Cloud-Conductor: An Inter-Cloud Broker for Executing ML Pipelines

Sayak Maity*
smaity@college.harvard.edu
Harvard University
Cambridge, Massachussetts, USA

Andrew Sima*
asima@college.harvard.edu
Harvard University
Cambridge, Massachussetts, USA

Will Cooper*
wcooper@college.harvard.edu
Harvard University
Cambridge, Massachussetts, USA

Abstract

The use of multiple cloud platforms for machine learning model training can provide cost savings compared to using a single cloud platform. In this paper, we evaluate the cost effectiveness of using multiple cloud platforms for ML model training by conducting a cost analysis of several scenarios. We compare the cost of training ML models on a single cloud platform to the cost of training the same models on multiple cloud platforms, using a variety of different combinations of cloud providers. Our results show that, in many cases, using multiple cloud platforms for ML model training can provide significant cost savings compared to using a single cloud platform. For the balanced configuration, we found that a multi-cloud configuration can be 7.1% cheaper and 2.4% faster than the closest comparable single-cloud configuration. We also discuss the factors that can influence the cost effectiveness of using multiple cloud platforms for ML model training, and provide recommendations for optimizing the cost of ML model training in the cloud.

1 Introduction

Cloud computing has enabled individuals and organizations to perform large scale computations at a cheaper cost than ever before. Within the last couple of decades, primary cloud providers such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure, have built data centers which collectively span most of the globe, permitting users to achieve high throughput and low latency from nearly any geophysical location. However, as vast datasets and cloud applications requiring ultra-low latency become more common, more and more practitioners are adopting a multi-cloud approach [5]. In the prominent white paper published earlier in 2022, "The Sky Above the Clouds," authors identify four main reasons for simplifying multi-cloud usage. Firstly, lower barriers to cloud usage will enable more individuals and organizations to take advantage of cloud resources, thereby expanding the cloud computing market. Secondly, if separate clouds become interoperable, more specialized clouds will emerge; new cloud providers will be more easily able to enter the market and provide value by lowering costs for particular types of computation. Third, it will allow for better integration of various computing models including edge-computing and on-premise computing. Lastly, an established multi-cloud interface would simplify both resilience and security of cloud applications, as there would be less application downtime when one center goes offline [2]. We add the additional point that, as the amount of data being generated across the world is growing at an increasing rate, being able to perform non-locality and non-cloud provider sensitive computations will prove cost, energy, and time efficient.

In order to achieve the above goals of simplifying inter- and intra-cloud computing, Sky Computing proposes the integration of an "inter-cloud broker" – a middleware whose responsibilities include but are not limited to: understanding multiple cloud ecosystems and the underlying hardware compatibilities, providing users with an outward facing platform with which to specify jobs and constraints, and provisioning compute power and distributing jobs securely and optimally [2]. We believe that such a middleware need not require human interference. As long as we can automate procurement of accurate data on costs of computation and data transfer across clouds, pure software should be able to aptly achieve such goals. Furthermore, as we support the idea of lowering barriers to entrance for cloud computing, we believe such middleware should not charge users a fee. As such, our final code is open source.

Our project, Cloud-Conductor, consists of two relatively distinct portions. First, we conduct experiments to compare metrics such as cost and time requirements across a variety of instance types and machine types in AWS (Amazon Web Services) and GCP (Google Cloud Provider). Running common machine learning preprocessing and training scripts, we are able to empirically examine the cost efficiency of different combinations of pipeline setups. Second, we build a rudimentary CLI (command line interface), which given a set of computation scripts and a cost constraint, is capable of automating the process of instantiating instances, running code, and transferring data, while optimizing for cost. Cloud-Conductor is currently built specifically to handle multi-stage machine learning pipelines. In section II of this report, we describe some of our background research, touching on topics which have recently entered the scene as a result of the Sky Computing paper. In section III, we further discuss our implementation's design. In section IV, we discuss results from both our initial experiments and from

*All authors contributed equally to this research.

our CLI's execution. Finally, in section V, we conclude and provide recommendations for future steps.

