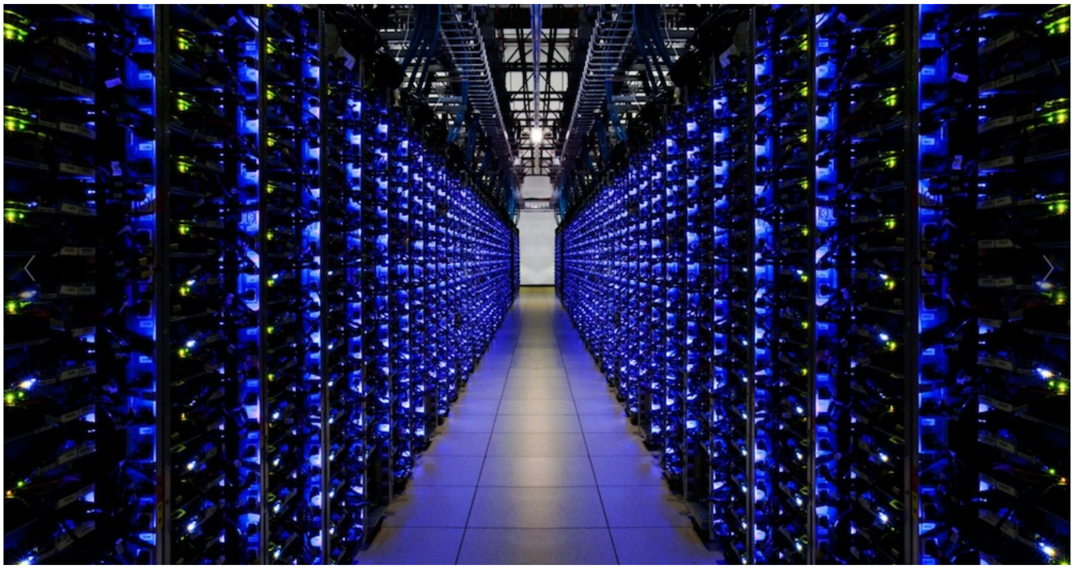

Figure 2.1: A Google Computing Cluster. [10]

3. Grid Computing: Grid computing is similar in concept to cluster computing, except it foregoes the requirement that all computers within the grid be relatively homogeneous. As such, the grid computing model accounts for a large degree of heterogeneity with respect to network membership, operating system, hardware, and more [21]. One popular example of a grid computing implementation is TeraGrid, a high-performance system containing multiple computers connected by an optical network, and used for specific scientific tasks [12]. While grid computing systems' lack of homogeneity increases flexibility, the resulting heterogeneity introduces significant complexity. Importantly, the distinction between grid and cluster computing is fuzzy and predominantly based on the uses of the system. Most grid implementations are used for very specific scientific computing tasks, as seen with TeraGrid, while clusters are traditionally more about offering general computational power.

Ultimately, because of simplicity, cost, and scalability, cluster computing is becoming the most prominent resource-intensive computing paradigm. Thus, cluster computing, and the accompanying cluster manager, is the focus of this thesis.

2.2 Cluster Management Paradigms

As was briefly mentioned in the introduction, cluster managers are responsible for admitting, scheduling, running, maintaining, and monitoring all applications and jobs a user wishes to run on the cluster.[1] Cluster managers can be thought of as the operating system for the cluster. Naturally,

[1] Application, job, and task are largely interchangeable names for computing work performed on the cluster.

cluster managers are extremely diverse, both in the types of applications and jobs they are best suited to running, and the method in which they seek execute their duties. At the most basic level, there are two types of workload that may be submitted to a cluster manager: production and batch. Production tasks are long-running with strict performance requirements and heightened penalties for downtime. Batch tasks are more flexible in their ability to handle short-term performance variance. In the context of a large company like Google, a production task would be serving a large website like Gmail or Google Search, which must be continuously accessible with low-latency and little downtime; a batch task would be analyzing advertising analytics data with MapReduce, which can fail or slow without significant external costs [24]. The type of tasks a cluster management system predominantly seeks to run dictate many of the cluster manager's implementation details.

