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

April 16, 2016

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 [46].

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 [46]. 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 [46].

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 [36].[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 [37]. 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 [49]. 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 [25].

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 [40]. Alternatively, a number of cluster managers measure themselves based on efficient resource utilization (ERU) [49]. 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 [48]. 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 [8] 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 [34]. 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." [46] 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 [38]. 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 [37]. Cloud computing, as implemented by Amazon Web Services [2], Microsoft Azure [28], and Google Compute Engine [8], continue to revolutionize the development 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. [33]

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 [46]. 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 [35]. 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 [49]. 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 [45]. 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 [45]. While two-level scheduling is lightweight, simple, and offers advantages with respect to data locality, it is not effective for long-running, production jobs [45]. 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 [45]. 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... [39]

Table 2.1: Overview of Cluster Management Paradigms.

	Job Type	**Scheduling Model**	**Open Source**
Borg [49]	Both	Monolithic	No
Omega [45]	Both	Shared state	No
Mesos [39]	Batch	Two-level	Yes
YARN [47]	Batch	Monolithic	Yes
Kubernetes [25]	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 [49]. 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 [45]. 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 [39]. 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 [47]. 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 [14]. 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 [9].

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 [42]. While both of these variations of auto-scaling accomplish the same goal, horizontal auto-scaling is a little 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] Historically, vertical auto-scaling has been more complex, although that is rapidly changing. The complexity of vertical auto-scaling depends on how the application is run.[13] If the application is run directly on a machine, then it is extremely difficuly to assign the application more resources without stopping it from running, as it would require transferring a running process to a new machine with more abundant resources.[14] The complexity slightly decreases, although is still considerable, when the application is running on a virtual machine. Virtual machines can claim or relinquish resources from their host, thus allowing the application the varying resources it needs. KVM, the hypervisor within the Linux kernel, implements this process through "balloon" drivers [5]. Finally, if the application is running within a container, performing vertical scaling is simple. Linux container implementaions allocate resources using cgroups, which assign set amounts of

[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.

[13] For background on the different manners of running applications on nodes in a cluster, please see the *Virtualization* section of the Appendix.

[14] In reality, it would be extremely rare for a cluster manager to run applications directly on the nodes of the cluster with no degree of isolation.

Table 2.2: Overview of Auto-scaling Paradigms.

	Reactive vs Predictive	**Implementer**
Threshold	Reactive	Amazon Web Services [4]
Time-Series Analysis	Predictive	Netflix [41]
Control-Theory	Both (currently reactive)	Kubernetes [16]

resources to certain processes. It is possible to modify the cgroup allocation while the process is still running, meaning vertical auto-scaling without stopping the application is trivial [43]. As containerization becomes increasingly prevalent, the difference in difficulty between horizontal and vertical auto-scaling decreases. The implementations of auto-scaling we examine focuses on horizontal auto-scaling, although the majority of research done for this thesis applies to vertical auto-scaling with only minor modifications [42].

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 [42]. 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 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.[15] 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 [4]. Furthermore, Netflix implements time-series analysis auto-scaling to help it respond to the varying demand placed on its services throughout the day [41]. Finally, Kubernetes implements control-theory auto-scaling [16]. 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

[15] 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.

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 [4].[16]

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 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 [42]. 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 [42].[17] 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.[18] 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

[16] 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]

[17] Netflix utilizes a combination of methods in their predictive time-series analysis auto-scaler, Scryer [50].

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

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.[19]

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 [42]. 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 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 [16]. 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

[19] Netflix's Scryer implements this combination of time-series analysis and threshold-based rule auto-scaling [50].

system that is predictive. We call this Model Predictive Control [42]. 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.[20] 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.

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.

[20] 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.

Chapter 3

Architecture

3.1 Overview of Kubernetes

An in-depth understanding of the architecture of Kubernetes, particularly with respect to the potential implementations of auto-scaling, is vital before progressing into a closer examination of the addition of Model Predictive Control in Kubernetes.

3.1.1 History

Active development on Kubernetes began at Google in 2014, and as a result, open-source developers have invested approximately 2 years of effort at the time of this thesis being written. However, as was mentioned in the background chapter, Kubernetes is not the first cluster manager developed at Google. Over a decade and a half of Google's experience informs Kubernetes. In addition, Kubernetes incorporates communally agreed upon new innovations in cluster management [32].

It is important to remember that, in the realm of cluster managers, Kubernetes is still young. While Kubernetes development has been ongoing for the past two years, version 1.0 of Kubernetes was released July 21, 2015 [24]. Again, this recent release means that at the time of this thesis' work, non-Google users have been running Kubernetes in production for less than a year. In contrast, Mesos and YARN, the other open-source cluster managers, have their origins in projects beginning approximately seven and ten years ago respectively. This history confers both advantages and disadvantages. Long running projects like Mesos and YARN have books, conferences, and multiple companies with business models situated upon the success of this open-source project. While Kubernetes enjoys some of these advantages, particularly related to Google's sustained support, it does not have the history of some other open-source cluster managers.[1] However, Kubernetes originality means there is lots of exciting innovation left to come, particularly with respect to auto-scaling. Part of why working on Kubernetes during this thesis is so exciting is that it presents the opportunity to contribute to an already impressive project with considerable future potential.

[1] However, Kubernetes has a tremendous amount of momentum and particularly impressive engagement considering its limited history.

3.1.2 Values

As with all large computer programs, a number of values drive the development of Kubernetes. The values must pertinent to this thesis are as follows:

- Portability: As will be seen in the next section, Kubernetes can be run in a number of different environments. As a result, Kubernetes presents potential solutions to a number of different problems, and increasing Kubernetes performance, as this thesis attempts to do, will assist in this ability.

- Extensibility: Kubernetes is built to easily incorporate pluggable new changes which can be easily enabled or disabled. Understanding the importance of pluggability will motivate this thesis' suggested modifications to Kubernetes.

- Automatic: Kubernetes places an emphasis on important cluster management tasks occurring without necessary user involvement. For example, if an application crashes, Kuberentes will automatically restart it. This emphasis on automation means this thesis' work on auto-scaling should be particularly valuable, as it improves Kubernetes ability to accomplish one of its fundamental goals.

In part, Kubernetes can be distinguished from other open-source cluster mangers based on these goals, and as previously mentioned, these goals align with the goals of this thesis [32].

3.1.3 Running Kubernetes

Finally, it is important to consider the ways in which Kubernetes can be run. Kubernetes is a cluster manager, and as such can be run any number of commodity computers. These computers can be provided in many different ways. Perhaps the simplest method is running Kubernetes on a single virtual machine on one's own commodity computer. This method simulates running a single-node cluster using Kubernetes as the cluster manager. Obviously this method does not facilliate running production applications. If one does wish to run production applications, the simplest method is using the Google Container Engine, which hosts Kubernetes for the user and the user just has to submit applications to run. Alternatively, the user can host Kubernetes themselves on a number of platforms including Google Compute Engine, Microsoft Azure, Rackspace, and Amazon Web Services. Kubernetes ability to run on a number of cloud providers allows the user to run Kubernetes with a cluster of the exact size needed for their applications. Overall, Kubernetes' flexibility assits the user in accomplishing a number of differnet aims [7].

3.2 Building Blocks of Kubernetes

As was mentioned in the general overview of Kubernetes, Kubernetes benefits not only from over a decade of experience of utilizing cluster managers at Google, but also from the distributed systems community's mass adoption of new technologies with low barriers to entry for new users. Kubernetes

both benefits from this trend, and contributes to making the benefits of distributed systems more accessible. The most prominent, increasingly popular technologies/paradigms influencing Kubernetes are containerization, or more specifically containerization, and microservices. A basic understanding of both will assist in understanding the Kubernetes basic, and auto-scaling specific, design.

3.2.1 Containerization

Google often describes Kubernetes as "an open source orchestration system for Docker containers." [25] From this statement, it is clear that containers, and more specifically Docker containers, inform much of Kubernetes. To begin, it is important to understand the benefits of containers in comparison to other similar options, and also to understand the technological underpinnings of containerization and the resulting weaknesses.

First, containers are not virtual machines. Though both seek to silo different applications running on the same physical machine, the manner in which they seek to accomplish this task is very different. Virtual machines sit between the operating system and the hardware, ensuring complete separation between two virtual machines running on the same physical machine. While this strong separation ensures applications running on separate virtual machines cannot affect each other, placing a virtual machine monitor between the operating system and the hardware has efficiency costs. More specifically, virtual machines are substantially more memory intensive, and have a higher startup cost, than lightweight containers [46]. In contrast, containers should be thought of as "lightweight wrappers around a single Unix process." [43, pg. 15] As such, containers require substantially less memory and can be started, destroyed, and restarted in a matter of seconds. However, these ephemeral benefits come at the cost of limited isolation between two containers running on the same physical host. While there are ways to increase container isolation, it is not possible to achieve the complete separation benefits of virtual machines [43].

More formally, containers can be thought of as a single Unix process bundeled with the operating system and application specific files needed to run said process. Containerizing a process involves saving a snapshot of a Linux operating system running the process. This snapshot can be given to any platform which can run the container, and the platform will then be able to simulate running the application on the operating system bundeled in the container.

So far, Docker has been the leading containerization platform. While Docker is a young platform, which has only been in development for since 2013, it has seen tremendous popularity and adoption. Again, while the containerization technology upon which Docker is built was a part of Linux for over a decade, it was the Docker platform that made it truly accessible [43]. Kubernetes is one of many options for running multiple containers in a production environment.

3.2.2 Microservices

Additionally, Kubernetes fits within the recent interest in, and adoption of, microservices. Microservices is a new paradigm of developing computer applications. It is particularly suited towards developing large, complex applications requiring many computing resources - the same types of applications that are typically benefit fromrequire being run on a cluster.

