A Hybrid Cloud Kubernetes Scheduler for Machine Learning Workloads

by

James Kieley

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved July 2021 by the
Graduate Supervisory Committee:

Ming Zhao, Chair
Dijiang Huang
Jia Zou

ARIZONA STATE UNIVERSITY

August 2021

ABSTRACT

Demand for processing machine learning workloads has grown incredibly over the past few years. Kubernetes, an open-source container orchestrator, has been widely used by public and private cloud providers for building scalable systems for meeting this demand. The data used to train machine learning workloads can be sensitive in nature, and organizations may prefer to be responsible for their data security and governance by housing it on on-premises systems. Hybrid cloud gives organizations the flexibility to use both on-premises and cloud infrastructure together, leveraging the advantages of both. While there is a long list of benefits, Kubernetes has limitations by design that limit a user's abilities in a hybrid cloud environment.

The Kubernetes control plane does not allow for the management of worker nodes across cloud providers. This boundary puts new responsibilities on the end-user when deploying a hybrid cloud workload. The end-user must create their clusters and specify which cluster the workload will be scheduled to ahead of time. The Kubernetes scheduler will not take the capacity of another cluster into account. To address these limitations, this thesis presents a new hybrid cloud Kubernetes scheduler that can create new clusters on-demand and burst machine learning workloads to a public cloud when on-premises resources are insufficient.

Workloads begin scheduling on an on-premises Kubernetes cluster. When the on-premises cluster's capacity is exhausted, a new Kubernetes cluster is created on-demand in a public cloud provider, and machine learning tasks waiting in the Kubernetes scheduling queue are dynamically migrated to the public cloud provider's Kubernetes cluster. The public Kubernetes cluster is dynamically sized and auto scaled based on the pending tasks' demand. When migrating tasks, the data dependencies among tasks are considered, and a region is dynamically chosen to reduce migration time and cost.

The scheduler is experimentally evaluated with real-world machine learning workloads, including predicting if a subscriber will stay with a subscription service, predicting the discount needed to retain a subscription customer, predicting if a credit card transaction is fraudulent, and simulated real-world job arrival behavior in a real hybrid cloud environment. Results show that the scheduler can substantially reduce the workload execution time by dynamically migrating tasks from on-premises to public cloud and minimizing the cost by dynamically sizing and scaling the public cluster.

DEDICATION

I dedicate this thesis work to my family. My loving wife, Lara Kieley who patiently supported me throughout the entire learning process. My children expressed love and praise during challenging moments. I am grateful for my parents, James and Gina Kieley, and grandparents John and Candy Kieley, who encouraged me to pursue postgraduate education.

ACKNOWLEDGMENTS

I thank NortonLifelock for their financial support throughout my master's program. Special thanks to Alex Tran and Maria Dossin from NortonLifelock, who supplied machine learning workloads. I acknowledge my committee chair, Professor Ming Zhao, for helping me at every stage of this thesis.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

With the rise of big data, machine learning, and cloud computing in the past decade, the amount of data collected and the sensitive nature of that data have increased. The multi-tenanted nature of the public cloud poses new security and privacy risks. In the past, researchers were able to exploit the multi-tenanted nature of a cloud provider by observing and predicting scheduling behavior. They then were able to successfully deploy an application on the same physical hardware as their attack target. Finally, they were able to extract private data from their attack target, despite the virtualization layers in-between the two applications Huang and Du (2014). When an organization decides to process their data using a public cloud provider, they entrust their data security, privacy, and governance to that cloud provider. Some organizations may prefer to own those responsibilities. Hybrid cloud gives organizations the flexibility to own those responsibilities for the data they consider most sensitive while still leveraging the public cloud's elasticity.

Containers changed software forever by providing a new layer of virtualization. This thin, inexpensive virtualization layer allows for more containers to be run on the same hardware than traditional virtual machines. With this increased software density, having many containers run in the same environment created a need for a new layer of management. Kubernetes addresses this need. Kubernetes provides APIs to deploy and manage many containers. It is extensible and encourages infrastructure and platforms to be container-centric instead of being machine or virtual machine-centric. Kubernetes acts as an abstraction layer between the user and public cloud provider, allowing them to deploy containerized workloads with the same APIs to

on-premise or public cloud infrastructure with no changes. Kubernetes brings additional benefits to the hybrid cloud environment such as automatic retries, self-healing, deployment history, rollbacks, workload replication, increased availability, and more.

Today it is not possible to enable hybrid cloud with a single Kubernetes cluster. The Kubernetes control plane is responsible for managing the Kubernetes nodes within the cluster. That logic has been written to explicitly exclude nodes in more than one public or private cloud. Google Cloud Platforms (GCP)'s Anthos overcomes this by providing a multi-cluster management layer. Kubernetes Federation overcomes this by providing Federated Kubneretes Resources (Federated across multiple clusters). While both Kubernetes Federation and GCP Anthos provide multi-cluster management, they both lack scheduling workloads between clusters (with optimized placement), they both require deployment to cluster mapping to be pre-defined, and there is no cluster assignment change post-deployment. A subset of problems addressed by Kubernetes requires more than one Kubernetes cluster or a multi-cluster configuration. Hybrid cloud represents a subset of the problem space address by multi-cluster configurations. See Figure 2.1.

The proposed HCS is designed to start with a single on-premise Kubernetes cluster. The cluster user begins scheduling workloads to the on-premise cluster. The HCS will detect when the capacity of the on-premise cluster has been exhausted by calling the Get Namespaced Pods REST API provided by the kube-apiserver. Then, it will create a new Kubernetes cluster in the cloud on-demand using the open-source cluster creator Kops. The cloud cluster's AWS instance types are chosen based on the total number of vCPUs and megabytes of memory required of the pending pods returned from the Get Namespaced Pods endpoint. The cheapest AWS region (based on the instance type) is chosen to save cost. After cluster creation, the HCS will move pods from the on-premise cluster's pending queue to the cloud cluster, thereby

bursting workloads to the cloud. The HCS ensures that data dependencies for these machine learning workloads are satisfied after migration by running pre-steps and post-steps before and after every job. The post-steps for training jobs uploads the trained model to AWS S3 and inserts a MongoDB MongoDB (2021) record with the named model and timestamp. Pre-steps for an inference job fetch the latest model by querying MongoDB for the latest trained model by name. The Pre-steps will also cache the trained model on the local Kubernetes node and check if the model resides there first before fetching it remotely from S3.

Three machine learning models named ARPU, Saved, and Credit Card Fraud evaluate the HCS's performance. Each machine learning model has two Kubernetes jobs. One job contains machine learning tasks to train the model. One job contains machine learning tasks to use the trained model to perform a prediction or inference. The Saved machine learning model is a mache learning pipeline. It takes as input to its training task the inference from the ARPU model. The Workload Deployer used in the evaluation deploys the training jobs and regular intervals and the inference jobs as random intervals to simulate real-world job arrival behavior. The on-premise Kubernetes cluster uses Kubernetes version v1.20.7. The cloud cluster uses v1.18.20. This difference in version is insignificant to our evaluation as the job deployment Representational State Transfer (REST) Application Programming Interface (API) and cluster capacity rules are the same between these two versions. Kubernetes version v1.18.20 is supported by the Kops cluster creator tool used to create cloud clusters dynamically. The on-premise Kubernetes cluster's bare metal hardware has 32 cores and 64GB of RAM with 8 cores and 16GB of RAM dedicated to the master node and 16 cores and 16GB of RAM dedicated to the worker node. For the baseline experiment, the cloud cluster is provisioned with the same number of CPUs and GB of RAM as the on-premise cluster. The HCS Improved experiment dynamically creates

	ASU Servers, On-premise Kubernetes Cluster	AWS, Cloud Kubernetes Cluster
Kubernetes Version	v1.20.7	v1.18.20
Master Node		
OS	Ubuntu 18.04.5	Ubuntu 20.04
CPUs	8	2
Memory	16GB	4GB
Worker Node		
OS	Ubuntu 18.04.5	Ubuntu 20.04
CPUs	16	Dynamic
Memory	16GB	Dynamic
Bare Metal		
OS	Ubuntu 18.04.5	
CPUs	32	
Memory	64GB	

Table 1.1: Test Bed

the cloud cluster with hardware based on the needs at runtime. Table 1.1 provides more information about the on-premise and cloud cluster configuration use.

The HCS results show that using dynamic region selection saved 16% of cloud cost on the master node and 20% on the worker node. Dynamic cluster creation, bursting to the cloud, and cloud auto-scaling saved 48% of the overall execution time of our workload, which is a mixture of training and inference machine learning workload tasks. The average turnaround time was increased when bursting workloads to the cloud as additional initialization tasks were needed to download docker images, download data dependencies and warm the local node cache already present for other nodes. However, because these tasks were executed with a higher degree of parallelism, overall execution time and cost were still reduced.

This thesis first reviews background information about Kubernetes, its architecture, capabilities and limitations. Then the state of the art or how current solutions

address the problem statement is discussed. The proposed HCS's behavior and implementation is reviewed. The baseline, experiments and results used to evaluate the HCS are reviewed. Potential future areas of research identified in this study are reviewed. Finally the conclusions drawn from this thesis research is discussed.

Chapter 2

BACKGROUND

Kubernetes is a distributed system. It comprises a set of processes that run on two groups of machines: master or control plane nodes and worker nodes. These collections of machines constitute a cluster. The six processes that make cluster-level decisions, known as the control plane, are: kube-apiserver, kube-controller-manager, kube-scheduler, and ectd. Two processes run on each of the worker nodes: kubelet and kube-proxy. The kube-scheduler process or Kubernetes scheduler will schedule application containers to worker nodes.