One varying factor in a cluster manager's implementation is the process by which the cluster manager schedules jobs.[2] There exist three predominant methods of scheduling: monolithic, two-level, and shared state. With monolithic scheduling, a single algorithm is responsible for taking the resource requests of all jobs and assigning them to the proper machine. With two-level scheduling, the cluster manager simply offers resources, which can then be accepted or rejected by the distributed computing frameworks.[3] Finally, with shared state scheduling, multiple different algorithms concurrently work to schedule jobs on the cluster [20]. Naturally, all of these methods have positives and negatives. While monolithic scheduling is initially simple to implement if the jobs being scheduled are homogeneous, a single-threaded monolithic scheduler does not allow nuanced processing of diverse jobs based on varying heuristics and guidelines. Attempts to support this nuance can create an incredibly complicated algorithm that is difficult to extend [20]. While two-level scheduling is lightweight, simple, and offers advantages with respect to data locality, it is not effective for long-running, production jobs [20]. Finally, while shared state scheduling removes the scheduler as both a computational and complexity bottleneck, it must take steps to guarantee global properties of the cluster and address the typical challenges of concurrent programs [20]. The chosen scheduling method effects the type of applications and distributed computing frameworks runnable on the cluster manager and the efficiency with which these applications and frameworks run.

A final distinction is the licensing and availability of the cluster manager's code. Because cluster managers are necessary only in the presence of vast amounts of data and computation, predominantly large corporations develop and utilize cluster managers. Often these cluster managers are kept within the confines of the corporation, or only explained by a brief paper or conference talk, with little source code available. In more unique cases, the company will open-source the source code, allowing anyone to view, modify, and run the cluster manager. Such open-sourcing presents a unique opportunity for researchers wishing to experiment with cluster managers, but lacking the resources to create their own from scratch. In rarer instances, a fully-developed cluster manager will originate from academic research. In unique scenarios, a large corporation will adopt a cluster manager originating in academia and the entire code base will be open-sourced. The availability of source code directly impacts the feasibility of pursuing experiments with already existing cluster management systems.

[2] Just like scheduling jobs on an operating system, scheduling jobs on a cluster equates to assigning jobs to resources on a machine in the cluster.

[3] Distributed computing frameworks are frameworks built to function over multiple machines. Some popular examples include Apache Hadoop, Apache Spark... [16]

Table 2.1: Overview of Cluster Management Paradigms.

	Job Type	**Scheduling Model**	**Open Source**
Borg [24]	Both	Monolithic	No
Omega [20]	Both	Shared state	No
Mesos [16]	Batch	Two-level	Yes
YARN [22]	Batch	Monolithic	Yes
Kubernetes [8]	Production	Monolithic	Yes

Naturally, cluster managers can vary in multiple additional ways. However, the previous three differences, highlighted in Table 2.1, recognize the most important distinctions in the context of this thesis. Given this understanding, we can now begin to examine specific cluster management implementations and defend our choice of Kubernetes as the cluster manager on which we will ask and answer our research questions.

2.2.1 Borg

While just recently described to the public in a 2015 paper, the Borg cluster manager from Google has been in production use for over a decade [24]. Borg incorporates many different objectives, seeking to abstract resource management and error handling, maintain high availability, and efficiently utilize resources on the cluster. It is responsible for managing the hundreds of thousands of batch and production jobs run everyday at Google. Borg utilizes a monolithic scheduler, as jobs specify the resources they need, and a single Borg scheduler decides whether to admit and schedule said jobs. Finally, Borg is not open-source. In fact, it was not even publicly announced or described until 2015, despite running at Google for over a decade. However, Kubernetes, Google's open-source cluster manager and the focus of this thesis, incorporates many of the lesson's learned from Borg.