In short, microservices are "small, autonomous services that work together." [44, pg. 2] More specifically, microservices reflect the following characteristics:

- Focused: Each microservice should perform a single task. Furthermore, multiple microservices should be as loosely coupled as possible. Each microservice should present a single API for performing the single task, with which other microservices can communicate without understanding more nuanced implementation details. [44]
- Stateless: Individual instances of microservices should be stateless, in that a single instance can be deleted at any time without loosing any state. This requirement often leads to application's being containerized, and communicating with external, more permanent databases.
- Concurrent: Multiple instances of a microservice should be able to work together to divide up the work presented to a single abstraction API. For example, if there is a microservice to support a search API, then it should be possible to run multiple instances of this search microservice and balance API requests between them so that they concurrently share work without interference.

Microservice design principles are particularly supportive of horizontal auto-scaling. Microservices should be easily replicable and easily replaceable. In other words, because microservices are stateless and can be run concurrently, it is possible to easily scale a microservice by replicating it and then dividing the work between the two. Auto-scaling down is supported, as it is possible to delete an instance of the microservice without worrying about loosing any uimportant state. When the design principles of microservices are followed, horizontal auto-scaling is possible, and its implementation supports Kubernetes goal of easy automation. Additionally, applications that follow microservice design principles are particularly easy to containerize.

As will become evident in the following sections, Kubernetes is designed to easily run applications developed based on the microservices model.

3.2.3 Summary

Again, it is important to remember that containerization, microservices, and cluster management are not entirely new innovations. Yet, what is unique and important is these advancements intended audience. While development to expand the technical merits of containerization, microservices, and cluster management continues, equally important, and more novel, work is being done to increase educational resources and promote accessibility. Kubernetes core value of automation, and this thesis specific goal of improving the performance of auto-scaling, coincides with this goal of accessibility, as it reduces the scope of concern for a new user of these technologies and allows them to focus on their application.

3.3 Components of Kubernetes

Kubernetes introduces multiple new abstractions for running containers in production. Discussions of auto-scaling are much easier if the vocabulary of Kubernetes is shared. While these terms are

Kubernetes specific, other cluster managerscontainer orchestrators may incorporate similar terms.

3.3.1 Pods

The pod is the smallest deployable unit that Kubernetes can create and manage. Importantly, the pod's role as the smallest abstraction means a user of Kubernetes does not interact with containers, except to the extent that containeres are contained within a pod. A pod can contain one or more applications, and one or more containers can comprise each of these applications. Multiple applications should run on the same pod if these applications need to be run on the same physical machine; otherwise, the applications should be in separate pods. Applications within a pod can see each other's processes, access the same IP network, and share the same hostname. Like well-designed containers within the microservices model, well-designed containers should be focused, stateless, and concurrent. Kubernetes assumes that pods can be deleted, created, and replicated at will. The user can either submit a single pod to Kubernetes to schedule and run on a node in the cluster, or pods can be created by a replication controller [21].

3.3.2 Replication Controllers

As mentioned in the previous section, pods can be created by a replication controller. A replication controller is responsible for ensuring that a given number of replica pods are currently running. It does this by restarting any deleted or terminated pods. Though the replication controller performs a seemingly simple task, it is useful for scaling the number of pods and for performing a rolling update of the pod [22].

Additionally, auto-scaling is closely associated with replication controllers, as autoscalers are attached to replication controllers for pods. When an auto-scaler is attached, a replication controller no longer ensures a given number of replica pods are running, but rather ensures that the number of replica pods will result in pods consuming certain percentages of resources. This relationship will be discussed in considerably more detail later when examining the architecture and implementation of auto-scaling in Kubernetes [16].

3.3.3 Services

The final main architectural building block of Kubernetes is the service. Services are necessary because pods are ephemeral. As a pod may be deleted or replaced at any time, it is important that no entity is trying to communicate with a specific pod, because that pod may disappear. What is needed is a consistent endpoint with which entities wishing to communicate with a pod can always contact. A service is just such a consistent endpoint. Services prevent a single, long-running access points for multiple replica pods, as it receives requests to the pods, and load-balances them among the replicas.[2] This endpoint is used within a Kubernetes cluster as pods wish to communicate with each other and it can also be exposed outside of the cluster. Again, the concept of a service is particularly important to auto-scaling, as when the replication controller creates pod replicas, the

[2]The replication of these pods is handled by a replication controller.

services ensures work is balanced across them. This load-balancing, and the knowledge of the new replicas on which to share the load, is entirely automated. [23]

3.4 Autoscaling in Kubernetes

3.4.1 Current Implementation

Currently Kubernetes implemenents horizontal pod auto-scaling. As this title suggests, auto-scaling in Kubernetes invovles creating an autoscaler for a specific replication controller such that the replica pods handling the requests load-balanced by the service operate within a specified resource range. Kubernetes implementation of auto-scaling is referred to as horizontal because auto-scaling occurs by creating replicas of each pod and then dividing the work among these replicas, as opposed to expanding the resources of any single pod. While not yet implemented, Kubernetes reliance upon containers means vertical auto-scaling is a possibility in the near future, as there are a variety of methods for increasing the resources available to a container without stopping execution [43].

As was distinguished in the background section, Kubernetes currently implements reactive feedback control auto-scaling, which is defined as when auto-scaling occurs to ensure the preservation of a certain state. Reactive auto-scaling can be visualized through Figure 3.1. This operation becomes clear when examining the execution of the horizontal pod autoscaler. To begin, the autoscaler operates as a control loop, as it queries pods at a set interval to determine their current resource utilization, and performs auto-scaling actions depending on the resulting utilization. At the start of this thesis, the only resource on which it is possible to determine auto-scaling behavior is CPU utilization percentage, but possible resources could conceivably expand to include percent memory utilized, percent network bandwidth used, etc. Based on the average CPU utilization percentage, the auto-scaler will create or delete replica pods to ensure average CPU utilization percentage is within the target range the user specificed when creating the autoscaler. The algorithm for determining the correct number of replica pods will be discussed in detail in the next section. The final implementation detail is that, in order to ensure auto-scaling is not attempted to frequently, once a scale up or scale down happens, no more auto-scaling will occur for a constant interval.[3] This waiting time ensures auto-scaling can actually take effect before it is potentially attempted again [17].

3.4.2 Algorithm

The current algorithm for determing the number of replica pods at each interval of the control loop is fairly simple. It is assumed that auto-scaling is occuring based on the pod's CPU utilization percentage, but this same algorithm could be applied to other resources without the loss of generality. To begin, it requires the definition of three variables. Let *TargetNumOfPods* be defined as the number of replica pods which should exist. The replication controller will be responsible for ensuring these pods exist. Let *SumCurrentPodsCPUUtilization* be found by multiplying the average pod CPU utilization by the current number of replica pods. Finally, let *TargetCPUUtilization* be the desired

[3]This interval is currently five minutes from the most recent rescaling for scale-down and three minutes from the most recent rescale for scale-up.

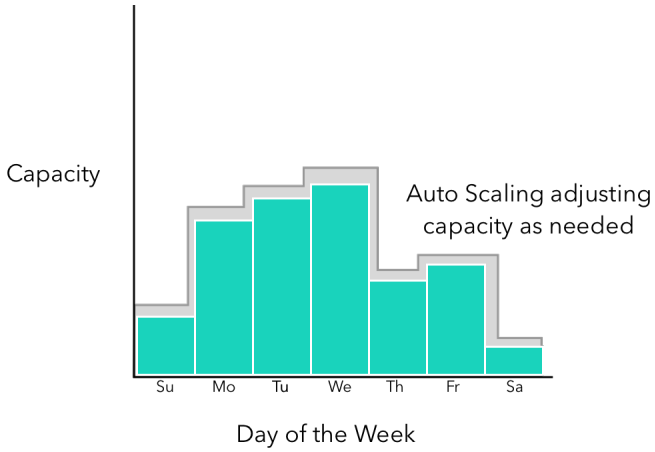

Figure 3.1: A Reactive Auto-scaled Application

level of per pod CPU utilization percentage as specified by the user when they created the autoscaler. The algorithm can now be written as follows:

$$\text{TargetNumOfPods} = \text{ceiling}(\text{SumCurrentPodsCPUUtilization} \,/\, \text{TargetCPUUtilization})$$

The *ceiling* function simply ensures that there is no attempt to create fractions of pods. Additionally, to keep auto-scaling from being overly sensitive, scaling will only occur if the current resource utilization is outside of a 10% tolerance range [16].

Implementing this thesis' new auto-scaling additions, namely utilizing predictive feedback control, will require modifications to this algorithm, which will be examined in detail later.

3.5 Predictive Autoscaling in relation to Kubernetes

3.5.1 Overview

As described in the Background chapter, this thesis examines the impacts of implementing Model Predictive Control auto-scaling, instead of the current Model Reactive Control auto-scaling. With a greater understanding of the building blocks of Kubernetes, and the current implementation of auto-scaling in Kubernetes, it is now possible to understand Model Predictive Control in a Kubernetes specific context. Given the Kubernetes specific auto-scaling implementaiton is horizontal pod auto-scaling, the updated, predictive version investigated in this thesis will be referred to as predictive horizontal pod auto-scaling.

Converting Kubernetes from reactive horizontal pod auto-scaling to predictive horizontal pod auto-scaling requires one major conceptual modification to the auto-scaling process. As the name predictive implies, auto-scaling no longer occurs in reaction to the current resource consumption of the pod. Rather, it occurs predictively based on the pod's predicted future resource consumption.

The amount of time ahead for which resource consumption is predicted is dependent on the amount of time it takes a replica pod to be ready to partake in the work the service is balancing across all replica pods.[4] This interval of time is referred to as *PodInitializationTime*.

Aside from the switch to using predictive resource measurement, the general architecture of predictive horizontal pod auto-scaling is similar to the architecture of horizontal pod auto-scaling. Again, the autoscaler operates as a control loop, yet at each looped interval, it utilizes the current resource utilization of pods to predict the resource utilization of the pods after *PodInitializationTime*. Assuming the resource in question is CPU utilization percentage, the auto-scaler will create or delete replica pods to ensure the future CPU utilization percentage will be within the target range the user specified when the autoscaler. It is important to remember that, while the new replica pods are created immediately, they are not actually initialized and reducing the work load of the other replica pods until after *PodInitializationTime*.