Depending on the cluster's infrastructure, it may require other processes within the control plane or worker nodes. When running on a public cloud provider, an additional control plane process runs the cloud-controller-manager. The cloud-controller-manager has various control loops that are used to synchronize the state between your cloud provider and your cluster et al. (2020a). The kube-apiserver facilitates communication between all the components. It exposes a REST API over Hypertext Transfer Protocol (HTTP) to interface with your cluster. Kubernetes uses Etcd for its state management and as its persistence layer for all Kubernetes objects. Ectd is a distributed persistent key-value store.

Kubernetes's abstractions provide a standard interface to deploy and run software independent of the underlying infrastructure. The proposed HCS will leverage these abstractions to provide a standard interface to deploy across traditional deployment boundaries. The boundaries between public and private cloud, multiple cloud providers, multiple regions, and availability zones within a cloud provider.

Kubernetes has a growing list of over fifty abstractions. The proposed HCS in

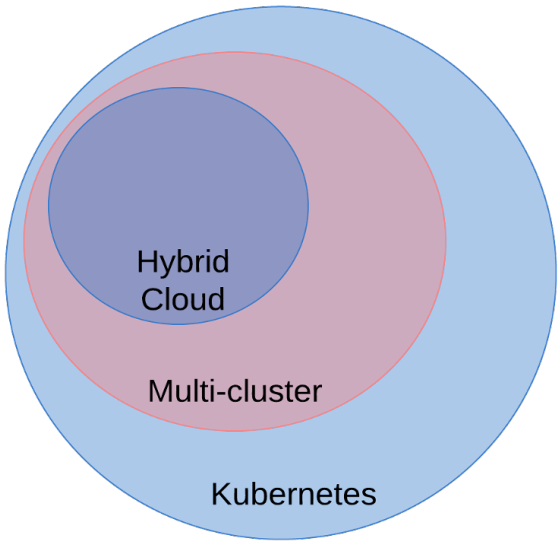

Figure 2.1: Problem Space

this thesis focuses on four: Cluster, Node, Job, and Pod. Every machine in the cluster, whether virtual or physical, is a Kubernetes *node*. A *pod* executes code within Kubernetes et al. (2020b). It differs from a container in that it can consist of one or many containers that collaborate on the same node.

A `deployment` defines a non-terminating process ex: web service, monitor, controller. A job is a parent object to a pod, a parent object to a container. It tracks multiple software deployments. It can be used to roll back to previous versions of a deployment, including the container image version. A deployment may parent many pods through a replicaSet.

A `job` defines a terminating process with a finite start and stop time, ex: cron job, batch job, machine learning workload. Like a deployment is a parent object of a pod. It records the process start and stop time. A job may parent many pods through multiple retries of a process failure.

Kubernetes's design is heavily influenced by control theory et al. (2020e). In control theory, there are two actors: an observer and a controller. The observer monitors a chosen variable's value. The controller first compares that value with

the desired value. Then, the controller chooses a corrective action to reconcile the difference. A Kubernetes controller is a non-terminating process that continuously monitors the cluster's current state and compares it to the desired state. When there is a discrepancy, it will choose from a list of actions to reconcile these differences. An example of this is the replication controller. When a pod is deployed with a replica set and a specified number of pods, it is the replication controller's responsibility to ensure that pod runs the given number of times. If the number of pods specified is three, but only one is running, the controller will spawn two new pods. If the number of pods specified is three and five runs, it will terminate two pods. It, therefore, reconciles the difference between the number of pods specified and the number of pods running et al. (2020f).

"A Kubernetes cluster can have no more than 5000 nodes. No more than 150000 pods. No more than 300000 total containers. No more than 100 pods per node" et al. (2020d). A single cluster cannot run in more than one region of the same cloud provider or in more than one cloud provider. There are even some limitations for running a single cluster in multiple availability zones, increasing clusters' availability by being redundant in more than one data center. When referring to the handling of multiple regions or cloud providers within a single cluster, one of the Kubernetes authors stated: "I think that an argument can be made both ways here. It depends on whether you prefer to weave the logic for handling nodes in multiple availability zones and cloud providers within a single logical cluster into the existing Kubernetes control plane codebase (which was explicitly not designed for this), or separate it into a decoupled Federation system (with possible code sharing between the two via shared libraries). The author prefers the latter"MacMillan and Saenger (2017)

2.1 Kubernetes Scheduling

The Kubernetes scheduler source code is written in Go Meyerson (2014) and can be found on GitHub under the Kubernetes Project et al. (2020c). When a pod definition is created using the Kubernetes API, there is no node assignment. The scheduler's responsibility is to determine the best node where the pod is schedulable and then assign that pod to the node. When the pod is assigned to a node, the scheduler will populate the nodeName attribute on the pod definition. The kubelet for that node will then detect that the pod definition has its nodeName and begin the pod startup sequence on its node.Burns and Tracey (2018a) The Kubernetes scheduler goes through a two-step process as it decides the placement of pods to nodes: It first runs the list of nodes through a list of predicate functions. Predicate functions return a true/false response if the pod is schedulable to that node Burns and Tracey (2018b).

2.2 Tiers of Scaling

Kubernetes provides several ways to scale today. The pod horizontal auto scaler increases the number of containers dedicated to that application, thereby increasing the number of processes and the amount of CPU and RAM dedicated. This is bounded by the number of nodes or machines available to the Kubernetes cluster. With one of two auto-scaling options, we overcome this limitation: The node vertical auto scaler or the cluster auto scaler. The node vertical auto scaler will leverage the cloud provider's capability to vertically scale the hardware available to the virtual machine, thereby increasing CPU and RAM or disc capacity without interruption to the node. The cluster auto scaler will leverage the cloud provider's capability to increase the number of nodes or machines dedicated to the cluster. A single Kubernetes cluster, however large, has its own set of limitations.

Chapter 3

STATE OF THE ART

Anthos Google (2021a) and Kubernetes Federation GitHub (2021a) both provide multi-cluster management. They do not provide scheduling behavior between clusters or optimized placement, and the deployment to cluster mapping is required to be predefined. Layer two schedulers do provide scheduling between clusters. They do not address: bursting to the cloud, dynamic cluster creation, optimization of cloud infrastructure based on pending queue needs, or provide data dependency migration for machine learning workloads. These are the advances that set the HCS apart from current solutions.

3.1 Anthos

Anthos refers to a collection of Google Cloud Platform (GCP)Google (2021b) offerings that address the Kubernetes multi-cluster space of problems. Google Kubernetes Engine (GKE) Google (2021d) creates managed Kubernetes clusters hosted on-premise, with GKE (On-Premise) in GCP or other cloud providers like AWS. GCP Fleets Google (2021c) define a logical group of Kubernetes clusters with an Application Program Interfaces (APIs) or web console to add or remove new Kubernetes clusters to the grouping. Anthos provides a centralized Kubernetes cluster monitoring solution that aggregates data from Kubernetes clusters into a single monitoring dashboard. GCP Fleets enables a user to define a multi-cluster deployment strategy for a single application. Anthos offers a managed service mesh similar to Istio that allows the user to specify network policies for intercluster and intracluster communication. The managed service mesh allows for additional network-level monitoring

of each cluster and intercluster communication secured by mTLS. Figure 3.1 shows an example of a multi-cluster configuration managed by Anthos. However, none of Anthos's or GCP's multi-cluster solutions provide a scheduler that will dynamically move workloads at runtime. Anthos, through GKE on-premise, can auto scale the number of Kubernetes worker nodes when additional on-premise resources are available. When on-premise resources are exhausted, it will take no action. The HCS will automatically move workloads from the on-premise cluster to the cloud cluster when on-premise resources have been exhausted. Anthos depends on the end-user to create and register Kubernetes clusters ahead of time. The HCS will automatically create a new Kubernetes cluster on-demand. By creating Kubernetes clusters on-demand, it can tune the size of the Kubernetes worker nodes provisioned based on the needs of the Kubernetes pending queue. Anthos does not inspect the Kubernetes pending queue as it does not create Kubernetes clusters on-demand based on workload state. Anthos does not automatically move data dependencies between clusters and instead relies upon the end-user to ensure that each cluster has the dependencies for the workloads. The HCS will move data dependencies between clusters to ensure that each machine learning workload uses the most recently trained model regardless of which cluster it runs. Deploying to an Anthos fleet requires the end-user to map the deployment to a specific cluster. With HCS, the user need only deploy to the on-premise cluster. The HCS will automatically move workloads and scale the cloud cluster as needed.

