SkyPilot: An Intercloud Broker for Sky Computing

Zongheng Yang*, Zhanghao Wu*, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj,
Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker[§], Ion Stoica
University of California, Berkeley [§]*UC Berkeley and ICSI*

Abstract

To comply with the increasing number of government regulations about data placement and processing, and to protect themselves against major cloud outages, many users want the ability to easily migrate their workloads between clouds. In this paper we propose doing so not by imposing uniform and comprehensive standards, but by creating a fine-grained two-sided market via an *intercloud broker*. These brokers will allow users to view the cloud ecosystem not just as a collection of individual and largely incompatible clouds but as a more integrated Sky of Computing. We describe the design and implementation of an intercloud broker, named SkyPilot, evaluate its benefits, and report on its real-world usage.

1 Introduction

The modern information infrastructure is built around three components. The Internet provides end-to-end network connectivity, cellular telephony provides nearly ubiquitous user access via increasingly powerful handsets, and cloud computing makes scalable computation available to all. These ecosystems obviously have many superficial differences, but perhaps their most fundamental difference lies in the degree of compatibility between providers in each of these ecosystems.

The Internet and the cellular infrastructure were designed with the goal of universal reachability. This required both uniform and comprehensive industry standards and broadly-adopted interconnection agreements (for Internet peering and cellular roaming) that led to a globally connected federation of competing providers. The cloud ecosystem has very different origins, emerging as a replacement for dedicated on-premise computing clusters rather than serving as an interconnected communication infrastructure. As a result, cloud providers began by emphasizing their differences rather than their similarities; though the clouds are all based on the same basic conceptual units (e.g., VMs, containers, and now FaaS), they initially differed greatly in their orchestration interfaces. These orchestration interfaces have become more similar over time, but some clouds continue to differentiate themselves through numerous proprietary service interfaces, such as for storage or key-value stores. In addition, clouds typically impose much higher charges on data leaving than on data entering, resulting in "data gravity" (i.e., the difficulty of moving jobs to another cloud due to the expense of transferring the data). The combination of proprietary service interfaces and data gravity have led to significant customer lock-in: it is hard for companies who have established their computational workloads on one cloud to move them to another.

However, as cloud computing has become a critical part of our computational infrastructure, enterprises are increasingly worried about how difficult it is to migrate workloads between clouds. There are two compelling reasons for wanting more freedom in workload placement. First, no business wants any critical part of their infrastructure tied to a single provider because such lock-in reduces their negotiating leverage and also makes the business vulnerable to large-scale outages at the provider. Second, there are now strict regulations about data and operational sovereignty that dictate where data can be stored and computational jobs run. Not all cloud providers have datacenters in all countries, so the inability to migrate jobs between cloud providers could be a painful roadblock to satisfying these new regulations. These two reasons are not theoretical problems whose solutions would be "nice-to-have"; the recent occurrence of large-scale cloud outages and the increasing number of government regulations are quickly making such a solution a "must-have" for large-scale users of the cloud. This paper is about how we can ease the migration of workloads through the rise of Sky Computing, a concept first introduced in [81] but significantly extended and more deeply explored here. Sky Computing is when users, rather than directly interacting with the cloud, submit their jobs to what we call *intercloud brokers* who handle the placement and oversee the execution of their jobs.

To explain our approach in more depth, we first review related concepts and recent developments (§2). We then (§3) describe our vision of Sky Computing and its transformative possibilities. We present the requirements, architecture, and

*Equal contribution.

implementation of an intercloud broker, named SkyPilot, that focuses on computational batch jobs (§4). We then demonstrate its benefits on several applications (§5). Finally, we share our experiences with early deployments (§6), survey related work (§7), and conclude (§8). While the body of this paper is devoted the technical characteristics of our system, in the appendix (§A.1) we speculate on how the cloud ecosystem might evolve once Sky Computing is more widely adopted.

SkyPilot is open source and available at `https://github.com/skypilot-org/skypilot`.

2 Related Concepts and Recent Developments

In this section we first review two concepts related to the ability to migrate workloads – standards and multicloud – and then discuss the recent progress towards compatibility.