3.5.2 Algorithm

A couple of changes need to be made to the current reactive horizontal pod auto-scaling algorithm in order for it to incorporate prediction. Again, we assume that auto-scaling is occuring based on the resource utilization metric of CUP utilization percentage. Additionally, some of the variables from the reactive horizontal pod auto-scaling algorithm appear again. Again, let *TargetNumOfPods* defined as the number of replica pods which should exist and let *TargetCPUUtilization* be the desired level of per pod CPU utilization percentage that the used specified when the autoscaler was created. Let the function *LineBestFit* be a line of best fit calculated to fit a plotting of observation time, x, and previous CPU utilization y.[5] Finally, let *PredictionTime*, t, be the time in the future for which we seek to predict the *FutureCPUUtilization*. We calculate *PredictionTime* as follows:

$$\text{PredictionTime} = \text{CurrentTime} + \text{PodInitializationTime}$$

We then use *PredictionTime* and the *LineOfBestFit* function to calculate *FutureCPUUtilization* as follows:

$$\text{FutureCPUUtilization} = \text{LineOfBestFit}(\text{PredictionTime})$$

The variable *SumFuturePodsCPUUtilization* can be defined by multiplying *FuturePodCPUUtilization* by the current number of replica pods. With these variables defined, it is possible to calculate *TargetNumOfPods* as follows:

$$\text{TargetNumOfPods} = \text{ceiling}(\text{SumFuturePodsCPUUtilization} \;/\; \text{TargetCPUUtilization})$$

[4]For example, a pod may run a web server takes multiple minutes to initialize, and until that web server is initialized, the pod cannot handle any web requests load balanced to it by the service

[5]We will discuss the method for calculating this line of best fit in the implementation section

The *ceiling* function definition and the tolerance range of 10% remainds from reactive horizontal pod auto-scaling algorithm. With these changes, *TargetNumOfPods* is now determined predictively.

3.5.3 Benefits of Predictive Autoscaling

Overall, this thesis seeks to answer the question of how to ensure Kubernetes uses as few resources possible while still supporting the application's resources needs. Within an understanding of the internal of Kubernets, it is now possible to understand the theoretical underpinnings of how predictive horizontal pod auto-scaling accomplishes the stated goal.

Specifically, utilizing prediction approves quality of service, without requiring any additional resources, thus achieving the goal of increasing the summation of efficient resource utilization and quality of service. Prediction increases quality of service, because it helps avoid the decreases in quality of service that can occur if pod initialization time is not taken into account. These decreases can best be understood in the context of a simple example. Imagine that if takes 10 minutes for a pod to initialize. Imagine that at 5:50pm reactive algorithm instructs the autoscaler that 10 pods should exist and at 6:00pm, the reactive algorithm instructs the autoscaler that 20 pods should exist. Using the reactive algorithm, the 10 replica pods will not be created until 6:00pm, and as a result of the 10 minute pod initialization time, they will not be ready to balance the load until 6:10pm. This delay means that from 6:00 - 6:10pm, 20 pods will be needed to keep the pods operating at a level that ensures high quality of service, but only 10 pods will be operating. Thus, for 10 minutes there will be low quality of service, as pods try to operate without enough resources. This delay is particularly troublesome if pod intialization time is particularly high or if there is a particularly high quality of service cost for pods operating outside of target resource consumption. A large part of the evaluation component of this thesis the prevalence and impacts of such situations.

Adding prediction avoids this problem, ensuring a consistently high quality of service. Consider the following example again, yet this time with predictive horizontal auto-scaling. Again, imagine the exact same scenario as above, yet this time, the predictive algorithm will instruct the creation of 10 replica pods at 5:50pm. By 6:00pm, when 20 replica pods are needed, 20 replica pods are initialized and ready to balance the load, ensuring that all pods operate at a higher quality of service than they would otherwise. Importantly, this modification does not decrease efficient resource utilization, as the same amount of resources are used as before, just at different times.

@TODO: Explain in greater depth how efficient resource utilization remains high.

It is clear that adding prediction has the potential to increase the summation of efficient resource utilization and quality of service for Kubernetes. To what extent it is able to manifest this potential will be made clear in the evaluation section.

3.6 Summary

In just a short time, Kubernetes has become a powerful option for open-source cluster management and container orchestration. Kubernetes is particularly important in that decreases the barrier of entry for exciting new technologies, particularly containers, and programming paradigms, particularly

microservices, and grants users to benefits of portability, extensibility, flexibility, and automation. Kubernetes is based on lower level objects called pods, replication controllers, and services, and builds upon these to implement a model reactice control method of auto-scaling, which it calls horizontal pod auto-scaling. This thesis builds upon this work to implement a model predictive control method of auto-scaling, entitled predictive horizontal pod auto-scaling. The belief is that predictive horizontal pod auto-scaling will increase the summation of quality of service and efficient resource utilization. This hypothesis will be verified through evaluation.

Chapter 4

Implementation

4.1 Technical Overview

As has been mentioned frequently throughout this thesis, Kubernetes is an open-source project sponsored by Google. Unsurprisingly, given the scope of the problems Kubernetes seeks to solve, the code base is very large. Overall, Kubernetes contains approximately three million lines of application specific code and thousands more of configuration scripts and documentation. The majority of application specific code is written in Go, a relatively recent open-source language also supported by Google. The influence of the Go programming language is seen throughout the project, particularly in Go's native support for parallel processing.

Given the size of Kubernetes, larger contributions adding substantial new functionality, including the contributions made by this thesis, follow a standardized contribution process. First, the developer must submit a proposal with their idea to the community. The community can make comments, and it must receive approval from core members of the Kubernetes if development is to be merged into the master release. Often this discussion involves members of the Kubernetes team focusing on the area relevant to the proposed changes. For this thesis, the proposal was considered by the internal auto-scaling team. After the proposal is approved, code containing the new feature, in this case predictive horizontal auto-scaling, will be added in parts over the course of several smaller contributions. Again, each request must be approved by the Kubernetes core team if it is to become part of the project. Development on Kubernetes follows the standardized Git workflow.

This thesis' specific contributions to Kubernetes, and their justifications, are described in the remainder of this section. The majority of the contributions are in the relatively small horizontal podautoscaler controller segment of the code, although an understanding of a greater portion of the code base was needed to make these changes. Overall, approximately 300 lines of codes were contributed to Kubernetes over the course of this thesis.

Currently, the modifications of this thesis have not been merged into the Kubernetes master branch that forms the basis for the majority of Kubernetes deployments. However, it is possible to deploy our modified version to many of the options for deploying Kubernetes should one desire. As our code continues to go through the submission process, we are optimistic that predictive auto-

scaling will be a feature in the standard Kubernetes deployment.

4.2 Enabling Additions

This section analyzes the specific additions enabling predictive auto-scaling. The changes are broken intro four groups: Recording Pod Initilization Time, Storing Previous CPU Utilizations, Autoscaling Predictively, and Enabling Predictive Autoscaling. The first two groups do not directly relate to predictive auto-scaling, but rather are the scaffolding that makes predictive auto-scaling possible. The third group, Autoscaling Predictively, builds upon the first two sections in implementing an alternative control flow, which when enabled, causes the auto-scaler to operate predictively instead of reactively. The last group of changes discusses how to turn in this alternative control flow, and thus enable predictive auto-scaling. While breaking this thesis' code into discrete compartments is in part just a reflection proper software design, it is also done to work within the framework of making open-source contributes to Kubernetes. Kubernetes preferences small, self-contained requests, which are able to function independently. Implementing the sections sequentially, ensures we never submit code which will not run or build.

4.2.1 Recording Pod Initialization Time

This thesis' implementation of predictive auto-scaling predicts *pod initialization time* into the future. As was made explicit in the *Benefits of Predictive Auto-scaling* section, this time frame allows any replica pods created to be ready to share in the work by the time in which they are needed.

Pod *initialization* time is distinct from pod *creation* time and the idea of a pod *running*. We define an initialized pod as one that is ready to perform computational work. This contrasts with a *created* pod, which merely signifies that all the containers within the pod are created. It does not signify that these containers have started running, performed any initialization tasks, and are now ready to share in computational work. The time necessary to create all containers for a pod is not equal to the time necessary for all containers within the pod to perform their computational work. Furthermore, a *running* pod is merely a pod in which all containers have been created, and at least one container is running or in the process of starting. This description does not guarantee that the containers have started and are ready to perform work [20]. With conceptions of pod creation and pod's running being insufficient to determine whether a pod has initialized, we must find an alternative mechanism.

Fortunately, Kubernetes defines the idea of a *Readiness Probe*. A readiness probe shows whether a pod is ready to handle incoming requests. Services use this probe to determine whether to pass work along to a newly created replica pod through ensuring that pods with long startup times do not receive proxy traffic until ready to handle it. Pods must implement a readiness probe, or else it will be assumed that if the pod is running, it is ready [20]. Each container within a pod defines its own readiness probe, by specifying an HTTP endpoint that will return a successful HTTP status code when sent a GET request if the replica pod is ready [26].[1] In terms of this thesis, a pod being

[1] There are alternative methods of defining a readiness probe, but specifying an HTTP endpoint is the most logical

ready and a pod being *initialized* are analogous terms.

We can rely on the existence of the readiness probe to measure the amount of time it took for a pod to initialize. Each pod records the time at which it was first ordered into existence. Subtracting the time at which the pod first came into existence from the time at which the readiness probe first returned *Success* results in the pod initialization time we desire. However, while Kubernetes records the time at which the pod first came into existence as *pod.CreationTimestamp*, it does not record the time at which the readiness probe first returned *Success*. Fortunately, Kubernetes pods record their current state and all states in which they have been, and the time at which they assumed said states. As such, the following algorithm calculates the average initialization time for all pods controlled by the auto-scaler.