3.2 Kubernetes Federation

Kubernetes Federation or KubeFed is part of the Kubernetes GitHub Project as a sub-project. It provides an interface for federated Kubernetes resources (federated across more than one Kubernetes cluster) It aims to provide new Kubernetes ab-

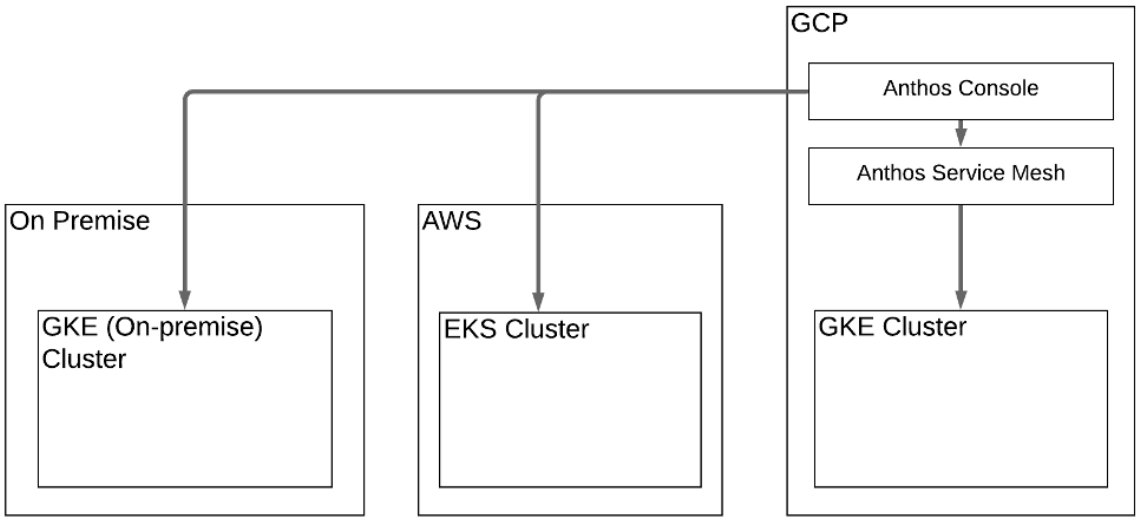

Figure 3.1: Anthos Diagram

stractions that manage more than one cluster. This project is not as mature as the Kubernetes Project.

Kubernetes Federation provides a way to spread a deployment across more than one cluster with a predefined definition called a Replica Scheduling Preference. The Replica Scheduling Preference supports use cases such as evenly distributing replicas across every cluster or having a weighted distribution where one cluster may have more replicas than another. The Kubernetes federation multi-cluster replica scheduler has one feature that schedules based on load called rebalance. However, it will continuously attempt to schedule to the desired predefined cluster even if it is at capacity. The Kubernetes federation user guide recommends that if this side effect is unacceptable, to disable rebalancing for that deployment.

The HCS does not require the end-user to specify a deployment to cluster mapping. Instead, the end-user need only schedule workloads to the single on-premise Kubernetes cluster. The HCS will then decide which workloads execute on the on-premise cluster and which workloads burst to the cloud based on the on-premise cluster's capacity. This design eliminates the need for the end-user to decide which

cluster the job executes on and can make optimizations at run time based on the workloads the on-premise cluster does not have the capacity to run.

3.3 Layer 2 Schedulers

A Kubernetes Layer two scheduler is a component that adds custom scheduling behavior as layer one and then delegates to the Kubernetes scheduler as layer two Youssef (2020). Several open-source layer two schedulers have been created to overcome shortcomings in the existing Kubernetes scheduler. Volcano (2020), addresses that the Kubernetes scheduler does not handle jobs in a (First In First Out) FIFO pattern. In other words, if many jobs arrive when Kubernetes does not have the capacity, it does not guarantee that the first job that arrived will be the first to be scheduled Wang (2019). This behaves contrary to what many users would expect.

The proposed HCS is a layer two scheduler. It first checks the capacity of the on-premise Kubernetes cluster, dynamically creates a Kubernetes cluster in the cloud with optimizations, auto-scaling, and then bursts workloads from on-premise to the cloud cluster. The dynamic cluster creation is what sets the HCS apart from other layer two schedulers. By delaying cluster creation until its needed, it enables the HCS to optimize for the needs of the pending Kubernetes Jobs and make cost-saving decisions at the time of creation.

Chapter 4

HYBRID CLOUD SCHEDULER

The proposed Hybrid Cloud Scheduler (HCS) accomplishes six optimizations and enables hybrid cloud with a single Kubernetes cluster. When the on-premise Kubernetes cluster has no more capacity, workloads will burst to the cloud. The HCS will create a Kubernetes cluster in AWS on-demand. This cluster's worker node will be sized based on the needs of the on-premise Kubernetes cluster's pending queue. A region for the on-demand Kubernetes cluster will be dynamically chosen to save cost and reduce transfer time. The on-demand Kubernetes cluster will be auto scaled. Each machine learning workload's data dependencies will be automatically migrated between clusters as needed.

The HCS's main thread performs a series of checks and is responsible for cluster creation. After cluster creation, it then spawns three control loops to accomplish its tasks: a Scale Watcher, Descheduler, and Overflow Watcher. The Overflow Watcher spawns groups of Job movers. Figure 4.1 shows a diagram of the control loops and threads that are spawned. The following sections review each of these thread's responsibilities below.

4.1 Main Thread

The main thread periodically pulls a list of pods from the pending queue by calling the Kubernetes API server of the on-premise cluster. This is done by calling `CoreV1ApiInstance.list_pod_for_all_namespaces` from the python Kubernetes client library with a field selector of `spec.nodeName==,metadata.namespace=default`. The Kubernetes REST API supports fields selectors to ensure the API response only con-

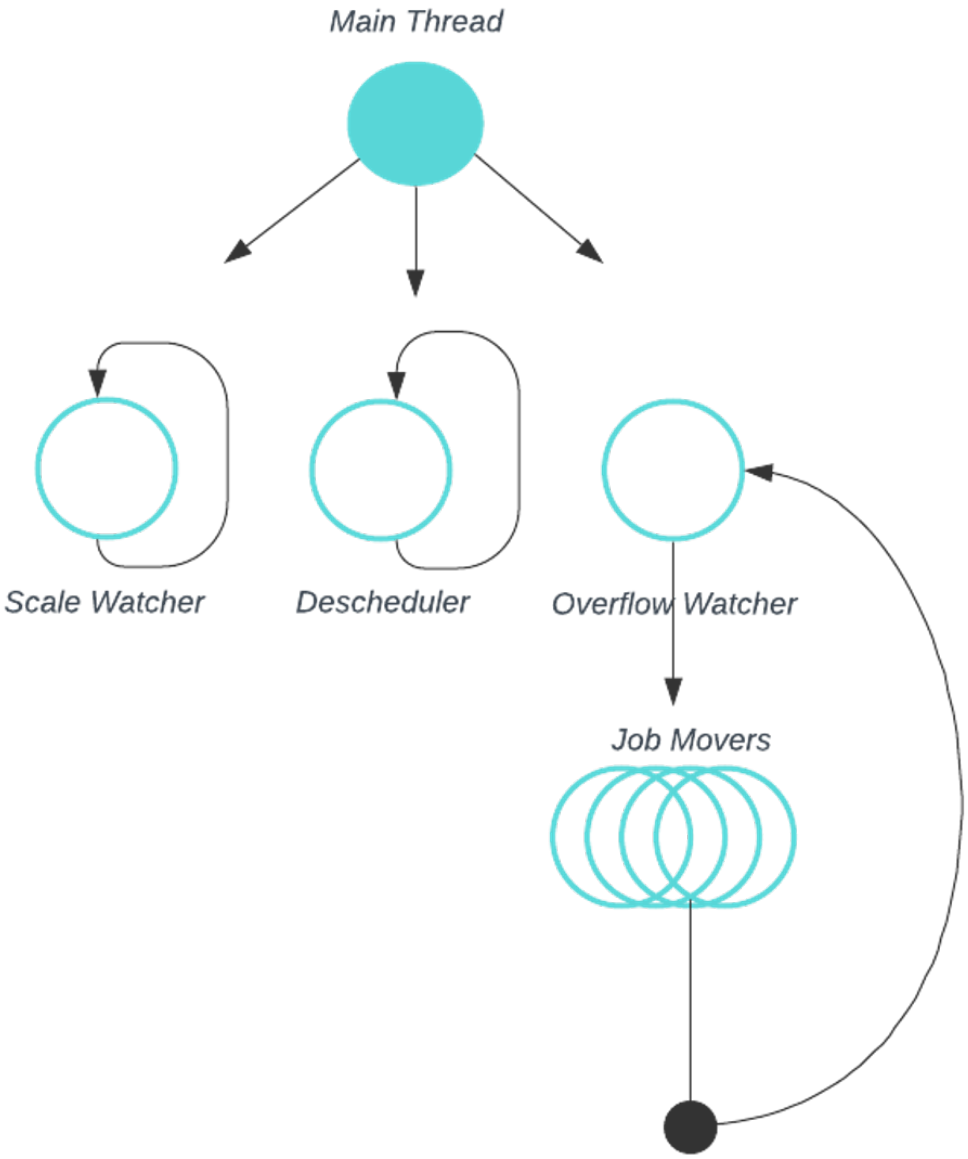

Figure 4.1: HCS Threads Diagram

tains the pods we need, thus reducing execution time and saving network bandwidth. When pods exist in the scheduling queue, the HCS knows the on-premise cluster's capacity is exhausted. It then creates the Kubernetes cluster on-demand in the cloud using Kops. Kops GitHub (2021b) is an open-source Kubernetes AWS cluster management Command Line Interface (CLI). After the cloud cluster has been successfully created, the HCS spawns the Scale Watcher, Descheduler, and Overflow Watcher control loops.

4.2 Dynamic Cluster Creation

When creating the cloud cluster, the HCS performs four steps. First, it chooses the AWS EC2 instance type based on need. Second, it chooses the AWS region and availability zone based on cost. Third, it creates the cluster using Kops. Finally, it downloads the authentication keys for the created cluster from AWS S3.