2.1 Why Not Just Adopt Standards?

The first question one might ask is if seamless migration is the goal, why not adopt a set of uniform and comprehensive cloud standards, as was done for the Internet and cellular? In fact, a decade ago IEEE proposed a set of Intercloud standards for portability, interoperability, and federation among cloud providers [88] involving an Intercloud Service Catalog and an Intercloud federation layer. There are two fundamental problems with this and other proposals for such uniform and comprehensive cloud standards. First, there is no incentive for the dominant clouds (i.e., those with large market shares) to adopt such standards; it would decrease their competitive advantage and make it easier for customers to move their business to other clouds. Second, users interact with clouds at many levels, using high-level service interfaces such as PyTorch [76] or TensorFlow [53] in addition to low-level orchestration interfaces such as Kubernetes [36]. If the goal is to make workload migration seamless, then all of these interfaces would need to be standardized. Requiring every cloud to standardize every interface is both unrealistic (as noted in the first objection) and unwise (because these higher-level interfaces have changed significantly over time, and standardizing them would greatly hinder innovation).

2.2 Why Isn't This Just Multicloud?

Multicloud is now an industry buzzword, and there are reports [33, 52] that most enterprises have, or will soon have, multicloud deployments; this would seemingly imply that our goal of seamless workload migration has already been realized. However, the common use of the term multicloud only requires that an enterprise have workloads on two or more clouds (e.g., the finance team runs their backend functions on Amazon while the analytics team runs their ML jobs on Google), *not* that they can easily move those workloads between clouds. It is clear, from everyone we have talked to in the industry, that moving many workloads between clouds remains difficult. The exceptions to this are the recent third-party offerings (e.g., by Trifacta, Confluent, Snowflake, Databricks, and others) that run on multiple clouds; users can indeed migrate their workloads that only use these services between clouds relatively easily (BigQuery, offered by Google, offers similar cross-cloud support). However, these are for specific workloads, and do not provide general support for workload migration.

In addition, there are several programming or management frameworks that support multiple clouds. JClouds [8] and Libcloud [10] offer portable abstractions over the compute, storage, and other services of many providers. However, the user still does the placement manually, whereas automatic placement is a key feature of Sky Computing. On the management front, Terraform [51] provisions and manages resources on different clouds, but requires the usage of provider-specific APIs, and also does not handle job placement. Kubernetes [36] orchestrates containerized workloads and can be run across multiple clouds (e.g., Anthos [5]). These frameworks, while quite valuable, focus on providing more compatibility in the lower-level infrastructure interfaces offered by the clouds (see §2.3), and as such are nicely complementary with Sky Computing but do not obviate the need for Sky Computing.

2.3 Growth In Interface Compatibility

Turning away from related concepts, we now discuss a recent development that Sky Computing will leverage. As noted before, users of cloud computing invoke a wide variety of computational and management interfaces. Many of these are open source systems that have become the *de facto* standards at different layers of the software stack, including operating systems (Linux), cluster resource managers (Kubernetes [36], Apache Mesos [63]), application packaging (Docker [27]), databases (MySQL [41], Postgres [43]), big data execution engines (Apache Spark [93], Apache Hadoop [89]), streaming engines (Apache Flink [57], Apache Spark [93], Apache Kafka [9]), distributed query engines and databases (Cassandra [7], MongoDB [39], Presto [44], SparkSQL [48], Redis [45]), machine learning libraries (PyTorch [76], TensorFlow [53], MXNet [58], MLflow [38], Horovod [79], Ray RLlib [66]), and general distributed frameworks (Ray [71], Erlang [55], Akka [1]). In addition, some of AWS's interfaces are increasingly being supported on other clouds: Azure and Google provide S3-like APIs for their blob stores to make it easier for customers to move from AWS to their own clouds. Similarly, APIs for managing machine images and private networks are converging.

These trends increase what we call *limited interface compatibility*, where both of these qualifiers are crucial. This compatibility applies only to individual interfaces and these interfaces are typically not supported by all clouds but by more than one. Our contention, based on what we see in the ecosystem, is that the number and the usage of these interfaces that have this limited compatibility – i.e., are supported on more than one cloud – is increasing, largely but not exclusively due to open-source efforts.