2 Related Work

Authors of the Sky Computing paper hosted a "sky camp" at University of California Berkeley in October 2022, a few months after the paper was published. The conference was designed to reinforce some of the ideas presented in the white paper, and presented some novel software and frameworks which work towards the sky computing vision. Of the presentations, we found Skyplane to be the most developed idea, and Skypilot to be the most captivating and most closely aligned with the idea of an inter-cloud brokerage. In this section we will delve briefly into these projects' designs and how they influenced our own ideas.

2.1 Skyplane

Skyplane's main goal is to facilitate wide-area bulk data transfers [2]. Its implementation is in response to the recognition of large scale WAN data transfers as a necessity for underlying multi-cloud computing infrastructure; for example, if a hypothetical futuristic inter-cloud broker decides that a portion of a client's job is most efficiently run on one platform but a latter portion is better suited for another platform, a potentially very large amount of data would have to be transferred over a wireless network, which can be expensive and slow. Furthermore, increasing amounts of data, particularly image and video data, is far outpacing the rate at which WAN bandwidth is able to sustain; in fact, WAN bandwidth growth has been decelerating for many years already [7].

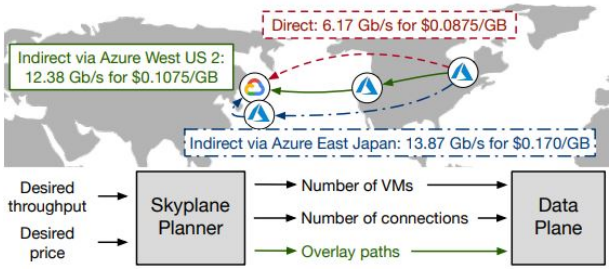

Figure 1. Skyplane's key insight, bulk data transfer overlay networks [4]

Although overlay networks have existed since the 2000s as a technique for application-level routing, Skyplane uses these overlay networks with price and elasticity in mind. Their key insight is that planning can be formulated with linear constraints, so that solving for the final overlay network can eventually be approximated by a relaxed linear program. When a client needs to transfer data from location A to location B, often these overlay networks can be calculated to determine an intermediate location C [2]. Solving the relaxed linear program achieves speedups from between 4.6X to 5.0X, depending on inter- or intra-cloud transfer. Using Skyplane's optimization model, clients can specify either a cost ceiling or a bandwidth floor with which they want their data to be transferred. We draw much of our inspiration from a similar notion: in a multi-stage computational job (any computational job which has natural partitioning points, such as a machine learning pipeline which includes preprocessing, training, and serving), some sort of path can be calculated to optimize for cost. Because we deal with the optimization of cost over computation rather than cost over pure data transfer, we actually have to take into account two cost factors: the cost of computation itself, as well as the cost of inter-cloud data transfer, when necessary.

2.2 Skypilot

Although Skypilot does not yet have an official paper out, we consider Skypilot to be the first tool created in the image of Sky Computing's intercloud broker proposal. When we first began to work on our own rendition of an intercloud broker, we were not aware of its existence, but since having looked more closely at their implementation, we have both gained deeper insight into building an intercloud broker, and have decided on a number of ways to set ourselves apart.

Skypilot's CLI takes a yaml file as input, which contains information such as number of virtual machines to instantiate, setup requirements, and the commands to run a certain script. In the background, Skypilot uses the Python linear programming library cxvpy in order to calculate optimization over different cloud providers. The optimization algorithm requires as input the runtime estimates of scripts, as well as the cost per unit time data for each of the instances being considered. From our deeper dive into Skypilot's inner workings, we were able to pinpoint three faults that we want to improve upon. Firstly, runtime estimates are *supplied by the user* [1]. This may lead to a faulty optimization and lower cost savings. Secondly, Skypilot does not actually account for data transfer and storage costs. For larger scale projects, this is an extremely important factor for the objective function! Finally, Skypilot only supports optimization over single scripts; as multi-part machine learning pipelines take up an increasingly large proportion of large volume cloud compute, optimization over multiple ordered scripts can lead to even further cost savings, in part because clients would not have to transfer data back to a local machine in between scripts. In our design section, we cover in detail how we implement basic versions of these changes.

