Temporary Thesis

**Introduction**

Starting a web-based company is a very risky endeavor. Recent statistics place the startup failure rate above 90% for varying reasons [1]. This figure should be frightening to many who are trying to begin their own business, especially since the cost of hardware for such an endeavor has traditionally resided in the thousands of dollars before the service can even launch. This, however, is the problem that the Cloud market specifically aims to rectify.

The Cloud, as it is today, is a collection of hardware resources which are owned by several large, established companies and rented out for use by other companies. This market takes many forms, such Google’s Compute Engine and Amazon AWS, but all essentially the same: users who wish to use these resources as a platform for their own service, or for whatever needs they may have, pay for the resources they want and are charged by the amount of time they use these resources. As with any resource someone wishes to buy, when they rent this hardware they are guaranteed to receive it and it cannot be taken away from them until they give it up freely. However, since users of Cloud services ask for the exact resource which they want, the providers of said services have very little control over which of their own servers they need to turn on and how much of these servers is actually being utilized for a given user.

For example, consider the case where only one user among thousands asks Google for their highest speed server type, but they only wish to utilize 10% of this server. Google must now turn on this server, and while it is only at 10% utilization, its power consumption is very high comparatively, most likely around 50% [7]. This leads Google to a situation where they are paying more to sell this resource than they are to receiving from its use, or at very least a situation where they are able to make more efficient use out of a product at almost no cost to them. Seeing this very clear issue, Amazon came up with a solution they called Amazon EC2 Spot Instances [2]. Spot Instances are Cloud computing resources that make up the unutilized space on currently active Amazon servers and are sold to the highest bidders attempting to utilize this resource for less than the standard rate of a Cloud resource. The catch to these is that they are not guaranteed as available to any user for any amount of time and thus can be taken away as soon as someone is willing to pay more for the resource or when the server is no longer needed for on-demand customers.

These spot instances, and their equivalents at other companies such as Google, have provided a unique opportunity. Companies seeking to minimize costs have the option to “take a chance” on these unreliable resources and utilize them as opposed to on-demand servers at full price. While on the surface this seems like a risky endeavor, especially for new companies who can’t risk intermittent service outages while trying to establish themselves, there would appear to be ways to avoid losing service while still benefitting from these affordable prices. **The problem we now face is how do we implement a service with 99.99% reliability for the minimum cost available?**

I believe the solution to this is to utilize pre-emptible VMs or spot instances in such a manner that their inherently failure-prone nature is hidden by having more available at a given time. By having more nodes available, you reduce the risk that some are taken away from you at any given time, as the traffic that was once there is just placed on the other available nodes. Currently, Google’s pre-emptible VMs cost 40% of their typical per hour costs for their smallest, most basic compute instance [6]. Given this, we know that if we can create a service that utilizes this type of instance, we can afford two and a half pre-emptible instances for every one on-demand instance we would need. That means that if we can implement a 99.99% reliable service using less than five pre-emptible instances for a service that would take two reliable instances, we have saved money.

The real solution will not be quite so simple though. The numbers and pricing above relate only to the smallest available standard compute engine instance. The final solution will first determine how big an instance will need to be to handle the pods we have designed. This instance size will be considered the baseline. Additionally, depending on the variety of pods needed for the entire service, there may be two or more “baseline” instance sizes to consider, considering that pods will be performing different functions and each type may require a different amount of computing resources. After the baselines are determined, we will need to determine how many on-demand baseline instances are needed to provide 99.99% reliability to some arbitrary **(SHOULD THIS NOT BE ARBITRARY?)** number of users. We will attempt to determine the lowest price possible for on demand instances by utilizing bigger instances with more pods (i.e. creating two pods on an instance that is two times the size of our baseline instance) until we are satisfied that we have found the lowest on-demand price possible.