We are basing our approach on the belief that this trend will

continue, and that leveraging this trend is far preferable to pursuing uniform and comprehensive standards. To paraphrase a quote attributed to Lincoln, we know that all interfaces are supported by some clouds, and some interfaces may be supported by all clouds, but we cannot and should not require that all interfaces be supported by all clouds.[1]

3 The Vision of Sky Computing

We first describe what Sky Computing is, and articulate why we see it as not just tactical but transformative.

3.1 What Is Sky Computing?

Given this increasing level of limited interface compatibility, how do we leverage it to ease workload migration? There are two key components. First, in order to reduce data gravity, clouds can enter into reciprocal free data peering; i.e., two clouds can agree to let users move data from one cloud to another without charge. With high-speed connections prevalent (many clouds have 100 Gbps connections to various interconnection points where they can peer with other clouds), we think such free peering can easily be supported, with its costs more than offset by the increase in computational revenue that it enables. One might worry about the delay that such transfers incur, but if the resulting computation times are superlinear in the data size (or linear with a reasonably high constant) then no matter how large datasets become, the networking delays will not be a major bottleneck.

The second component, and the one we focus on for the rest of this paper, is what we call *intercloud brokers*. In this paper we describe our intercloud broker, which is designed specifically for *computational batch jobs* (§4). While batch jobs (e.g., ML, scientific jobs, data analytics) represent only a fraction of today's diverse cloud use cases, their computation demands are growing quickly [74] and are responsible for the recent surge of specialized hardware [15, 22, 23]. Thus, we have started with a broker designed for batch jobs as a tractable but common and rapidly growing workload. We expect future versions of the broker will address a wider range of workloads, and provide a broader set of features, but that is not our focus here. In addition, we expect that eventually there will be an open market in intercloud brokers that charge a small fee for their brokerage service; some of those brokers will be general purpose and others more tailored to specific workloads, as ours is.

An intercloud broker takes as input a computational request that is is specified as a directed acyclic graph (DAG) in which the nodes are *coarse-grained* computations (e.g., data processing, training).[2] For lack of a better term we call these computations "tasks". The request also includes the user's preferences about price and performance.

[1] The following adage is widely but incorrectly attributed to Lincoln: "You can fool part of the people some of the time, you can fool some of the people all of the time, but you cannot fool all the people all of the time."

[2] This is informed by workflow systems [6] that are now the de facto standard for orchestrating complex batch applications.

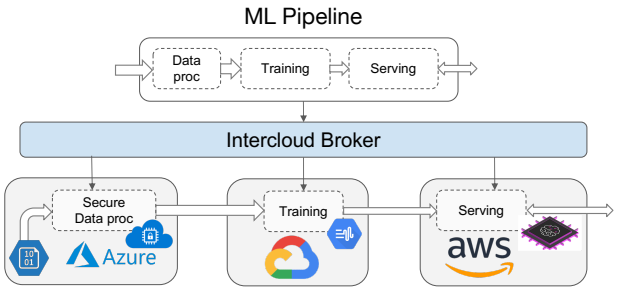


Figure 1: **An ML pipeline running on top of Sky.** The goal is to minimize cost while processing the input data securely.

The intercloud broker is then responsible for placing these tasks across clouds. Unlike existing multicloud applications which run an application instance per cloud, an intercloud broker can run a single application instance across several clouds. For example, Figure 1 shows a machine learning (ML) pipeline with three tasks: data processing, training, and serving. The user may wish to minimize the total cost while processing data securely. The intercloud broker might decide to run data processing on Azure Confidential Computing [16] to anonymize data and thus protect data confidentiality, training on GCP to take advantage of TPUs [23], and serving on AWS to take advantage of the Inferentia accelerator [15].

The ability to partition applications enables the emergence of specialized clouds. For example, a cloud provider can build a successful business by just focusing on a single task, such as ML training, and offering the best price-performance for that task; see §A.1 for a more detailed discussion of this.