2.2.2 Omega

Also originating at Google, Omega is a cluster manager seen as an extension of Borg [20]. While retaining the mission and goals of Borg, Omega differs in implementation. Specifically, it considers a new method of assigning jobs the resources they need on the cluster by implementing shared state, instead of monolithic, scheduling. As previously mentioned, shared state scheduling allows multiple scheduling algorithms to work in parallel to assign tasks to the resources they request. Research on Omega shows this parallel scheduling implementation both eases the complexity of adding new scheduling behaviors and offers competitive scaling performance. Additionally, in Omega, all schedulers are aware of the entire state of the cluster, meaning that a job's resource allocation can be varied after the job begins to execute. This flexibility offers significant performance improvements. Like Borg, Omega is not open-source. However, like Borg, much of the work done on Omega is incorporated into Kubernetes.

2.2.3 Mesos

Lest we think all cluster managers originate at Google, we now examine Apache Mesos, a cluster manager originating at University of California, Berkeley [16]. Mesos is considerably more lightweight than either Borg or Omega. We can think of Mesos as functioning like the kernel of an operating system (i.e, the Linux kernel) while a system like Borg is like an entire Linux distribution (i.e, Red Hat, Ubuntu, etc…) It does not seek to monitor the health of jobs or provide user-interfaces for viewing the current state of a job, leaving those tasks to the distributed computing framework. Additionally, Mesos predominantly focuses on quick-running, high-volume batch jobs, and is not particularly suited to long-running, high-availability production jobs. In part, Mesos' job scheduling implementation dictates the singularity of the job types Mesos efficiently processes. Mesos utilizes two-level scheduling, in which the cluster management system simply offers, not assigns, available resources to the distributed computing framework. Predominantly, this decision is made to ensure data locality. [4] Finally, Mesos is interesting in that it is entirely open-source, yet still in use at some of the largest tech companies such as AirBNB and Twitter.

2.2.4 YARN

We now briefly discuss Apache YARN. YARN, an acronym for Yet Another Resource Negotiator, is a cluster manager initially built for use with Apache Hadoop [22]. However, it is now possible to use YARN with a variety of distributed computing frameworks. Unsurprisingly given YARN's initial use case, YARN is predominantly used for running batch jobs.[5] Like Mesos, YARN aims to support data-locality, again a predominant advantage for batch processing. YARN schedules resources by allowing distributed computing framework application masters to request resources from a single resource master. The application master can then assign these resources to specific tasks at its leisure. As a single resource master is assigning all of these resources, YARN is a monolithic scheduler. Similar to Mesos, YARN is both entirely open-source and in use at major corporations like Yahoo.

2.2.5 Kubernetes

Finally, we arrive at the cluster manager that is the focus of this thesis: Kubernetes. Kubernetes also originates at Google, although it is open source. Kubernetes is also the most recent of the cluster managers we consider in this background chapter. [6] The recent explosion in popularity of containerization[7] heavily impacts the development and implementation of Kubernetes. Specifically,

[4] Data locality is a measure of if the task has the necessary data on its machine, or if it must make a costly request across the network for the data. Mesos works to ensure data locality by allowing multiple frameworks to function on the same machines, and thus share data, and also for frameworks to dictate their own resources, such that they can work to ensure data locality. If running a large number of data processing tasks with extremely high volumes of data, data locality can be essential to efficient cluster operation.

[5] At the time of the publication, YARN was just beginning to be used for production jobs; however, its main focus from creation has been batch processing.

[6] Kubernetes became public in the summer of 2014.

[7] We discuss both the motivations and technology behind containerization in the Appendix. Briefly, containerization is packing everything an application needs to run into a single *container* and then running that container on any desired computer.

Kubernetes is seen as part of a new paradigm of developing applications through the use of microservices.[8] Kubernetes predominantly focuses on effectively running service jobs, which require high-availability and potentially varying amounts of resources. Additionally, Kubernetes currently utilizes a simple monolithic scheduler, although there are plans to grow the scheduler in the future. [6] Finally, Kubernetes is an open source project, yet is also used in production at Google, and available to the public through the Google Container Engine. [5]

We choose Kubernetes as the cluster manager on which to conduct our experiments for a number of reasons. First, unlike many of the aforementioned cluster managers, Kubernetes focuses on service jobs. Services jobs have particularly stringent requirements for availability and are also the most likely to have varying resource needs. Both of these conditions are closely linked with the previously stated goals of this thesis, and Kubernetes is the cluster manager that stands to benefit the most if we achieve our goals. Additionally, Kubernetes is the only open source cluster manager focusing on long-running services. Working with a open source cluster manager allows us to benefit from the previous work of others, as well as expand the potential benefits of any successful work.