3 Design

In this section, we will briefly describe our initial experiment design, as well as the results which inspired our CLI; our CLI will be described in more detail. The repository we will reference can be found here.

3.1 Experimental Design

Before beginning development of our inter-cloud framework, we conducted a series of experiments to evaluate the feasibility of lowering costs and increasing efficiency through the use of multiple cloud platform s for computation. We chose the MNIST digits dataset [3] as a basis for our experiment results, and selected two of the industry's largest cloud providers, AWS and GCP, as the platforms for our computational experiments.

Our initial assumption was that the variance in GPU cost across the different platforms would be the primary factor in the effectiveness of our framework, Cloud-Conductor. By carefully distributing tasks that can be more efficiently performed on GPUs, we believed that Cloud-Conductor would be able to make up for the time and resources lost during data transfer overhead. To test this, we developed experimental code that included a data augmentation preprocessing script and a training script written with Tensorflow.

We executed these scripts across a range of instances on both AWS and GCP, including c4, c5n, g4, and g5 on AWS, and e2 on GCP. We also experimented with a TPU instance on GCP. Our scripts kept track of the time spent on each task, and we used the respective billing APIs of the platforms to determine the costs of different combinations.

After analyzing the results, we were able to show the feasibility of lowering costs through the use of multi-cloud deployment while maintaining similar performance. In certain cases, with constraints on the number of CPUs available, distributing a preprocessing and training pipeline across AWS and GCP was up to two times cheaper than using only AWS. This demonstrated the potential for Cloud-Conductor to significantly reduce costs for certain computational tasks.

3.2 CLI Design

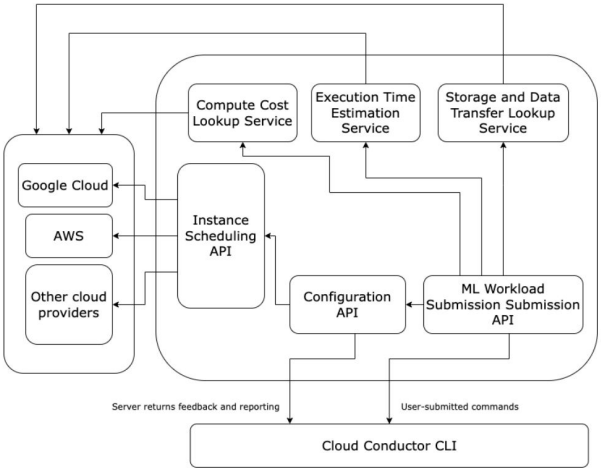

Figure 2. Cloud-Conductor's system design

With the Cloud-Conductor command-line interface (CLI), users are able to specify a bundle of scripts and the desired order in which they should be run across different cloud providers. The Cloud-Conductor configuration API automatically detects the user's initial data size, which our time-estimator.py script uses to provide estimates on how long tasks will take on various machines. Unlike Skypilot's implementation, which requires users to manually input estimates of task completion times, Cloud-Conductor uses the input data size, along with measured data from our initial experiments, to determine runtime estimates. Due to the time constraints of this project, we have specialized time-estimator.py to have specific functions for estimating runtimes for preprocessing tasks and training tasks. These functions reference a JSON file containing our empirical values for processing capacity across CPUs and GPUs, although a client may choose to manually input their own values. This allows for more accurate estimates of task runtimes, which in turn allows for more efficient use of computational resources across multiple cloud platforms.

Once we have obtained time estimates for the various tasks to be performed across different types of instances on different cloud providers, this information is passed to our optimization script. The optimization script uses these parameters as input, along with values from our Compute Cost Lookup Service, which is able to specify the cost per unit time of running any particular instance. Currently, our Compute Cost Lookup Service is a JSON file with values that have been updated in the past couple of months. Similarly, Cloud-Conductor's Storage and Data Transfer Lookup Service references a JSON file to determine the extra costs associated with transporting data from one cloud provider to another. This is in contrast to Skypilot's implementation, which does not consider data movement costs (although it is less necessary for an application that is running a single script).

