Cost Optimization Through Unifying Multi-Cloud Resources

A Major Qualifying Project Report:
Submitted to the Faculty of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
By

Synella Gonzales

Synella Gonzales

[illegible]

Elsa Luthi

[illegible]

Benjamin Nickerson

Date: January 22, 2019

Approved:

Dr. Tian Guo, Major Advisor

Abstract

The goal of this project is to create a multi-cloud web interface that provides users with the cheapest resource provisioning options from Amazon Web Services and Google Cloud Platform. The user can choose between predefined allocations based on workloads or specify a custom amount of resources needed. In addition, our application handles deployments to respective cloud providers. By handling the end-to-end functionality of finding cloud resources and managing deployments, the user is able to optimize costs from multiple providers.

Executive Summary

It is predicted that by 2020, 87% of enterprise workloads will be hosted on the cloud [1]. Migration is due to the fact that cloud technologies allow businesses to economically distribute their computing needs and data storage. Ability to “rent” the kind of computing power or storage space offered by top cloud service providers like Amazon Web Services (AWS) or Google Cloud Platform (GCP) at such a cheap rate is beneficial for individual users. Cloud service providers remove the bottleneck of individuals needing to meet the hardware requirements of their workloads. For example, an individual might need to have enough storage space for their data or have a processor powerful enough to perform their computations.

However, the process of deploying cloud resources can be tedious and overwhelming for users with no prior experience with cloud services. For example, there are a variety of different cloud providers each offering different pricing options for each resource. In addition, it can be challenging to know the optimal amount of resources needed for specific workloads.

The goal of this project is to create a platform that will optimize this process. We built a web application that allows the user to enter information about their workload. Then we developed an algorithm that optimizes the cost of their instance creation across two cloud providers: AWS and GCP. This work will streamline user’s work, removing numerous manual steps for choosing instance types and sizes, and simplifying the deployment process.

Through an extensive testing suite, we were able to gain valuable insight on the performance of our system. First, we confirmed the correctness of our *find_instance* and *find_instance_workload* functions. We also found that AWS provides more storage options compared to GCP. Furthermore, caching the list of instances and images from both cloud providers improved the runtime of our pricing algorithm. Lastly, AWS was faster than GCP in deploying cloud resources.

There are a variety of engineering efforts and system algorithm improvements that can be made to our application. An engineering effort we recommend is collecting user credentials to the respective cloud provider in order to deploy resources to their specific account. Possible implementations include prompting user for credentials to cloud providers, using a token system, or instructing users how to create private keys with AWS and GCP. A system algorithm improvement we recommend is integrating machine learning to our cost optimization algorithms. Implementing machine learning models can be difficult and time consuming, therefore we suggest it as a direction for future work.

In summary, we successfully created a multi-cloud prototype that achieved the goal of recommending the best price to the user. Our system provides a strong basis for future development. We hope that the system can be improved and utilized by users looking to find the best options for deploying to the cloud.

Table of Contents

Abstract 2

Executive Summary 2

Table of Contents 3

Table of Figures **5**

Table of Tables **6**

Chapter 1: Background **7**

Chapter 2: Literature Review **13**

Chapter 3: Implementation **24**

I. System Overview 24

II. System Software Architecture 26

III. Interface Detailed Design 28

Chapter 4: Results 30

Chapter 5: Recommendations and Future Work 50

References 50

Appendix A 54

Appendix B 59

Appendix C **68**

Appendix D **68**

Appendix F **71**

Table of Figures

Figure 1: Hardware Model for Virtual Machines [5] 8
Figure 2: Diagram showing the difference between Virtual Machines (VMs) and Containers [7] 9
Figure 3: The Top Four Cloud Deployment Model Properties [10] 11
Figure 4: Interface for Cisco CloudCenter, a current multi-cloud deployment platform [24] 22
Figure 5: High-level architecture overview for our multi-cloud interface 24
Figure 6: Flow of information for our multi-cloud solution 25
Figure 7: Home Page 28
Figure 8: Custom Options Page 28
Figure 9: Workload-Based Options Page 29
Figure 10: Top Options Page 29
Figure 11: Deployment Page 30
Figure 12: Launching an AWS Deep Learning AMI using our platform 33
Figure 13: Caching vs. Non-Caching Runtime 40
Figure 14: Runtime Metrics for Deployments 42
Figure 15: Total Deployment and Initialization Time 44
Figure 16: Full Deployment and Initialization Time 44
Figure 17: Analysis from AWS after running the Tensorflow and Keras project on the platform-recommended image 45
Figure 18: Launching an AWS Deep Learning AMI using AWS' dashboards 46
Figure 19: Step 3. Look up Amazon Marketplace Pricing Information 47
Figure 20: AWS analysis of TensorFlow and Keras project running on the recommended t2.micro instance 47

Table of Tables

Table 1: Levels of Abstraction 11
Table 2: Cost Models 13
Table 3: Utility Based Pricing Models 14
Table 4: Service Based Pricing Models 14
Table 5: AWS Instance offerings for In-memory workloads [30] 18
Table 6: Google Cloud Engine offerings for in-memory workloads [31] 18
Table 7: AWS Instance offerings for machine and deep learning workloads [30] 18
Table 8: Google Cloud Engine offerings for machine and deep learning workloads [31] 18
Table 9: AWS Instance offerings for general purpose workloads [30] 20
Table 10: Google Cloud Engine offerings for general purpose workloads [31] 20
Table 11: Correctness tests for find_instance function based on user input 31
Table 12: Caching vs Non-Caching Template 32
Table 13: Deployment Testing Template 33
Table 14: First Iteration of Correctness Tests 35
Table 15: Results of Second Iteration of find_instance 36
Table 16: Results of find_instance_workload 37
Table 17: Test Results for Caching 38
Table 18: Test Results for Non-Caching 39
Table 19: Deployment Runtime of AWS and GCP 41
Table 20: Deployment and Initialization Tests for AWS 43
Table 21: Amazon's Recommendation vs. Platform Recommendation 48

Chapter 1: Background

I. Introduction

Computing resources are in high demand due to computationally expensive workloads. Streaming, the Internet of Things (IoT), machine learning, and artificial intelligence are quickly becoming ingrained into everyday life. Managing substantial workloads raises challenges for developers. Cloud computing offers a variety of tools and services to help alleviate issues that developers face when creating and managing their applications.

Businesses are turning towards cloud computing to host applications. It is predicted that by 2020, 87% of enterprise workloads will be hosted on the cloud [1]. However, cloud solutions raise additional challenges. Vendor lock-ins, resource management, and cost optimization remain a critical concern for maintaining cloud-based solutions [1]. Before addressing the goal of our project, it is necessary to gain a basic understanding of cloud concepts and technologies.

II. Definition of Key Terms

Cloud Computing

Cloud computing allows convenient and on-demand network access to shared computing resources [2]. These resources can be both provisioned and deployed with little effort or interaction with the cloud service provider. According to the National Institute of Standards and Technology (NIST), there are five key benefits of cloud computing: on-demand self-service, broad network access, resource pooling, elasticity, and measured quality of service.

Service Level Agreement

The pricing of cloud computing is based on Service Level Agreements (SLAs) that are arranged by the producer and consumer. In cloud computing, a large part of this agreement is the quality of service that the producer provides. The SLA is usually comprised of an assurance, time period, scope, guarantee, recognition, and a way measure any violations [3].

Virtualization

Virtualization is a key component of cloud computing, especially when resource pooling. It reduces the amount of hardware needed by allowing multiple consumers to utilize an application hosted by a single machine. When software replaces hardware, the efficiency of delegating and consolidating resources is increased. There are a few different forms of virtualization: operating system (OS), platform, storage, network, and application [3]. Application virtualization is the focus of our project.

Virtual Machine

A Virtual Machine (VM) is a computer file, typically called an image, that acts like a physical computer [4]. Virtual machines run like any other program and give the user the same experience the actual machine. For example, a Windows user can have a Linux virtual machine installed on their hard drive that allows the user to run Linux programs on their physical Windows machine.

One of the key features of a virtual machine is that it is completely isolated. This means that it is unaffected by anything happening on the host operating system, and in turn it has no effects on the OS. Multiple virtual machines can be used on the same computer to do different tasks; one for a testing environment, and another for development. Virtual machines are run and managed by a piece of software that sits between the virtual machine and the host operating system called a hypervisor [5]. The relationship between the computer hardware, operating machine, hypervisor, and virtual machines is shown below in *Figure 1*.

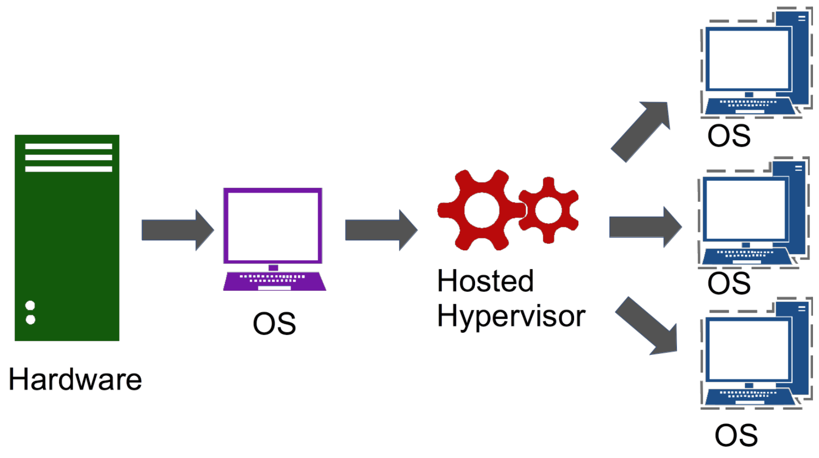


Figure 1: Hardware Model for Virtual Machines [5]

Containers

Containers are designed to virtualize a single application [6]. Containers create an isolation boundary at the application level rather than at the server level. Isolation means that events in one container only affect that container, and they do not impact the virtual machine or the server. Containers can also remove potential compatibility problems between applications that reside on the same operating system.

Management of containers is a common issue. The size of the system is a critical concern for updating containers. Larger systems may require multiple container configurations which make updating a container difficult. However, container management systems such as Docker, Kubernetes, and Fragate provide services to alleviate these issues. Management services provide methods for deploying and maintaining containers. For example, Docker provides two ways to deploy a container: create an image to run in a container or download a pre-created image. Similar to virtual machines, container utilization raises new challenges for developers.

Container security is a concern for end-users. In order to understand it thoroughly we looked at a case study to about security concerns related to Docker. Previously, Docker containers had to have root access. This was incredibly unsafe because if the container was to become compromised, it would have root access to the entire OS. Now, Docker containers utilize user namespaces, allowing the containers to run as specific users [6].

The integrity of the images used to deploy containers is another challenge for developers. Container images that are downloaded from third parties are not guaranteed to be secure. As a result, Docker has tried to remedy this by adding a feature called Docker Content Trust [6], where images are verified and scanned for vulnerabilities.

The key difference between containers and virtual machines is that containers are isolated in their deployment. As shown below in *Figure 2*, virtual machines do not need a full operating system to be installed with the container. They can operate with only the resources they need to perform the task they were designed for (libraries, software, etc.). Containers are also portable; once the container is created, it can be deployed to different servers easily.

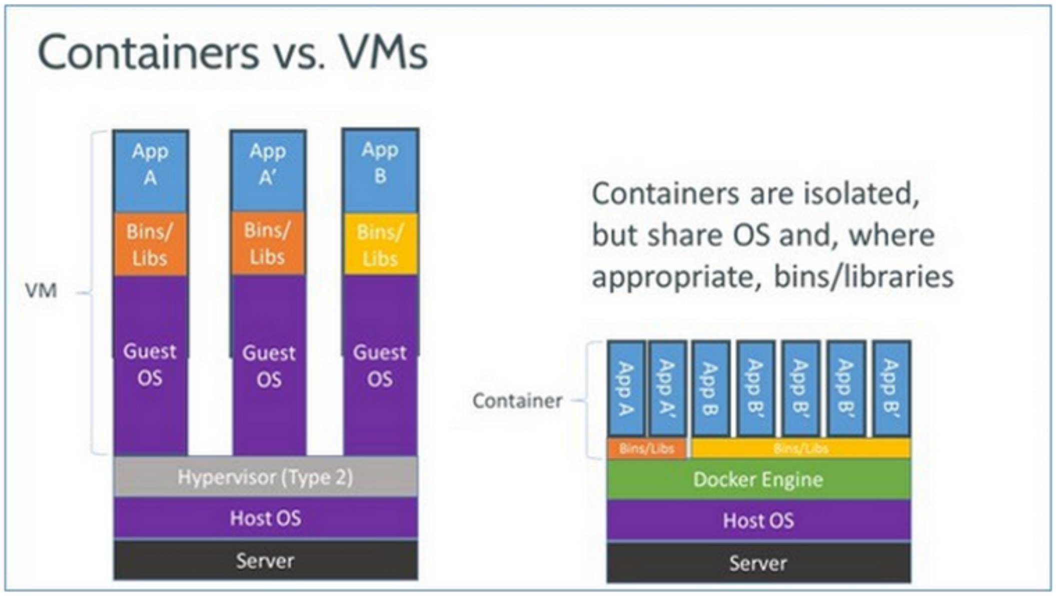


Figure 2: Diagram showing the difference between Virtual Machines (VMs) and Containers [7]

Workload

A workload is a computing task that is completed in a given amount of time [8]. Cloud computing offers a variety of resources optimized for different types of workloads. Common cloud workloads include large streaming, in-memory database, big data storage, and machine and deep learning workloads. Comparisons of workloads can be found in Chapter 2.

Deployment

Deployment is the act of making a resource available on a cloud platform. To choose which platform gets to use a particular resource, it is necessary to analyze a cloud deployment model. A cloud deployment model shows how the cloud obtains and manages its resources [9]. The main deployment models are: Public, Private, Hybrid, and Community

Public cloud deployment model is the most common and well-known model. The cloud is a data center that offers every user the same service [9]. The data center is the information that has been deployed to the cloud, from all users of the service.

Private clouds have a similar set-up to public clouds; however, the data center is internally hosted by the customer [8]. This model is helpful when managing sensitive data, because it allows the customer to control access to shared information. Furthermore, only users who have access to the private cloud can use its pool of resources. This model is typically more expensive than public clouds due to additional costs related to on-premise data center management.

Hybrid clouds are a combination of public and private clouds. This model is suited for large companies because it combines the privacy of the private cloud and the computing resources of the public cloud [8]. This is the cloud deployment model our project aims to utilize.

Community cloud is a model in which a group of users utilize a private cloud model [9]. An example of this form of cloud deployment is the United States government. *Figure 3* further outlines the deployment models and some of their properties.

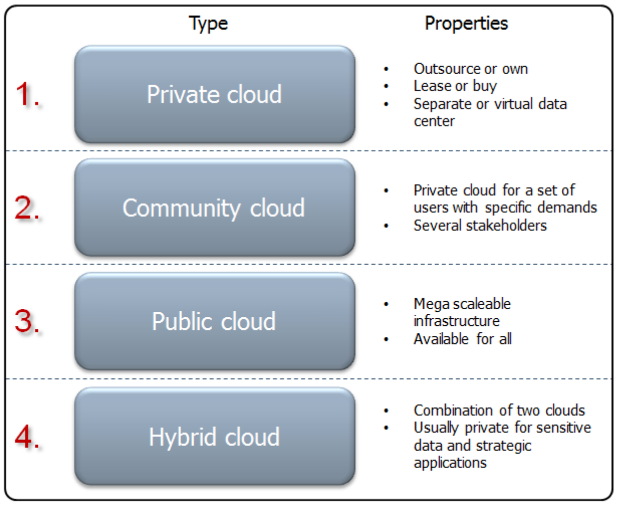

Type
Properties
1.
Private cloud
• Outsource or own
• Lease or buy
• Separate or virtual data center
2.
Community cloud
• Private cloud for a set of users with specific demands
• Several stakeholders
3.
Public cloud
• Mega scaleable infrastructure
• Available for all
4.
Hybrid cloud
• Combination of two clouds
• Usually private for sensitive data and strategic applications

Figure 3: The Top Four Cloud Deployment Model Properties [10]

For the scope of this project, it is important to define multi-cloud versus hybrid cloud. Multi-cloud is the use of multiple cloud providers independently from each other. Hybrid-cloud is the implementation of a workload that spans across multiple clouds, specifically public versus private. In this project we will be working on a multi-cloud system.

Cloud Service Provider

A cloud service provider offers a platform where customers can utilize their cloud-based infrastructure and storage services [11]. Each cloud provider utilizes their own pricing model, based on the services that they offer to their customers. These differences can often be associated with the level of abstraction that the cloud service providers outline for their customers.

Levels of Abstraction

The level of abstraction allows companies using the cloud to decide what type of service they need from the cloud service provider. The different levels all have different services being provided, and thus have different ways of measuring their quality of service. In *Table 1*, the different levels of abstraction are defined according to the service that is associated with them [3].