2.3 Auto-scaling Paradigms

We now consider a subset of cluster management closely related to this thesis' goal of ensuring efficient resource utilization and quality of service. Specifically, we introduce auto-scaling, a method of ensuring each application has the necessary amount of computational resources to handle varying external demands.[9]

To better understand both the implementation and benefits of auto-scaling, let us consider a simple scenario. Imagine you have an application running on a cluster for a week. On Monday, it will need x resources.[10] On Tuesday through Thursday, the application will need $2x$ resources. Finally, on Friday the application will again need x resources. Without auto-scaling, we are forced to assign a constant amount of resources to the application running on our cluster. However, there is no constant amount of resources that can meet the two goals of efficiently utilizing the cluster's resources and ensuring the application has the resources needed for maintaining a high-level of service. Specifically, if we assign the application x resources, then on Tuesday through Thursday the application will not have the amount of resources it needs to handle its load, and quality of service will deteriorate. Alternatively, if we assign the application $2x$ resources on the cluster, then on Monday and Friday we will essentially be wasting x resources on the cluster, as they are assigned to an application that does not need them. Such waste indicates inefficient resource utilization.

We use auto-scaling to address the inability of statically allocated resources to efficiently handle all the variances in application load. Auto-scaling allows us to assign an application more or less resources based on the status of the cluster. In our previous example, perfect auto-scaling would

[8] Again, we will discuss microservices in greater detail in the Appendix. Essentially, microservices are the division of applications into small, easily scalable services which communicate with each other across the network.

[9] Importantly, auto-scaling is made possible by current models of cluster/cloud computing, in which it is possible to automatically obtain and relinquish computing resources within a very short time frame. Obviously, if adding computing resources requires buying a new physical server, as was the case before the advent of cloud computing, auto-scaling becomes impossible.

[10] By *resources*, we mean CPU, memory, etc…

allow us to assign the cluster x resources on Monday and Friday and $2x$ resources on Tuesday through Thursday. Through auto-scaling, we accomplish both goals: our application has the needed resources for a high quality of service, and our cluster is efficiently utilizing resources by only allocating the application what it needs. Overall, auto-scaling can make applications on a cluster more performant and the cluster more cost-effective.

Given an understanding of the importance of auto-scaling, we now begin to examine the differing implementations of auto-scaling. Auto-scaling implementations differ in how the cluster manager assigns an application the new resources it needs[11] and how the cluster manager makes its auto-scaling decisions. Given these two points of variation, there are two predominant characteristics that shape the nature of an auto-scaling implementation. The first is if the auto-scaling is horizontal or vertical. The second is if the auto-scaling is reactive or predictive.

1. Horizontal vs. Vertical: We begin by examining the difference between horizontal and vertical scaling. Let us begin by assuming there is an application assigned x resources by the cluster manager. The application faces external load such that it needs $2x$ resources to operate with an acceptable quality of service. There are now two options. In *vertical* auto-scaling, the cluster manager will attempt to assign the application the needed $2x$ resources without halting the execution of the application. In *horizontal* auto-scaling, the cluster manager will create another instance of the application, so that there are two machines each with x resources ($2x$ resources in summation). The load will be split between the two new instances of the application, meaning each machine handles half the requests and requires only the x resources it has [19]. While both of these variations of auto-scaling accomplish the same purpose, horizontal auto-scaling is much simpler. Given we know how to create an instance of a virtual machine running the application, an entirely safe assumption considering we already created one such instance, it is fairly trivial to create another instance, and then split the load between these two instances using standard methods of load balancing. [12] Contrast this well-defined duplication with the more questionable, difficult process of changing the resources given to a virtual machine without stopping the execution of the application. Importantly, there exists no consistent, cross-platform method of modifying the amount of resources granted to a virtual machine instance without stopping the execution of the virtual machine. As such, vertical auto-scaling is rarely seen in practice; every implementation of auto-scaling we examine focuses on horizontal auto-scaling [19].