Unlike Skyplane's optimizer, we do not formulate our optimization problem as a relaxed linear program. We consider two methods for our algorithm instead: a greedy search algorithm and a constraint satisfaction search algorithm. The problem contains the following constants:

$c \in CP$ (cloud provider c from set CP)
$r \in R$ (region r from set R)
$i \in I$ (instance type i from set I)
$t \in T$ (task t from set T)
$et_{t,c,r,i} \in ET$ (estimated runtime of task t on cloud provider c in region r on instance i)
$ec_{t,c,r,i} \in EC$ (unit cost of running a task on cloud provider c in region r on instance i)
$edt_{t,(c_1,r_1,i_1),(c_2,r_2,i_2)} \in EDT$ (estimated data transfer time of task t from machine 1 to machine 2)
$edc_{t,(c_1,r_1,i_1),(c_2,r_2,i_2)} \in EDC$ (estimated data transfer cost of task t from machine 1 to machine 2)

Typically the specific regions and instance types will vary with the cloud provider, though for simplification of notation we do not take this into account in this formulation. Estimated runtime of tasks, unit cost of running tasks, and data transfer values of tasks from one set of {c, r, i} to another are provided by our time estimation service, Compute Cost Lookup Service, and Data Transfer Lookup Service. The optimization problem's decision variables are the following binary variables:

$X_{t,c,r,i} \in X$ (binary variable representing whether or not to run task t on cloud provider c in region r on instance i)

There are 2 simple constraints: every job must be run exactly once on a machine, and the jobs must execute in the order specified by the user. As of now, our problem's CLI only uses an objective function to minimize the overall cost of running the pipeline, although the code supports three other types of objective functions, including minimizing time, maximizing value (equivalent to minimizing the product of time * cost), or minimization of a custom weighting of time and cost, where the user decides which weight to use.

We first consider a greedy method. The greedy method simply computes the best machine to run each task on according to the objective function. As large cloud providers often charge egress fees but not ingress fees, the greedy algorithm factors in egress fees to the cost that is associated with running a task on a particular machine, thus accounting for data transfer overhead [4]. This method has an extremely fast runtime on the user's local computer; only (# tasks * # of cloud providers * # of regions * # of instance types) options need to be searched. One drawback of this approach, however, is that in order to ensure optimality, we need to make the assumption that egress fees from a particular region and cloud provider remain constant no matter where the destination is located; otherwise, a greedy method that performs entirely separate calculations for each task will not be accurate because egress fees cannot be calculated correctly. With the relaxed assumption that data egress fees to any regions will not outweigh the price of compute, this method could prove useful.

Alternatively, in our actual current implementation, we formulate the optimization problem as a constraint satisfaction problem. There are a few reasons as to why such a formulation may be more effective than either the aforementioned greedy method or Skyplane's relaxed linear program. Firstly, if the time estimation values and cost estimation values are reliable, an exhaustive CSP state space search guarantees an optimal solution with the lowest objective value. Secondly, using a state space search enables more complex conditional constraints to be easily applied. For example, if a new cloud provider whose services do include ingress fees, it would be straightforward to take this into account. This method is also able to take into account varying egress fees.

While our optimizer's search algorithm has a higher time complexity than linear programming algorithms due to the nature of an exhaustive search, we note that the number of states the algorithm must search is limited by the number of scripts in the pipeline, as well as the number of types of instances that a user is considering using. Assuming that a pipeline contains no more than 5 scripts, and that we perform a search across <20 types of machines per cloud provider, modeling the optimization problem as a constraint satisfaction problem will still be able to run quickly on a user's local machine. We provide a simple heuristic that truncates branches of the search when the accumulated cost rises above the final cost of a previous solution, further improving the efficiency of the optimization process.