After this testing is complete for the on-demand instances, we will begin working with pre-emptible VMs. We will have our maximum cost allowable for Google Compute Engine, and from here we will attempt the lowest cost possibilities using pre-emptible VMs. Determining what these will be should be simple, as Google has guaranteed pricing on their pre-emptible instances. Unlike Amazon Spot Instances, these costs will always remain the same, so determining the lowest cost possible after having determined the computing power needed will be a matter of simple math. However, we will then need to implement this and determine the reliability and revocation rate of the types of servers we have chosen. Unlike Spot Instances, pre-emptible VMs are still very new and lack the revocation history data needed to make accurate predictions about which instances are likely to fail before it happens. This means we will need to take reliability data as we test. As we gather reliability data, we can then make models to determine how many extra of every instance type should be made in order to keep service at 99.99% reliability. Once we have found the lowest cost solution as a function of per hour cost and reliability of server types, we will finally be able to determine if the pre-emptible VM instances have offered us the lowest possible cost solution on the Google Compute Engine.

To demonstrate this, I intend to create a basic web service and implement it on the cloud using Amazon Spot Instances or Google Pre-emptible Virtual Machines, which is Google’s unguaranteed cloud service created in response to Amazon Spot Instances. This decision is being influenced by numerous factors. Amazon has a much more established spot market, providing very useful past data about their various spot markets that can be used to accurately predict the likely future prices and make the most informed purchasing decisions. However, the Amazon Spot Market is far more complex and utilizes a bidding system which adds to the difficulty in making a perfect algorithm. Google’s Pre-emptible VMs offer a fixed price and a much more fluid integration with Kubernetes, a service that will be utilized to manage server instances. They are also, however, newer and lack the history data of revocations that would be useful in making confident purchasing decisions and they put an absolute limit of 24 hours on any pre-emptible instance they sell. One major factor in deciding is determining how well Kubernetes will be able to manage Amazon Spot Instances across many markets simultaneously. Some claim that “Kubernetes was designed to make working with containers on Google Compute Engine easier” [3]. Using the Google Cloud was the intention for Kubernetes very early on, and it makes sense that this fact would make Google the much more intuitive choice for service.

This web service will be managed by a 2-level management algorithm. The first level acts as a traffic analyzer to estimate the current load on each of the available servers and determine when it is necessary to obtain more resources to maintain reliability and when it is acceptable to drop superfluous resources. When it is decided that the number of resources must change, it advises the second level to respond accordingly. The second level management is the implementation of Kubernetes to manage the Cloud resources. Kubernetes is a very new, open-source project managed by Google which is used to manage server resources across many servers in as efficient a manner as possible.

Kubernetes uses a Master-Minion structure to manage resources, meaning we divide up our resources into a master node and minion nodes. In this case, a node refers to an amount of computing resources needed for something. There is only ever one Master node and it resides on a physical or virtual resource that should never be revocable. The Master node contains the main instance of Kubernetes as well as the top level of our management system in our case. The minions, however, can reside on revocable, lossy instances and in our case this will be the vast majority of them. It should be noted that there is nothing preventing the creation of one or more minions on the Master node’s instance as long as the necessary resources are available. The Master node controls all instances of minion nodes in various ways described in the following pages and decides which minions are to be kept, which are to be disposed of, and when more minions need to be activated.

Kubernetes utilizes a Replicated State Machine approach to providing web services. This means that Kubernetes creates a certain number of copies of a service and directs users wishing to access the service to the least utilized instance of the service. The number of copies of a service is called the desired state and is a setting within Kubernetes. When something happens to change the number of available services, such as one instance of the cloud that was available to Kubernetes previously failing or being revoked, Kubernetes will see some pod has failed and will bring an identical one up in its place. To create an identical pod, Kubernetes references the label attached to the pod that has failed and accesses a template of that pod to create the next instance. The desired state is the parameter being managed by the first level of management, which means Kubernetes will bring up and close down resources as it is instructed by the first level of management. All of this management of nodes occurs on the Master node and all of the resources managed are minion nodes.

The use of the desired state and template files to manage resources makes Kubernetes ideal for integration into a cloud based service utilizing unstable instances. It is a system designed entirely to deal with faults that happen and take resources away from a system. When utilizing lossy instances such as Google Pre-emptible VMs as we are in this study, a system with fault tolerance built in is not only beneficial, but essential. We will, however, need to perform tests to determine how quickly Kubernetes can react to the different situations that occur and cause faults.