The HCS chooses the instance type by pulling the list of pods from the pending queue by calling the Kubernetes API server of the on-premise cluster. Every container within each pod definition will have a requested CPU and memory limit. Located here on the pod definition: `pod.spec.containers.[].resources.requests['cpu/memory']` This is a requirement for the Kubernetes scheduler. If no value exists for the request limit, then the Kubernetes scheduler will continue to schedule an unlimited number of pods onto the node even after the node becomes overwhelmed, leading to an unhealthy node and unhealthy set of pods on that node. By summing all the containers request limits for CPUs, the HCS will know how many CPU cores it needs to satisfy the pending queue's demand. This is likewise done for the megabytes of RAM needed. The HCS compares the CPU cores and megabytes of RAM needed for each instance type from the c5 instance family, choosing the smallest instance type that satisfies those needs.

The HCS chooses the AWS region by querying the per region, per availability zone cost for the given instance. This is done by calling the python AWS client library boto `client.get_products(ServiceCode='AmazonEC2', Filters=filters)`. The least expensive region and availability zone are chosen.

The instance type, region, and availability zone parameters are then passed to the cluster creation operation. The cluster is created by calling `kops create cluster` with the following parameters

```
kops create cluster \
  --node-count 1 \
  --node-size $CLUSTER_NODE_INSTANCE_TYPE \
  --master-size t2.medium \
  --zones=$AWS_ZONE \
  $NAME
```

The final step taken by the HCS main thread is to download the authentication keys for the AWS cluster from amazon S3. This is done with boto's S3 `download_file` function

```
s3_client = boto3.client('s3')
s3_client.download_file(...)
```

The HCS Scale Watcher watches the cloud cluster's pending queue by calling `CoreV1ApiInstance.list_pod_for_all_namespaces` from the python Kubernetes client library with a field selector of `spec.nodeName==,metadata.namespace=default`. When the Kubernetes API returns more than zero pods, the Scale Watcher starts the cloud cluster horizontal scale operation. Then it waits for the scale operation to complete. This control loop iterates every 60 seconds. The horizontal scaling opera-

tion uses Kops to increase the size of the AWS instance group that backs the Kubernetes worker nodes. First the horizontal scaling operation downloads the Kops YAML definition `kops get --name $CLUSTER_NAME -o yaml`. Then the `spec.minSize` and `spec.maxSize` of the Kubernetes node instance group is increment by one. Then the horizontal scaling operation updates the remote configuration files in S3. Finally, the horizontal scaling operation performs a kops update cluster call, adding the additional node and waiting for the task to complete with a ten-minute timeout.

4.3 Descheduler

During multiple tests of the HCS Scale Watcher, it was discovered that the Kubernetes scheduler will overcommit a newly added Kubernetes worker node when there is a large pending queue. See the following error when scheduling a pod `'phase': 'Failed', 'reason': 'OutOfcpu', 'message': 'Pod Node didn't have enough resource: cpu, requested:'`. The Kubernetes GitHub issue list shows that there have been several issues raised like this error in past versions of the Kubernetes scheduler. To overcome this issue, deleting the pod manually was necessary. Otherwise, the Kubernetes scheduler would not attempt to reschedule the pod, and it would remain in a non-scheduled, failed state. The HCS resolves this issue by spawning the Descheduler. The Descheduler periodically pulls a list of pods by calling the Kubernetes API server of the cloud cluster. This is done by calling `CoreV1ApiInstance.list_pod_for_all_namespaces` from the python Kubernetes client library with a field selector of `status.phase=Failed,metadata.namespace=default`. Any pods that are returned by the Kubernetes API are deleted, retriggering the scheduling of those pods. The Descheduler captures and logs the failure reason to ensure that other errors are not missed. The Descheduler control loop repeats every five seconds.

4.4 Overflow Watcher

The Overflow Watcher periodically pulls a list of pods from the pending queue by calling the Kubernetes API server of the on-premise cluster. If pods exist, it will spawn a Job Mover for every job. It will wait for the results of every job mover joining the threads. The Overflow Watcher control loop repeats every sixty seconds. The first iteration of the Overflow Watcher would move each job sequentially until all jobs were moved. Due to the high number of blocking network calls needed to move each job, moving all the jobs took far too long. By spawning a Job Mover for each of the found pods, we ensure that these network calls are executed concurrently and with the highest degree of parallelism. The Overflow Watcher will wait for all of the Job Moves to complete before terminating its current loop. The Overflow Watcher control loop repeats every sixty seconds.

4.5 Job Mover

First, each Job Mover will fetch the Kubernetes Job corresponding with the pending pod. This is done by fetching the pending pods as previously described. All the jobs from the on-premise cluster are fetched using `BatchV1ApiInstance.list_namespaced_job('default', timeout_seconds=60, watch=False)` from the python Kubernetes client library. The pending pods are matched to their corresponding jobs by checking that the job metadata name match the pending pod's metadata label job name.

```
pending_pod.metadata.labels['job-name'] == job.metadata.name
```

The job-name label is automatically set on the pod when the Kubernetes job controller creates the pod for that job. The HCS then duplicates the job from the on-premise cluster to the cloud cluster. The Job is duplicated by performing a python deepclone

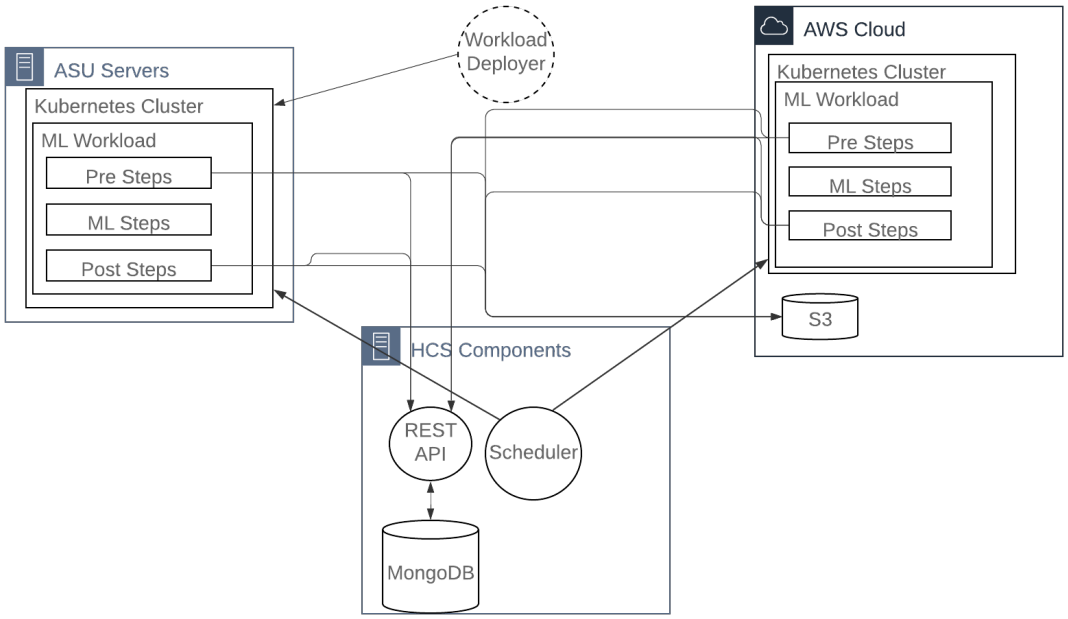

Figure 4.2: HCS Diagram

of the job object returned from the python Kubernetes client library. The cloned job object is then passed to the create job operation `BatchV1ApiInstance.create_namespaced_job(body=job, namespace='default')` Third, The HCS deletes any pods associated with that job from the on-premise cluster. Finally, it deletes the on-premise job.

4.6 Data Dependency Migration

The HCS ensures data dependencies are satisfied for machine learning workloads regardless of where workloads have been migrated. When a machine learning model training task is completed, the trained ML model is uploaded to S3, and a record is inserted into a MongoDB instance with the name of the trained model, a timestamp, and the S3 file location. An overview of this process is displayed in Figure 4.2.

When an ML inference task uses a trained model, it performs a lookup through an exposed REST API of "what and where" is the most recently trained model. The

REST API queries MongoDB and returns the name of the most recently trained model, its timestamp, and its S3 location. Then a check is performed to see if that trained model already exists on disk on the Kubernetes node. If the model exists on disk already, it is used. If the model does not exist on disk, an operation is performed to fetch the model from S3 and persist it to a hostDir Kubernetes Volume. Persisting it to a hostDir Kubernetes Volume allows other pods running on that node to use the same shared disk space.

Chapter 5

RESULTS

Existing machine learning models with simulated real-world job arrival behavior are used to evaluate the performance of the HCS. The Workload Deployer simulates real-world job arrival behavior. It spawns one thread per Kubernetes Job that contains machine learning tasks and deploys machine learning training jobs at regular intervals and inference jobs at random intervals. The hardware and software used for the On-premise and Cloud Kubernetes clusters can be found in Table 1.1.

The evaluation of the HCS consists of three experiments. First, the on-premise cluster baseline runs the entire workload on the on-premise cluster without any intervention. Second, the hybrid cloud naive baseline bursts workloads from the on-premise cluster to the cloud cluster without dynamic cluster creation, horizontal cluster scaling, or dynamic worker node sizing or region cost optimizations enabled. Finally, the hybrid cloud improved scheduler experiment runs the same workload, starting with only the on-premise cluster. Then, the HCS creates the cloud cluster on-demand with all of the optimizations enabled.

5.1 Machine Learning Workloads

Three machine learning models are used to evaluate the HCS: ARPU, Saved, and Credit Card Fraud. Each with its training and inference tasks. The ARPU Model attempts to answer the business question: "How much of a discount do we need to provide a customer to retain them?". Retain refers to keeping a customer subscribed who is currently paying for a subscription product. NortonLifelock provided this model. As input, it takes records of offers made to customers and if they accepted

or rejected those offers along with customer attributes such as what products they purchased and how long they have been a customer. This model uses a Linear Regression. It outputs the predicted discount they need to stay in United States Dollars (USD).