- For all pods controlled by the auto-scaler.
- If the pod was ever in the ready state, find the first time that occurred. If the pod has never been in the ready state, skip this pod.
- If a *InitializationTime* value has not already been recorded for this pod, then subtract the *CreationTime* from the time at which the pod entered the ready phase, and record this time as *PodInitializationTime*
- Add the value for *InitializationTime* to *TotalInitializaitonTime*.
- Divide *TotalInitializationTime* by the total number of pods that have been ready at some point.

This algortihm allows us to record the initialization time for each pod controlled by the auto-scaler, as well as calculate the average pod initialization time for all pods controlled by this auto-scaler.[2]

At this point, all recorded initialization time values factor into the average. This inclusion could lead to a problem with extreme outliers drastically affecting the average value. Perhaps in future iterations of predictive pod auto-scaling it would make sense to only average values that fit within some multiple of the standard deviation.

With this implementation, we can now use initialization time to determine how far into the future to predict the state of the application on the cluster, and thus how far into the future to auto-scale.

4.2.2 Storing Previous CPU Utilizations

The addition to record pod initialization time described in the previous section tell us how far into the future we want to predict the resource usage of the application. However, we still need a method of predicting the resource usage of the application at this point in time. To make this prediction of future CPU utilization values, we need to store multiple previous CPU utilization values.

within the context of this thesis.

[2]We record the value in the *Annotations* map for each pod, which is basically a map for writing values that are not necessarily in the Pod API object. If predictive auto-scaling becomes particularly popular, *InitializationTime* may be included as a field in the Pod API object, although making such a change would be a fairly substantial process.

It is necessary to make a couple of changes to the Kubernetes code base in order to faciliatate this additional storage. To start, consider the contents of a *HorizontalPodAutoscaler* object, which is just an Kubernetes api object representing an auto-scaler. This object contains two objects entitled *HorizontalPodAutoscalerSpec* and *HorizontalPodAutoscalerStatus*. Traditionally within Kubernetes, *Spec* represents the desired state of the object and *State* represents the current state of the object. Before the addition of predictive auto-scaling, the *HorizontalPodAutoscalerStatus* object contained fields for the current and desired number of replicas, the last time at which scaling occured, and the current average CPU utilization percentage for all pods controlled by this auto-scaler object. This last field, entitled *CurrentCPUUtilizationPercentage*, provides part of the information needed to estimate the predict the future CPU utilization percentage [15]. However, to make any decent estimation, we need to know at the previous CPU utilization percentages at least as far into the past as we wish to predict into the future.

Thus, we start tracking a new field entitled *PreviousCPUUtilizationPercentages*. This field is a list of average previous CPU utilization percentages in integer form. Fortunately, Kubernetes already implements code that updates the *CurrentCPUUtilizationPercentage* value at a set duration interval. We modify the code such that everytime a new *CurrentCPUUtilizationPercentage* is recorded, we add onto the queue of *PreviousCPUUtilizationPercentages*. We implement our queue such that it is of a fixed length, and newer utilization percentage readings bump older ones from the queue should we exceed the total number of observations we feel we need to keep to be able to make accurate predictions about the future. In other words, at time t_j, the *HorizontalPodAutoscaler* object will have access to CPU utilization percentage values from t_i to t_j, where t_i is the first observation recorded after $t_i - (p * c)$, where p is the amount of time it takes an average pod being run by this auto-scaler to initialize,[3] and c is a constant multiplier.[4] In our queue, we additionally record the time at which previous CPU utilization observation was taken, meaning that we can easily find a simple line of best fit for the graph in which time is the independent variable and CPU utilization is the dependent variable. Extrapolating with this line of best fit allows us to predict the future CPU utilization of our pods.

4.2.3 Autoscaling Predictively

Now that we have methods for calculate the average pod initilization time and keep records of previous average CPU utilizations, we can now auto-scale predictively. To do so, we must add an alternative branch of execution to the current method of reactive auto-scaling.

With reactive auto-scaling, Kubernetes calculates the average current CPU utilization across all pods and divides this number by the target CPU utilization percentage to obtain the *UsageRatio*. This usage ratio is then multipled by the current number of replica pods, resulting in the desired number of replica pods. A similar process occurs through predictive auto-scaling, with one significant change. Instead of calculating the average current CPU utilization, we seek to calculate the average

[3]The amount of time it takes an average pod being run by this auto-scaler to initialize is how far into the future we seek to predict the state of the cluster.

[4]In our current implementation, c has a value of 1, indicating that we will predict based on past data equal in distance from the current time as the point in the future at which we wish to predict.

predicted CPU utilization at *PodInitializationTime* into the future. Given this value, we again calculate *UsageRatio* and multiply that value by the current number of replica pods to give us the target number of replica pods.

Thus, the final step to implementing predictive auto-scaling is calculating average predicted CPU utilization. There are a variety of different methods that we could use for making this prediction, but for our initial experiments, we choose the simplest method of generating a linear line of best fit for a plotting of *Time*, x, as the independent variable and *CPUUtilization*, y as the dependent variable. As the points on our plot, we use all of our previous CPU utilization measurements and the current CPU utilization measurement, with *Time* recorded as Unix seconds[5] and CPU utilization recorded as a average percent CPU utilization. Given these separate lists of x and y values, we can now find a line of best fit. We define the line of best fit as the line minimizing the squares of any derivations between predicted and actual average CPU utilization. We calculate the slope of this line, b, with the following equation:

$$b = \frac{Cov_{xy}}{Var_x}$$

Cov_{xy} is a measure of how *Time* and *CPUUtilization* vary with respect to each other, while Var_x is a measure of the squared standard deviation of all the different *Time* values from the mean *Time* measurement.

Additionally, it has been proven that a linear line of best fit calculated in this manner will pass through the sample mean of *Time* and *CPUUtiization* respectively. Thus, we can calculate the y-intercept of our line of best fit, a, by subtracting $b * \overline{x}$ from $\overline{y}$. With these follows, we have a linear line of best fit, which we can use to make simple predictions about future CPU utilization [27].

Our final step is to get a *Time* value to plug into this line of best fit to receive a future prediction. As we want to predict *PodInitializationTime* into the future, we simply add *PodInitializationTime*, measured in seconds, to the current time, for a new value t_p. By calculating $a + b * t_p$, we have a prediction of the future average CPU utilization. As mentioned previously, given this prediction, we can now plug it in as if it was the current CPU utilization, and proceed with the typical reactive method of auto-scaling.

@TODO Get a better source to cite than the Radford article.

Naturally, our method here is making a considerable assumption that a linear line of best fit can accurately model CPU utilization and also the amount of time we attempt to extrapolate when making our prediction is not too extreme. Our evaluation section will help us how concerned we should be about these potentially challenging aspects. Fortunately, should a linear line of best fit not prove to be a suitable first attempt, our implementation is designed such that it would be easy to examine a number of alternative modelling solutions, such as quadratic, exponential, or logarithmic lines of best fits. It would even be possible to try all the different modelling options, and ultimately select the one that had proven itself most accurate.

[5]Unix seconds are the number of seconds from a specific date in 1970

4.2.4 Enabling Predictive Autoscaling

With all of the pieces for predictive auto-scaling in place, the final step is making it so that a user running pods on the Kubernetes cluster can turn prediction on and off. We seek a lightweight method for accomplishing this, so as to not have to make drastic changes to the Kubernetes user interface before being confident that predictive auto-scaling is a generally useful addition.

Fortunately, Kubernetes makes it possible to attach key/value pairs to any resource through the concept of annotations. As was mentioned in the section on recording pod initialization time, the annotations of an object are a useful way to record information without going through the long process of making an official update to the API for an object in Kubernetes. Thus, to turn on auto-scaling for an individual auto-scaler object, we simply record in the annotations for that object the key *predictive* with the value *true*. This annotation can easily be done using Kubernetes command line client [18]. For example, if our auto-scaler had the name *foo*, the following command would turn on prediction.

```
kubectl annotate hpa foo predictive=‘true’
```

It follows that auto-scaling could just as easily be turned off by rewriting the annotated value to anything other than *true*.

```
kubectl annotate hpa foo predictive=‘false’
```

Within the code for determining the number of replia pods to create when auto-scaling, we can easily check if the *predictive* annotation is *true* for that particular auto-scaler. If the value is not set, or anything other than true, the auto-scaler will reactively auto-scale as normal. This flexibility allows different auto-scalers to utilize different auto-scaling methods. Additionally, it allows a single autoscaler to utilize different auto-scaling methods throughout its lifetime.

If the benefits of predictive auto-scaling are substantial, then it will make sense to transition this value from one recorded in *Annotations* to a more permananet field defined a autoscaler's *HorizontalPodAutoscalerSpec* object. A field entitled *Predictive* could be added to this object which, if set to true, would turn on predictive auto-scaling. This addition would enable to configure a autoscaler to perform predictively from the start, as opposed to having to first create the autoscaler and then turn on prediction. Making prediction a part of the static configuration for an autoscaler has the benefit of linking predictive behavior with all other pod state, especially as configuration files are used among multiple projects. However, until substantial benefits of predictive auto-scaling are demonstrated, it does not make sense to undertake this effort.

4.3 Overview

In all, we have added predictive auto-scaling to Kubernetes, as well as a creating infrastructure to make this addition possible. We added calculation of the average pod initialization time as well as storing previous measurements of CPU utilization and the time at which they occurred. These two pieces of information allowed us to create a linear line of best fit which we used to predict the future

CPU utilization, and thus predict the number of replica pods we would need in the future. In short, it allowed us to predictively auto-scale. Finally, we specified a mechanism for turning predictive auto-scaling on and off. Our version of Kubernetes, with predictive auto-scaling operational, can be deployed to Amazon Web Services or Google Cloud just like any other Kubernetes version.