In addition, the intercloud broker provides benefits even when the application (i) entirely runs on a single cloud, by automatically choosing the cloud that best matches the user's preferences and choosing the best region and zone within that cloud, or (ii) uses services[3] provided only by a single cloud, by placing a task on that cloud but still having the freedom to use other clouds for the other tasks.

3.2 Why Is This Transformational?

There are three reasons, each from a different perspective, why we see this as a transformational change in cloud computing, not as merely a tactical mechanism for workload migration.

User's Perspective: When using an intercloud broker, users are no longer interacting with individual clouds, but with a more integrated "Sky" of computing. They merely specify their computation and their criteria, and the broker then places the job. This makes it significantly easier to use the cloud, and may lead to increased cloud adoption. Note that such an interface hides the heterogeneity between and within clouds. Users no longer need to research which clouds have the best prices, or offer a particular service. This also applies *within* individual clouds, because different regions within a cloud

[3] By "service" we mean the compute services or a hosted service provided by one or more clouds, such as hosted Apache Spark (e.g., EMR [4], HDInsight [17]) and hosted Kubernetes (e.g., EKS [3], GKE [32], or AKS [18]).

can offer different hardware options and different prices.

Competitive Perspective: Note that by serving as an intermediary between users and clouds, the intercloud broker is creating a fine-grained two-sided market for computation: users specify their tasks and requirements, and clouds offer their interfaces with their pricing and performance. Job placement is no longer driven mostly by measures to promote lock-in (e.g., proprietary interfaces and data gravity), but increasingly by the ability of each cloud to meet the user's requirements through faster and/or more cost-efficient implementations. This means that the clouds, in order to increase their market, will likely start supporting interfaces that are commonly used in jobs, driving the market towards increased compatibility.

Ecosystem Perspective: Once there is a two-sided market established, the cloud ecosystem can transition from one in which all clouds offer a broad set of services and try their best to lock customers in, to one in which many clouds focus on becoming part of a computational Sky, where they can specialize in certain tasks because the intercloud broker will automatically direct computations to them if they best meet user needs for those particular tasks; the economic analysis in the appendix (§A.1.2) makes this case more precisely.

This vision should be tempered with several doses of reality. First, while we envision some clouds will embrace the vision of Sky Computing by focusing on compatible interfaces and adopting reciprocal free data peering, we expect others, particularly those with dominant market positions, to continue with lock-in as a market strategy. Nonetheless, the presence of a viable alternative cloud ecosystem will set the bar for innovation and meeting user requirements, so all users will benefit. Second, we assume that the creation of Sky Computing will be a lengthy process that will start slowly and gradually gather momentum. Our goal in this paper is to investigate how to start this transformation, not to define its ultimate form. As such, we start with with an intercloud broker for batch jobs—a small but important set of workloads. Third, given our focus on the early stages of the Sky, we do not provide solutions to several problems that must eventually be addressed, such as how to troubleshoot failures that occur with applications running across multiple clouds.

4 Intercloud Broker

We now present an intercloud broker that targets *batch applications*. We first review the requirements of such a broker, and then propose an architecture. Finally, we describe our implementation of the resulting design, called *SkyPilot*.

4.1 Requirements

Cataloging cloud services and instances. There is a huge and growing number of services, instances, and locations[4] across clouds. As shown in Table 1, the top three public clouds alone provide hundreds of compute VM types in dozens of regions across the globe. Even for a simple request of a 4-vCPU VM in the "compute-optimized" family—advertised by all three clouds—there are at least 90 choices within the US in terms of region and VM type. Furthermore, each cloud has hundreds of software services (e.g., hosted Kubernetes/Spark, blob storage, SQL databases) to choose from. This is clearly beyond what can be navigated manually by ordinary users.

[4]We use "locations" to refer to regions and zones, collectively.

Cloud	Regions	Zones	VM types
AWS	20 (US: 4*)	64 (US: 15*)	$\geq$ 558
Azure	51 (US: 8*)	124 (US: 23*)	$\geq$ 714
GCP	35 (US: 9)	106 (US: 28)	$\geq$ 155

Table 1: **Top public clouds with their myriad choices of locations and compute instance types**. Data is gathered from each cloud at the time of writing. *Not counting government cloud regions.