Faults can be caused by a number of sources. The first reason is the failure of the server, or physical resource, on which the service is based. There are numerous specific reasons for this type of fault occurrence. It can happen as a result of power loss or a power surge at the site where the server is held. Some physical trauma to the server itself could cause it to fail. Or simply by being forced to shut down or restart would also cause this issue. The second reason for server failure is caused by resource issues on the server. This can be caused by software draining too many resources, either intentionally or accidentally, and causing the system to become overloaded. The system can also become overloaded by too many requests for service, which could also be unintentional or a malicious attack. In addition to these, our system has an additional fault cause, which is the possibility of losing VM instances at any time. Unlike the others, this is almost sure to be a very consistent issue that will have to be dealt with constantly and quickly, whereas the others are generally rare.

Using Kubernetes and the Google Compute Engine together can help address many of these faults as discussed previously. As discussed before, Kubernetes will always be regulating the number of servers active to make sure it is the desired number to meet our availability goals. However, the fact that our system is on the cloud means that the VM instances are usually separated onto different machines in different areas, diffusing the impact of power outages and environment related issues. Our traffic analyzer coupled with Kubernetes desired state and round-robin traffic distribution will keep heavy-traffic interruptions from happening. DDOS attacks to cause intentional heavy traffic to our system should be dealt with in a basic securities check.

Kubernetes creates containers, which are sealed application packages that exist as seemingly separated portions of a server resource. Essentially, these containers are seemingly independent instances of computing resources (like separate VM instances) that are managed as such, but which are actually created by dividing a larger machine up so that you can make more efficient use out of this machine. Containers and Virtual Machines are very similar in their respective uses, however there a benefits to each. A Virtual Machine has the benefit of flexibility. By utilizing a Virtual Machine, one could choose to use any Operating System they wanted to use regardless of that of the host. The benefit to using Containers, however, is that they utilize the Operating System of their host, thus negating the overhead and resource waste created by implementing a separate Operating System on top of another Operating System. By using Containers and minimizing or eliminating this waste of computer resources, we reduce the amount of computing resources we require drastically and can save on operating costs. The only requirement to make use of Containers in the cloud is that you must be able to create your service in the type of environment provided by the cloud service providers, but as they offer very typical Operating System choices this is easily done in most cases.

Kubernetes then organizes these containers into pods, which are groups of containers that work together to achieve some goal. Pods are the smallest deployable units as viewed by Kubernetes [5]. They can contain from one to any number of containers, however all containers within a pod need to exist on the same physical or virtual host. Pods are also seen as “immutable”, meaning that once a pod has been created by Kubernetes it remains unchanged for its lifetime. Kubernetes creates pods based on need. It uses a replication controller and its desired state value to determine how many pods should exist any time. If this desired state is not meant, the number of active pods is either increased or decreased. Every pod has a template that it is created from, so in the case that Kubernetes must create more pods, like at startup or when traffic increases enough, it refers to the template of a pod that is needed and creates a pod from that template.

Kubernetes also has something known as a Service to manage Pods. There is an important abstraction between the service we are trying to create and what is meant when referring to a Kubernetes Service. Our service will be a simple web application of some type. Right now, the intention is to create one with two main components: a front end user interface with minimal complexity, and a backend that supports the front end and likely has a database for some yet to be specified use.

A Kubernetes service is simply another component of the Kubernetes system. The Kubernetes service serves to crucial roles: to act as a staple, fixed component that is not as disposable as individual pods, and to provide load balancing for the pods we have available. A Kubernetes service exists on the master node, and thus it should never be in danger of failing as our master node should never be lost. Each service targets a specific type of pod with a specific label and a specific port. The service has an IP address given to it, referred to as the “cluster IP” [8]. Requests for the desired functionality of the pods managed by a service are received by the service at this IP and then distributed to the pods in a round robin fashion to keep pods under equal load [9].