The saved model attempts to answer the business question: "How likely will this customer be retained if we reach out to them?" NortonLifelock also provided this model. As input, it takes the ARPU model's inference value, customer attributes, and records of offers made to customers and if they are accepted or rejected. This model uses a pipeline for training as it takes the inference of another model as input and uses a Linear Regression. The output of this model is the predicted likelihood of being retrained in a probability percentage.

The Credit Card Fraud model attempts to answer the business question: "Is this a fraudulent transaction we should prevent?" This is a public model from Kaggle Kaggle (2021). This model takes as input example fraudulent and legitimate credit card transactions. This model uses a Logistic Regression for model training. The output is the predicted likelihood of being fraudulent in a probability percentage.

5.2 Workload Baseline On-Premise

The workload baseline experiment is performed to establish the characteristics of the workload to be scheduled in all of our experiments. What happens to a single cluster when it is overwhelmed by machine learning workload jobs? Are the Jobs eventually processed? How much longer does it take to process?

An overview of all running pods throughout the experiment can be seen in Figure 5.1.

The blue line represents the total number of running pods. It shows the running pods' count from its base number before any workloads are scheduled to their max-

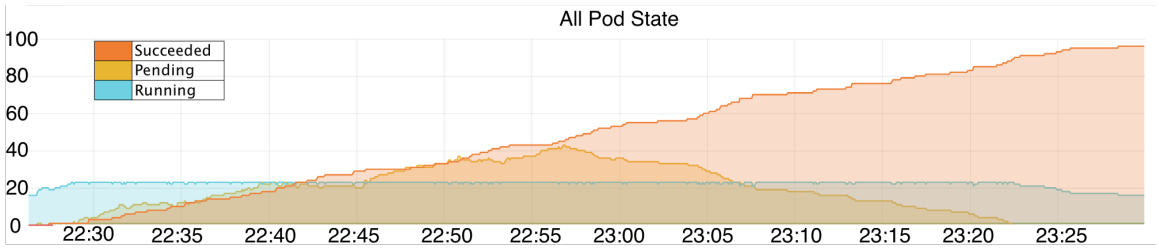

Figure 5.1: Workload Baseline Experiment - On-premise All Pod State

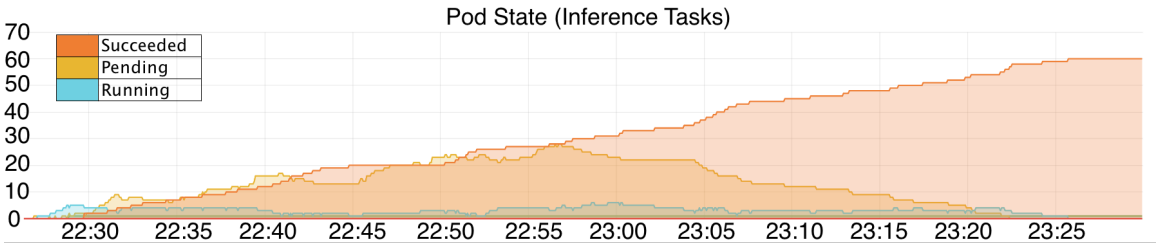

Figure 5.2: Workload Baseline Experiment - On-premise Inference Pod State

imum number. The blue line's maximum number represents the cluster's maximum capacity. When new jobs arrive, and the Kubernetes cluster is at its maximum capacity, pods will enter the pending queue. The yellow line in this figure shows the pending queue. It shows that the cluster becomes overwhelmed at about 22:30, and the pending queue size grows over time from this point. The orange line grows at 22:30. The orange line represents pods with the succeeded status or, in other words: machine learning workloads that have been executed and completed. At 22:57, the number of pods in the scheduling queue hit their climax. After this point, all jobs have arrived at the Kubernetes cluster, and the pending queue slowly drains as jobs are completed, shown by the growing orange line and declining yellow line.

The differences in arrival behavior between training and inference ML jobs can be seen in Figures 5.2 and 5.3. The training ML jobs, shown in Figure 5.3 arrive at regularly scheduled intervals. The inference ML jobs, shown in Figure 5.2 arrive more frequently and arrive randomly instead of in regular intervals. This difference in arrival behavior can be seen by comparing the yellow line between Figures 5.2 and 5.3 showing the total count of pending inference or training ML jobs, respectively.

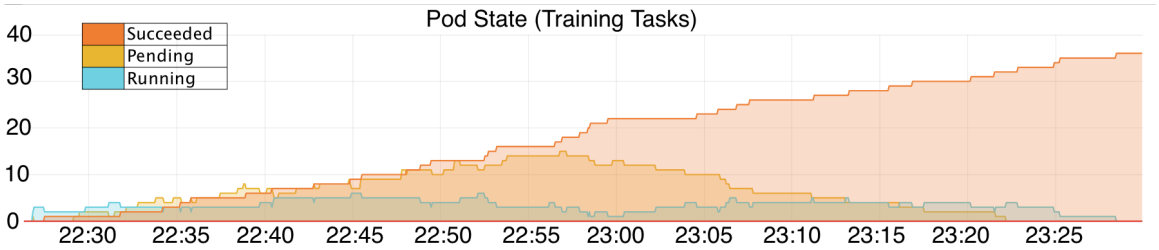

Figure 5.3: Workload Baseline Experiment - On-premise Training Pod State

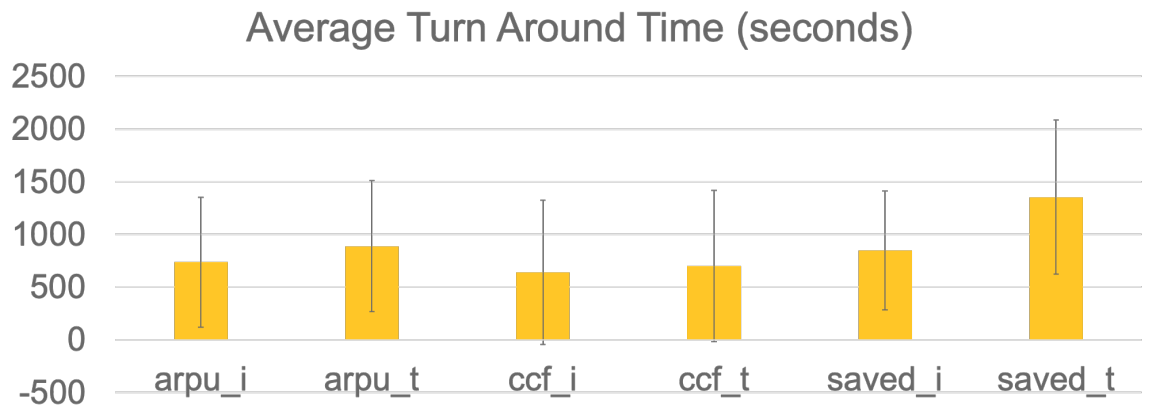

Figure 5.4: Workload Baseline Experiment - Model Performance

This shows the simulated job arrival behavior implemented in the HCS's workload deployer. The workload deployer is not a part of HCS but is used to evaluate the HCS's effectiveness. If either CPU or memory utilization does not have enough free space for the new pod's memory or CPU request limits, it will mark the node as at capacity and send the pod to the scheduling queue. Figure 5.4 shows the average model performance with the +/- the standard deviation as error bars. The on-premise cluster took 3244 seconds to process the entire workload, including queuing, processing, and completing.

5.3 Hybrid Cloud Scheduler Baseline Naive Overflow

The hybrid cloud scheduler naive overflow baseline shows how this workload behaves when the HCS migrates workloads from on-premise to the cloud without optimizations. Those optimizations are on-demand cluster migration, dynamic sizing of

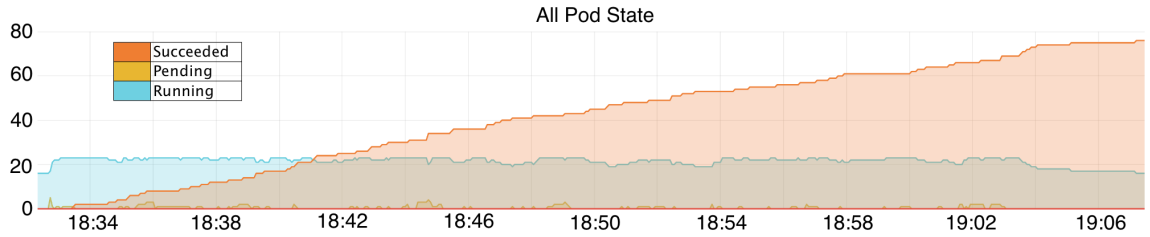

Figure 5.5: HCS Baseline Naive Overflow - On-Premise Cluster All Pod State

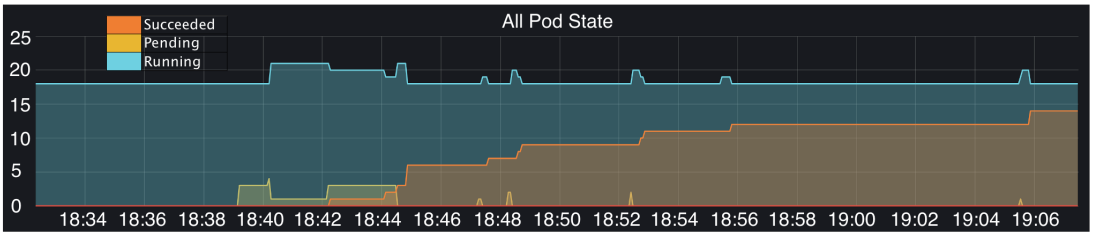

Figure 5.6: HCS Baseline Naive Overflow - Cloud Cluster All Pod State

the cloud cluster, and dynamically choosing a region to optimize transfer speed and cost. This baseline demonstrates the time, and costs savings these HCS optimizations provide. Without dynamic cluster creation, we must create the cloud cluster ahead of time, as is the recommended action with many of the multi-cluster configurations and tools available today. The cloud cluster is created in advance with a default worker size of an AWS EC2 instance type of c5.4xlarge. A c5.4xlarge has the same amount of memory and CPU cores as the on-premise cluster in the closest AWS region to our location (us-west-1). Observing the yellow pending scheduling queue line in Figure 5.5 shows that pods no longer increasingly build up over time.