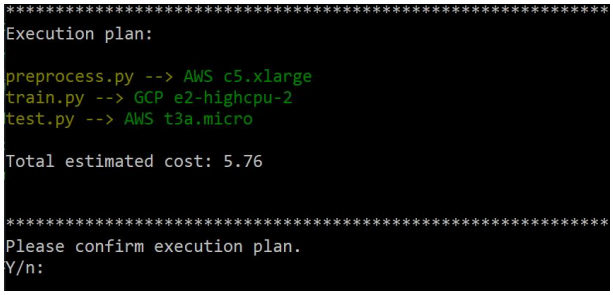

Figure 3. Example image of Cloud-Conductor's CLI in action

The output of the optimizer is in the form of a matrix, where each row represents a task and each column represents a particular instance. Each row contains a single value of 1, indicating which instance the respective task should be run on. At this point, Cloud-Conductor will display the execution plan and total estimated cost of computation and prompt the user for execution confirmation. This output is then interpreted by the main script, which instantiates instances accordingly via the AWS and GCP Python libraries. In order to optimize for cost, Cloud-Conductor only instantiates an instance right before it is needed, rather than instantiating all instances once confirmation is received from the user. Although this adds instance-transition overhead, we believe it to be the most cost-effective choice, because starting a machine and waiting for it to warm up typically takes less than a minute. For our rudimentary implementation, we have simplified the process of actually running code on these instances by using an SSH client, paramiko, on the user's local machine to execute commands remotely. For AWS, a security key pair containing the pem file for an instance and a security group dictating the instance's traffic protocols are generated before the instance begins to run, so that paramiko can remotely run commands on the instance. Similarly, when a GCP instance needs to be used in the pipeline, a key is generated. Once a pipeline has finished executing, Cloud-Conductor handles cleanup by terminating

instances, relieving the user of the need to manually manage these resources, avoiding additional costs.

We would like to note here that Cloud-Conductor does not handle or facilitate the process of acquiring or integrating AWS or GCP credentials; instead, the CLI assumes the user already has these credentials configured.

4 Evaluation

We study whether using multiple public clouds for a machine learning pipeline (including preprocessing and training) can be more cost-effective and faster than using a single public cloud. In this evaluation section, we will present the results of our experiments that compare the performance and cost of using multiple public clouds versus a single public cloud for preprocessing and ML training tasks. We will also discuss the implications of these results and provide insights on the potential benefits of using multiple public clouds for these types of tasks.

4.1 Preprocessing & Data Augmentation

Our preprocessing script [6] is a parallelized script that performs data augmentation for the MNIST dataset [3]. We leverage Python's `multiprocessing` library, which provides tools for easily parallelizing scripts, allowing them to run on multiple cores simultaneously. Once all of the workers have completed the data augmentation on their assigned chunks, the script uses the get method of the worker pool to retrieve the augmented data from each worker and combine it into a single dataset. This dataset will be 9x larger as a result of the data augmentation process, and it could be used for ML training. Overall, using a parallelized script for data augmentation speeds up the process and makes it more efficient, allowing for faster and more cost-effective training of ML models on the MNIST dataset.