2. Reactive vs. Predictive: We continue by examining the distinction between reactive and predictive auto-scaling. At the simplest level, reactive auto-scaling reacts to the current state of the application and cluster, while predictive auto-scaling reacts to the future state of the application and cluster [19]. While reactive auto-scaling must only consider one time-frame when gathering and interpreting information, predictive auto-scaling must consider many different time-frames with respect to the most accurate method of projecting past metrics into

[11] Auto-scaling implementations can also take away resources from an application running on a cluster.

[12] Claiming simplicity assumes the application is written such that it can be replicated with no unexpected modifications on operation.

Table 2.2: Overview of Auto-scaling Paradigms.

	Reactive vs Predictive	**Implementer**
Threshold	Reactive	Amazon Web Services [3]
Time-Series Analysis	Predictive	Netflix [18]
Control-Theory	Both (currently reactive)	Kubernetes [7]

future metrics. However, predictive auto-scaling has the advantages both of historical insight and allowing the cluster manager to decrease the time-costs of certain actions by performing them before a reactive cluster manager would suggest.[13] Finally, techniques for auto-scaling can be both reactive and predictive as they incorporate both current and projected cluster and application metrics to make auto-scaling decisions.

A number of the major providers of cloud computing resources offer auto-scaling, as can be seen in Table 2.2. The most prominent of these providers is Amazon, which supports threshold-based horizontal auto-scaling on EC2 virtual machine instances [3]. Furthermore, Netflix implements time-series analysis auto-scaling to help it respond to the varying demand placed on its services throughout the day [18]. Finally, Kubernetes implements control-theory auto-scaling [7]. We will examine threshold, time-series analysis, and control-theory auto-scaling in detail in the remainder of this chapter.

2.3.1 Threshold-based Rule Policies

The simplest method of auto-scaling is threshold-based rule policies. Threshold-based rule policies are reactive, as they perform scaling behaviors if the current state of the application and host machine is not in accordance with predefined rules. The rules predominantly relate to per machine resource utilization levels. For example, a rule could be that if the average CPU utilization percent for all of the machines is above 80%, then a new machine should be created. This rule would be accompanied with an additional scale-down rule stating that if the average CPU utilization percentage for all of the machines is below 20%, then a machine should be deleted. The most popular implementation of threshold-based rule policies for auto-scaling comes from Amazon Web Services.[14] [3].

Threshold-based rule policies offer both advantages and disadvantages with respect to auto-scaling. Predominantly, these advantages and disadvantages arise from threshold-based rule policies' conceptual simplicity. Because threshold-based rule policies are reactive and based on simple metrics like CPU utilization percentage and memory usage, they are simple to write. However, they are difficult to write well, as it is difficult to predict how certain rules will respond to the varying external circumstances. One particular difficulty arises with respect to handling the nebulous time between

[13] We will spend considerable time later on this concept. Basically, predictive auto-scaling makes it easier to account for the amount of time necessary to perform horizontal auto-scaling (i.e. creating a new virtual machine instance running the application). If we know we need a machine in the future, we can start creating it before it is needed, so it is ready by the time it is needed. With reactive auto-scaling, we do not know we need the replication until the current state of the application and cluster indicates it. Thus, we must wait for the application to be created and ready to run, while the application continues to operate with sub-optimal resources.

[14] More specifically, Amazon Web Services uses threshold-based rule policies for auto-scaling with respect to EC2 instances. EC2 instances are essentially rentable cloud virtual machines [1]

when the threshold is crossed and auto-scaling triggers the creation of a new application, and when the newly created application can start running and balancing the load.

Overall, there are a number of variables that must be considered when determining the impact of threshold-based rule policies, reinforcing that while it is easy to conceive of a threshold-based rule for auto-scaling, it can be difficult to write a threshold-based rule having the desired effect if an application will face varying external metrics.