Instead, pods are moved from the on-premise cluster's scheduling queue to the cloud cluster's scheduling queue, which can be seen in Figure 5.6.

This stands in contrast to Figure 5.1 from the experiment performed in Section 5.2. The differences in ML job arrival behavior can be seen both in the on-premise and cloud cluster. This is shown in figures 5.7, 5.8, 5.9, 5.10.

The ARPU inference, ARPU train, credit card fraud train, and saved train jobs

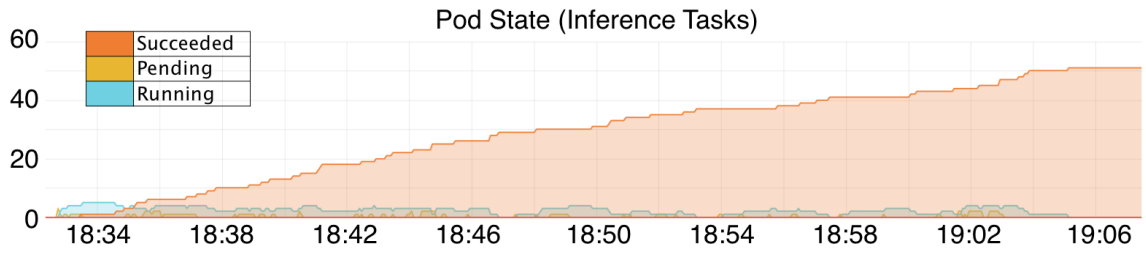

Figure 5.7: HCS Baseline Naive Overflow - On-Premise Inference Pod State

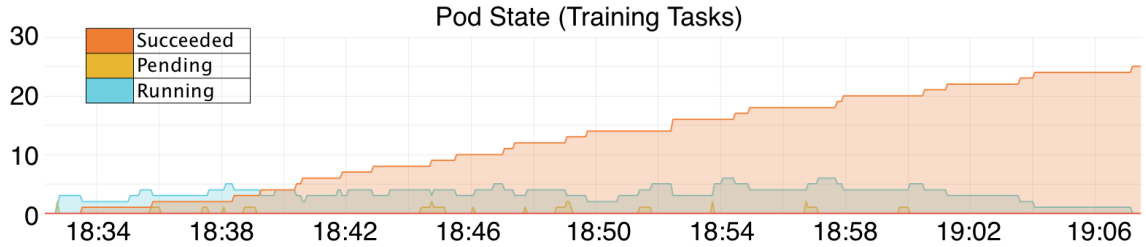

Figure 5.8: HCS Baseline Naive Overflow - On-Premise Training Pod State

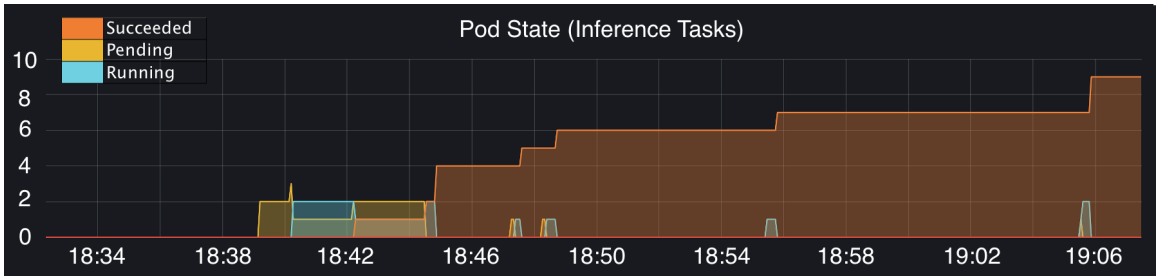

Figure 5.9: HCS Baseline Naive Overflow - Cloud Inference Pod State

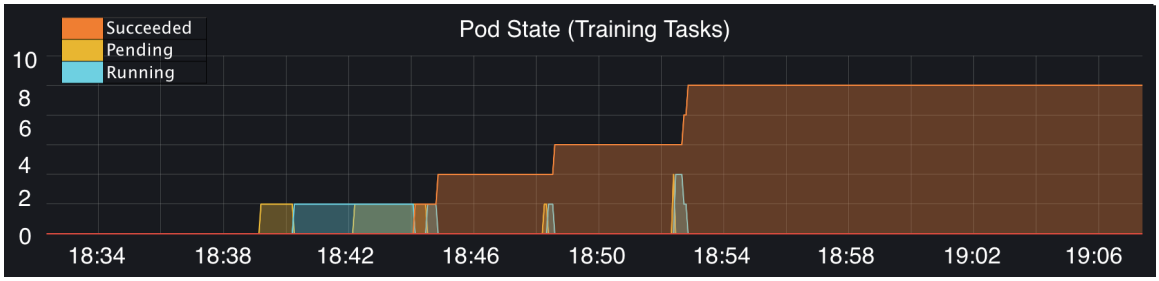

Figure 5.10: HCS Baseline Naive Overflow - Cloud Training Pod State

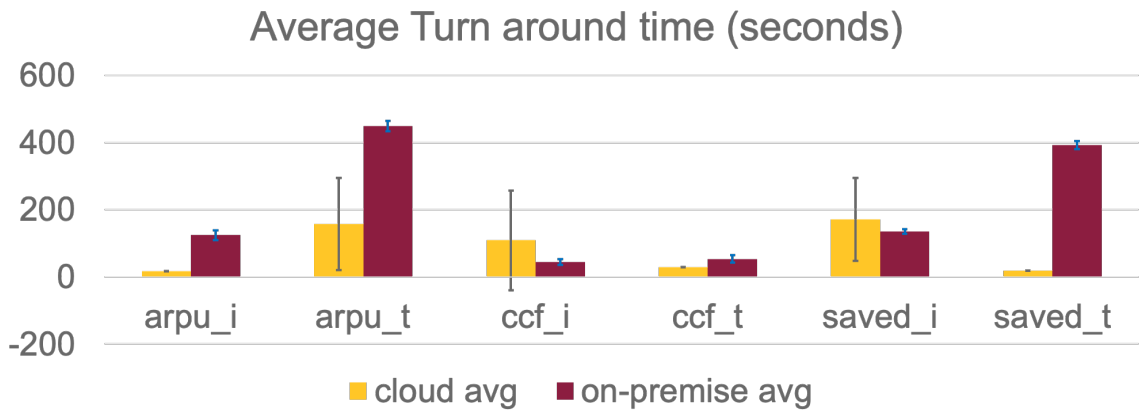

Figure 5.11: HCS Baseline Naive Overflow - Model Performance

performed better on the cloud cluster, while the saved inference and credit card fraud inference models performed better on the on-premise cluster. Figure 5.11 shows the average model performance with the +/- the standard deviation as error bars.

The total time taken to process the workload between the on-premise and cloud clusters was 1847 seconds. The cloud cluster was active the entire time.

5.4 Hybrid Cloud Scheduler Improved Overflow

The HCS improved overflow experiment is performed with the same workload processed in the previous two experiments. However, all of the optimizations are enabled.

The first optimization can be seen in the first 10mins of Figure 5.12. The yellow line shows that the number of queued pods continues to grow until 8:50, when it drops to zero. During this time, the HCS is dynamically creating the cloud cluster with a worker node sized to meet the needs of the pods in the scheduling queue and in the US-based region that will provide the most cost savings.

Figures 5.13 5.14 shows those pods broken into inference and training jobs. Figures 5.15 5.16 show the migrating jobs arriving in the cloud cluster for the first time.