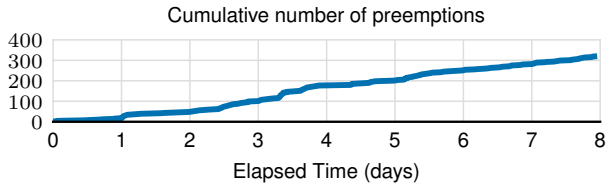


Figure 2: **Dynamic resource unavailability:** preemptions over time from a real-world bioinformatics workload trace. The workload ran for 8 days, using 24 large-CPU spot VMs on GCP, us-west1.

To provide the automatic placement of jobs, the broker must catalog the variety of instances and services, the APIs to invoke these services, and the subset of clouds and regions where these offerings are available.

Even after they have been cataloged, these many options are hard to navigate. Thus, the broker should expose filters on common attributes to applications so that they can easily narrow down the many options across clouds. For compute instances, filters may include the number of vCPUs, RAM, and accelerator types. For managed services (e.g., hosted analytics), filters may include the service or the package version (e.g., AWS EMR 6.5, or Apache Spark 3.1.2). Moreover, the broker should allow an application to choose specific services or instances supported only by one cloud.

Tracking pricing and dynamic availability. The price and availability of resources can vary dramatically across clouds and even regions or zones in the same cloud, often, but not always, following a diurnal pattern [73]. The variations are especially acute for *scarce resources* (§5.4), such as GPUs or preemptible spot instances that many applications use due to their lower costs, and change over time.

To illustrate the potential changes in resource availability, consider a real user's application: a bioinformatics task running for 8 days on 24 spot VMs on GCP (see §5.2 for more detail). When a VM is preempted, it waits for another spot VM to become available. Figure 2 shows the cumulative number of preemptions over time. Note that preemptions happened every day and at *unpredictably* different rates (e.g.,

compare day 3–4 vs. day 4–5). The application experienced 319 preemptions, a preemption every 36 minutes on average.

Thus, the broker should track the availability and pricing to provide applications with the best choices at run time. One challenge is that clouds do not publish availability information explicitly. The broker may have to learn about availability implicitly by observing preemptions or allocation failures of both on-demand and spot resources in different locations.

Dynamic optimization. Recall that the goal of the broker is to meet the application's cost and performance requirements under various constraints, such as data residency. This means the broker should choose the types of instances or services, clouds, and locations to run the tasks in the application DAG. This is a challenging optimization problem because of (1) the sheer number of choices (Table 1), (2) DAG topologies becoming complex (Figure 10), and (3) the unpredictable resource availability and price changes during the application's provisioning or run time (Figure 2).

As a result, the broker should implement a dynamic optimizer that can reflect the current resource availability and prices, and quickly find an optimal execution plan out of the large search space. To use up-to-date prices, the broker needs to compute the execution plan whenever an application starts. In addition, when a task in an application DAG cannot run as the broker originally planned due to availability changes, the broker needs to generate a new execution plan by *re-optimization* during the application's run time.

Managing resources and applications. Once the optimizer decides the placement of an application, the broker must provision the resources and free them when the application terminates. This involves starting and *reliably* shutting down instances on various clouds, or creating and terminating services (e.g., sending requests to a hosted service like AWS EMR). While these lifecycle operations may seem straightforward, bugs or failures can easily lead to inconsistencies between the broker state and the cloud provider state (e.g., leaking instances or intermediate data), which can be costly.

In addition, the broker must manage the execution of the application, i.e., start an application's task when its inputs are available, possibly restart it in case of failures or preemptions, and move the task's inputs across clouds/regions, if remote.

4.2 Architecture

Given these requirements, we propose an intercloud broker architecture consisting of the following components (Figure 3).

Catalog. The catalog records the instances and services available in each cloud, detailed locations that offer them, and the APIs to allocate, shut down, and access them. It also stores the long-term prices for on-demand VMs, data storage, egress, and services (typically these prices do not change for months). The catalog can provide filtering and searching functionalities. The catalog can be based on information published by the clouds, listed by a third party, or collected by the broker.