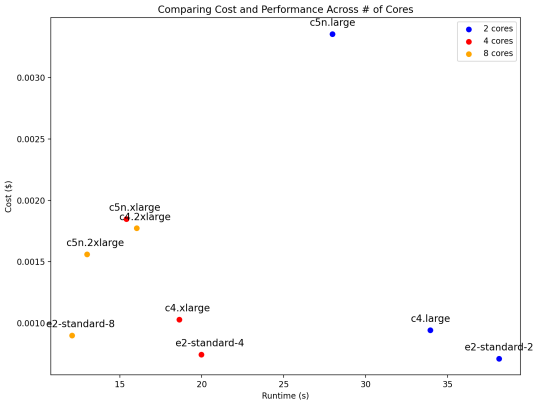

Figure 4. Comparing Cost and Performance Across # of Cores

Our methodology involved running 10 trials of preprocessing. The figure above represents our averaged results of the trial runs. We observed that because of the parallelization and relatively good scalability, using an instance with more cores can also be more cost-effective when running on public clouds. This is because public cloud providers like AWS and GCP charge customers based on the amount of time their software is running, so using more cores to complete tasks faster can help reduce the overall cost of running the software on the cloud. Typically the pricing models on the cloud providers is that using twice as many cores will cost twice as much, but the parallelized code is theoretically able to run twice as fast so the cost works out to be similar. In practice, scalability isn't perfect because of overhead and parts of the process that aren't fully parallel, but it isn't far off from this theoretical framework. Out of the GCP E2 instances, we see that there are similar costs between the 2, 4, and 8 core versions, but the 8 core runs significantly faster than the 4 or 2 core. For the AWS C5n instance, we see that the 8 core version is cheaper and faster than the 4 core and 2 core versions. Also, for the AWS C4 instance, we see that the 2 and 4 core versions cost about the same, but the 4 core is significantly faster. The 8 core version of the C4 instance is slightly faster but also more expensive. The takeaways here are that 8 and 4 cores for preprocessing make more sense than 2 cores. We tested larger amounts of cores like 16 cores but they happen to be more expensive to run because the parallel scalability doesn't give enough of a speedup to offset the cost of the more expensive rates for larger instances. The takeaways here are that we'd want to focus on 4 and 8 core configurations for performing the processing. Also, the fastest and near-cheapest server for this was the GCP e2-standard-8 in our testing, and GCP instances tended to be cheaper than the AWS equivalents. Hence, from this experiment, we would expect that a GCP E2 instance would be the better choice for doing preprocessing.

4.2 Training

For training, we compared the cost and performance of GCP TPUs, AWS G5 instances (which use an NVIDIA A10G GPU), and AWS G4 instances (which use an NVIDIA T4 GPU), running 10 trials of each cloud machine configuration on our implementation of the ResNet-18 model [6] for the MNIST dataset [3]. The figures below represent our averaged results of the trial runs.

A TPU, or Tensor Processing Unit, is a specialized type of hardware designed specifically for running machine learning (ML) models. The TPU provides performance gains over the traditional GPU-based systems like AWS G5 and G4. This is because TPUs are specifically designed for this type of workload and can process data faster than GPUs.

In terms of cost, however, an AWS G5 and G4 instance is more affordable than a TPU. The TPUs is a specialized piece of hardware and is more expensive to rent. AWS G5 and G4

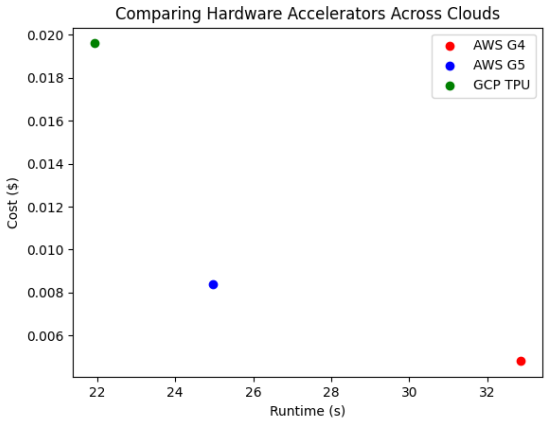

Figure 5. Comparing Hardware Accelerators Across Clouds

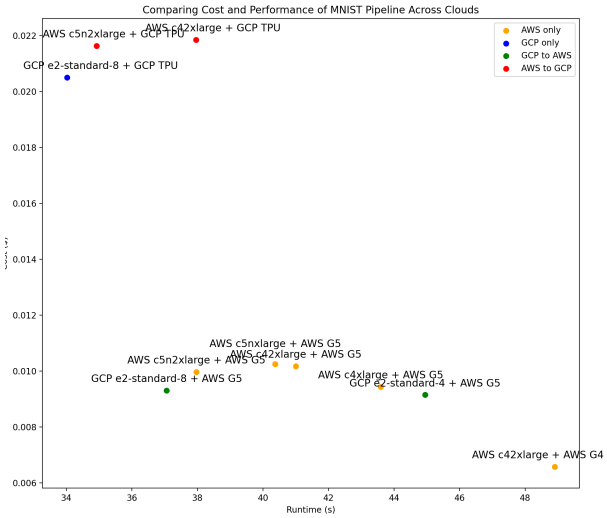

Figure 6. Comparing Cost and Performance of Pipeline Across Clouds

instances, on the other hand, are more widely available and can be rented at a lower cost. We observe that the TPU is twice as expensive as the G5 instance, but is only 12.8% faster than the G5 for this task.