Table 1: Levels of Abstraction

Level of Abstraction	**Service Provider Performs**
Infrastructure as a service (IaaS)	Storage of information, or the ability to transfer files
Platform as a service (PaaS)	Provides a platform that includes hardware, and an operating system
Software as a service (SaaS)	Provides a platform that includes hardware, an operating system, and software
Business process as a service (BPaaS)	A whole business function
Information as a service (INaaS)	Database of information (such as laws or tax codes)

III. Problem Statement

Multi-cloud solutions act as a broker between the variety of cloud service providers. In order to optimize cost, some workloads can span across multiple clouds. Different types of workloads may be better suited for certain cloud service providers in terms of pricing. By implementing a cost optimization algorithm, our multi-cloud solution aims to give users the best price while maintaining quality of service for their workload. It is our belief that we can reduce costs by a small amount and optimize the usage of cloud resources.

Analysis of our solution will be based on cost comparisons between two major cloud providers and the results of our algorithm. We want to ensure that the provisioned resources are being utilized, and analyze metrics such as CPU usage, storage capacity, and deployment time.

The goal of our project is to provide a cost-effective multi-cloud solution that focuses on individual users and their workloads. To achieve this goal, we have three main objectives:

1. Optimize cost of cloud resources for different types of workloads.
2. Create a web interface for multi-cloud resource deployments to AWS and GCP.
3. Evaluate the correctness and resource utilization metrics of our multi-cloud resource manager.

Chapter 2: Literature Review

In order to provide a cost-effective multi-cloud system, it is necessary to gain an understanding of the existing market. This chapter outlines the pricing models of leading cloud providers, analyzes the market surrounding existing container and cloud technologies, and defines common workloads leveraged by the cloud. In addition, it explores existing multi-cloud technologies, highlights two implementation methods of intelligent resource provisioning, and introduces human-computer interaction concepts that will be adopted by our multi-cloud solution.

I. Pricing Models

Cloud Computing is a billion-dollar industry that thrives on a large consumer base. In order to keep up with the industry, it is imperative for companies to provide their product at a competitive price. Cost models are algorithms that are used to decide how much money a project will cost. Once a cost model for a project is generated, a pricing model can be derived so that the provider can profit from their offered services. There are a few types of cost models that are used in cloud computing listed below in *Table 2* [12].

Table 2: Cost Models

Model	Description	Examples
Subscription-cost pricing model	Over period of time, there is a subscription charge paid	IBM SmartCloud for Social Business
Advertising-based cost model	The consumer is not charged, or rarely charged due to the amount of ads	free TV provider
Market-based cost model	The consumer is charged on a timed per-usage basis, where the cost fluctuates based on demand	Amazon EC2 Spot Instances
Group buying cost model	The more consumers there are in a group the less an individual has to pay	Groupon

Pricing models define the price that is paid in order to receive the value of a product or a service [12]. There are a few different pricing models including utility, service, performance, and marketing-based pricing models [3]. Utility and service-based price models are the most common in the cloud computing market [3]. The utility-based pricing model is described as when the consumer is monitored and pays accordingly. Examples of utility-based models are described in *Table 3* [3].

Table 3: Utility Based Pricing Models

Model	Description	Abstraction Level (best use)
Consumption	Pay for the resources used	IaaS, PaaS
Transaction	Pay for the number of transactions	BPaaS, INaaS, SaaS
Subscription	Pay for a time period of service (usually a month)	All abstraction levels

Service-based pricing models use the risk, or money saved as defined in an SLA to compute how much the consumer will pay [12]. Examples of service-based pricing models are listed in *Table 4* [12]. We will be using these pricing models in order to get a better idea of the cloud services we are dealing with, and how they compare to each other.

Table 4: Service Based Pricing Models

Model	Description	Abstraction Level (best use)
Fixed Price	Made up of a nonrecurring price and a recurring price	All abstraction levels
Volume-Based	Cost based on the amount of users, storage, transaction speed, or volume of resources used	All abstraction levels, but mainly IaaS or PaaS
Tiered Price	Creates a tiered model that is based on the volume, cost, or SLA	All abstraction levels

II. Market Overviews

Container Technologies

The current industry standard for container technology is Docker [13]. Docker originated as an experiment for deploying Linux containers onto PaaS platforms [14]. According to Docker, they provide the strongest default isolation capabilities in the industry [15]. The company gained popularity due to its simplicity -- the code was isolated and portable, making it easy for developers to deploy the same code into testing and development environments.

As more companies started using Linux containers, they needed an easy way to manage them due to the isolation of containers. Eventually, Google developed a solution for this in its Kubernetes technology. Container technology had only been around for approximately four years when Kubernetes entered the stage [16]. Many companies were rushing to become the industry standard for container deployment and management, and eventually Kubernetes saw a rise in adoption [16]. Kubernetes gained popularity because of its stability and adaptability. Forrester Research stated in their 2018 Cloud Predictions that Kubernetes beat out competitors for container orchestration [17]. Together, Docker and Kubernetes are a powerful combination that our team can use to accomplish our goal of creating a multi-cloud deployment platform. Deployments to containers are often viable solutions for smaller workloads where creating new virtual machine is not needed.

Cloud Technologies

The cloud computing market is growing rapidly. Two of the largest cloud services are Amazon Web Services (AWS) and Google Cloud Platform (GCP) [18]. The leading cloud infrastructure technology platform is AWS [18]. According to Amazon, AWS is a variety of cloud products that include computing, storage, databases, analytics, and networking. Due to their resources, AWS controls around a third of the cloud marketplace, and over half of the IaaS market [18]. In addition, the service provides management and developer tools for their customers [19]. GCP is one of Amazon's top competitors [20]. Currently, GCP offers a variety of services including computing and hosting, storage, networking, big data, and machine learning [21]. The largest VMs in the cloud marketplace belong to GCP [18]. AWS and GCP are of interest, because together they control a majority of the cloud computing marketplace.

There are several key differences between AWS and GCP. Amazon has more offerings in terms of integration with other technologies. For example, if a user needs a cloud SQL solution, GCP and AWS offer in the solutions that they offer. GCP offers a MySQL solution, while AWS offers multiple options such as Aurora, MariaDB, Oracle, and Microsoft SQL Server. Furthermore, Amazon has more data centers across the world [18]. This may be beneficial for users in countries where some services may be blocked internationally. On the other hand, GCP offers more configurability for instances on their cloud platform [18]. This allows for the user to configure their deployments optimally for each process, using only the resources they need. GCP also has more informative dashboard, allowing the user to know exactly what resources are being used and where [18].

Due to the scope of our project, we decided to focus on GCP and AWS for our multi-cloud solution. It is important to understand the differences between GCP and AWS as each cloud service provider offers different services at competitive prices. AWS utilizes what they call "pay-as-you-go" pricing and offer a 1-year free trial [19]. GCP has a $300 credit that lasts a year, and a free tier the customer can utilize before committing to their "pay-as-you-go" pricing scheme [21]. Both AWS and GCP are market-based cost models, that use a consumption-based pricing model. GCP also includes a tiered price-based pricing model [22]. Although one option may be cheaper, the user may want the more expensive option for additional functionality. These issues should be considered when developing a multi-cloud solution.

III. Existing Multi-Cloud Technologies

Enterprises are increasing their demand for cloud management tools. A scan of the existing market shows that there are a wide range of cloud management tools that offer an array of solutions. However, there is a lack of cohesion in the current market as many of the cloud management offerings do not cover all enterprise needs. Many companies utilize a variety of tools for a holistic solution to cloud management [23].

Cloud management platforms (CMP) are a series of multifunctional tools that manage the cloud solutions across a variety of cloud providers. The success of CMPs vary as many enterprises have to implement other tools for deployments [23]. The requirements for cloud management involve three major categories: access management, service management, and service optimization

Service management, the focus of our project, deals with provisioning, orchestration, and automation. It also includes governance and policy, monitoring and metering, and multi-cloud brokering. CMPs are more likely to provide capabilities to service management over the other two major categories [23]. CMPs need to enforce policies on which provider to use and what resources can be consumed from that provider. In addition, CMPs should reduce lock-ins with one cloud provider and provide a variety of pricing options for optimization [23].

There are many vendors that provide CMP solutions. There are two technologies that we wanted to highlight for different reasons. The first is Cisco CloudCenter. Cisco CloudCenter defines itself as IT-as-a-Solution (ITaaS). It acts as a service broker for multiple cloud providers via a single interface. It offers one-click deployment and includes cost controls and reporting mechanisms on usage statistics [24]. The reason for highlighting this particular technology is the UI that it utilizes. The UI provides a clean way to deliver cost reporting that many CMP's in the market fail to report on. The dashboard section shows the percentage of active virtual machines from each cloud provider, usage statistics for each provider, and the status of each of the running applications. Cost reporting and usage data is an important aspect of a multicloud interface and analyzing this Cisco's interface from a user perspective will provide benefits when designing our application.

The second technology is HashiCorp Terraform. This technology provides a mechanism to make infrastructure as a code portable to other cloud providers. The goal of Terraform is to provide reliable deployments and prevent vendor lock-ins. In addition, it urges developers to extend Terraform to fit new needs [25]. The reason for highlighting Terraform is due to its multi-cloud deployments and flexibility of extending the API to fit our needs.

In addition to fully functional solutions, there are currently a variety APIs that provide low-level abstraction [26]. In this project, we will implement and extend an API for our multi-cloud interface. The API's that were analyzed include:

1. Apache Libcloud (Python)
2. Apache jCloud (Java)
3. Fog (Ruby)
4. Libretto
5. Pkgcloud

After analyzing each technology, it was concluded that Apache Libcloud provided us with the greatest benefits for developing a multi-cloud solution. It is supported by a variety of cloud and container technologies, and it supports an entirely code-based solution [27]. This mechanism allows us the freedom to develop our own UI design. In addition, it also supports methods for load balancing, block storage, and object storage across a variety of cloud providers [27]. Apache jCloud is related to Libcloud and the main difference is that it implements a Java-based solution [26].

Fog is a Ruby-based solution that does have many of the mechanisms present in Apache Libcloud and Apache jCloud. However, often the code quality is not as strong and many of the drivers are missing tests [26]. In an effort to focus on a fully-functional web interface, we felt that a Python-based solution is better suited for web development. Finally, Libretto and Pkgcloud did not provide the full functionality we were looking for in our solution.

IV. Workloads

Cloud computing offers a variety of resources optimized for different types of workloads. A workload is a computing task that is completed in a given amount of time [10]. The types of workloads that are often leveraged by cloud computing require different configurations of resources in order to optimize performance.

Common cloud workloads include large streaming workloads, in-memory database workloads, big data storage workloads, and machine and deep learning workloads. *Large streaming workloads* are associated by high throughput. The difficulty related to this workload is that the end user can cancel the interaction at any moment. Bandwidth to the user and network speed from storage to the server are major concerns with this type of workload [28]. *In-memory database workloads* deal with large amounts of data that is accessed both quickly and often. It is concerned with real-time processing and speed is the highest priority [28]. *Big data storage workloads* manage large databases that are updated in small amounts periodically. The main concern related to this workload is availability, integrity, and security [28]. *Machine and deep learning workloads* are computationally expensive and utilize large amounts of data [29]. Cloud service providers often offer resources that are better suited for these types of workloads [30][31]. In addition to these common workloads, we also want to define *general purpose workloads* in relation to our project. This type of workload is related to non-intensive tasks that are not resource dependent for optimization.

AWS and GCP each offer virtual machine configurations that are designed to optimize performance based on certain workloads. In addition to the pre-configurations, users also have the ability to define their own resources. In *Table 5* and *Table 6*, we highlighted a minimum and maximum resource configuration that AWS and GCP offer for in-memory workloads. These tables illustrate some of the resource levels needed for this type of workload. In addition, each of the tables only show a small fraction of the total offerings available with AWS and GCP. In *Table 7* and *Table 8*, we highlighted configurations for machine and deep learning workloads. Finally, in *Table 9* and *Table 10,* we show the resource configurations that are often utilized for general purpose workloads. This table is particularly relevant to our project because we are focused on general purpose workloads.

General purpose workloads are instrumental to our project because of its lack of dependency on resources. This fact will allow us the freedom to experiment with both virtual machine solutions and container solutions. In addition, we will be able to analyze usage statistics and the accuracy of our algorithms. As we feel comfortable in the accuracy of our solution, we may extend our scope to include machine-learning workloads as well.

Table 5: AWS Instance offerings for In-memory workloads [30]

Type	# of vCPU	Memory (GB)	Storage (xGB)	Dedicated bandwidth EBS volumes (Mbps)
x1e.xlarge	4	122	1x120	500
x1e.32xlarge	128	3,904	2x1920	14,000
x1.16xlarge	64	976	1x1920	7,000
x1.32xlarge	128	1,952	2x1920	14,000

Table 6: Google Cloud Engine offerings for in-memory workloads [31]

Type	# of vCPU	Memory (GB)	Max number persistent disks (PD)	Max total PD size (TB)	Local SSD
n1-ultramem-40	40	961	16	64	No
n1-ultramem-160	160	3,844	16	64	No

Table 7: AWS Instance offerings for machine and deep learning workloads [30]

Type	# of vCPU	Memory (GB)	Storage (GB)	Dedicated EBS bandwidth (Mbps)
c5.large	2	4	EBS-only	3,500
c5d.18xlarge	72	144	2x900 SSD	3,500

Table 8: Google Cloud Engine offerings for machine and deep learning workloads [31]

Type	# of vCPU	Memory (GB)	Max number persistent disks (PD)	Max total PD size (TB)
n1-highcpu-2	2	180	16	64
n1-highcpu-96	96	864	16	64

Table 9: AWS Instance offerings for general purpose workloads [30]

Type	# of vCPU	CPU credits per hr	Memory (GB)	Storage (type)
t3.nano	2	6	0.5	EBS-only
t3.medium	2	24	4	EBS-only
t3.large	2	36	8	EBS-only
t3.2xlarge	8	192	32	EBS-only

Table 10: Google Cloud Engine offerings for general purpose workloads [31]

Type	# of vCPU	Memory (GB)	Max number persistent disks (PD)	Max total PD size (TB)
n1-standard-1	1	3.75	16	64
n1-standard-96	96	360	16	64

V. Intelligent Resource Provisioning

Intelligent resource provisioning is a critical component to multi-cloud solutions. In order to optimize costs based on different workloads, our system needs to provision resources in a way that does not sacrifice quality of service (QoS). Quality of service relies on the resources ensuring that response time and cost constraints are met. Often, QoS is defined by the cloud service providers by SLAs [32].

There are two concepts that are implemented for intelligent resource provisioning: optimization algorithms and machine learning. In order to explain the first concept, we highlight a system called CherryPick. CherryPick utilizes Bayesian optimization for resource provisioning. This method estimates confidence intervals of both cost and running time for different possibilities of cloud configurations [33]. One of the goals is to find an optimal solution to minimize the cost algorithm.

$$C(\vec{x}) = P(\vec{x}) * T(\vec{x})$$

$T(\vec{x})$ is equal to the running time and must satisfy $T(\vec{x}) \leq Tmax$, or the total tolerated running time.
$P(\vec{x})$ is equal to the price per unit of time for all of the virtual machine's needed in a particular cloud configuration.

[33]

The reason that Bayesian Optimization is needed is due to the fact that calculating the running time under all of the different configurations would result in high total runtime [33]. Bayesian optimization computes a confidence interval based on samples that come from the cost algorithm C(x) [33].

The second concept is utilizing machine learning for intelligent resource provisioning. The system that utilizes this method is PARIS. PARIS created an interface system that provisions resources for different workloads based on machine learning algorithms [34].

There are two stages related to machine learning: an offline stage and an online stage. During the offline stage, information is collected about different VM types. In addition, benchmark tests are generated with a variety of different resource requirements [34]. During the online stage, workload queries are created by executing the user's task on reference VMs that are currently running. Then, the information from the offline stage and information from the online stage are combined to successfully model the user's workload performance [34]. PARIS does acknowledge that there are difficulties modeling workload resource requirements due to executions utilizing different resources at different times.

Intelligent resource provisioning is a critical component to our project. As a result, we need to understand the different implementations that currently exist today. The two concepts utilized by the systems above are viable solutions to our multi-cloud solution. However, our approach will use cost optimization algorithms similar to CherryPick. Machine learning, although useful, adds additional resource and data requirements that exceed the scope of our project.

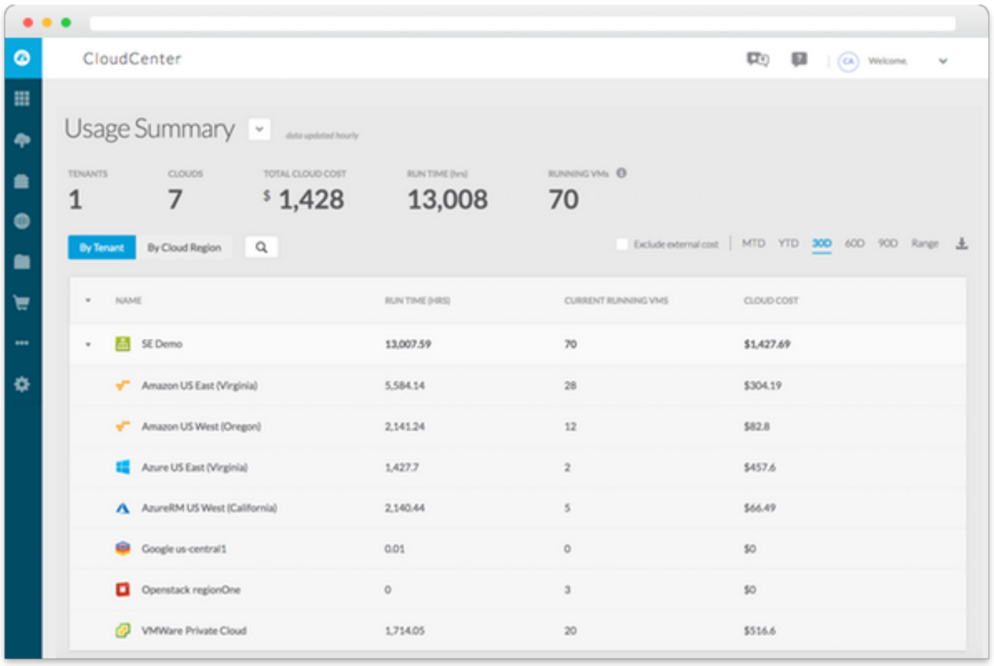
VI. Human-Computer Interaction (HCI) Principles

Since we are building a web-application, we need to keep in mind the current human-computer interaction principles to ensure a quality user experience [35].