Between 8:23 and 8:57, the yellow line shows that the pending queue of the cloud

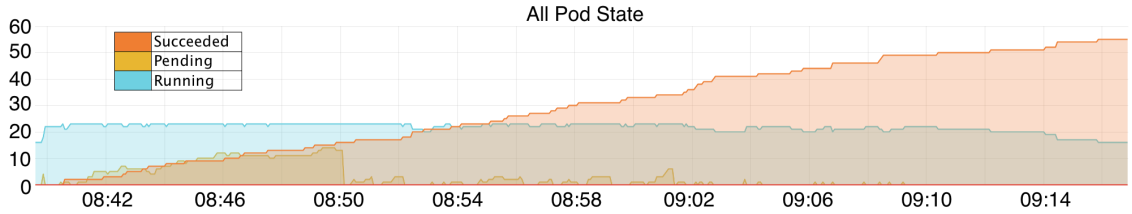

Figure 5.12: Scheduler Improved Experiment - On-premise All Pod State

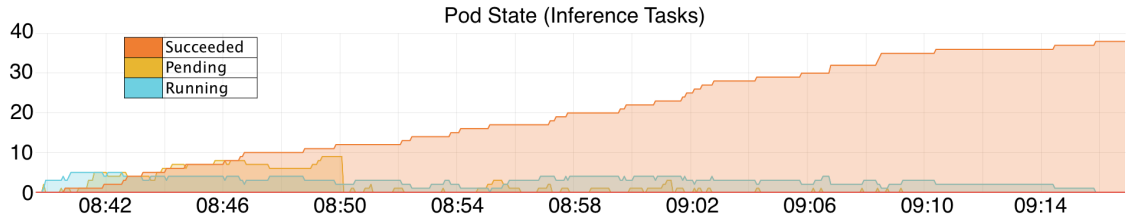

Figure 5.13: Scheduler Improved Experiment - On Premise Inference Pod State

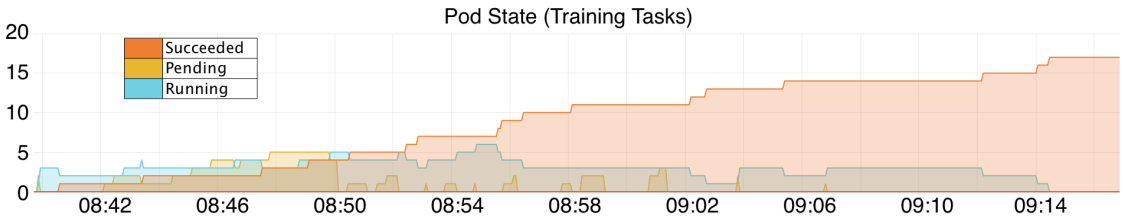

Figure 5.14: Scheduler Improved Experiment - On Premise Training Pod State

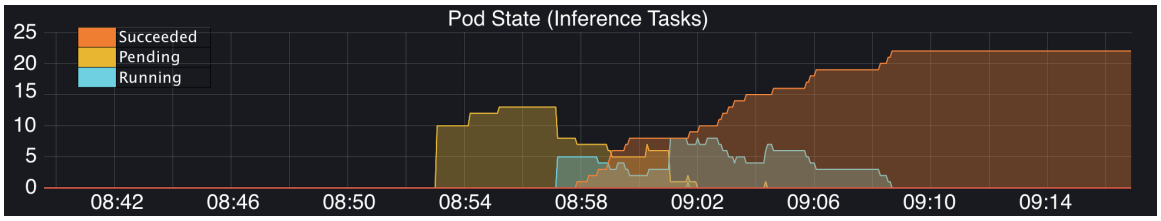

Figure 5.15: Scheduler Improved Experiment - Cloud Inference Pod State

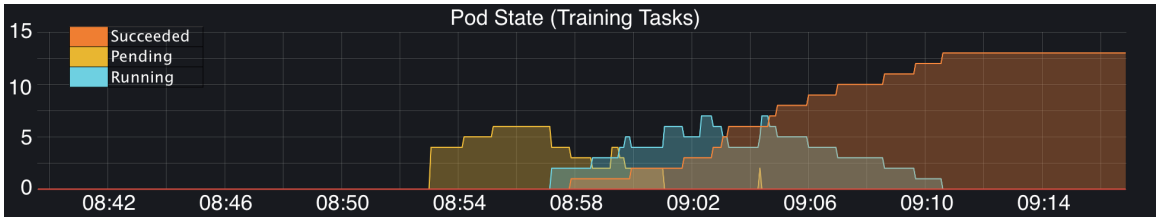

Figure 5.16: Scheduler Improved Experiment - Cloud Training Pod State

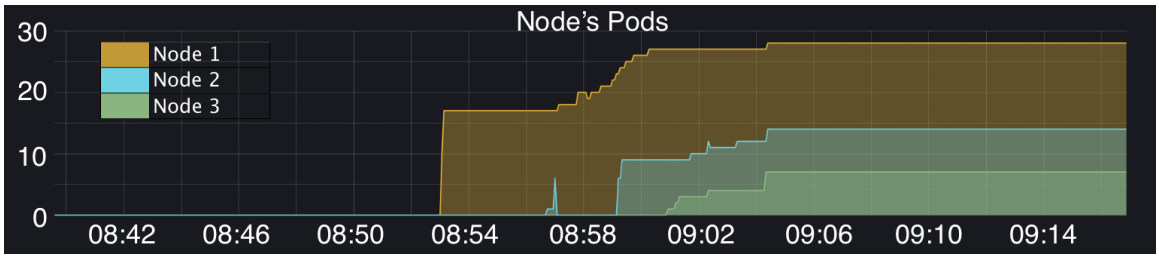

Figure 5.17: Scheduler Improved Experiment - Cloud Node Load

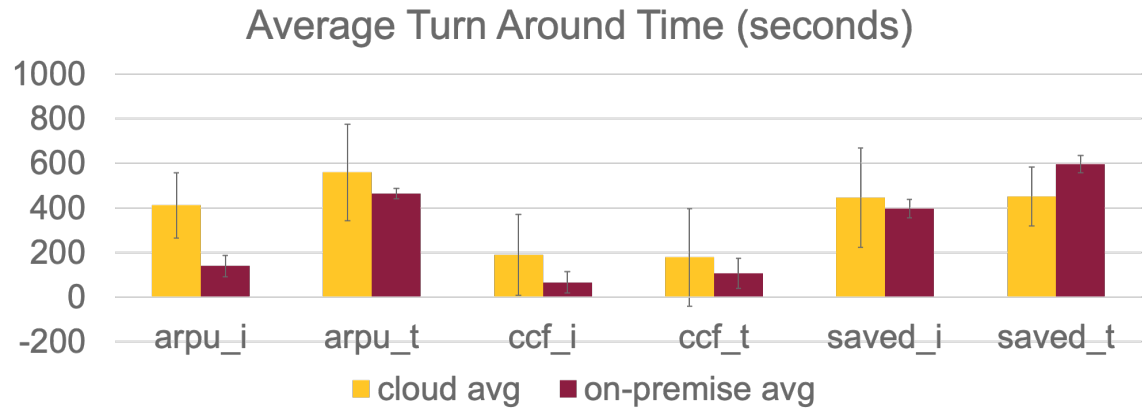

Figure 5.18: Scheduler Improved Experiment - Model Performance

cluster is huge. This is address by HCS' auto-scaling. As the pending queue continues to remain large, we can see two auto-scaling events. The addition of two new worker nodes and their load is displayed in Figure 5.17. The yellow line represents the number of pods scheduled to node 1, the blue line represents the number of pods scheduled to node 2, and the green represents the number of pods scheduled to node 3. As new nodes are added to the cluster, there are drastic drops in pending pods as they transition to running pods.

Figure 5.18 shows the ML jobs performance.

The total time taken to process the workload between the on-premise and cloud clusters was 1847 seconds. Node 1 is active for 1440 seconds, node 2 is active for 1080 seconds, and node 3 is active for 780 seconds.

The HCS Improved regional cost optimization saved 16% of the cost for the t2.medium master node and 20% savings on the c5.4xlarge worker node as shown

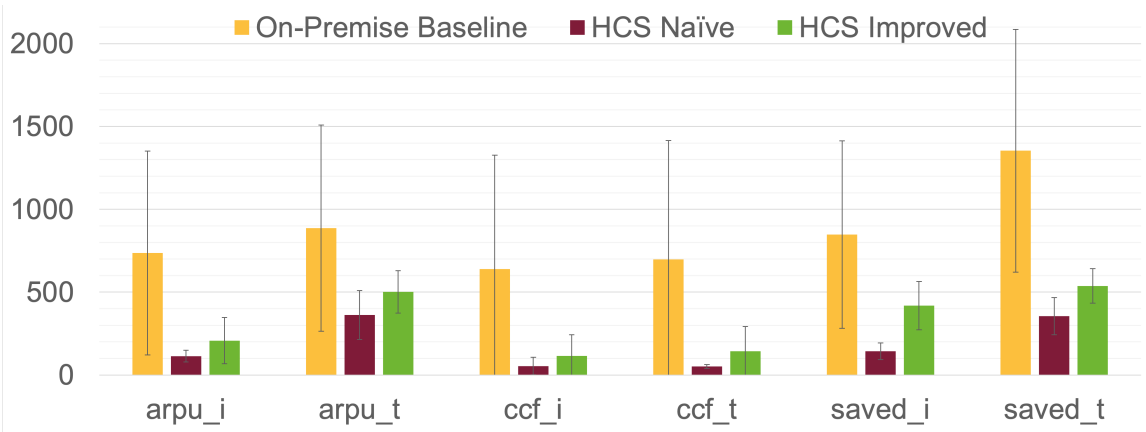

Figure 5.19: Final Experiment Comparison, Average Turn Around Time (seconds)

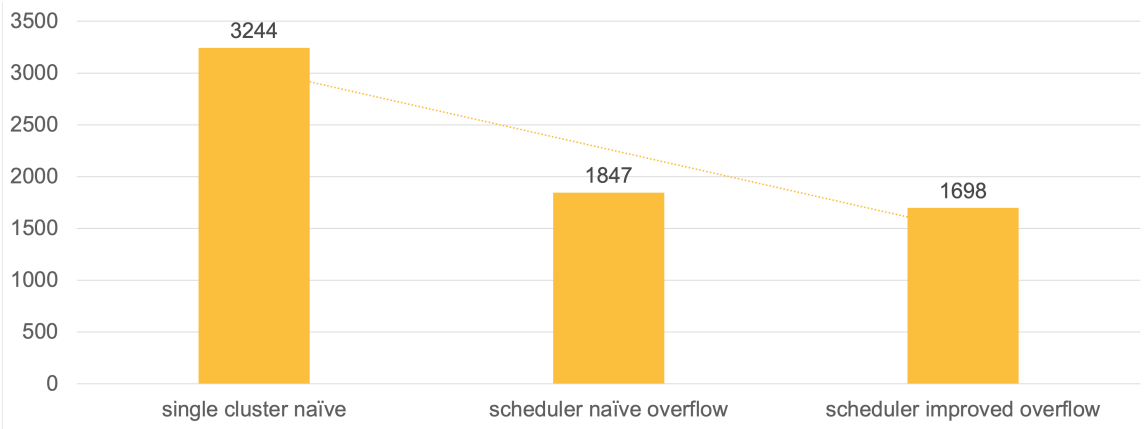

Figure 5.20: Time Savings, Total Experiment Run time (seconds)

in Figure 5.21. The HCS Improvements saved 48% execution time over the On-Premise baseline and 8% over the HCS Naive as shown in Figure 5.20. The HCS Naive experiment has the lowest average turnaround time per model, as shown in Figure 5.19. Each of the machine learning tasks had the most significant average turnaround time in the on-premise baseline experiment.