Chapter 5

Evaluation

Given an addition of auto-scaling, it is now necessary to progress to evaluation. This section contains a more thorough definition of what entails successful modification of current Kubernetes auto-scaling behavior, the metrics used to determine if efforts were successful, an analysis of the control groups with which the new predictive auto-scaling will be compared, a description of the environment and process used for evaluation, the results of said evaluations, and the real world impacts of this thesis' findings.

5.1 Goals of Evaluation

In the introduction section of this thesis, we defined success as implementing modifications to Kubernetes that will increase the summation of the efficient resource utilization and quality of service metrics. Careful evaluation will confirm whether we succeeded in this objective. In all, we will seek to determine both the extent and significance of the addition of predictive auto-scaling in comparison to other methods of allocations resources with respect to our goal summation, and highlight the scenarios in which the addition of predictive auto-scaling is the most, and the least, beneficial.

5.1.1 Predictive Auto-scaling's Impact

One important aspect of evaluation will be presenting a number of different scenarios, based on independent variables that will be discussed later, in which to evaluation predictive auto-scaling in comparison to the alternative methods of assigning resources. This type of evaluation will focus on how predictive auto-scaling is an improvement, or regression, on the currently existing options. Furthermore, this type of evaluation will focus on to what extent predictive auto-scaling's difference from current implementations, is either statistically or practically significant. Finally, this type of evaluation will provide a number of visualizations, allowing for easily accessible comparisons between different types of resource assignment in different scenarios.

5.1.2 Scenario Analysis

Our evaluation will also attempt to provide further insight into predictive auto-scaling through analysing the impacts of predictive auto-scaling in relation to itself. In other words, we will seek to identify the scenarios in which benefits or determinents of predictive auto-scaling are the most pronounced, and also the scenarios in which predictive auto-scaling has little impact. This method of analysis will be useful in recommending when to enable predictive auto-scaling and also will likely suggest avenues for future work.

5.2 Evaluation Metrics

Before progressing any further, it is important to state the metrics we will use to measure the efficacy of predictive auto-scaling. We seek specific metrics relating to the general concepts of efficient resource utilization and quality of service. A selection of a "typical" service to run on Kubernetes influences how exactly we will measure efficient resource utilization and quality of service. We posit a web application as the best choice for a representative application. A number of factors support our decision. First, Kubernetes is built for running long-term, stable service jobs, and a well-built web application desires to be both long-running and crash free [14]. Second, Kubernetes focuses on running ephemeral, containerized applications which can be started or stopped at any time. Containerized web applications achieve these goals as, as long as the database layer is abstracted, web applications are stateless and can restart with few ramifications. Third, applications on Kubernetes should be concurrent, meaning replicas can be added or removed to divide the work any individual application must handle. Stateless web applications are easily parallelized, as the requests can be easily distributed across all of the replicas. Finally, much of the appeal of the simplicity of Kubernetes is that a user can construct a containerized application and then pass it to an externally managed Kubernetes cluster for hosting. This hosting simplicity is particularly appreciated by burgeoning startups, who typically develop web applications and may face extreme variance in external demand.

5.2.1 Efficient Resource Utilization

We define efficient resource utilization as a measure of whether an application has enough, but not too many resources given to it by the operating system or the cluster manager. For example, editing a text file in Vim on a super computer would be terrible efficient resource utilization, while running a web browser on a laptop is proper efficient resource utilization. In the context of Kubernetes, an application with poor efficient resource utilization would be a web server that uses many replica pods, each reserving considerable resources, to serve a very low volume of web requests.

@TODO We will just have to make it clear that if we are using Idle CPU percentage of measure efficient resource utilization, than the lower the value, the better the efficient resource utilization.

Specifically, we measure efficient resource utilization with respect to Kubernetes through examining the percentage of idle CPU. The amount of CPU that a pod reserves is the summation of all resources that containers within the pod reserve [13]. If our application is only using a small amount of that reserved CPU to run, than a large amount of CPU will be left idle. The larger the

percentage of CPU that is left idle, the worse the efficient resource utilization, as many resources are just sitting. Our specific metric for efficient resource utilization will be measured in percentage of CPU that is idle and we name it *Idle CPU*.

The efficient resource utilization metric has direct links to the costs of running applications on a cluster manager. If applications are given resources they do not need, and the cluster manager does not reclaim these unused resources, then additional applications added to the cluster must claim new resources. The inability to utilize inefficient applications' wasted resources requires the expansion of the cluster and an increase in cost that will be felt both by those running the cluster and those running an application on the cluster.

5.2.2 Quality of Service

We additionaly define quality of service as a measure of how will an application is accomplishing its goal. There does not exist a singular consistent specific metric for measuring quality of service, as measures of quality of service are dependent on the specific application. Furthermore, it is difficult to measure quality of service as a variety of difficult to control for external factors impact an application's ability to perform its goal.

In the context of the typical web application run on Kubernetes, we measure quality of service based on the *response time* to an HTTP request. An application with a high quality of service will have a low response time, while an application with a low quality of service will have a high response time. Our specific metric for quality of service will be measured in seconds.

The quality of service metric has links to the type of application which can be run on Kubernetes. As Kubernetes supports as best as possibile a high quality of service, more and more important applications will run on Kubernetes. For example, if Kubernetes works to improve the quality of service of an application, important web applications serving vital medical data or political information will seek Kubernetes as a platform on which to run.

5.2.3 Summation of ERU and QOS

Importantly, it is an improvement of the summation of efficient resource utilization and quality of service metrics that we are most interested in. It is easy to improve quality of service by decreasing efficient resource utilization, as we can just assign the application the largest amount of resources it could ever need. It is equally easy to improve efficient resource utilization by decreasing quality of service, as we can just assign an application the fewest amount of resources it will ever use. As such, we want to ensure that this thesis improves efficient resource utilization or quality of service, without negatively impacting the other. This realization leads us to evaluating our modifications to auto-scaling by measuring its impact on the summation of efficient resource utilization and quality of service.

While conceptually summing efficient resource utilization and quality of service is simple, care must be taken when combining the specific metrics of *idle CPU* and *response time*. The metrics are measured in unrelated units. Furthermore, the scale for these metrics may be entirely different, meaning that small changes in one could completely overshadow larger changes in the other.

We combine these *eru* and *qos* through the following process. First, we gather all measurements of *eru* and *qos*, distinguished by the variables E_A and Q_A respectively. Importantly, if we are seeking to compare the summation of *eru* and *qos* across multiple trials, than E_A and Q_A is *all* summations, not just the values from a single trial. Next, we define e_t and q_t as the respective *eru* and *qos* measurements at time t. We first normalize these measurements, by subtracting the individual observation value from the mean of all observations, and dividing by the standard deviation of all observations. This operation leaves us with ne_t and nq_t respectively. Then, as with our current implementation of the *eru* and *qos* metrics, smaller values indicated "better" performance, and with respect to summation of *eru* and *qos*, we expect larger values to indicate "better" performance, we negate ne_t and nq_t. Finally, we sum $-ne_t$ and $-nq_t$ together to get s_t, where s_t is the summation of eru and qos at time t. In short, we add the negation of the z-score for *eru* and *qos*. Mathematically, this process can be written as follows:

$$ne_t = ((e_t - \text{MEAN}(E_A))/\text{STDDEV}(E_A))$$
$$nq_t = ((q_t - \text{MEAN}(Q_A))/\text{STDDEV}(Q_A))$$
$$s_t = -ne_t + -nq_t$$

Given a measurement of the summation of *eru* and *qos* for a set of observations, in which *eru* and *qos* have an equal impact in the summation, it is now possible to compare the summations of *eru* and *qos* within the different scaling methods or traffic patterns included in our evaluation trials.

5.3 Control Groups

To determine the impact of adding prediction to horizontal pod auto-scaling, we must estabish baseline standards of performance with which we can perform comparisons. These "normal" standards of performance come from the control group, which includes all of the methods of scaling pods before this thesis. These methods can be divided into the two general categories of *static* and *reactive auto-scaling*. The inclusion in our control group of all previous methods of auto-scaling allows us to answer the fundamental question of this thesis: does adding prediction to Kubernetes auto-scaling improve its ability to reliably and resourcefully run containerized applications?

5.3.1 Static

The first, and the simplest method, of scaling pods is *static* provisioning. Static provisioning requires one wishing to deploy an application on Kubernetes to determine ahead of time a constant amount of pods for that application. Any desire to update that static value will require a manual change. Put simply, with the static method there will be a constant number of pods, and the application will have a constant amount of resources, throughout its entire lifetime, regardless of the amount of work the application is asked to perform.

There are multiple possible heuristics for statically assigning resources to an applications, as it is possible to over, under, or average provision.

- **Over Provision**: With over provisioning, an application is given the greatest amount of resources that it will ever require. With respect to horizontal pod auto-scaling, over provisioning means the user of the application statically sets the replication controller to ensure that x pods always exist, where x is the number of pods needed to maintain high quality of service when the application is asked to perform the most work.[1] While over provisioning ensures a high quality of service, it has extremely poor efficient resource utilization, as can be seen in Figure 5.1.

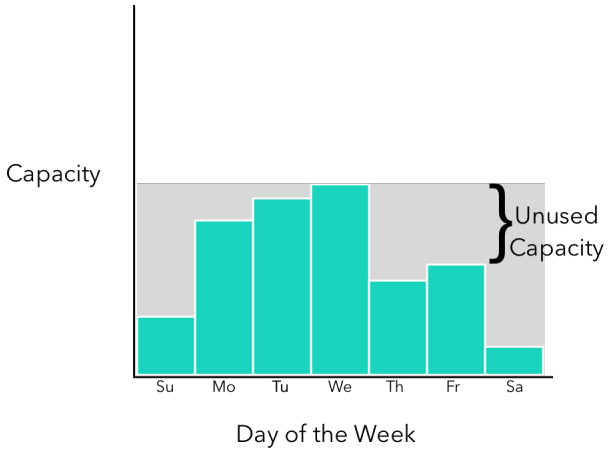

Figure 5.1: A Over Provisioned Application

- **Under Provision**: With under provisioning, an application is given the least amount of resources that it will ever require. Again, in the context of horizontal pod auto-scaling, under provisioning leads to user to statically set the replication controller to ensure the existence of y pods, where y is the number of pods needed to maintain quality of serice when the application is asked to perform the least work. Under provisioning ensures efficient resource utilization, as the application will never reserve any resources and then leave them idle. However, in all situations except for when the application performs the minimum possible amount of work, quality of service will suffer because the application does not have enough resources.