1. Define the ergonomics of the interface.
2. Determine dialogue design and interface style.
3. Choose a presentation style and screen design [35].

The first step allows us to guarantee two things. First, the user must understand the input they must give to the program for it to work properly. Second, the user must understand the output our program will return. These two combined makes for an acceptable user experience. However, if we want to ensure a *quality* user experience, we want to follow the second and third steps.

Current implementations of multi-cloud deployment platforms are very clean and easy to follow, shown in *Figure 4* below.



NAME	RUN TIME (HRS)	CURRENT RUNNING VMS	CLOUD COST
SE Demo	13,007.59	70	$1,427.69
Amazon US East (Virginia)	5,584.14	28	$304.19
Amazon US West (Oregon)	2,141.24	12	$82.8
Azure US East (Virginia)	1,427.7	2	$457.6
AzureRM US West (California)	2,140.44	5	$66.49
Google us-central1	0.01	0	$0
Openstack regionOne	0	3	$0
VMWare Private Cloud	1,714.05	20	$516.6

Figure 4: Interface for Cisco CloudCenter, a current multi-cloud deployment platform [24]

Cisco CloudCenter, a current multi-cloud solution, offers a variety of services for their users [24]. Reducing complexity through the user interface allows the user to quickly manage and deploy workloads. By analyzing this solution, we can gain an understanding of the configurations we may need for our approach.

Cisco CloudCenter requires users to link their different accounts with cloud service providers to the application [24]. In addition, it allows the user to choose SLA agreements that bet fit the application they wish to deploy. After, the user can choose a desired environment for their application and Cisco handles the deployment [24]. The process is efficient and streamlined which increases the user's understanding of the system.

For our application to be successful, we need to implement the proper HCI and web-development designs in order to provide our users with the best possible experience. User experience, often not at the forefront of development, is essential for our multi-cloud solution to be effective. We need to provide an interface that is both intuitive and easy to use.

VII. Conclusion

As more enterprises turn towards cloud computing to handle their workloads, cost optimization will be a critical concern. Due to the variety of pricing models between cloud service providers, it is often difficult to predict the best price. Our multi-cloud solution aims to provide the most accurate prediction of price among providers. We intend to give users optimal solutions that fit their needs, but at the best price.

Chapter 3: Implementation

I. System Overview

The web interface we developed is designed to take user provided input about a predefined workload option or custom resource options and generate a virtual machine recommendation aimed to minimize costs and fulfill workload demands. A high-level overview of our architecture can be found in *Figure 5*.

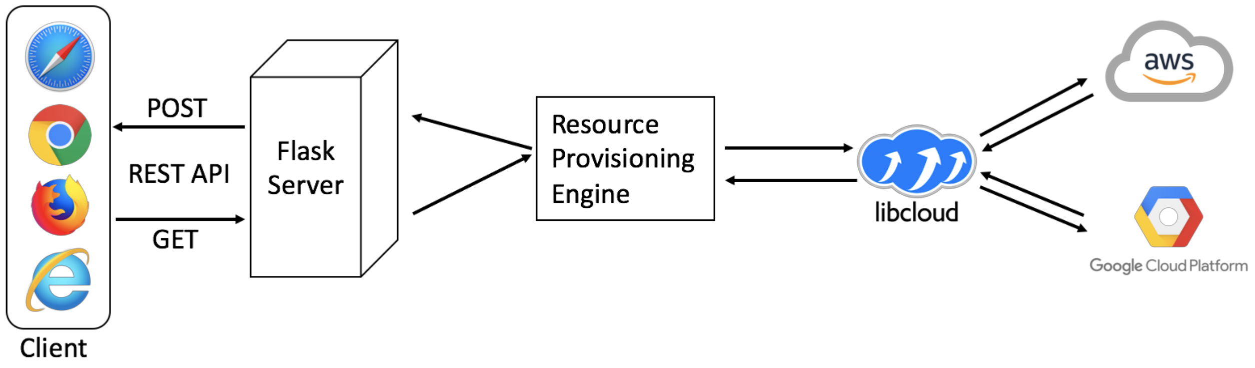


Figure 5: High-level architecture overview for our multi-cloud interface

The front-end web server is written in HTML and Python. In addition, we utilized Flask, a Python web framework, for our background server. We integrated the Apache Libcloud API that will allow us to connect to the different cloud service providers, specifically AWS and GCP. Due to the Apache Libcloud API, a Python server is better suited for our project. Communication between the client and server is done using the RESTful API, specifically GET and POST requests. These methods allow us to get input from the user and provide output to the web clients. In addition, we have a resource provisioning engine that computes the top three optimal cloud resources based on the user-specified requirements. The flow of information within our system can be found in *Figure 6.*

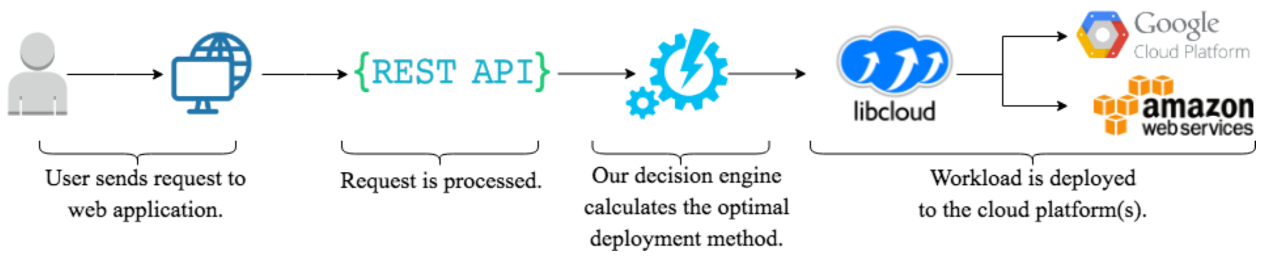

{REST API}
libcloud
Google Cloud Platform
amazon web services
User sends request to web application.
Request is processed.
Our decision engine calculates the optimal deployment method.
Workload is deployed to the cloud platform(s).

Figure 6: Flow of information for our multi-cloud solution

The workflow of the system begins when the user initiates a request to the multi-cloud system. There are two options the user can choose: user-defined or custom. After the user initiates the request by providing the needed input, it is sent to the backend server using the RESTful API. Our decision engine then calculates the three best options, based on cost, and sends it back to the user for further input. The user then chooses a deployment option and a request is sent through the Apache Libcloud API to the respective cloud provider.

Design Constraints

As we conducted our background research, we narrowed the scope of our project which resulted in design constraints. First, our system design will only use two cloud service providers: AWS and GCP. In addition, an active cloud account with both providers is needed. We do not require the user to input account information for each of the cloud providers and instead conduct a proof of concept using a general set of credentials. In the future, user credentials will need to be collected in order to pass the resources to the user. Second, we used Flask as our web server because it is Python-based, and it integrates with the also Python-based Apache Libcloud API.

II. System Software Architecture

Flask

We chose to use Flask instead of Node.js because we will be integrating with Apache Libcloud, which is a Python-based package. Flask is a micro web framework for Python. It provides us with reusable code and extensions for common web application operations, such as request handling, creating database connections, and authentication. Flask will also help us set up the application's points of interaction. Furthermore, Flask gives us access to the Python standard libraries, which will speed up our development process.

RESTful APIs

We will utilize GET and POST methods for transferring user input between the client and the web server. The GET method will retrieve the user input. The POST method will output the responses to the client.

Apache Libcloud

Apache Libcloud is a multi-cloud API that can connect to multiple cloud providers. This API supports both GCP and Amazon. Specifically, we are focused on the integrated Cloud APIs for Amazon EC2, Amazon S3, and the Google Compute Engine API. Apache Libcloud is Python-based and will cleanly integrate with the Flask web server.

Resource Provisioning Engine

The resource provisioning engine will be our optimization algorithm that computes the cloud configurations for lowest cost among cloud service providers. Cost optimization algorithms are essential to our multi-cloud solution. Both AWS and GCP are queried in which a list of all resources are generated. Then, the lists are filtered based on the user's input and the top three lowest prices are chosen.

Inputs

The inputs that are required by the user are based on two categories: *user-specified* and *workload-specified*. The user-specified input allows the customer to provision their own resources, and workload-specified provides different configurations based on types of workloads.

User-Specified Input:

- Instance Name
- Operating System (Linux, Windows, Redhat, CensOS, etc.)
- Memory (GB)
- Storage (GB)

Workload-Specified Input:

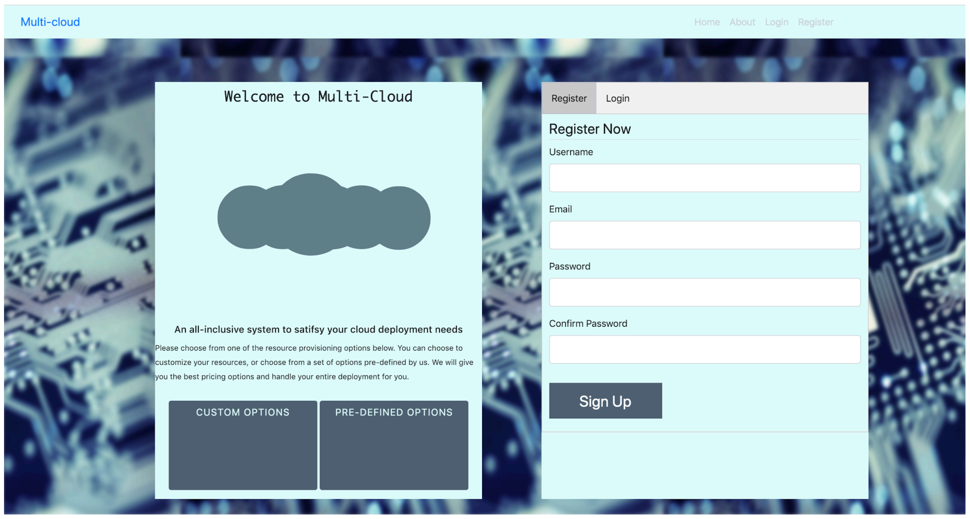
- Workload type (Machine and deep learning, big data storage, general purpose, etc.)

Outputs

The output of our system happens in two phases: the *initializing phase* and the *deployment phase*. During the initialization phase, the pricing options are provided to the user in which they must choose their desired option. In addition, we highlight the optimized cost for the user. During the deployment phase, we provide users information about their workload deployment. This phase sends information to the user about their virtual machine, the status of the deployment, and information on how to access it. In addition, we plan to also provide information on resource utilization such as CPU usage over time, storage usage, etc.

IV. Interface Detailed Design

The home page of our interface will have our application name, and a summary of what our application accomplishes. We will have links to resources that could be useful to users, as well as detailed information about our project. From this page the user is able to navigate to the custom options page, or the predefined workload page based on their preferences. An example of what this looks like is pictured below in *Figure 7.*



An all-inclusive system to satifsy your cloud deployment needs

Please choose from one of the resource provisioning options below. You can choose to customize your resources, or choose from a set of options pre-defined by us. We will give you the best pricing options and handle your entire deployment for you.

Figure 7: Home Page

If the user selects that they want to specify their own cloud resources, the web application will navigate them to the custom options page pictured in *Figure 8.* Here the user is prompted to enter the resources they want to use.

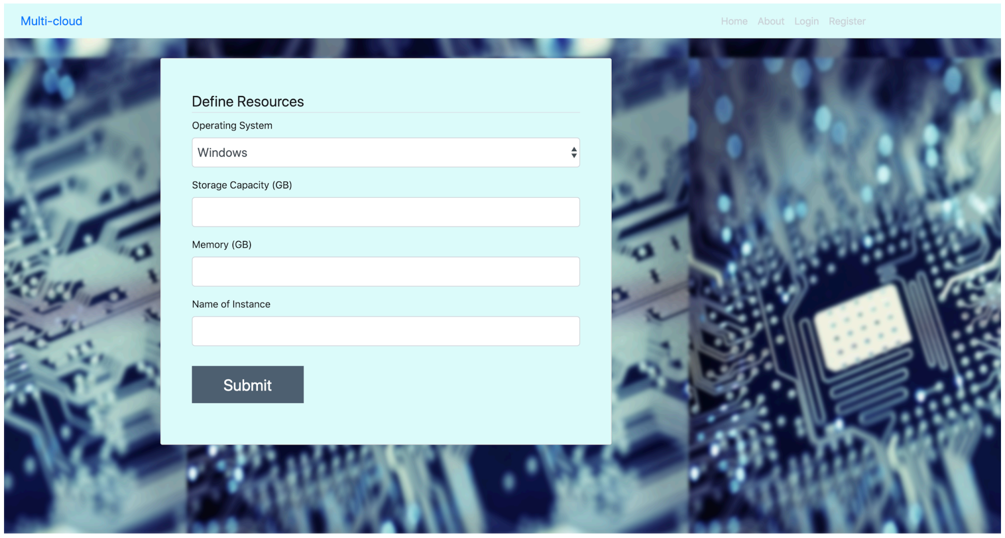


Figure 8: Custom Options Page

If the user selects that they want to choose an option based on a set of predefined resources based on workload, they will be taken to the Workload-Based page pictured in *Figure 9.* After the user selects their workload, they will be taken to the top options page.

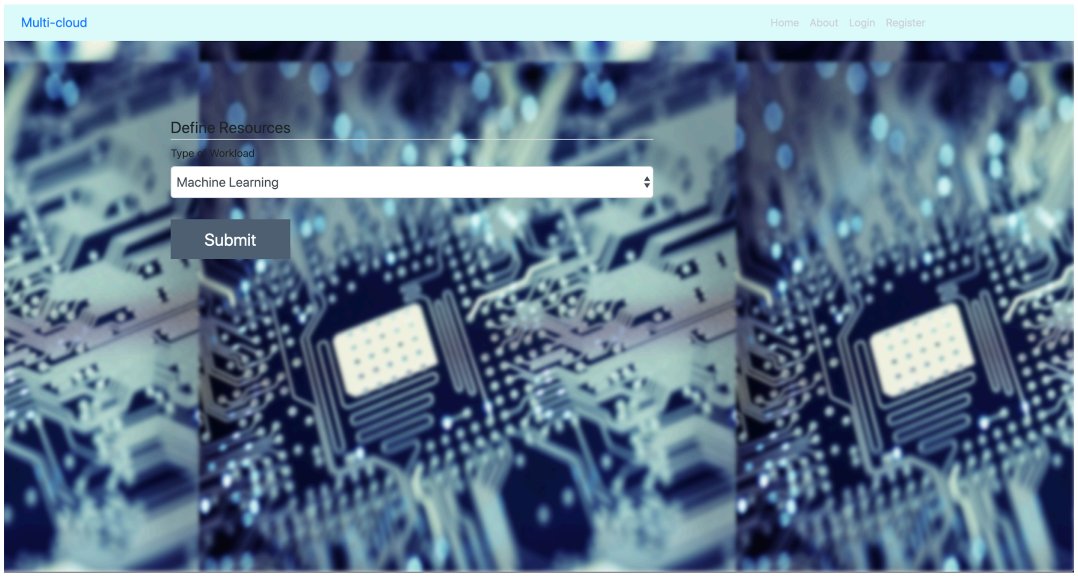
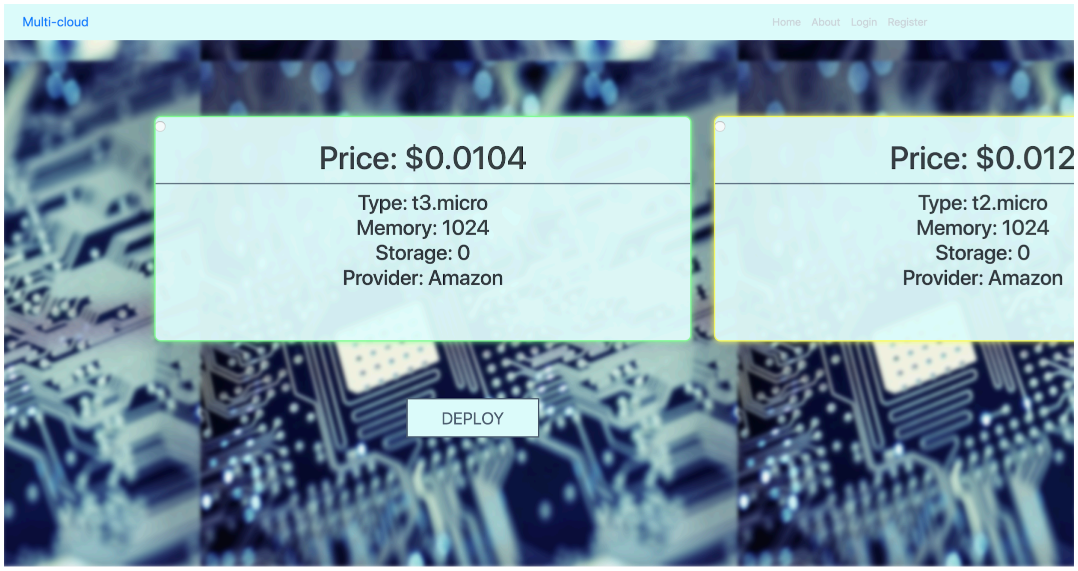


Figure 9: Workload-Based Options Page

The top options page is where the top cloud provisioning selections is displayed. An algorithm will run based on the user input, and our system returns a maximum of three cheapest cloud resource options. It will also outline the specific resources associated with each option and the price. The user will be able to scroll between the options, as displayed in *Figure 10* below.