Overall, the decision to use a GCP TPU, AWS G5 instance, or AWS G4 instance will depend on the specific needs and requirements of a given ML project. The relative performance is highly dependent on the specific workload that is run. Hence, if someone wants maximum performance, they would choose a TPU, but for this use case most practitioners would likely prefer the cost/performance tradeoff offered by a G5 instance. If cost is a serious priority, then there is a case for the G4 instance to be chosen as well. The takeaway here is that running the ML training on AWS would make sense for most users instead of leveraging GCP TPUs.

4.3 Results

In this experiment, we compared the performance and cost of different cloud instances for running an ML pipeline involving preprocessing and training. Based on the results of the previous two experiments, it appears that using the cheaper CPU instances of Google Cloud Platform (GCP) and the cheaper GPU instances of Amazon Web Services (AWS) would provide a desirable balance between cost and performance.

Indeed, this turned out to be the case when testing the full pipeline. We found that the combination of GCP E2 and AWS G5 instances was one of the fastest configurations, only trailing two configurations that used Tensor Processing Units (TPUs). However, the TPU configurations were significantly more expensive for the slight performance improvement they provided. In contrast, the combination of GCP E2 and AWS G5 instances was both faster and cheaper than the AWS-only configurations, even while accounting for data transfer costs and time.

Overall, our experiment suggests that using a combination of GCP E2 and AWS G5 instances can provide a good balance between cost and performance for an ML pipeline involving preprocessing and training. It turns out to be 7.1% cheaper and 2.4% faster than the AWS C5n and AWS G5 configuration. At the same time, the performance of the GCP E2 and AWS G5 combination approached the performance of significantly more expensive trials with TPU configurations.

5 Conclusion

In its early stage, Cloud-Conductor is a valuable tool for optimizing the use of cloud resources for running scripts. By taking a set of scripts as input and utilizing cost and performance data from multiple cloud providers, Cloud-Conductor is able to determine the best cloud providers to run the scripts on, resulting in improved efficiency and cost savings. Our experiments and implementation demonstrate the feasibility and potential benefits of managing computational tasks across multiple cloud platforms as opposed to a single cloud platform.

In the future, we'd like to explore and experiment with distributing the workload across different regions, which can help to reduce the overall cost of running the ML pipeline. For example, if preprocessing and training tasks are distributed across different regions, it may be possible to take advantage of lower-cost regions for certain tasks and reduce the overall cost of running the ML pipeline.

Also, using multiple clouds could also provide redundancy and fault tolerance, which can help to ensure that the ML pipeline is able to continue running even if one of the clouds experiences an outage or other issue. This can help to prevent costly downtime and ensure that the ML pipeline is able

to complete its tasks as efficiently and cost-effectively as possible. We would like to implement and test this type of functionality in the future.

References

[1] 2022. Skypilot. https://github.com/skypilot-org/skypilot

[2] Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E. Gonzalez, Joseph M. Hellerstein, Michael I. Jordan, Anthony D. Joseph, Michael W. Mahoney, Aditya Parameswaran, David Patterson, Raluca Ada Popa, Koushik Sen, Scott Shenker, Dawn Song, and Ion Stoica. 2022. The Sky Above The Clouds. https://doi.org/10.48550/ARXIV.2205.07147

[3] Li Deng. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29, 6 (2012), 141–142.

[4] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. 2022. Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays. https://doi.org/10.48550/ARXIV.2210.07259

[5] Paul Miller. 2018. A clear multicloud strategy delivers business value. https://www.forrester.com/report/A-Clear-Multicloud-Strategy-Delivers-Business-Value/RES128781

[6] Andrew Sima, Will Cooper, and Sayak Maity. 2022. Cloud Conductor. https://github.com/bbytiger/cs243_final_proj

[7] Ashish Vulimiri, Carlo Curino, P. Brighten Godfrey, Thomas Jungblut, Jitu Padhye, and George Varghese. 2015. Global Analytics in the Face of Bandwidth and Regulatory Constraints. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 323–336.

Received 11 December 2022