- **Average Provision**: With average provisioning, an application is given the average amount of resources that it needs. With respect to horizontal pod auto-scaling, average provisioning guides the user to statically set the replication controller to maintain z pods, where z is the number of pods needed to maintain quality of service when the application is asked to perform the average amount of work. Average provisioning can be seen as somewhat of a middle ground between under and over provisioning, offering decent quality of service and efficient resource utilization, as can be seen in Figure 5.2.

[1] In this discussion, references to *most*, *least*, and *average* work assume that there exist bounds on the work the external environment can ask the application to do.

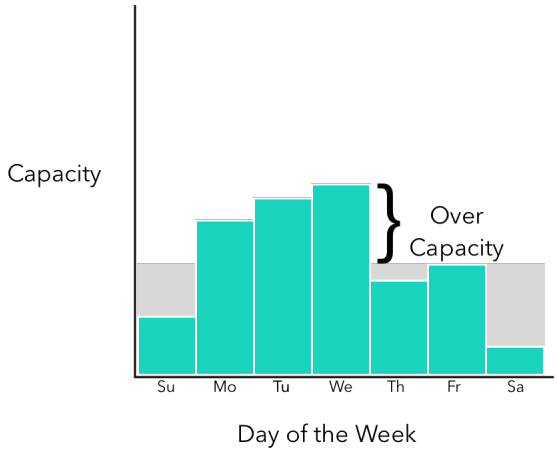

Figure 5.2: A Average Provisioned Application

Our discussion of static provisioning in Kubernetes makes the assumption that creating a static number of pods equates to reserving a static amount of resources. In the default case, the previous statement is not necessarily true, yet it is possible to craft specialized pods which validate this equality. To start, remember that pods contain containers. When Kubernetes receives a pod, it seeks to schedule all of its containers on a physical node within the cluster. By default, containers within the pod run with no bounds on their CPU and memory beyond the constrictions the physical node on which they are scheduled. As such, declaring x number of pods does not give any guarantees of resource usage, as the amount of resources available to the containers within the pod vary drastically based on their specific node [19]. This variability challenges the stability of static provisioning, and weakens its ability to serve as a control group with which we can compare predictive horizontal auto-scaling.

Fortunately, there is a way to modify configure Kubernetes such that a static number of pods equates to a static number of resources.[2] This configuration involves setting resource requests and limits for each container within the pod. A resource request for a container indicates the minimum amount of resources that should always be available. A pod will not be scheduled on a node within the cluster, unless that node can guarantee the requested amount of resources to all containers within the pod. A resource limit for a container indicates the maximum amount of resources that a container can claim. Depending on the resource, a container exceeding the maximum amount of resources will either be throttled (CPU) or killed (memory).[3] A pod's resource requestlimit is the summation of the resource requestlimit for all of its containers. Setting a container's, or pod's, resource request equal to its resource limit essentially guarantees that the existence of a pod represents the claiming and utilization of a static amount of resources [13]. Ensuring static provisioning is reserving a consistent amount of resources allows us to still examine idle CPU percentage, our way of investigating efficient resource utializaiton. We additionally incorporate similar CPU limits and requests when testing

[2]Right now in Kubernetes, resources relates to either CPU or memory. CPU is requested in cores. Memory is requested in bytes of RAM.

[3]It is also possible to configure Kubernetes such that a container using too much CPU is killed.

reactive and predictive auto-scaling, again because it makes it easier to reason about idle CPU.

In the interest of reducing the amount of length evaluation jobs that we run, perform one test in which we compare average static provisioning, reactive auto-scaling, and predictive auto-scaling for a single test pattern and pod initialization time. Ultimately, we do not devote significant time to considering any type of static provisioning, as we are confident that it will at best be equal to reactive auto-scaling. Thus if our implementation of predictive auto-scaling outperforms reactive auto-scaling, we are confident that it will also outperform any type of static provisioning.

5.3.2 Reactive Auto-scaling

Additionally, previous to this thesis, it was possible to scale applications in Kubernetes using horizontal, reactive pod based auto-scaling. We examine the implementation and utilization of this method of scaling in depth in the *Autoscaling in Kubernetes* section, so we will not repeat it here.

However, some additional detail assists in understanding the potential values of CPU utilization percentage that result from reactive horizontal auto-scaling. Remember that the current implementation of reactive horizontal auto-scaling occurs by having the auto-scaler create sufficient replica pods such that each replica pods operates within a specified range of CPU utilization percentage. [4] If all replica pods initialize and share in the work immediately, than we could expect CPU utilization percentage to consistently stay within a small range of the value the user specified. For example, if the user instructed the auto-scaler to auto-scale such that all pods utilized 70% of available CPU, and the range was ± 2, then we would expect CPU utilization, and thus efficient resource utilization, to stay within a constant range. However, our justification for introducing prediction the horizontal pod auto-scaling is an understanding that replica pods will not always immediately initialize. During this delay while pods' initialize, we expect to see a decrease in quality of service. Establishing reactive horizontal pod auto-scaling as a control group assists us in determining whether the addition of prediction improves upon what existed in Kubernetes before this thesis began.

5.4 Independent Variables

We examine two independent variables, traffic request pattern and pod initialization time, with respect to the different scaling types' performance. In other words, for under-, average-, and over-static, reactive, and predictive auto-scaling, we examine the impacts of varying the request pattern of traffic to our testing application and the impacts of differing pod initialization times. This allows us to determine under what combinations of traffic request patterns and pod initialization time predictive auto-scaling is most effective, and also under what combinations it is the least effective. Furthermore, because we utilize the same independent variables for all of the different scaling types, and because these independent variables are relevant to all scaling types, we can make comparisons across the scaling types. For example, we could determine that predictive auto-scaling outperforms

[4] If we decide to enact resource requestslimits on our replica pods, we can also calculate the exact amount of resources being utilized, as opposed to just a percentage. However, we are not particularly concerned about non-percentage values because we measure efficient resource utilization using CPU utilization percentage, instead of total CPU utilization.

reactive auto-scaling most when pod initialization time is lengthy and the traffic pattern is a simple linear slope, but predictive auto-scaling exhibits very little difference from reactive auto-scaling when pod initialization time is very small and we see a *flash crowd* traffic pattern.

5.4.1 Pod Initialization Time

As will be discussed in detail in the later *Tools* section, we've created a web server application that allows us to specify any *pod initialization time* we desire. As such, we have considerable flexibility with respect to what initialization time values we test. To begin, we decided to test the following values: 5s and 135s.

We believe 5s and 135s are important independent variables to test because they are indicative of different classes of applications that could be run on Kubernetes. A pod initialization time of 5s is commonly found among web application frameworks, as we showed when writing a simple HTTP server in Go and using the Spring Java web application framework [30]. There web frameworks initialized, and were ready to serve requests, in 1s and 5s respectively. It is our strong belief that any web framework that simply needs to initialize, without any external communicating our data, will accomplish this task in under 5s. Thus, testing a pod initialization time of 5s allows us to consider how any web application framework would perform with predictive auto-scaling.

We derive a pod initialization time of 135s from a simulation of downloading a shard database file, loading it into a database, and then starting a web application to process data from said database. More specifically, our sample pod downloaded a 25.2MB file and then batch inserted it into an Elasticsearch database [6]. It then started a web application framework. In all, this process took approximately 135s, although it is easy to imagine the time varying based on the size of the initial shard file. The file we downloaded came from an ElasticSearch tutorial on prepopulating a database [10], and thus we believe it is a fairly representative size. By examining pod initialization times of 5s and 135s, we are confident that we are capturing the majority of the potential use cases for predictive auto-scaling.

We believe that there will be a *sweet spot* of pod initialization times for which predictive auto-scaling demonstrates the most benefits over reactive auto-scaling. Obviously the smaller the pod intialization time value, the lesser the difference between reactive and predictive auto-scaling, as predictive auto-scaling predicts into the future a time closer and closer to the current moment. However, the larger the pod initialization time value, the further into the future we must predict the state of the application. While large pod initialization times have the potential for considerable benefits when using predictive auto-scaling, we can only realize that potential if predictions of future application state are accurate. As the prediction window gets larger and larger, accuracy becomes substantially more difficult to obtain.

5.4.2 Traffic Request Pattern

We are also interested in the impacts of different traffic patterns on scaling performance. As such, we send our test web server application web requests in a variety of patterns and examine which traffic pattern the scaling method handles well, and which traffic pattern the scaling method does

not handle well. Moreover, we also examine scaling methods in relation to each other with respect to different traffic patterns, seeking to answer for which traffic patterns predictive auto-scaling is beneficial and which traffic patterns predictive auto-scaling is detrimental or meaningless.

In this thesis we examine two different traffic patterns that we feel are fairly indicative of the different traffic patterns a web application may face. We entitle these patterns *increase-decrease*, and *flash-crowd*.[5] We describe each of these patterns, and offer a visual representation for each, below. Each pattern runs for one hours and makes a maximum of 20 requests per second. These values were chosen to give space for the lengthier pod initialization times and to keep the network from becoming too congested respectively. Additionally, each traffic pattern sends no requests for the first five minutes to ensure that the Kubernetes pods have had time to initialize by the time the first request is sent.

- **increase-decrease**: The traffic pattern *increase-decrease*, visible in Figure 5.3, represents a web server facing constantly increasing and then constantly decreasing load. This scenario can be seen as representing for example, a restaurant in which people increasingly visit the site as it becomes closer and closer to a meal time, and decreasingly visit the site as the meal time becomes farther and farther away. After five minutes of silence, our load generator builds from sending 0 requests per seconds to 20 requests per second over the course of 30 minutes. After reaching the apex, the traffic generator then reduces from sending 30 requests per seconds to sending 0 requests per second, again over the course of 30 minutes.