The HCS Improved experiment had a lower total execution time while having a more significant average turnaround time due to the additional worker nodes added during the cloud cluster's horizontal auto scaling event. After each additional worker node was added, additional initialization steps were needed, including downloading the docker images for the first time and downloading the data dependencies for the

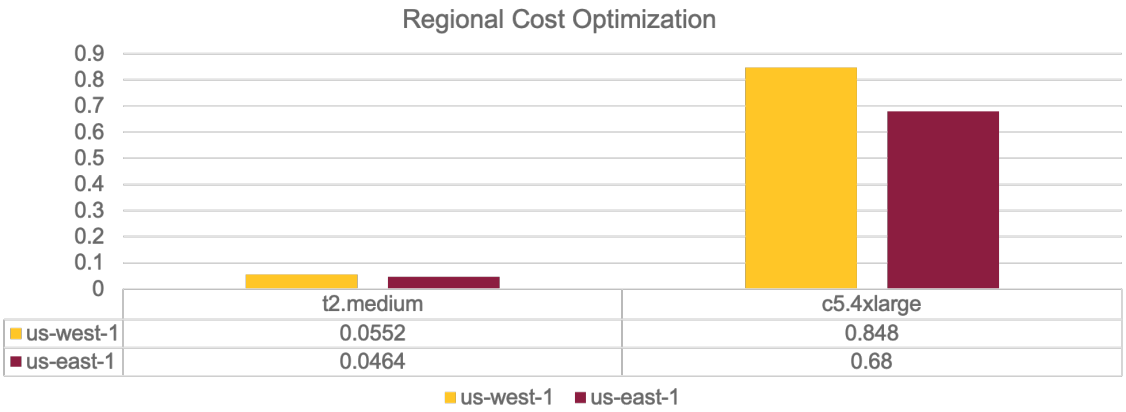

Figure 5.21: Regional Cost Savings, Total Cost Savings (USD/per hour)

first time. These initialization tasks increased the total average turnaround time. However, due to these expensive initialization tasks happening parallel to other worker nodes executing machine learning tasks, the total experiment execution time was still reduced.

Chapter 6

FUTURE RESEARCH

6.1 Regional

What impacts could confining a cluster to a single region have? Cloud providers today provide an ever-growing impressive list of services. Cloud providers bound support for each of those services by region. One service may be available in region A, while not available in region B. Users of Kubernetes bound to a single region may find that they need a service not support in their region. A naive solution to this problem would be to deploy a second cluster to the region where the desired service is available. This introduces the problem of needing to specify which cluster to deploy to. This challenge could be overcome by extending the proposed HCS. The user could specify what needs the workload has ex: [connect-to-cloud-service-x] the scheduler will be aware of n number of clusters, which clusters support cloud service x and dynamically schedule them accordingly. This allows the user scheduling workloads onto Kubernetes to specify the workload's needs and allow the scheduler to choose the appropriate cluster, just as the Kubernetes scheduler handles choosing the appropriate worker node.

6.2 Cloud Provider

Just as regional boundaries can impact what services are available to us, the same is valid with cloud providers. Kubernetes clusters deployed in AWS will have native connectivity to AWS services like AWS DyanmoDB. This, however, would limit the native connectivity to other cloud services such as Google Cloud Platform BigQuery.

Native connectivity can be defined as inter-data-center communication and platform-specific mechanisms to establish connectivity, authorization, and integration of the service easily. While some workloads will require service A (from cloud provider D)and others Service B (from cloud provider E), we can allow the multi-cluster scheduler to handle the placement of these workloads such that their service and cloud provider dependencies are met without the user manually specifying the exact cluster. This allows room for further scheduling optimizations while still satisfying the workloads requirements and no changes needed to the end Kubernetes user.

Suppose the HCS was enabled to review the cross-availability zone, cross-region, even cross-cloud provider data points at the time of scheduling and at the time of scale. Using these additional data points could make an additional optimized placement decision.

Chapter 7

CONCLUSIONS

This thesis's public contributions include: open-sourcing the Hybrid Cloud Scheduler developed, the code used to execute and analyze the results of each experiment Kieley (2021), reporting scheduling issues found to the Kubernetes GitHub project and sharing the statistical findings of the experiments run.

The open-sourced GitHub project includes nine git repositories. The tsis-scheduler repository contains the HCS scheduling logic. The tsis-orchestrator repository includes the automated steps of each experiment, including running the Workload Deployer and HCS. The tsis-create-cluster-build repository includes the source code of automated cluster creation and build steps to create the needed docker image file. The tsis-create-cluster repository includes the CircleCI CircleCI (2021) YAML to execute a parameterized cluster creation CircleCI job. The tsis-workload-deployer repository includes the source code of the Workload Deployer, including simulated job arrival behavior. The tsis-rest-api repository houses the HCS REST API component used to read and write to and from MongoDB the latest trained model information. The tsis-grafana-terraform repository hosts the terraformed Grafana dashboard configuration to reproduce the pod state over time charts. The tsis-reporter repository exports data about jobs and pods to excel after an experiment run. The tsis-clearer repository clears out experiment-specific data captured between experiment runs. These repositories could be used to rerun the experiments with different machine learning workloads or apply the HCS in new ways. The HCS was able to save up to 48% execution time by bursting to the cloud and up to 20% cost savings on cloud infrastructure.

BIBLIOGRAPHY

Burns, B. and C. Tracey, *Managing Kubernetes : operating Kubernetes clusters in the real world*, chap. 5 (O'Reilly Media, Sebastopol, CA, 2018a), an Overview of Scheduling.

Burns, B. and C. Tracey, *Managing Kubernetes : operating Kubernetes clusters in the real world*, chap. 5 (O'Reilly Media, Sebastopol, CA, 2018b), scheduling Process.

CircleCI (2021), "Continuous Integration and Delivery", `https://circleci.com/`.

et al., B. E. P. (2020a), "Kubernetes components .", `https://kubernetes.io/docs/concepts/overview/components/`.

et al., B. E. P. (2020b), "Pods", `https://kubernetes.io/docs/concepts/workloads/pods/#what-is-a-pod`.

et al., B. S. (2020c), "Kubernetes core generic scheduler logic", `https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/core/generic_scheduler.go#L112`.

et al., K. B. (2020d), "Building large clusters", `https://kubernetes.io/docs/setup/best-practices/cluster-large/`.

et al., T. B. (2020e), "Controllers", `https://kubernetes.io/docs/concepts/architecture/controller/`.

et al., T. T. (2020f), "Replicationcontroller — kubernetes", `https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/`.

GitHub (2021a), "kubernetes-sigs/kubefed", `https://github.com/kubernetes-sigs/kubefed`, original-date: 2018-01-09T14:21:03Z.

GitHub (2021b), "kubernetes/kops", `https://github.com/kubernetes/kops`, original-date: 2016-06-27T22:01:15Z.

Google (2021a), "Anthos Modern Application Platform", `https://cloud.google.com/anthos`.

Google (2021b), "Cloud Computing Services", `https://cloud.google.com/`.

Google (2021c), "Introducing fleets | Anthos", `https://cloud.google.com/anthos/multicluster-management/fleets`.

Google (2021d), "Kubernetes - google kubernetes engine (gke)", `https://cloud.google.com/kubernetes-engine`.

Huang, X. and X. Du, "Achieving big data privacy via hybrid cloud", in "2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)", pp. 512–517 (IEEE, 2014).

Kaggle (2021), "Kaggle: Your Machine Learning and Data Science Community", https://www.kaggle.com/.

Kieley, J. (2021), "Jkieley-Masters-Thesis", https://github.com/Jkieley-Masters-Thesis.

MacMillan, J. and G. Saenger (2017), "Kubernetes cluster federation", https://github.com/kubernetes/community/blob/master/contributors/design-proposals/multicluster/federation.md.

Meyerson, J., "The go programming language", IEEE software **31**, 5, 104–104 (2014).

MongoDB (2021), "The most popular database for modern apps", https://www.mongodb.com.

Volcano (2020), "Introduction", https://volcano.sh/en/docs/schduler_introduction/.

Wang, Y. (2019), "Priorityqueue is far from fifo for same priority pods", https://github.com/kubernetes/kubernetes/issues/83834.

Youssef, D. A. (2020), "Why the kubernetes scheduler is not enough for your ai workloads", https://www.cncf.io/blog/2020/08/10/why-the-kubernetes-scheduler-is-not-enough-for-your-ai-workloads/.