Price: $0.0104

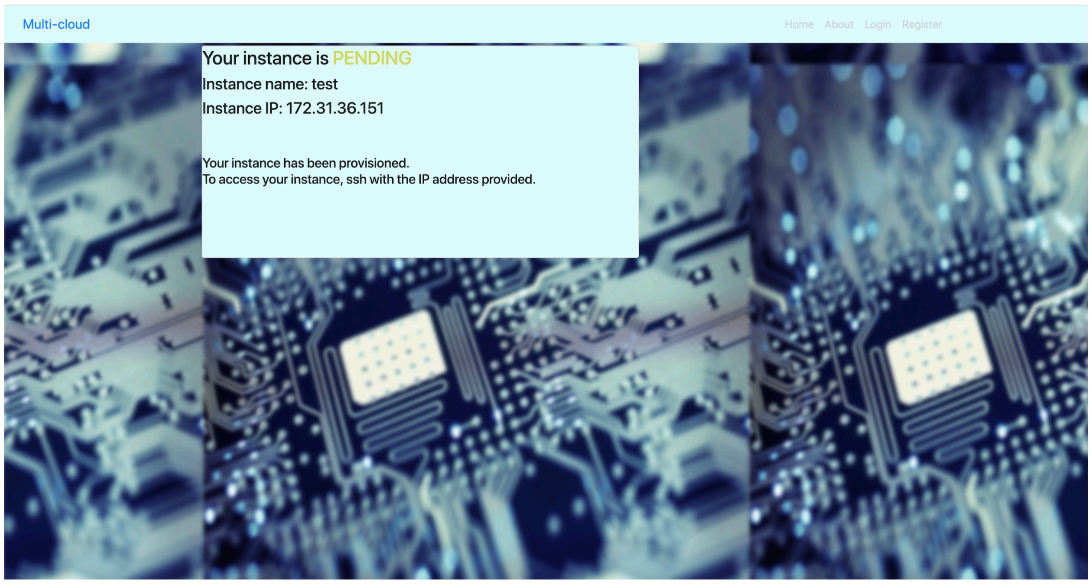
Type: t3.micro
Memory: 1024
Storage: 0
Provider: Amazon

Price: $0.012

Type: t2.micro
Memory: 1024
Storage: 0
Provider: Amazon

Figure 10: Top Options Page

The last page in the process that the user is taken to is the deployment page shown in *Figure 11* below. The user is taken to the deployment page after an option is selected and the deploy button is pressed. It will show the name of the instance, an IP address, and the state of the deployment (pending or ready).



Your instance is PENDING

Instance name: test

Instance IP: 172.31.36.151

Your instance has been provisioned.
To access your instance, ssh with the IP address provided.

Figure 11: Deployment Page

Chapter 4: Results

In the previous chapter, we outlined the design of our system and highlighted specific functionality that would help us achieve our goal for this project. In this chapter we analyze the multi-cloud interface to determine correctness, efficiency, and optimality of the solutions. The metrics we used to measure performance include runtime of our cost optimization functions, cloud resource usage statistics, deployment times of AWS and GCP cloud resources, and runtime tradeoffs of our caching implementation.

I. Procedure

Correctness Tests

Table 11, shown below, was created for testing the correctness of our two main functions: *find_instance* and *find_instance_workload*. In order to evaluate the functions, we needed to analyze the results from our algorithms with the results manually taken from AWS and GCP, respectively. The input column consists of the parameters entered into our function. The expected values column is the results that are taken directly from AWS and GCP. In *Table 11,* the actual values are the results that are returned using our pricing function, and the final column is used to indicate whether the expected and actual values matched.

Table 11: Correctness tests for *find_instance* function based on user input

Test #	Input	Expected	Actual	Success (Y/N)
1	1GB Memory 1GB Storage	t3.nano (0.5GB, 0GB, $0.0052)	t3.nano (0.5GB, 0GB, $0.0052)	Y

In the first step, the input is defined for the purpose of recreation and documentation. For the *find_instance* function, the input includes memory and storage. For the purpose of consistency, the memory is converted to gigabytes in the table although the function takes megabytes for that parameter. In the second step, we manually found the three cheapest options based on memory and storage requirements on the two cloud providers' websites. Finally, we created a unit testing Python program that automated our testing process and printed out the returned results. This testing program can be found in *Appendix A*. After running the program, we compared the results between the expected return values and actual values.

This testing template can be used for the *find_instance_workload* function. The only difference is that the input is a string indicating the type of predefined options. These options include 'ml', 'gp', and 'im', which stand for machine learning, general purpose, and in-memory, respectively.

Design Optimization Tests

In addition to the correctness tests, we also tested runtime based on the design decision of implementing caching when finding a virtual machine size from AWS or GCP. *Table 12* is a template used for collecting runtime results from caching versus non-caching. The *find_instance* and the *find_instance_workload* columns contain the runtime, in milliseconds, of each function. The final column is the absolute value of the difference between the second and third columns. There will be two sets of *Table 12*, one that is non-caching and one that is caching.

Table 12: Caching vs Non-Caching Template

Test #	find_instance (ms)	find_instance_workload (ms)	Difference (ms)
1	1000	1000	0
2	1000	1000	0
3	1000	1000	0

We created a testing program to measure the runtime of each of our functions. This can be found in *Appendix A*. In addition, we created a flag in the pricing file that switched between caching versus non-caching in order to understand if the design choice improved efficiency. We performed 50 tests for each algorithm for caching and non-caching, totaling 200 test cases.

Measuring Deployment Time

Table 13 is a template used for collecting data used to measure the performance of our functions and deployment times. For this set of tests, deployment time is defined as when the request is sent to the respective cloud provider, via the Apache Libcloud API function, to when the function returns. We are interested in runtime and deployment time of our system in order to measure its efficiency. The metric we used for measuring runtime is milliseconds and recorded in column two. The provider column is used to differentiate between AWS and GCP.

Table 13: Deployment Testing Template

Test #	Deployment Time (ms)	Provider
1	1000	AWS
2	1000	GCP

We created a testing program to calculate the total runtime of our two main functions. In addition, we also measured the total time elapsed for a deployment to complete. This testing program can be found in *Appendix B*. For deployment testing, we gathered results from 50 test cases from AWS and 50 test cases from GCP, totaling 100 test cases.

Resource Usage Analysis

Another way we tested the system was by running projects of our own on the recommended images. In this specific case study, we focused on our machine-learning recommendations for AWS. We wanted to see if the allocated resources (CPU and memory) from the recommended image are able to support the user's workload based on the user input. From this, we used AWS' built-in analytics tools to view the various statistics of our launched instances.

Machine Learning with Tensorflow and Keras

Amazon AWS has specific images for machine and deep learning workloads, as well as images that have pre-loaded libraries for workloads that have specific dependencies. In this case study we compared our platform's recommendation versus the AWS dashboard for creating an instance and ran a machine learning project requiring both TensorFlow and Keras libraries. The project also included an 11.59 MB dataset.

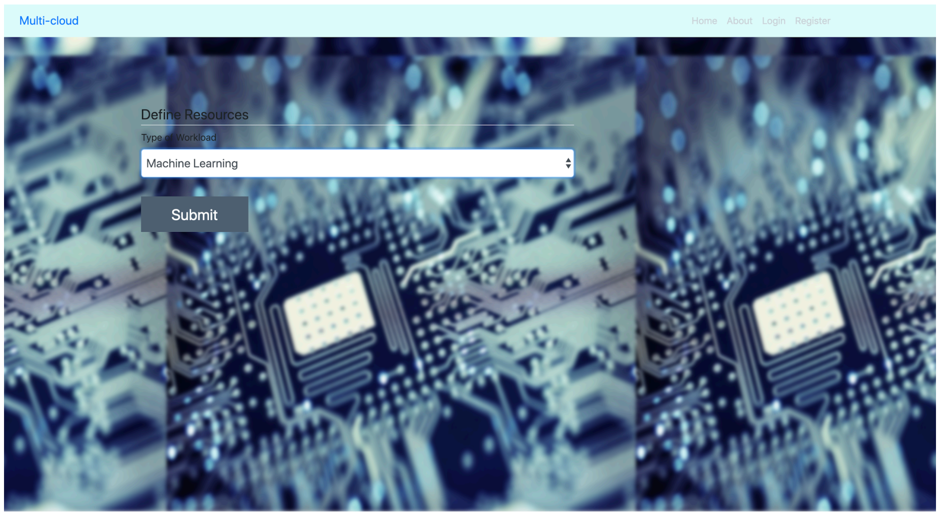


Figure 12: Launching an AWS Deep Learning AMI using our platform

On our platform, the test only required two steps:

1. User input.
2. User selection of recommended options.

In order to run this test, we needed to choose a workload-based launch from our system and compare it to a manual instance option from AWS. After the instances are created, we then ran the machine-learning project and analyzed the resource usage statistics.

II. Testing

Find_Instance and *Find_Instance_Workload*

Find_instance and *find_instance_workload* are the two most important functions of our system. They provide the user with a list of valid options based on resource-defined or workload-defined inputs. As a result, we chose to test both correctness and runtime as the metrics for analyzing performance. The first set of tests performed focused on correctness and utilized *Table 14*. We split the tests into two groups and started with *find_instance.* Testing went through two iterations due to a bug in the pricing information of our system. The results of the two iterations for the correctness tests can be found in *Table 15* and *Table 16*.

First Iteration

Table 14: First Iteration of Correctness Tests

Test #	**Input**	**Expected**	**Actual**	**Success (Y/N)**
1	0.8GB Memory 0 GB Storage	t3.micro (1GB, 0GB, $0.0104) t2.micro (1GB, 0GB, $0.012) g1-small (1.7GB, 0GB, $0.027)	t3.micro (1GB, 0GB, **none**) t2.micro (1GB, 0GB, $0.012) g1-small (1.7GB, 0GB, $0.027)	N
2	16 GB Memory 10 GB Storage	r5d.large (16GB, 75GB, $0.144) m5d.xlarge (16GB, 150GB, $0.226) m2.xlarge (17.1GB, 420GB, $0.245)	r5d.large (16GB, 75GB, **none**) m5d.xlarge (16GB, 150GB, **none**) c5.2xlarge (16GB, 120GB, $0.245)	N

After running two tests during the first iteration, we noticed that prices were being returned as *none,* highlighted in *Table 14*. Instances of those types on AWS are not free of charge. As a result, we investigated where the pricing data is being generated from. The Apache Libcloud API stores a default pricing file that is used for setting the price of each size from AWS and GCP. Reviewing the file showed that it had not been updated with new prices or instance types from either cloud provider. As a result, we had to manually enter the pricing options for the missing instances into the pricing file. Small discrepancies in price between actual values and expected values are due to the pricing file not being updated.

Second Iteration

Table 15: *Results of Second Iteration of find_instance*

Test #	Input	Expected	Actual	Success (Y/N)
1	0.8GB Memory 0 GB Storage	t3.micro (1GB, 0GB, $0.0104) t2.micro (1GB, 0GB, $0.012) g1-small (1.7GB, 0GB, $0.027)	t3.micro (1GB, 0GB, $0.0104) t2.micro (1GB, 0GB, $0.012) g1-small (1.7GB, 0GB, $0.027)	Y
2	16 GB Memory 10 GB Storage	r5d.large (16GB, 75GB, $0.144) m5d.xlarge (16GB, 150GB, $0.226) m2.xlarge (17.1GB, 420GB, $0.245)	r5d.large (16GB, 75GB, $0.144) m5d.xlarge (16GB, 150GB, $0.226) m2.xlarge (17.1GB, 420GB, $0.245)	Y
3	25 GB Memory 10 GB Storage	n1-highmem-4 (26GB, 10GB, $0.2606) r5d.xlarge (32GB, 150GB, $0.305) i3.xlarge (30.5GB, 950 GB, 0.343)	n1-highmem-4 (26GB, 10GB, $0.2606) r5d.xlarge (32GB, 150GB, $0.305) i3.xlarge (30.5GB, 950 GB, 0.343)	Y
4	125 GB Memory 175 GB Storage	r5d.4xlarge (128GB, 600GB, $1.221) i3.4xlarge (125GB, 3800GB, $1.373) r3.4xlarge (125GB, 320GB, $1.463)	r5d.4xlarge (128GB, 600GB, $1.152) i3.4xlarge (125GB, 3800GB, $1.248) r3.4xlarge (125GB, 320GB, $1.33)	Y
5	500 GB Memory 500 GB Storage	x1.16xlarge (976GB, 1920GB, $6.669) x1e.8xlarge (960GB, 960GB, $6.672) r5d.24xlarge (768GB, 3600GB, $6.912)	x1.16xlarge (976GB, 1920GB, $6.669) x1e.8xlarge (960GB, 960GB, $6.672) r5d.24xlarge (768GB, 3600GB, $6.912)	Y

6	1000 GB Memory
	
	2000 GB Storage
		x1.32xlarge(1952 GB, 3840GB, $13.338)
		x1e.32xlarge(1952GB, 3840GB, $26.668)
			x1.32xlarge (blank GB, 3840GB, $13.338)
			x1e.32xlarge (blank GB, 3840GB, $26.668)
				Y

7	5000 GB Memory 0 GB Storage	None	None	Y
8	0 GB Memory 0 GB Storage	t3.nano (0.5GB, 0GB, $0.0052) t2.nano (0.5GB, 0GB, $0.0059) f1-micro(0.6GB, 0GB, $0.008)	t3.nano (0.5GB, 0GB, $0.0052) t2.nano (0.5GB, 0GB, $0.0059) f1-micro (0.6GB, 0GB, $0.008)	Y

Table 16: Results of find_instance_workload

Test #	Input	Expected	Actual	Success (Y/N)

1	“ml”	c5.large (4GB, 0GB, 0.085)
		c5d.large(4GB, 50GB, 0.096)
		c5.xlarge(8GB, 0GB, 0.17)
			c5.large (4GB, 0GB, 0.085)
			c5d.large(4GB, 50GB, 0.096)
			c5.xlarge(8GB, 0GB, 0.17)
				Y

2	“gp”	t3.nano(0.5GB, 0GB, $0.0052)
		t2.nano(0.5GB, 0GB, $0.0059)
		f1-micro(0.6GB, 0GB, $0.008)
			t3.nano (0.5GB, 0GB, $0.0052)
			t2.nano (0.5GB, 0GB, $0.0059)
			f1-micro (0.6GB, 0GB, $0.008)
				Y

3	“im”	n1-highmem-2 (13GB, 10GB, $0.126) r4.large(15.25GB, 0, $0.133) n1-highmem-4 (26GB, 10GB, $0.252)	n1-highmem-2 (13GB, 10GB, $0.126) r4.large(0, $0.133) n1-highmem-4 (26GB, 10GB, $0.252)	Y

The second iteration consisted of 8 tests for *find_instance* and 3 tests for *find_instance_workload*. Due to the updated pricing file, our results align with our expected values. The *find_instance* set of tests attempted to encompass a wide range of user defined inputs, including edge cases such as resources that were too large or too small. In addition, the tests included variations in storage options. The results showed limited options for GCP when storage was requested by the user. In test cases 2-6 from *Table 15*, only AWS instances were recommended for the user. However, when storage was not included in test cases 1 and 8, GCP often occupied at least one of the top three options.

Find_instance_workload consisted of 3 tests based on the possible input parameters. Due to the scope of our project, we only created three workload-specific options: general purpose, in-memory, and machine learning. The correctness is based on the cheapest options based on AWS and GCP optimized virtual machines for each of the three workloads. For example, GCP’s “*n1-highmem”* and AWS’ *“r4/r5”* group of virtual machines are optimized for in-memory workloads. Therefore, the cheapest options from that group will be returned to the user. The expected values and actual results matched for each workload test.