**It is my hypothesis that by gauging the traffic of a service built on unreliable cloud resources we can manage the amount of cloud instances needed to provide 99.99% reliability of a service for the minimum cost available.**

As mentioned previously, the intended method to test this is to create a web service with an internet accessible front end. Using this method what we will have is a stateless service. A stateless service is one whose performance does not rely on holding the specific state of a machine at any given time and being able to return to this state in the event of a failure. Our web service will not depend on state at any given time, as there is no state to return to. A state in this context refers specifically to holding onto session information that would be very important to the user’s experience. For instance, a state could simply refer to holding onto the user’s login information in an application that requires a login. If we need to change server instances after losing a server, it would be very annoying to a user who had to log back in when we transitioned to a new server.

This type of a service would be referred to as stateful. As the name implies, it would be very important to somehow hold onto current state information of all current instances of an application or service. Kubernetes would also be useful in this type of service to the same extent that it is useful for our own needs. The difference is a method of storing state information in a way such that it would not be lost when an instance is lost would be required. The reason a stateless service rather than a stateful one has been selected is simplicity. The tests we run are going to focus entirely on the availability of a service we create on unreliable instances and Kubernetes speed in bringing instances back up afterwards. While it would be interesting to see the time added to recreate stateful instances, it is not a worthwhile endeavor in this case and would not help in solving the question of whether we can create a service for less money and the same availability using unreliable resources.

It should also be noted that Kubernetes is useful for batch processing as well as web services. Batch processing is a method of distributing a large task, or many smaller tasks that have been allowed to accumulate, across an adequate number of computing resources so that the task(s) can be handled quickly and at the same time. This is a sort of stateful service, where the progress made on the batch and the results must be noted as work is being performed on them. That way, when an instance is lost, Kubernetes is able to bring up one that begins at the point where the other left off. This would also be an option for our testing, however it has the issue addressed before of all stateful services: it adds a level of complexity to our design that is unnecessary and adds nothing to our goals.

The choice of 99.99% availability equates to a decision that it is simply an acceptable amount of downtime for a web service to have. In a year, a 99.99% available service will have less than an hour of unexpected downtime. That means in a single day, a system can be expected to be down for about eight and a half seconds. Out of an entire day, that is an unnoticeable amount and could easily be attributed to latency or a spike in internet activity temporarily, even if occurring all at once. If 99.99% proves to be too lofty of a goal, 99.9% also seems to be an acceptable downtime rate. It equates to just under one and a half minutes of downtime per day. While this is a large amount to happen all at once, if it is distributed fairly evenly throughout the day, it would show signs of a slower service, but still within an acceptable threshold of user experience.

**Algorithm:**

To solve the issue before us, we need to come up with a sufficient algorithm. This algorithm must be able to:

1. Edit Kubernetes configuration files to change settings
2. Direct Kubernetes in which types of instances to obtain
3. Direct Kubernetes in which spot markets to choose (Amazon)
4. Analyze traffic data in order to determine how many instances are needed
5. Review past spot market data about revocation rates in order to make wise purchasing decisions

[1] <https://s3.amazonaws.com/startupcompass-public/StartupGenomeReport2_Why_Startups_Fail_v2.pdf>

[2] <https://aws.amazon.com/ec2/spot/>

[3] <https://www.ctl.io/developers/blog/post/what-is-kubernetes-and-how-to-use-it/>

[4] <https://cloud.google.com/container-engine/docs/pods/>

[5] <https://docs.openshift.com/enterprise/3.0/architecture/core_concepts/pods_and_services.html>

[6] <https://cloud.google.com/compute/pricing>

[7]<http://www.dell.com/downloads/global/products/pedge/en/BladePowerStudyWhitePaper_08112010_final.pdf> (Shows power usage of its servers on page 10, figure 4)

[8] <https://github.com/kubernetes/kubernetes/blob/master/docs/user-guide/services.md>

[9] <http://kubernetes.io/v1.0/docs/user-guide/services.html#why-not-use-round-robin-dns> (round robin distribution)

[10]