2.3.2 Time-series Analysis

We now examine an additional, substantially more complex, form of auto-scaling situated upon predictive time-series analysis. Time-series analysis seeks to find a repeating pattern in application load, and then horizontally auto-scale the application based on these patterns [19]. For example, if time-series analysis indicated a pattern in the application needed $2x$ resources every Friday at 5pm, it would be possible to auto-scale the application to $2x$ resources at this time. If we are able to compose a number of these observations, we can create a policy for the entire auto-scaling behavior of the given application by evaluating the application's predicted external environment and determining the resources the application will need to operate in said environment.

There are a variety of techniques for conducting time-series analysis auto-scaling including pattern matching, signal processing, and auto-correlation [**?**].[15] Like threshold-based rules for auto-scaling, there are significant advantages and disadvantages to time-series analysis. Unlike threshold-based rules which are marked by simplicity, time-series analysis is significantly more complex. This complexity allows time-series analysis to be particularly fine-grained and effective at responding to external changes when said changes have a pattern.[16] However, time-series analysis requires a large amount of data and also substantial mathematical knowledge; it certainly cannot be implemented as easily as specifying a few simple thresholds. Additionally, while time-series analysis works well for auto-scaling with respect to patterns, it does not work well when the external application load is random, or incorporates elements of randomness. As such, predictive time-series analysis is often combined with reactive threshold-based rule auto-scaling to ensure the benefits of both.[17]

2.3.3 Control theory

Our next auto-scaling technique is predicated on control theory. Control theory is normally used for reactive auto-scaling, although it can also be used in a predictive context. The simplest implementation of a control system with respect to auto-scaling utilizes feedback controllers [19]. Abstractly, a feedback model functions by continuously examining a set of output parameters, and then tweaking a set of desired input parameters in an attempt to ensure the output parameters maintain some desired state. More concretely with respect to auto-scaling, the output parameters would be the current state of the application instances, such as the percent CPU utilization or the amount of memory the instances were using. The input parameters would be the number of instances of the

[15] Netflix utilizes a combination of methods in their predictive time-series analysis auto-scaler, Scryer [25].

[16] An example of a change with a pattern would be Netflix users who are more likely to watch TV at 10pm than 10am.

[17] Netflix's Scryer implements this combination of time-series analysis and threshold-based rule auto-scaling. [25]

application currently running. A feedback model can implement auto-scaling as the number of application instances will vary in accordance to the external load on the application, which will ensure that the application instances maintain certain operation metrics. For example, we could specify that the feedback controller should auto-scale applications such that all application instances utilize 70% of the CPU.

Done correctly, feedback control theory offers substantial advantages over threshold-based rules. Specifically, it is as simple to write auto-scaling specifications with control theory as it is to write specification with threshold-based policies, as in both the author simply defines well-understood resource metrics. Yet, it is easier to determine the effects of feedback control systems. When a new instance is created as the result of the violation of a threshold-based rule, we do not exactly know what the result will be with respect to the metrics we care about. However, with a feedback control system, we are certain about the results of the auto-scaling, as we auto-scale specifically to ensure the maintenance of certain metrics.

Kubernetes currently implements auto-scaling through a feedback control system. While we will spend substantially more time discussing the Kubernetes auto-scaling implementation later, the basics are as follows. The user specifies a target resource metric, for example CPU utilization. At a specified time interval, Kubernetes then examines the current values of the resource metric, and updates the number of application instances to ensure the current actual value equals the target value [7]. In the context of control theory, the output is the CPU utilization for each machine and the input is the number of application instances, which varies to ensure the output is at the proper level. Using this method, it is possible to auto-scale such that the application is always running at 50% CPU utilization.

Predictive Feedback Control

As previously mentioned, feedback control systems are typically reactive, meaning that the metrics used are based on the current state of the system. However, we can also consider a feedback control system that is predictive. We call this Model Predictive Control [19]. Again, we will spend significantly more time discussing predictive feedback control later. Put simply, it is an implementation of feedback control based auto-scaling, but the outputs are predictions about the future state of the application instance, instead of the current state of the application instance.