Caching Optimizations

During the creation of our system, we made several design decisions for improving the overall efficiency. Caching was an optimization decision that was made due to delayed connection times to each cloud provider when gathering virtual machine sizes. The goal was to only have to gather virtual machine information from the cloud providers every 10 days versus connecting each time a user makes a request. We gathered results for each function implementing caching and non-caching. We needed to gather results on both designs in order to analyze the overall cost savings. *Table 17* contains the first 15 test results for non-caching and *Table 18* contains the first 15 test results for caching. Full results can be found in *Appendix C.*

Table 17: Test Results for Caching

Test #	find_instance (ms)	find_instance_workload (ms)	Difference (ms)
1	59	52	7
2	53	53	0
3	63	52	11
4	59	53	6
5	61	51	10
6	53	52	1
7	65	53	12
8	57	53	4
9	60	51	9
10	51	53	2
11	62	53	9
12	57	53	4
13	60	52	8
14	52	53	1
15	64	55	9

Table 18: Test Results for Non-Caching

Test #	find_instance (ms)	find_instance_workload (ms)	Difference (ms)
1	1418	1179	239
2	1131	1389	258
3	1145	1207	62
4	1118	1248	130
5	1058	1248	190
6	1202	1051	151
7	1219	1124	95
8	1526	1203	323
9	1458	1046	412
10	1250	1224	26
11	1073	1170	97
12	1333	1412	79
13	1150	1384	234
14	1013	1282	269
15	1231	1574	343

After gathering the results of 200 test cases, we noticed that caching significantly improved the overall performance of *find_instance* and *find_instance_workload*. *Figure 13* shows the average runtime for each function when caching is implemented versus not implemented.

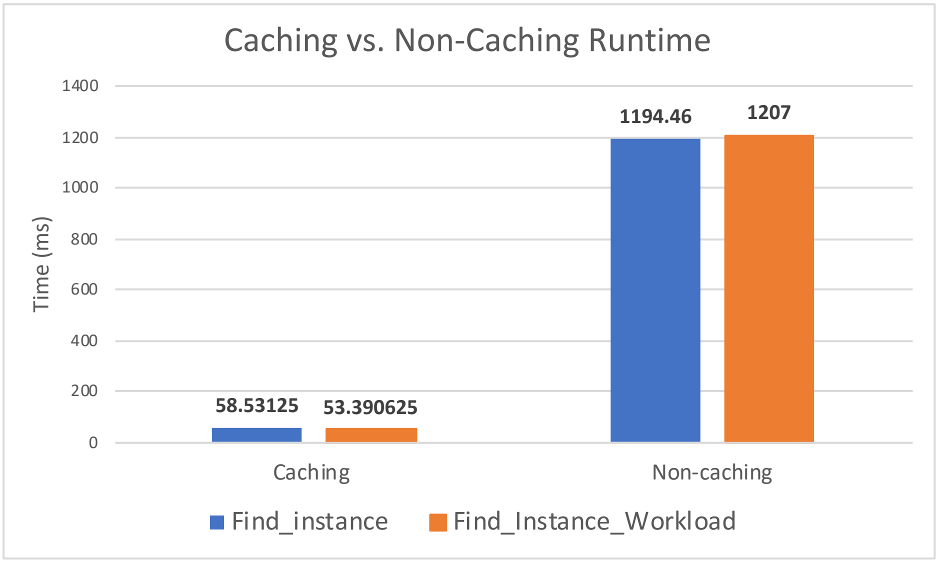


Figure 13: Caching vs. Non-Caching Runtime

According to the figure, non-caching results in an average of 1194.46 milliseconds for *find_instance* and 1207 milliseconds for *find_instance_workload*. When caching is implemented, there is a significant reduction in the average runtime for each function, resulting in 58.53 milliseconds and 53.39 milliseconds, respectively. We then concluded that implementing caching in our system significantly improved the efficiency of our system due to not having to connect to the cloud providers for each user request.

Measuring Deployment Time

Deployment time is critical to the functionality of our web interface. Since we are using two different cloud providers, it is our assumption that the total deployment time varies depending on each provider. In addition, we also need to know how long before the user can use the newly deployed resources. *Table 19* displays the first 15 test cases for measuring deployment runtime. The full set of tests can be found in *Appendix D*.

Table 19: Deployment Runtime of AWS and GCP

Test #	Milliseconds	Provider	Test #	Milliseconds	Provider
1	1370	AWS	1	25000	GCP
2	1692	AWS	2	24000	GCP
3	2081	AWS	3	20000	GCP
4	1661	AWS	4	15000	GCP
5	1409	AWS	5	18000	GCP
6	1514	AWS	6	20000	GCP
7	1253	AWS	7	15000	GCP
8	1385	AWS	8	19000	GCP
9	1864	AWS	9	18000	GCP
10	1398	AWS	10	20000	GCP
11	3386	AWS	11	32000	GCP
12	1460	AWS	12	20000	GCP
13	1843	AWS	13	18000	GCP
14	1289	AWS	14	17000	GCP
15	2312	AWS	15	25000	GCP

We created a unit testing Python program that calculated the runtime for deploying to AWS and GCP. We then collected the values and stored them in a table. Full results can be found in *Appendix E*. As shown in *Figure 14*, AWS deploys much faster than GCP.

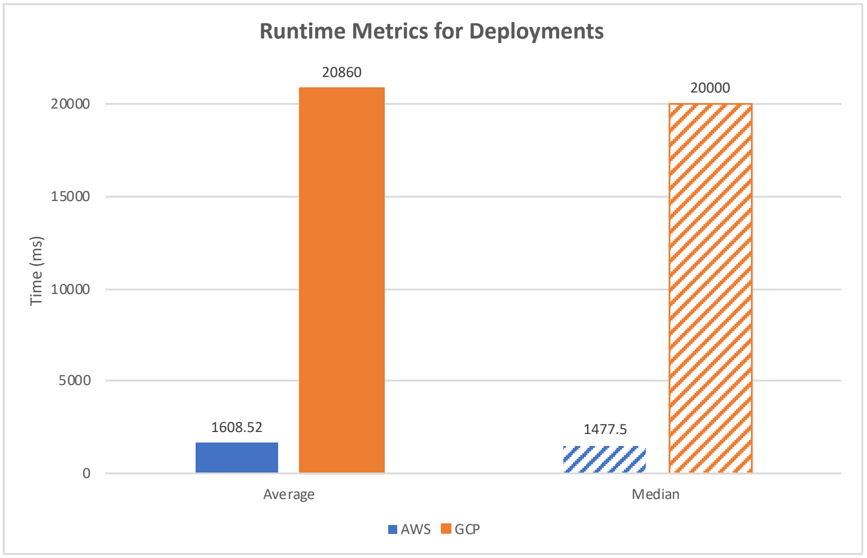


Figure 14: Runtime Metrics for Deployments

The average deployment time for GCP is 20,860 milliseconds compared to 1608.52 milliseconds for AWS. Deploying to AWS is much more efficient then deploying to GCP. However, this data does not illustrate the reason why GCP is significantly longer than AWS.

In order to better understand where the bottleneck was happening, we measured deployment time with the time until the instance is ready. When a user deploys to the cloud, there is a brief window of time in which the resources are being provisioned and initialized. We measured the time of each of these parts to see if the initialization time contributes to the gap in deployment times. Initialization time is synonymous with the provisioning time it takes for GCP and AWS to get the resources ready for the user.

Table 20: Deployment and Initialization Tests for AWS

Test #	Initialization Time (ms)	Deploy Time (ms)	Total (ms)
1	11972ms	1677	13649
2	8255	1115	9370
3	9315	1610	10925
4	10143	1504	11647
5	8102	1783	9885
6	9274	1216	10490
7	9224	2235	11459
8	7957	1356	9313
9	8348	1200	9548
10	8488	1215	9703
11	8272	1518	9790
12	7234	1996	9230
13	12740	1231	13971
14	10097	1321	11418
15	8735	1162	9897

Table 20 contains 15 of the test results for AWS deployment and initialization times. *Appendix F* has the full set of test results for AWS and GCP. After gathering results from 50 deployments to both cloud providers, we concluded that the reason GCP deployments are longer than AWS is due to the fact that the Apache Libcloud API includes initialization time in its deployment function for GCP.

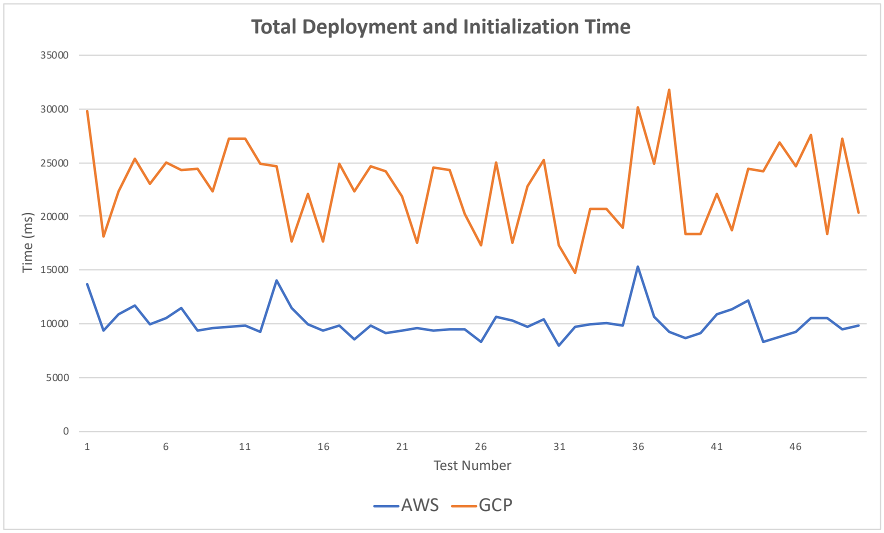


Figure 15: Total Deployment and Initialization Time

Figure 15 shows the total deployment time, including initialization time, for each of the 50 tests to AWS and GCP. Based on the data, the gap between deployment times is shortened when initialization time is taken into consideration. However, the average time still demonstrates that AWS is faster deploying and provisioning resources. *Figure 16* shows the average and median deployment and initialization times. The average difference between AWS and GCP is 12,683 milliseconds, or approximately 12.68 seconds.

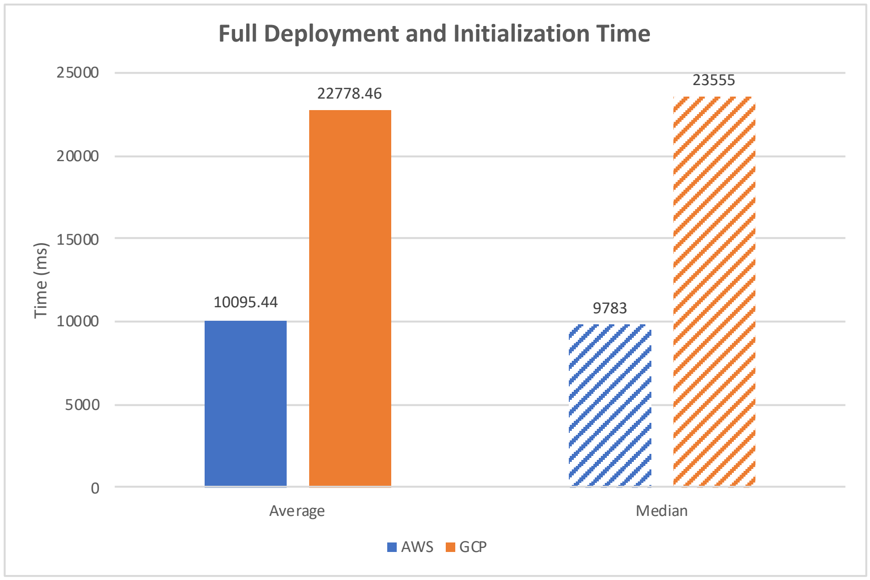


Figure 16: Full Deployment and Initialization Time

Resource Usage Analysis

We chose the workload-based launch of machine-learning in our system, filled in our workload information, and received the recommendation to launch an "*Amazon Base Deep Learning AMI"* with an instance type *"c5.large"*. The *"c5"* family of AWS EC2 instances are compute-optimized and better equipped to handle the large amounts of computing power many machine learning projects require. Furthermore, machine learning workloads typically require large datasets, so more memory is also required for optimal performance. In total, it took 2 steps to choose a recommended option.

This option included 2 virtual CPUs and 4 GB of memory to run a 12.9 KB program, while also storing the 11.59 MB dataset on the virtual machine. It cost $0.085 per hour. The analysis from AWS' built-in analytics tool, shown in *Figure 17* below, proves that this option was in fact the cheapest option that was capable of handling that workload.

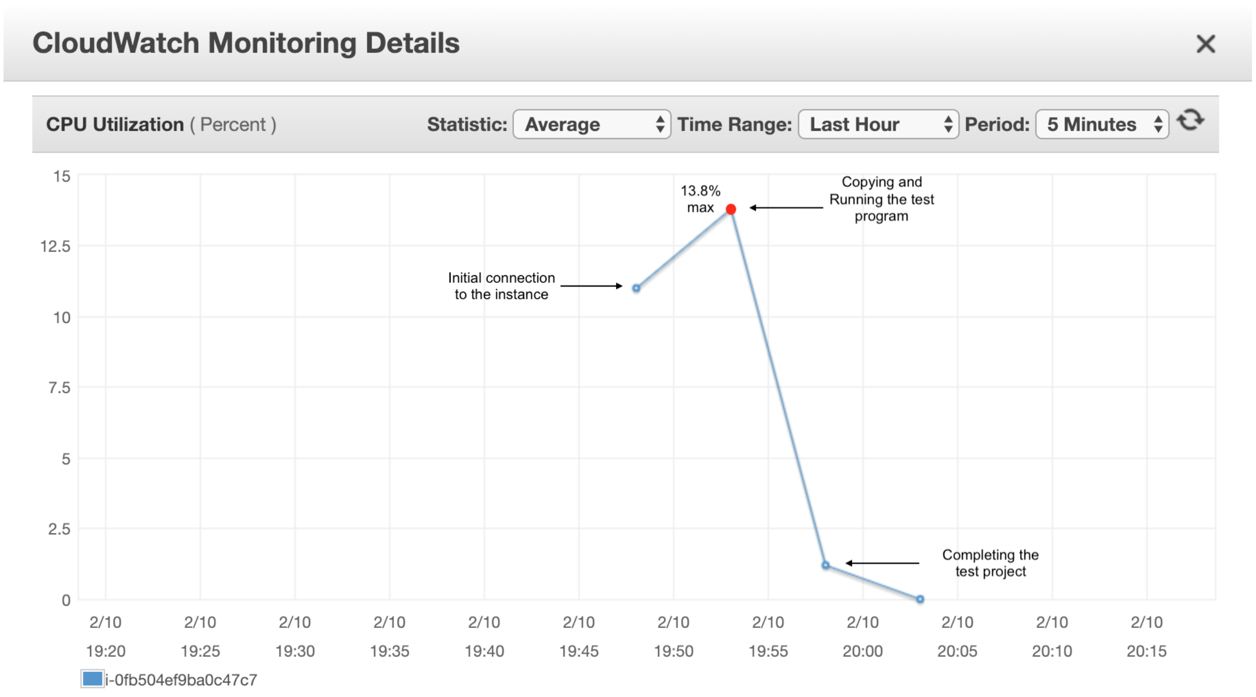
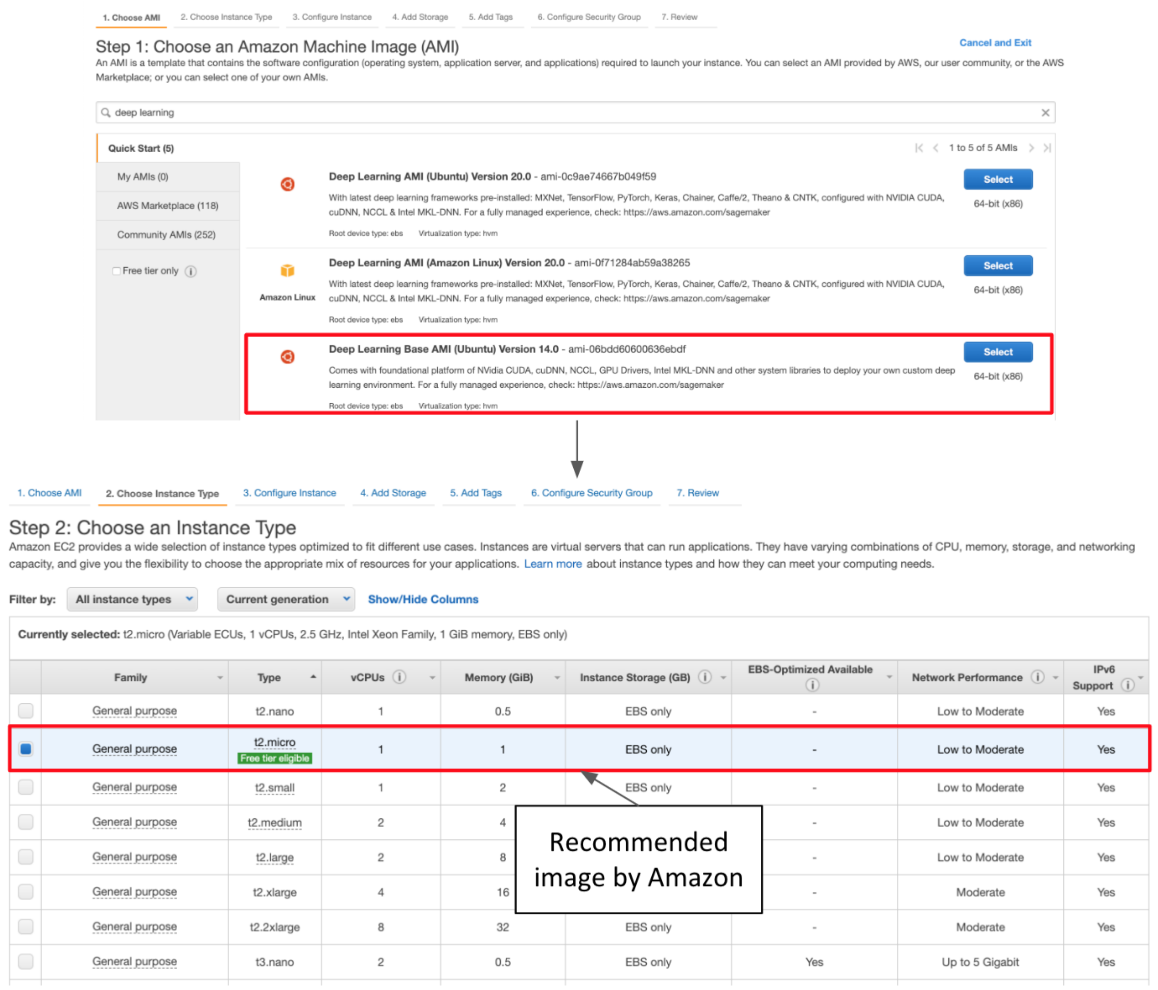


Figure 17: *Analysis from AWS after running the Tensorflow and Keras project on the platform-recommended image*

As shown in the figure above, the "*c5.large"* instance type utilized at most only 13.8% of its CPU to run and complete the testing program, and only required the instance to run for 20 minutes. Each of the steps took little time and effort, from the secure shell (SSH) connection to the secure copy protocol (SCP) of the test program from a local machine to the instance. In addition, there was no wait for the instance to be ready to run the program, and it finished within 5 minutes. The entire process of running a program on an instance launched from our platform was less than 1 hour.