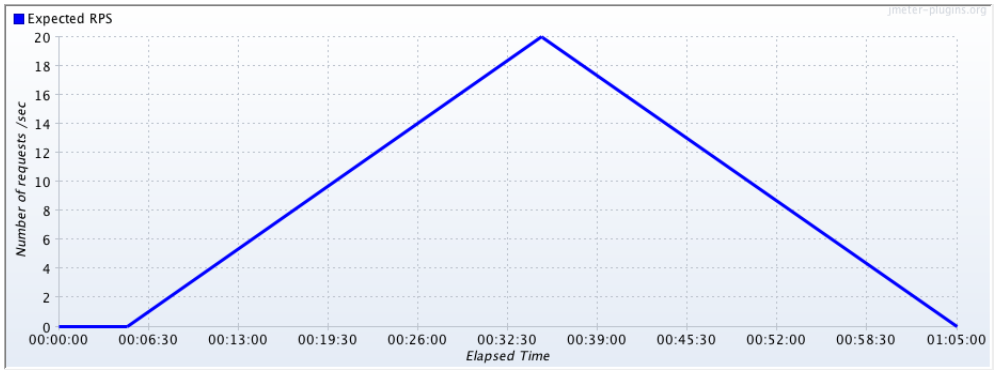

Figure 5.3: The increase-decrease Traffic Pattern

- **flash-crowd**: The traffic pattern *flash-crowd*, visible in Figure 5.4, represents a web server facing a suddenly increasing, and then suddenly decreasing, amount of load. This scenario is indicative of, for example, a newsite which will suddenly receive a short-lived burst of traffic when a major story hits. After five minutes of silence, our load generator slowly builds from sending 0 requests per second to sending 5 requests per second, over the course of 28 minutes. Then, in just 2 minutes, our load generator jumps from sending 5 requests per second to sending 20 requests per second. After reaching the apex, the traffic generator decreases from

[5]There are of course an infinite number of traffic patterns that we could examine, and examining other options is an exciting opportunity for future work.

sending 20 requests per second to sending 5 requests per second, again in just 2 minutes. Finally, our load generator decreases from sending 5 requests per second to 0 requests per second, over the course of 28 minutes.

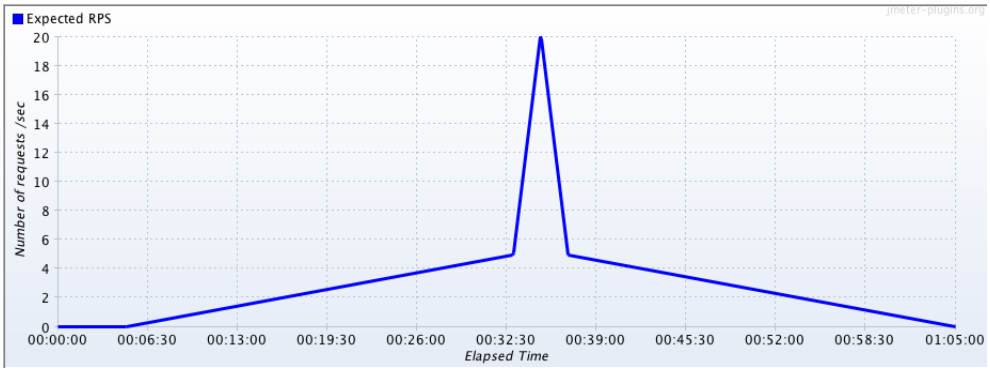

Figure 5.4: The flash-crowd Traffic Pattern

5.5 Methodology

Kubernetes does not currently contain either the tools or processes for performing the evaluation this thesis needs. This section gives an overview of tools and processes we created for testing different types of scaling in Kubernetes.

5.5.1 Tools

Pursuing the evaluation method previously outlined in this thesis requires both the creation of new tools and the utilization of previously existing open-source tools.

test-server

We seek to measure variations in a web server application's efficient resource utilization and quality of service across a variety of scaling methods. Many of the previous sections of this thesis, and the majority of this thesis' contributions to Kubernetes, relate to implementing predictive horizontal auto-scaling as a new scaling method. Yet, to evaluate we must also create a web server application. Ultimately, the created web server application, entitled *test-server*, allows us refined control over the independent variables we wish to control during experimentation.

test-server is a HTTP server written in Go. It defines two API endpoints, "/" and "/ready".

- **"/" - Load**: Traffic generators simulating user requests will send GET requests to the endpoint at "", also known as the *Load* endpoint. This endpoint has two functions. First, it must perform a task that is somewhat CPU intensive so as our application uses enough CPU that auto-scaling may be triggered. Second, it must record the percent of idle CPU and an estimation of request response time as measures of efficient resource utilization and quality

of service respectively. The function assigned to handle the GET request accomplishes these tasks as follows. Upon receiving a GET request to this endpoint, the handler function records the start time of the function and records the starting percentage of CPU that is idle. It then executes a computationally intensive task, which we define as repeatedly encrypting a passphrase. Once this task is over, we once again record the time and the percentage of idle CPU. Having these two values allows us to calculate average amount of idle CPU and function exeuction time, which function as measurements of efficient resource utilization and quality of service. The function then records these values as a new point in the database, along with a an indicator of the values of independent variables in place for this trial (i.e. pod initialization time, scaling method). The function finishes by returning a status code of 200.

- **"/ready" - Ready**: The *Ready* endpoint is used to allow us fine grained control over pod initialization time during experimentation. Kubernetes determines if a pod is initiliazed based on a *ReadinessProbe*. This probe was discussed in greater detail earlier, but suffice to say it works by defining an HTTP endpoint that will return a successful status code if the pod is initialized. During experimentation, we are particularly interested in how predictive auto-scaling is effected by different pod initialization times. Thus, we utilize the following setup. First, when defining our pod, we specify the *ReadinessProbe* should use "/ready" as the HTTP endpoint. The function handling responses for requests to "/ready" works by reading in an environment variable indicating the pod initialization time value we wish to test.[6] The function parses this time, and waits that amount of time before returning a status code of 200. This wait ensures that we can control exactly when the *ReadinessProbe* will receive a successful response, and thus can control exactly the amount of time before a pod initializes.

This application closely fits within the guidelines microservice theory establishes and it is easy to containerize it and then run it on Kubernetes. It is *focused*, as it performs the single task of returning to HTTP requests. It is *stateless*, as all containers communicate with an external database, that is not stored within the container. The container can be deleted and restarted at anytime with no errors or sustained interrupts. Finally, the container can easily be replicated and run concurrently, as a load balancer can distribute discrete requests across multiple replica HTTP servers with no threat of race conditions. It is thus simple to package this webserver into a container, and then package said container into a pod.

As was previously mentioned, the pod containing the container for this application implements a *ReadinessProbe* pointing to the "/ready" endpoint. Additionally, the pod limits and requests an explicit amount of CPU, currently .5 cores, for the container running the application. This ensures measures of efficient resource utilization are consistent across scaling method. Finally, our pod contains a number of environment variables including *SCALING_METHOD* and *INITIALIZATION_TIME* which allow us the fine grained control and measuring previously mentioned. Additionally, regardless of the scaling method, we define a replication controller for ensuring whatever the scaled amount of pods exist, and a service for balancing requests across all *test-server* containers

[6] Any method of running containers, whether locally during testing or in a pod on Kubernetes will allow us to easily set environment variables.

and exposing *test-server* to an external IP address that can receive our generated traffic.

InfluxDb

Our evaluation strategy generates an immense amount of time-series data, which we need to store in a database that we can later query and manipulate. *InfluxDB*, an open source time-series database from InfluxData, is a natural choice for recording such data. InfluxDB makes it easy to collect, store, manage, and visualize the large amount of data that we will generate. InfluxDB provides a simple HTTP API which we use both for recording our data throughout are evaluations, and also for querying and modifying it later. Additionally, it provides a number of integrations into common visualization platforms, which are used to generate many of this thesis' graphs [11]. Finally, InfluxDB has a variety of hosting options, meaning our containerized application does not have to hold any data, and thus remains stateless and concurrent.

JMeter

As mentioned in the discussion of independent variables, we need to generate substantial traffic conforming to a specific pattern. Fortunately there a variety of open-source tools that can assist us in this task. Ultimately, the one selected for this thesis is Apache JMeter [3]. JMeter has considerable functionality, but most appealing is its *Throughput Shaping Timer* plugin [31]. This plugin allows us to use JMeter to send HTTP requests with a vary specific pattern, by specifying exactly how many requests should be sent per second, and for what length of time they should be sent. Using this tool, it is easy to create our previously described patterns of traffic that we seek to test.

We containerize JMeter and place it into a pod which we run on Kubernetes. This gives us considerably more resources with which to generate traffic, as opposed to trying to generate a high volume of network requests from a laptop. Importantly, we run JMeter within a different namespace than *test-server* pods, and also impose CPU limits on JMeter, to ensure that our *test-server* pods are isolated from JMeter's traffic generation regardless of how much traffic JMeter generates.

5.5.2 Evaluation Process

Given the tools created and utilized for evaluation, we must consider how we will utilize these tools in order to create a robust, automated testing environment.

Hosting

For evaluation, we must run *test-server* on a hosted Kubernetes instance. It is necessary to use a hosted Kubernetes instance, instead of just running Kubernetes locally, because we will be sending our pod an amount of traffic too great for any single commodity machine to handle. Additionally, we want to simulate running Kubernetes in as realistic a production environment as possible, and of course all instances of running Kubernetes in production require hosting Kubernets on external cloud servers.