Adding prediction to feedback control offers significant benefits. The most significant benefit is accounting for the application's start up time when auto-scaling. With predictive feedback control, we can create instances of the application so they are ready as soon as they are needed. For example, if it takes 10 minutes for our application to be created and ready to operate, and we predict we will need to application at 4pm, we can begin building it at 3:50 pm, so it is ready as soon as needed.[18] Ultimately, we hypothesize that adding a predictive component to Kubernetes' current feedback control auto-scaling will allow us to auto-scale in a manner that maintains efficient resource utilization and improves quality of service.

[18] If we were using reactive auto-scaling, we would not know we needed the application until 4pm, and it would not be ready to run until 4:10 pm.

2.4 Summary

In summation, we discussed the variety of methods for performing resource intensive computational tasks, before focusing on cluster computing. We examined a variety of popular cluster managers, and investigated the distinguishing characteristics of Kubernetes, the cluster manager which is the focus of this thesis. Finally, we examined a variety of methods of auto-scaling applications running on cluster managers, and suggested a new auto-scaling method for Kubernetes entitled model predictive control. The remainder of this thesis seeks to show the effectiveness of model predictive control in modifying the current implementation of auto-scaling in Kubernetes, such that we will improve quality of service while maintaining efficient resource utilization.

Chapter 3

Appendix

3.1 Technologies Underlying Kubernetes

3.1.1 Containerization

Virtualization

Containerization

Docker

3.2 Trends Motivating Kubernetes

3.3 Microservices

Bibliography

[1] https://aws.amazon.com/ec2/.

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

[3] Auto scaling developer guide. http://aws.amazon.com/documentation/autoscaling/.

[4] Google compute engine. https://cloud.google.com/compute/.

[5] Google container engine. https://cloud.google.com/container-engine/docs/.

[6] Kubernetes design overview. http://kubernetes.io/v1.1/docs/design/README.html.

[7] Kubernetes horizontal pod autoscaler proposal. https://github.com/kubernetes/kubernetes/blob/master/docs/design/ho
pod-autoscaler.md.

[8] Kubernetes website. http://kubernetes.io.

[9] Microsoft azure. https://azure.microsoft.com/en-us/.

[10] Google cluster. https://talks.golang.org/2012/splash/datacenter.jpg, 2012. [Online; accessed Nov 13, 2015].

[11] Baker, M., and Buyya, R. Cluster computing: the commodity supercomputer. *Software-Practice and Experience 29*, 6 (1999), 551–76.

[12] Beckman, P. H. Building the teragrid. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences 363*, 1833 (2005), 1715–1728.

[13] Corporation, I. D. The digital universe of opportunities: Rich data and the increasing value of the internet of things, 2014.

[14] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley Publishing Company, USA, 2011.

[15] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. Cluster-based scalable network services. *SIGOPS Oper. Syst. Rev. 31*, 5 (Oct. 1997), 78–91.

[16] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R., Shenker, S., and Stoica, I. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2011), NSDI'11, USENIX Association, pp. 295–308.

[17] Isard, M. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev. 41*, 2 (Apr. 2007), 60–67.

[18] Jacobson, D., Yuan, D., and Joshi, N. Scryer: Netflix's predictive auto scaling engine, 2013.

[19] Lorido-Botrán, T., Miguel-Alonso, J., and Lozano, J. A. Auto-scaling Techniques for Elastic Applications in Cloud Environments. Research EHU-KAT-IK, Department of Computer Architecture and Technology, UPV/EHU, 2012.

[20] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, 2013), pp. 351–364.

[21] Tanenbaum, A. S., and Steen, M. v. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[22] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (New York, NY, USA, 2013), SOCC '13, ACM, pp. 5:1–5:16.

[23] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 18:1–18:17.

[24] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (Bordeaux, France, 2015).

[25] Yuan, D., Joshi, N., Jacobson, D., and Oberai, P. Scryer: Netflix's predictive auto scaling engine - part 2, 2013.