However, on Amazon’s “Create Instance” dashboard it took 4 steps to create an instance.



Step 1: Choose an Amazon Machine Image (AMI)

An AMI is a template that contains the software configuration (operating system, application server, and applications) required to launch your instance. You can select an AMI provided by AWS, our user community, or the AWS Marketplace; or you can select one of your own AMIs.

deep learning

Quick Start (5) — 1 to 5 of 5 AMIs

My AMIs (0)
AWS Marketplace (118)
Community AMIs (252)
Free tier only

Deep Learning AMI (Ubuntu) Version 20.0 - ami-0c9ae74667b049f59
With latest deep learning frameworks pre-installed: MXNet, TensorFlow, PyTorch, Keras, Chainer, Caffe/2, Theano & CNTK, configured with NVIDIA CUDA, cuDNN, NCCL & Intel MKL-DNN. For a fully managed experience, check: https://aws.amazon.com/sagemaker
Root device type: ebs Virtualization type: hvm
64-bit (x86)

Amazon Linux
Deep Learning AMI (Amazon Linux) Version 20.0 - ami-0f71284ab59a38265
With latest deep learning frameworks pre-installed: MXNet, TensorFlow, PyTorch, Keras, Chainer, Caffe/2, Theano & CNTK, configured with NVIDIA CUDA, cuDNN, NCCL & Intel MKL-DNN. For a fully managed experience, check: https://aws.amazon.com/sagemaker
Root device type: ebs Virtualization type: hvm
64-bit (x86)

Deep Learning Base AMI (Ubuntu) Version 14.0 - ami-06bdd60600636ebdf
Comes with foundational platform of NVidia CUDA, cuDNN, NCCL, GPU Drivers, Intel MKL-DNN and other system libraries to deploy your own custom deep learning environment. For a fully managed experience, check: https://aws.amazon.com/sagemaker
Root device type: ebs Virtualization type: hvm
64-bit (x86)

Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. Learn more about instance types and how they can meet your computing needs.

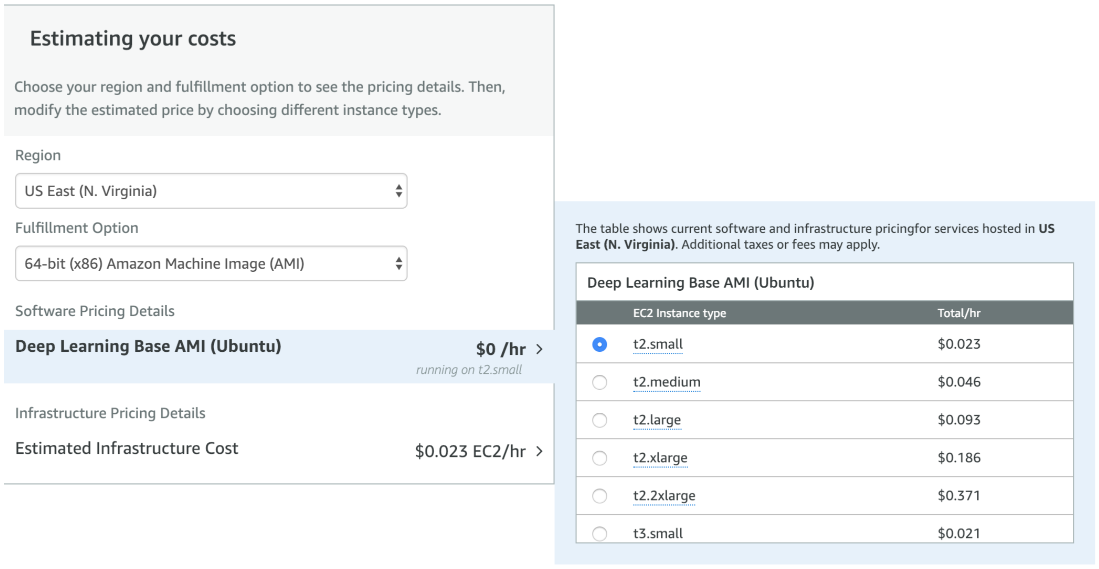
Filter by: All instance types | Current generation | Show/Hide Columns

Currently selected: t2.micro (Variable ECUs, 1 vCPUs, 2.5 GHz, Intel Xeon Family, 1 GiB memory, EBS only)

	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance	IPv6 Support
☐	General purpose	t2.nano	1	0.5	EBS only	-	Low to Moderate	Yes
■	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.small	1	2	EBS only	-	Low to Moderate	Yes
☐	General purpose	t2.medium	2	4		-	Low to Moderate	Yes
☐	General purpose	t2.large	2	8		-	Low to Moderate	Yes
☐	General purpose	t2.xlarge	4	16		-	Moderate	Yes
☐	General purpose	t2.2xlarge	8	32	EBS only	-	Moderate	Yes
☐	General purpose	t3.nano	2	0.5	EBS only	Yes	Up to 5 Gigabit	Yes

Figure 18: Launching an AWS Deep Learning AMI using AWS’ dashboards

Steps 1 and 2, pictured in *Figure 18* above, require entering a workload type and being prompted to use the recommended image from AWS’ marketplace. It is important to note that this selection screen does not display the price-per-hour for every instance. This makes it more difficult for the user to perform an effective cost-benefit analysis as they will have to look up the prices on a different page (*Figure 19* below).


Estimating your costs

Choose your region and fulfillment option to see the pricing details. Then, modify the estimated price by choosing different instance types.

Region: US East (N. Virginia)

Fulfillment Option: 64-bit (x86) Amazon Machine Image (AMI)

Software Pricing Details

Deep Learning Base AMI (Ubuntu) **$0 /hr** running on t2.small

Infrastructure Pricing Details

Estimated Infrastructure Cost $0.023 EC2/hr

The table shows current software and infrastructure pricingfor services hosted in **US East (N. Virginia)**. Additional taxes or fees may apply.

Deep Learning Base AMI (Ubuntu)

	EC2 Instance type	Total/hr
●	t2.small	$0.023
○	t2.medium	$0.046
○	t2.large	$0.093
○	t2.xlarge	$0.186
○	t2.2xlarge	$0.371
○	t3.small	$0.021

Figure 19: Step 3. Look up Amazon Marketplace Pricing Information

As shown in *Figure 19,* the recommended image "*t2.micro"* is not even on the pricing information on AWS Marketplace's listing for the Deep Learning Base AMI (Ubuntu). The closest instance type to the recommended option is a "*t2.small"* instance, costing $0.023 per hour. Step 4 requires going back to the console with this new information and finalizing the instance launch.

Once the deployment was chosen, running the TensorFlow and Keras project on the recommended instance proved to be difficult. Results are shown in *Figure 20* below.

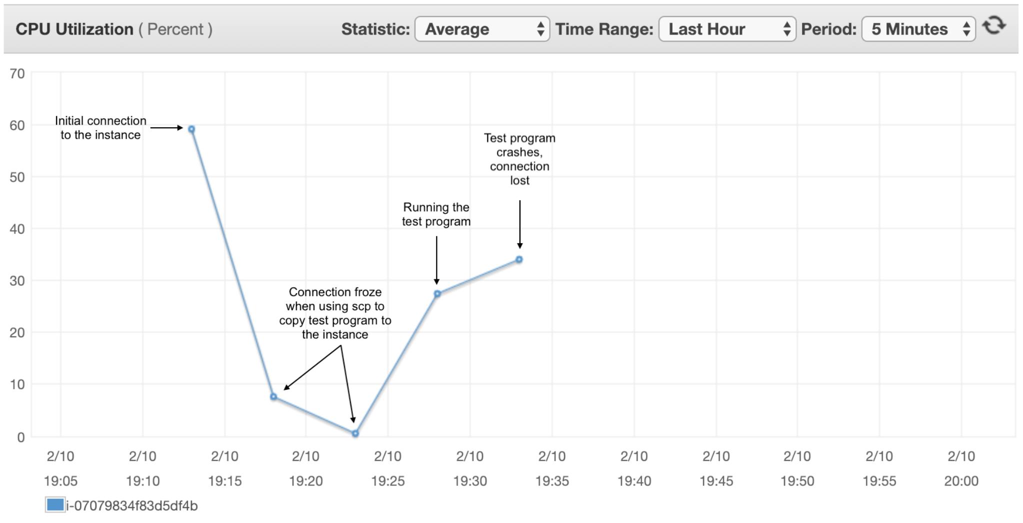


Figure 20: AWS analysis of TensorFlow and Keras project running on the recommended "t2.micro" instance

The instance struggled to connect via SSH and failed when copying the program from the local disk. When the program began to run, the instance started to lag, and eventually the instance crashed and the SSH connection was lost. Clearly, the AWS recommended image was not suited for the type of workload that was running, as the project failed to complete. A full comparison between our platform and the AWS console can be seen below in *Table 21*.

Table 21: Amazon's Recommendation vs. Platform Recommendation

Instance	**# Steps to Launch Instance**	**Total Time Running**	**Max CPU Utilization (%)**	**Completed Task?**	**Total Cost to Complete Task**
AWS Recommended t2.micro	4	20 minutes	59.14%	**No**	**N/A**
Multi-Cloud Recommended c5.large	2	20 minutes	13.8%	Yes	$0.028

III. Conclusion and Summary of Results

The goal of our project was to create a multi-cloud web interface that minimizes costs and provides optimizations for different types of workloads. Based on the results of our system, we achieved optimal costs for resource assignments. The correctness tests illustrated that based on custom resource provisioning options, our algorithm returned the optimal instance results. In addition, we were able to evaluate the efficiency of our system and deployment times from AWS and GCP. Finally, we were able to test specific workloads on recommended options of our system. The summary of results are as follows:

- Our algorithm for *find_instance* and *find_instance_workload* returned results that matched the expected values. Table XX and XY contain the full results of these tests.
- For default instances, AWS provides more options for storage than GCP.
- Caching significantly improved performance and reduced the average runtime by approximately 1,135.92 milliseconds for *find_instance* and 1,153.61 milliseconds for *find_instance_workload*. Refer to Figure XYZ for full results.
- AWS is significantly faster in deploying resources than GCP. The average deployment to AWS took 1,608.52 milliseconds, while the average to GCP took 20,860 milliseconds.
- For the Apache Libcloud API, GCP includes initialization time in its deployment function. The initialization time for GCP is 0 milliseconds, while the average initialization time for AWS is 8707.12 milliseconds.
- Our algorithm returned a more expensive instance per hour, but it was more efficient in completing the machine-learning workload task which reduced the overall cost of running the project.

Chapter 5: Recommendations and Future Work

The goal of this project was to optimize the cost of multi-cloud resources based on the needs of the user. We created a multi-cloud web interface that allowed the user to specify needed resources and recommended the best virtual machine options from two cloud providers. Based on the results, were able to successfully recommend the cheapest options to users and successfully deploy to the respective provider. However, there are a variety of engineering efforts that can be made to our system.

The first effort is collecting user credentials to the respective cloud provider in order to deploy resources to their specific account. Currently, our application acts as a proof of concept in which we demonstrate the flow of the system and deploy based on a common set of credentials. This method is not possible if the web interface were to be used commercially. Possible implementations include prompting user for credentials to cloud providers, using a token system, or instructing users how to create private keys with AWS and GCP.

The second engineering improvement to our system is to allow the user to choose an image for deployment. Currently, we have a set of basic images that are used based on the user-specified operating system. Different projects may require certain dependencies to be installed on the system, such as Keras or TensorFlow. AWS and GCP offer images with pre-installed packages that can be used for deployments. A possible implementation is to create a second page in which the user can choose specific dependencies, and the system then finds a possible image that complies with those requirements.

In addition to the potential engineering efforts, there are additional improvements that can be made to the system algorithm. The first improvement is integrating machine learning to the cost optimization algorithms. Machine learning can provide an accurate recommendation to the user. Choosing a model and training set can be difficult, which is why we recommend it as a potential direction for future project work.

The second recommended improvement to the system algorithm is to add a feedback option. A feedback option could range between a simple thumbs up or thumbs down button on recommendations to allowing the user to save favorite instances to their account. Feedback acts as a way to collect data from users, which in turn could be used to train the machine learning algorithm. Due to the complexity of implementing each of the engineering efforts and system algorithm improvements, we recommend these options for future project work.

In summary, we successfully created a multi-cloud prototype that achieved the goal of recommending the best price to the user. Our system provides a strong basis for future development. We hope that the system can be improved and utilized by users looking to find the best options for deploying to the cloud.

References

[1] L. Columbus, “83% Of Enterprise Workloads Will Be In The Cloud By 2020”, *Forbes*, Jan. 7, 2018. [Online], Available: https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/#596a74e46261. [Accessed: Sept. 9, 2018]

[2] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," 2011.

[3] N. B. Ruparelia, *Cloud Computing.* Cambridge, MA: The MIT Press, 2016.

[4] Microsoft, “What is a Virtual Machine and How Does it Work,” Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/. [Accessed: Sept. 9, 2018].

[5] Hypervisor Market – Latest Trends and Key Drivers Supporting Growth Through 2025 Research Report By Application, Products Research Industry Analysis, Growth, Size, Share, Trends, Forecast to 2025,” Latest Industry News, Aug. 6, 2018. [Online]. Available: https://www.latestindustrynews.com/11491/hypervisor-market-latest-trends-and-key-drivers-supporting-growth-through-2025-research-report-by-application-products-research-industry-analysis-growth-size-share-trends-forecast-to-2/. [Accessed: Sept. 9, 2018]

[6] Containers: R. Shapland, “Cloud containers -- what they are and how they work,” SearchCloudSecurity. [Online]. Available: https://searchcloudsecurity.techtarget.com/feature/Cloud-containers-what-they-are-and-how-they-work. [Accessed: Sept. 9, 2018].

[7] S. J. Vaughan-Nichols, “What is Docker and why is it so darn popular?,” *ZDNet*, 21-Mar-2018. [Online]. Available: https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/. [Accessed: 10-Sep-2018].

[8] Technopedia, “Workload,” *Technopedia*, 2018. [Online]. Available: https://www.techopedia.com/definition/13544/workload [Accessed: Sept. 13, 2018]

[9] Apprenda. (2018). Deployment to the Cloud. [Online] Available at: https://apprenda.com/library/cloud/deployment-to-the-cloud [Accessed Sept. 9, 2018].

[10] Vold, N. (2018). *Cloud basics – Deployment models*. [Online] Visma Corporate Blog. Available at: https://www.visma.com/blog/cloud-basics-deployment-models/ [Accessed 9 Sept. 2018].

[11] Microsoft Azure, “What is a cloud service provider?”, *Microsoft Azure.* [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-a-cloud-provider/. [Accessed: Sept. 13, 2018]

[12] Z. Mahmood, Cloud Computing; Challenges, Limitations and R&D Solutions. (2014th ed.) Cham: Springer International Publishing, 2014.

[13] M. Heusser, “30 essential container technology tools and resources,” *TechBeacon*, 31-Jul-2018. [Online]. Available: https://techbeacon.com/30-essential-container-technology-tools-resources. [Accessed: 15-Sep-2018].

[14] C. Matsumoto, “Why Docker and Google Kubernetes Are Like PaaS Done Right,” *SDxCentral*, Aug. 17, 2015. [Online]. Available: https://www.sdxcentral.com/articles/news/why-docker-and-google-kubernetes-are-like-paas-done-right/2015/08/. [Accessed: Sept. 15, 2018].

[15] “Why Docker,” *Docker*, 11-Sep-2018. [Online]. Available: https://www.docker.com/why-docker. [Accessed: 15-Sep-2018].

[16] “4 Reasons Why Kubernetes Is Hot,” *Network Computing*, 18-Jun-2018. [Online]. Available: https://www.networkcomputing.com/data-centers/4-reasons-why-kubernetes-hot/704546519. [Accessed: 15-Sep-2018].