There are a couple of different options for a hosting service which will provide the machines for our Kubernetes cluster. The simplest method of using Kubernetes is to use *Google Container Engine*. Google Container Engine is a verison of Kubernetes hosted by Google itself. The number of machines managed by Kubernetes is completely abstracted, as the user only interacts with the Kubernetes command line tool, *kubectl*. While this method is admired for its simplicity, it is not feasible for this thesis. Because we want to be able to test our modifications to Kubernetes, without waiting for them to be accepted in the stable version of Kubernetes used for Google Container Engine, we must instead use a platform that allows greater control [7].

Fortunately, Kubernetes can be run on a number of cloud providers, including *Google Compute Engine*, *Amazon AWS*, and *Microsoft Azure* [7]. The Kubernetes source code provides a number of simple scripts for configuring one of these providers to run Kubernetes. Importantly, the version of Kubernetes running on these providers can be any version we desire, meaning that we can test our modified version that incorporates predictive auto-scaling, even if our updates are not yet merged into a stable Kubernetes master version. Because of previous development experience, we decided to pursue hosting on Amazon AWS. Kubernetes typically runs 1 *m3.medium* EC2 instance as the master and 4 *t2.micro* instances as workers, all running in the *us-west-2a* region. These defaults make sense for the workload we expect [29].

@TODO Document how we are going to send requests (they probably shouldn't all be sent from my local computer right?)

Additionally, for simplicity's sake, we decided to host the InfluxDB database instance used for storing our evaluation data. It would have also been possible to run an instance of InfluxDB ourselves on an Amazon EC2 machine, but the potential cost benefit did not message the time and complexity costs. Because all of the data being stored is small key/value pairs, we only use a 10GB Storage, 1GB RAM, 1 Core machine [12].

K8s Configuration

Additionally, we need a method for configuring *test-server* to incorporate the different variables that we wish to test. Specifically, we need a way to ensure that *test-server* can be run on a Kubernetes cluster utilizing a variety of different methods for scaling, and also that we can control the amount of time it takes for a pod to initialize. In addition, we need *test-server* to know the exact values of its independent variables so that it can record them to the database, ensuring all data is properly labelled. Our *test-server* application reads all of these dynamic values from Unix *environment* variables.

It is possible to utilize Kubernete's configuration language to work with this method of controlling independent variable values through environment variables. We place our containerized *test-server* application within a pod, and Kubernetes allows the specification of environment variables within a pod. The only issue is that these environment variables in the pod configuration file must be static. We solve this issue by creating a template of our pod configuration file, with indications of the dynamic environment variables. We can then run a custom python that reads in configuration values, and creates distinct configuration files incorporating each of these values. Thus, each different

file specifies a different pod configuration. Utilizing the proper set of independent variables for a pod is as simple as creating a pod from the correct configuration file. This entire process has been automated, meaning this implementation detail has been largely abstracted.

Running Tests

Given the powerful tooling described in the previous section, the process for running a set of tests is completely automatable and quite simple. Each test must specify a traffic pattern, a scaling method, and a pod initialization time. These variables influence Kubernetes' *ReplicationController* and *HorizontalPodAutoscaler* objects. Thus, while some configuration files for Kubernetes test objects are consistent between tests, the configuration files for the aforementioned varying *ReplicationControllers* and *HorizontalPodAutoscalers* must be selected based on which test we are running.

We do this selection using environment variables, which allow us to run the tests with the following single *make* command.

```
export TS_RC=test-server-controller-reactive-5s.yaml;
export TG_RC=traffic-generator-increase-decrease-test-plan.yaml; export HPA=TRUE; make
test_start
```

The above command indicates a wish to start a test instance with reactive auto-scaling, a five second pod initialization time, and an *increase-decrease* traffic pattern. It also rebuilds all containerized applications and ensures the configuration files are up to date.

Once complete, all of the pods, replication controllers, services, and autoscalers on Kubernetes can be destroyed with the following *make* command.

```
make test_stop
```

As such, running a single test requires very little human involvement. The only task is monitoring the tests to ensure they are no errors. Particularly, we are concerned about writing our metrics to the database failing, Kubernetes running out of space to schedule replica pods, and potential errors in our predictive auto-scaling implementation. Fortunately, either Kubernetes provides, or we have implemented, methods for highlighting such errors and making the necessary adjustments to the evaluation process. In addition, after all the tests run, we are able to determine what percent of requests to *test-server* were successful, and take action accordingly.

Interpreting Results

Just as we have automated the process for running our evaluation tests, we have also automated the process for interpreting the results from these tests. As all of the results from our evaluation tests are stored in InfluxDB, we need to write a script that retrieves these results in aggregated 1 minute intervals, sums eru and qos for each observation, and generates graphs and summary statistics comparing the difference in the summation of eru and qos for predictive and reactive auto-scaling. We run this same script for all 4 combinations of traffic pattern and pod initialization time that we are interested in. A more in-depth discussion of the analysis performed and the combinations of traffic pattern and pod initialization time that we wish to test will occur in the Results section.

		Pod Initialization Time	
		5s	135s
Traffic Pattern	increase-decrease	(5s, i-d)	(135s, i-d)
	flash-crowd	(5s, f-c)	(135s, f-c)

Table 5.1: The Test Matrix

5.6 Results

With our test information generated, and the architecture for processing our test information in place, we can now interpret our results to determine the impact of predictive auto-scaling.

5.6.1 Impact of Predictive Auto-scaling

As has been discussed in the previous sections, we are interested in visualizing and understanding the difference in performance between predictive and horizontal auto-scaling for two different pod initialization times and two different traffic patterns. As such, we have four different tests on which we compare ERU and QOS: increase-decrease traffic pattern with 5s pod initialization time, increase-decrese traffic pattern with 135s pod initialization time, flash-crowd traffic pattern with 5s pod initialization time, and flash-crowd traffic pattern with 135s pod initialization time. This grouping of tests can be visualized through the matrix seen in Table 5.1.

For each test on the matrix, we provide two different sources of information. First, we generate a graph comparing the summation of eru and qos across the 60 minute evaluation time for predictive and reactive auto-scaling. Additionally, we provide statistical measurements for the difference of predictive and reactive auto-scaling at the same point in the evaluation sequence (i.e. we compare the summation of eru and qos after 10 minutes for predictive with the summation of eru and qos after 10 minutes for predictive). We process all of the observations that we recorded in 1 minute intervals, for which we calcualte the mean eru and qos observations. With respect to statistical measurements, we calculate a one-sided p-value based on the null hypothesis that difference between the summation of eru and qos for predictive and reactive auto-scaling is 0. As we are interested in seeing if predictive auto-scaling performs better than reactive auto-scaling, we calculate a one-sided p-value, p, with the alternative hypothesis that the difference between the summation of eru and qos for predictive and reactive auto-scaling is greater than 0. We test for significance at the 5% significance level, meaning that if $p < 0.05$, we can reject our null hypothesis in favor of our alternative hypothesis that predictive auto-scaling performs better than reactive auto-scaling.

5s and increase-decrease

135s and increase-decrease

5s and flash-crowd

135s and flash-crowd

5.6.2 Scenarios

Pro Predictive Auto-scaling Scenarios

Anti Predictive Auto-scaling Scenarios

5.7 Implications

5.8 Summary

Chapter 6

Conclusion

6.1 Future Work

6.2 Summary of Contributions

Bibliography

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

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

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

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

[5] Automatic ballooning. http://www.linux-kvm.org/page/Projects/auto-ballooning.

[6] Elasticsearch. https://www.elastic.co/.

[7] Getting started: Get started running, deploying, and using kubernetes. http://kubernetes.io/docs/getting-started-guides/.

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

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

[10] Import some data. https://www.elastic.co/guide/en/kibana/3.0/import-some-data.html.

[11] Influx data. https://influxdata.com/.

[12] Influxdb pricing. https://customers.influxdb.com/pricing.

[13] Kubernetes compute resources. http://kubernetes.io/v1.1/docs/user-guide/compute-resources.html.

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

[15] Kubernetes horizontal pod autoscaler object. https://github.com/kubernetes/kubernetes/blob/release-1.2/docs/design/horizontal-pod-autoscaler.md.

[16] Kubernetes horizontal pod autoscaler proposal. http://kubernetes.io/v1.1/docs/design/horizontal-pod-autoscaler.html.

[17] Kubernetes horizontal pod autoscaler user guide. http://kubernetes.io/v1.1/docs/user-guide/horizontal-pod-autoscaler.html.

[18] Kubernetes kubectl annotate. https://cloud.google.com/container-engine/docs/kubectl/annotate.

[19] Kubernetes limit range. http://kubernetes.io/v1.1/docs/admin/limitrange/README.html.

[20] Kubernetes pod states. http://kubernetes.io/docs/user-guide/pod-states/.

[21] Kubernetes pods documentation. http://kubernetes.io/v1.0/docs/user-guide/pods.html.

[22] Kubernetes replication controllers. http://kubernetes.io/v1.0/docs/user-guide/replication-controller.html.

[23] Kubernetes services. http://kubernetes.io/v1.0/docs/user-guide/services.html.

[24] Kubernetes v1 released - cloud native computing foundation to drive container innovation. http://googlecloudplatform.blogspot.com/2015/07/Kubernetes-V1-Released.html.

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

[26] Kubernetes working with containers. http://kubernetes.io/docs/user-guide/production-pods/.

[27] Linear regression tutorial for biology 231. http://www.radford.edu/ rsheehy/-Gen_flash/Tutorials/Linear_Regression/reg-tut.htm.

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

[29] Running kubernetes on aws ec2. http://kubernetes.io/docs/getting-started-guides/aws/.

[30] Spring by pivotal. https://spring.io/.

[31] Throughput shaping timer. http://jmeter-plugins.org/wiki/ThroughputShapingTimer/.

[32] What is kubernetes? http://kubernetes.io/v1.1/docs/whatisk8s.html.

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

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

[35] 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.

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

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

[38] 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.

[39] 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.

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

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

[42] 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.

[43] Matthias, K., and Kane, S. *Docker Up & Running: Shipping Reliable Containers in Production.* O'Reilly Media, Inc., California, USA, 2015.

[44] Newman, S. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media, Inc., California, USA, 2015.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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).

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