[17] S. McCarty and IDG Contributor Network, “What you need to know (now) about container standards,” *InfoWorld*, Nov. 16, 2017. [Online]. Available: https://www.infoworld.com/article/3237645/containers/what-you-need-to-know-now-about-container-standards.html. [Accessed: Sept. 15, 2018].

[18] R. Aboukhalil, “A Tale of Two Clouds: Amazon vs. Google” *Medium*, March 13, 2017. [Online]. Available: https://medium.com/@robaboukhalil/a-tale-of-two-clouds-amazon-vs-google-4f2520516a38. [Accessed: Sept. 15, 2018].

[19] Amazon, “AWS Pricing,” *Amazon Web Services,* 2018. [Online]. Available: https://aws.amazon.com/pricing/. [Accessed: Sept. 14, 2018].

[20] C. Baun, M. Kunze, J. Nimis, and S. Tai, *Cloud Computing Web-Based Dynamic IT Services*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[21] Google, “About the GCP Services,” *Google Cloud*, June 28, 2018. [Online]. Available: https://cloud.google.com/docs/overview/cloud-platform-services. [Accessed: Sept. 15, 2018].

[22] Google, “GCP Pricing”, *Google Cloud,* 2018. [Online]. Available: https://cloud.google.com/pricing/. [Accessed: Sept. 14, 2018].

[23] D. Smith and M. Govekar, "Market guide for cloud management platforms," Gartner, Inc, April 25,. 2017.

[24] Cisco, “Cisco CloudCenter Solution Use Case: Hybrid IT as a Service,” *Cisco*, 2017. [PDF]. Available: https://www.cisco.com/c/dam/en/us/products/collateral/cloud-systems-management/cloudcenter/solution-overview-c22-737217.pdf. [Accessed: Aug. 28, 2018].

[25] Hashicorp. Introduction to Terraform. Available: https://www.terraform.io/intro/index.html.

[26] A. Shaw, "Multi-cloud, what are the options? Part 1- Low level abstraction libraries," Sept. 18, 2016. [Accessed: Aug. 28, 2018].

[27] The Apache Software Foundation. Apache Libcloud. Available: https://libcloud.readthedocs.io/en/latest/compute/index.html.

[28] W. Mulia, N. Sehgal, S. Sahoni, J. Acken, C. Stanberry, & D. Fritz, “Cloud Workload Characterization,” in *IETE Technical Review,* vol. 30, no. 5, p. 382-397, Sept. 1, 2014. [Online]. Available: https://doi.org/10.4103/0256-4602.123121. [Accessed: Sept. 17, 2018].

[29] M. Guignard, M. Schild, C. Bederián, N. Wolovick, & A. Vega, “Performance Characterization of State-Of-The-Art Deep Learning Workloads on an IBM Minsky Platform,” in *Frontiers in AI and Software Engineering,* 2018. [Online]. Available at: http://hdl.handle.net/10125/50591. [Accessed: Sept. 18, 2018].

[30] Amazon, “Amazon EC2 Instance Types,” *Amazon Web Services,* 2018. [Online]. Available: https://aws.amazon.com/ec2/instance-types/. [Accessed: Sept. 14, 2018].

[31] Google, “Machine Types”, *Google Cloud*, Aug. 3, 2018. [Online]. Available: https://cloud.google.com/compute/docs/machine-types. [Accessed: Sept. 14, 2018].

[32] S. Singh, I. Chana, “Cloud resource provisioning: survey, status and future research directions,” *Knowledge and Information Systems*, vol. 49, no. 3, p. 1005–1069, Dec. 2016. [Online]. Available: https://doi.org/10.1007/s10115-016-0922-3

[33] O. Alipourfard, H. Liu, J. Chen, S. Venkartaraman, M. Yu, M. Zhang, “CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics,” in *Proceedings of the*

14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17), Boston, MA, March 27-29, 2017. [Online]. Available: https://www.usenix.org/system/files/conference/nsdi17/nsdi17-alipourfard.pdf

[34] J. Gonzalez, B. Hariharan, R. Katz, B. Smith, N. Yadwadkar, "Selecting the Best VM across Multiple Public Clouds: A Data-Driven Performance Modeling Approach," in *Proceedings of SoCC '17*, Santa Clara, CA, Sept. 24–27, 2017. [Online]. Available: https://doi.org/10.1145/3127479.3131614

[35] A. Dix, *Human-computer interaction*. Harlow: Pearson Prentice-Hall, 2011. [PDF].

Appendix A

```
# ----------------------------------------------------------------------------------
#                               FIND_INSTANCE
# ----------------------------------------------------------------------------------
find_instance_results = []

# TEST 1
start = datetime.now()
instance, top_three, valid_instances = find_instance(0, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print("-------------------------")
print("                    FIND_INSTANCE              ")
print("-------------------------")
print(" -------- TEST 1 --------")
print("Total time: " + str(final))



# TEST 2
start = datetime.now()
instance, top_three, valid_instances = find_instance(16000, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 2 --------")
print("Total time: " + str(final))


# TEST 3
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 3 --------")
print("Total time: " + str(final))


# TEST 4
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 4 --------")
print("Total time: " + str(final))
```

```
# TEST 5
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 5 --------")
print("Total time: " + str(final))


# TEST 6
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 6 --------")
print("Total time: " + str(final))


# TEST 7
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 7 --------")
print("Total time: " + str(final))


# TEST 8
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 8 --------")
print("Total time: " + str(final))


# TEST 9
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 9 --------")
print("Total time: " + str(final))
```

```
# TEST 10
start = datetime.now()
instance, top_three, valid_instances = find_instance(800, 0)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_instance_results.append(final)
print(" -------- TEST 10 --------")
print("Total time: " + str(final))


# ---------------------------------------------------------------------------------
#                             FIND_INSTANCE_WORKLOAD
# ---------------------------------------------------------------------------------
find_workload_results = []

# TEST 1
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print("--------------------------")
print("                  FIND_WORKLOAD           ")
print("--------------------------")

print(" -------- TEST 1 --------")
print("Total time: " + str(final))


# TEST 2
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 2 --------")
print("Total time: " + str(final))


# TEST 3
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 3 --------")
print("Total time: " + str(final))


# TEST 4
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
```

```
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 4 --------")
print("Total time: " + str(final))


# TEST 5
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 5 --------")
print("Total time: " + str(final))


# TEST 6
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 6 --------")
print("Total time: " + str(final))


# TEST 7
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 7 --------")
print("Total time: " + str(final))

# TEST 8
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 8 --------")
print("Total time: " + str(final))


# TEST 9
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
```

```
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 9 --------")
print("Total time: " + str(final))


# TEST 10
start = datetime.now()
instance, top_three, valid_instances = find_instance_workload('ml')
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
find_workload_results.append(final)
print(" -------- TEST 10 --------")
print("Total time: " + str(final))


print("--------------------------")
print("                  FINAL RESULTS                ")
print("--------------------------")
print(find_instance_results)
print(find_workload_results)
```

Appendix B

```
# ---------------------------------------------------------------------------------
#                                   FULL DEPLOYMENT TEST
# ---------------------------------------------------------------------------------


# ---------------------------------------------------------------------------------
#                               AWS DEPLOYMENT TESTS
# ---------------------------------------------------------------------------------
print("--------------------------")
print("    AWS Deployments              ")
print("--------------------------")

full_aws_ids = []

# TEST 1
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test1')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)

print(" -------- TEST 1 --------")
print("Total time for deployment: " + str(final))


# TEST 2
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test2')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 2 --------")
print("Total time for deployment: " + str(final))

# TEST 3
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test3')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 3 --------")
print("Total time for deployment: " + str(final))

# TEST 4
```

```
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test4')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 4 --------")
print("Total time for deployment: " + str(final))

# TEST 5
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test5')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 5 --------")
print("Total time for deployment: " + str(final))

# TEST 6
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test6')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 6 --------")
print("Total time for deployment: " + str(final))

# TEST 7
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test7')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 7 --------")
print("Total time for deployment: " + str(final))

# TEST 8
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test8')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
```

```
print(" -------- TEST 8 --------")
print("Total time for deployment: " + str(final))

# TEST 9
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test9')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 9 --------")
print("Total time for deployment: " + str(final))

# TEST 10
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[0]
deployed_node = deployment(node, 'rhel', 'test10')
full_aws_ids.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
print(" -------- TEST 10 --------")
print("Total time for deployment: " + str(final))


for i in full_aws_ids:
        destroy("Amazon", i)

# ---------------------------------------------------------------------------------

# ---------------------------------------------------------------------------------
#                                   GCP DEPLOYMENT TEST
# ---------------------------------------------------------------------------------
print("--------------------------")
print("    GCE Deployments               ")
print("--------------------------")

full_node_list = []

# TEST 1
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test1')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
        user_node = get_node("Google", deployed_node.name)
        flag = 2
```

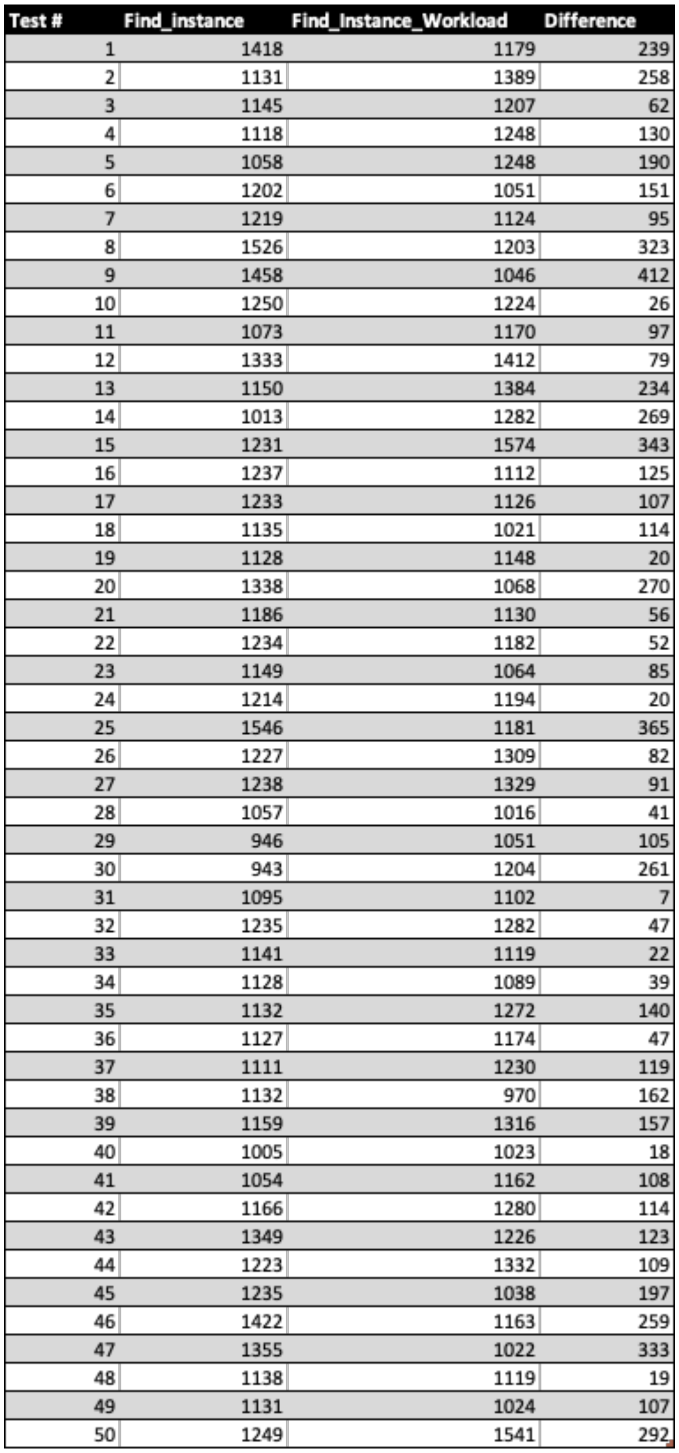
```
if flag == 1:
        ready_time = final
else:
        final2 = datetime.now().second
        ready_time = final2 - final
print(" -------- TEST 1 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 2
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test2')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
        user_node = get_node("Google", deployed_node.name)
        flag = 2
if flag == 1:
        ready_time = final
else:
        final2 = datetime.now().second
        ready_time = final2 - final
print(" -------- TEST 2 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 3
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test3')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
        user_node = get_node("Google", deployed_node.name)
        flag = 2
if flag == 1:
        ready_time = final
else:
        final2 = datetime.now().second
        ready_time = final2 - final
print(" -------- TEST 3 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 4
```

```
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test4')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
        user_node = get_node("Google", deployed_node.name)
        flag = 2
if flag == 1:
        ready_time = final
else:
        final2 = datetime.now().second
        ready_time = final2 - final
print(" -------- TEST 4 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 5
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test5')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
        user_node = get_node("Google", deployed_node.name)
        flag = 2
if flag == 1:
        ready_time = final
else:
        final2 = datetime.now().second
        ready_time = final2 - final
print(" -------- TEST 5 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 6
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test6')
full_node_list.append(deployed_node)

end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
```

```
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
		user_node = get_node("Google", deployed_node.name)
		flag = 2
if flag == 1:
		ready_time = final
else:
		final2 = datetime.now().second
		ready_time = final2 - final
print(" -------- TEST 6 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 7
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test7')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
		user_node = get_node("Google", deployed_node.name)
		flag = 2
if flag == 1:
		ready_time = final
else:
		final2 = datetime.now().second
		ready_time = final2 - final
print(" -------- TEST 7 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 8
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test8')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
		user_node = get_node("Google", deployed_node.name)
		flag = 2
if flag == 1:
		ready_time = final
else:
		final2 = datetime.now().second
		ready_time = final2 - final
print(" -------- TEST 8 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 9
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test9')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
        user_node = get_node("Google", deployed_node.name)
        flag = 2
if flag == 1:
        ready_time = final
else:
        final2 = datetime.now().second
        ready_time = final2 - final
print(" -------- TEST 9 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

# TEST 10
start = datetime.now()
instance_options = find_instance(800, 0)
node = instance_options[1][2]
deployed_node = deployment(node, 'linux', 'test10')
full_node_list.append(deployed_node)
end = datetime.now()
delta = end - start
final = int(delta.total_seconds() * 1000)
flag = 1
user_node = get_node("Google", deployed_node.name)
while user_node.state == 'pending':
        user_node = get_node("Google", deployed_node.name)
        flag = 2
if flag == 1:
        ready_time = final
else:
        final2 = datetime.now().second
        ready_time = final2 - final
print(" -------- TEST 10 --------")
print("Total time: " + str(final))
print("Time until ready: " + str(ready_time))

gce_destroy_all(full_node_list)
# ---------------------------------------------------------------------------------
```

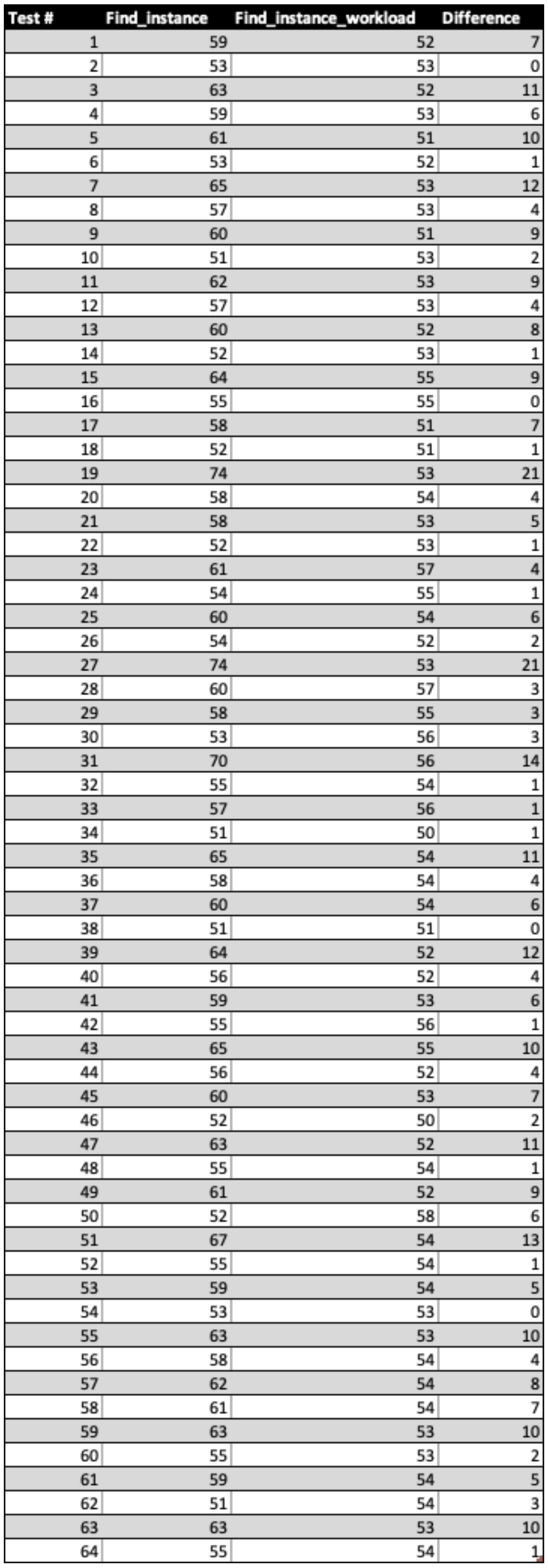
Appendix C

Non-Caching



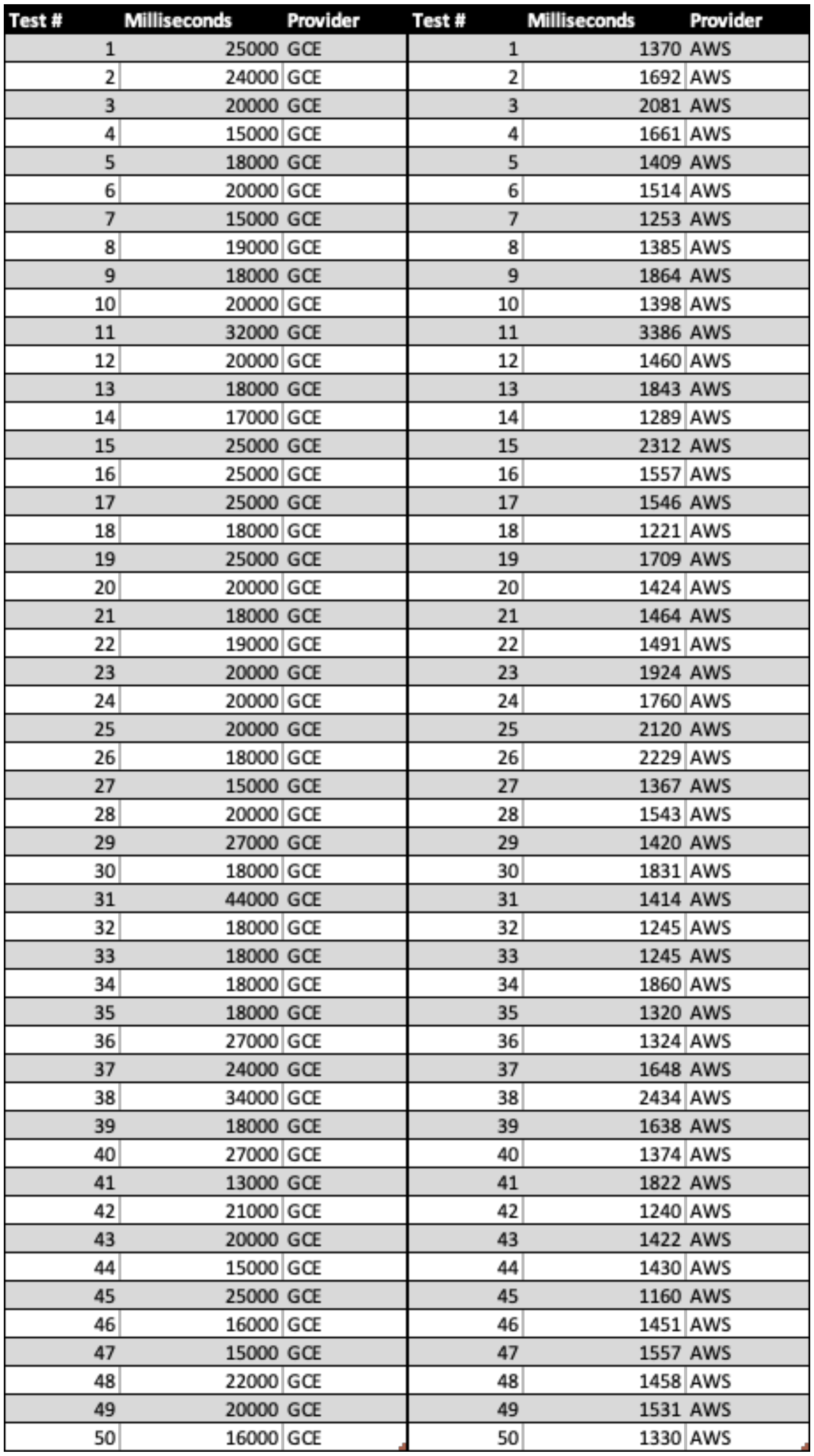
Test #	Find_instance	Find_Instance_Workload	Difference
1	1418	1179	239
2	1131	1389	258
3	1145	1207	62
4	1118	1248	130
5	1058	1248	190
6	1202	1051	151
7	1219	1124	95
8	1526	1203	323
9	1458	1046	412
10	1250	1224	26
11	1073	1170	97
12	1333	1412	79
13	1150	1384	234
14	1013	1282	269
15	1231	1574	343
16	1237	1112	125
17	1233	1126	107
18	1135	1021	114
19	1128	1148	20
20	1338	1068	270
21	1186	1130	56
22	1234	1182	52
23	1149	1064	85
24	1214	1194	20
25	1546	1181	365
26	1227	1309	82
27	1238	1329	91
28	1057	1016	41
29	946	1051	105
30	943	1204	261
31	1095	1102	7
32	1235	1282	47
33	1141	1119	22
34	1128	1089	39
35	1132	1272	140
36	1127	1174	47
37	1111	1230	119
38	1132	970	162
39	1159	1316	157
40	1005	1023	18
41	1054	1162	108
42	1166	1280	114
43	1349	1226	123
44	1223	1332	109
45	1235	1038	197
46	1422	1163	259
47	1355	1022	333
48	1138	1119	19
49	1131	1024	107
50	1249	1541	292

Caching



Test #	Find_instance	Find_instance_workload	Difference
1	59	52	7
2	53	53	0
3	63	52	11
4	59	53	6
5	61	51	10
6	53	52	1
7	65	53	12
8	57	53	4
9	60	51	9
10	51	53	2
11	62	53	9
12	57	53	4
13	60	52	8
14	52	53	1
15	64	55	9
16	55	55	0
17	58	51	7
18	52	51	1
19	74	53	21
20	58	54	4
21	58	53	5
22	52	53	1
23	61	57	4
24	54	55	1
25	60	54	6
26	54	52	2
27	74	53	21
28	60	57	3
29	58	55	3
30	53	56	3
31	70	56	14
32	55	54	1
33	57	56	1
34	51	50	1
35	65	54	11
36	58	54	4
37	60	54	6
38	51	51	0
39	64	52	12
40	56	52	4
41	59	53	6
42	55	56	1
43	65	55	10
44	56	52	4
45	60	53	7
46	52	50	2
47	63	52	11
48	55	54	1
49	61	52	9
50	52	58	6
51	67	54	13
52	55	54	1
53	59	54	5
54	53	53	0
55	63	53	10
56	58	54	4
57	62	54	8
58	61	54	7
59	63	53	10
60	55	53	2
61	59	54	5
62	51	54	3
63	63	53	10
64	55	54	1

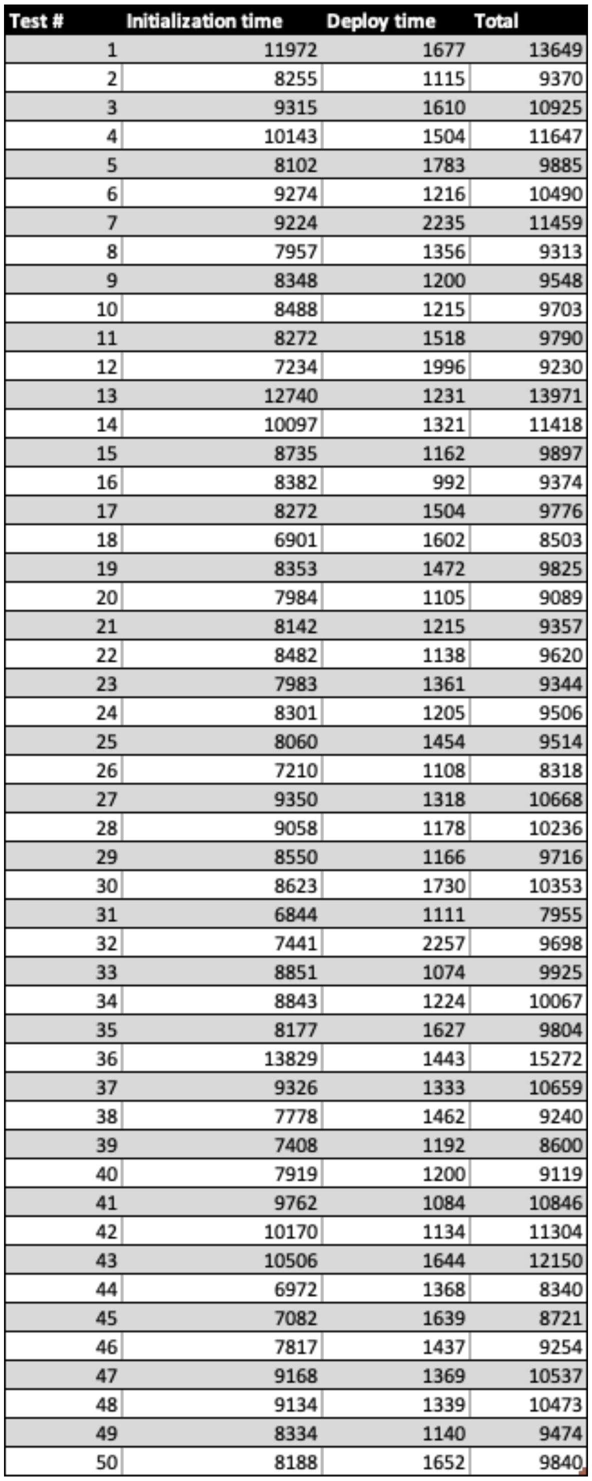
Appendix D



Test #	Milliseconds	Provider	Test #	Milliseconds	Provider
1	25000	GCE	1	1370	AWS
2	24000	GCE	2	1692	AWS
3	20000	GCE	3	2081	AWS
4	15000	GCE	4	1661	AWS
5	18000	GCE	5	1409	AWS
6	20000	GCE	6	1514	AWS
7	15000	GCE	7	1253	AWS
8	19000	GCE	8	1385	AWS
9	18000	GCE	9	1864	AWS
10	20000	GCE	10	1398	AWS
11	32000	GCE	11	3386	AWS
12	20000	GCE	12	1460	AWS
13	18000	GCE	13	1843	AWS
14	17000	GCE	14	1289	AWS
15	25000	GCE	15	2312	AWS
16	25000	GCE	16	1557	AWS
17	25000	GCE	17	1546	AWS
18	18000	GCE	18	1221	AWS
19	25000	GCE	19	1709	AWS
20	20000	GCE	20	1424	AWS
21	18000	GCE	21	1464	AWS
22	19000	GCE	22	1491	AWS
23	20000	GCE	23	1924	AWS
24	20000	GCE	24	1760	AWS
25	20000	GCE	25	2120	AWS
26	18000	GCE	26	2229	AWS
27	15000	GCE	27	1367	AWS
28	20000	GCE	28	1543	AWS
29	27000	GCE	29	1420	AWS
30	18000	GCE	30	1831	AWS
31	44000	GCE	31	1414	AWS
32	18000	GCE	32	1245	AWS
33	18000	GCE	33	1245	AWS
34	18000	GCE	34	1860	AWS
35	18000	GCE	35	1320	AWS
36	27000	GCE	36	1324	AWS
37	24000	GCE	37	1648	AWS
38	34000	GCE	38	2434	AWS
39	18000	GCE	39	1638	AWS
40	27000	GCE	40	1374	AWS
41	13000	GCE	41	1822	AWS
42	21000	GCE	42	1240	AWS
43	20000	GCE	43	1422	AWS
44	15000	GCE	44	1430	AWS
45	25000	GCE	45	1160	AWS
46	16000	GCE	46	1451	AWS
47	15000	GCE	47	1557	AWS
48	22000	GCE	48	1458	AWS
49	20000	GCE	49	1531	AWS
50	16000	GCE	50	1330	AWS

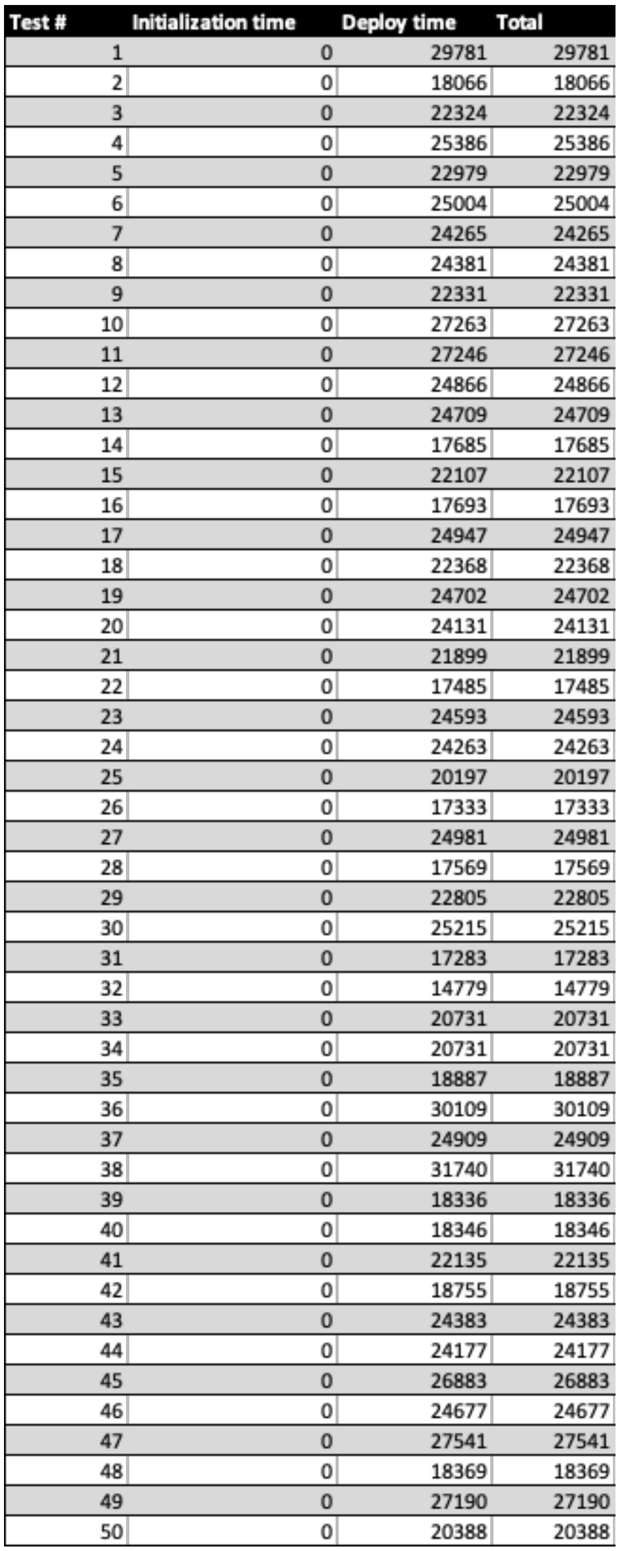
Appendix F

AWS



Test #	Initialization time	Deploy time	Total
1	11972	1677	13649
2	8255	1115	9370
3	9315	1610	10925
4	10143	1504	11647
5	8102	1783	9885
6	9274	1216	10490
7	9224	2235	11459
8	7957	1356	9313
9	8348	1200	9548
10	8488	1215	9703
11	8272	1518	9790
12	7234	1996	9230
13	12740	1231	13971
14	10097	1321	11418
15	8735	1162	9897
16	8382	992	9374
17	8272	1504	9776
18	6901	1602	8503
19	8353	1472	9825
20	7984	1105	9089
21	8142	1215	9357
22	8482	1138	9620
23	7983	1361	9344
24	8301	1205	9506
25	8060	1454	9514
26	7210	1108	8318
27	9350	1318	10668
28	9058	1178	10236
29	8550	1166	9716
30	8623	1730	10353
31	6844	1111	7955
32	7441	2257	9698
33	8851	1074	9925
34	8843	1224	10067
35	8177	1627	9804
36	13829	1443	15272
37	9326	1333	10659
38	7778	1462	9240
39	7408	1192	8600
40	7919	1200	9119
41	9762	1084	10846
42	10170	1134	11304
43	10506	1644	12150
44	6972	1368	8340
45	7082	1639	8721
46	7817	1437	9254
47	9168	1369	10537
48	9134	1339	10473
49	8334	1140	9474
50	8188	1652	9840

GCP



Test #	Initialization time	Deploy time	Total
1	0	29781	29781
2	0	18066	18066
3	0	22324	22324
4	0	25386	25386
5	0	22979	22979
6	0	25004	25004
7	0	24265	24265
8	0	24381	24381
9	0	22331	22331
10	0	27263	27263
11	0	27246	27246
12	0	24866	24866
13	0	24709	24709
14	0	17685	17685
15	0	22107	22107
16	0	17693	17693
17	0	24947	24947
18	0	22368	22368
19	0	24702	24702
20	0	24131	24131
21	0	21899	21899
22	0	17485	17485
23	0	24593	24593
24	0	24263	24263
25	0	20197	20197
26	0	17333	17333
27	0	24981	24981
28	0	17569	17569
29	0	22805	22805
30	0	25215	25215
31	0	17283	17283
32	0	14779	14779
33	0	20731	20731
34	0	20731	20731
35	0	18887	18887
36	0	30109	30109
37	0	24909	24909
38	0	31740	31740
39	0	18336	18336
40	0	18346	18346
41	0	22135	22135
42	0	18755	18755
43	0	24383	24383
44	0	24177	24177
45	0	26883	26883
46	0	24677	24677
47	0	27541	27541
48	0	18369	18369
49	0	27190	27190
50	0	20388	20388