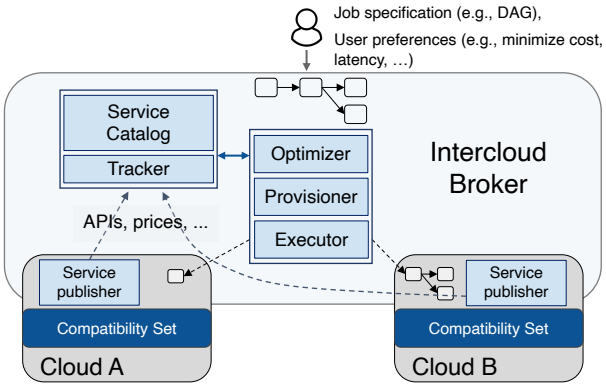


Figure 3: **Architecture of the intercloud broker.**

Tracker. This component tracks spot prices (which can change more frequently, e.g., hourly or daily) as well as resource availability across clouds and their locations.

Optimizer. The optimizer takes as inputs (1) the application's DAG and its requirements, and (2) the instance and service availability as well as their prices provided by the catalog and tracker, and then computes an optimal placement of the tasks. Upon resource availability and price changes, the optimizer may perform re-optimization.

Provisioner. This component manages resources (§4.1) by allocating the resources required to run the execution plan provided by the optimizer, and freeing them when each task exits. To handle unpredictable capacity and user quota errors, the provisioner implements automatic failover, where it asks the optimizer for a new placement plan if the provision fails. Failures are also reported to the tracker.

Executor. The executor manages the application (§4.1) by packaging each application's tasks and running them on the resources allocated by the provisioner.

In the future, we imagine intercloud brokers will offer more sophisticated services such as troubleshooting across clouds, providing more detailed performance measurements for specific applications on each cloud, the equivalent of spot-pricing but across clouds, reselling services at lower than listed prices (similar to the travel industry), and advanced configuration features for security and/or networking.

Furthermore, we expect a commercial broker to provide billing support to enable a user to have a single account with the provider of the intercloud broker, which then pays for the services rendered by each cloud on behalf of the user, and charges the user back. In our current deployment, our users have direct accounts with the three major clouds, so this functionality is not needed.

4.3 SkyPilot: An Implementation

We have implemented *SkyPilot*, which follows the architecture described in §4.2 with one difference: instead of implementing the tracker as a centralized component, SkyPilot distributes it between the catalog that refreshes prices daily, and the provisioner that tracks and caches provisioning failures.

SkyPilot is written in ≈21,000 lines of Python code, and has involved several person-years so far. It currently supports AWS, Azure, and GCP. It is being used by users from 3 universities and 4 other organizations; we report our deployment experience in §6. Next, we first describe SkyPilot in detail, then discuss the services in the compatibility set it uses.

Application API. As mentioned earlier, an application is specified as a DAG of coarse-grained tasks. Example tasks include a Spark job to process data, a Horovod [79] job to train a model, or an MPI job for HPC computations. A task starts when all of the tasks that provide its inputs have finished. Each task is self-contained and includes its executable and all library dependencies (e.g., packaged as a Docker image).

A task specifies its *input and output locations* in the form of cloud object store URIs. Optionally, a task can provide the *size estimates* of its inputs and outputs to help the optimizer estimate the cost of data transfers across clouds.

Each task specifies the *resources* it requires. For flexibility, resources are encoded as labels, such as "cpu: 4" or "accelerator: nvidia-v100", an idea we borrow from cluster managers such as Borg [85], Mesos [63], and Condor [82]. The optimizer uses these resource labels to search the service catalog for a set of feasible candidates for each task. If desired, the user can short-circuit the optimizer's selection by explicitly specifying a cloud and an instance type.

The user optionally specifies the number of instances for each task by a "num_nodes: *n*" label, which defaults to 1. Since we target coarse-grained batch jobs, our users have not found this a burden. In the future, we plan to support autoscaling or intelligently picking the number of instances [54, 84].

Finally, the user supplies an optional *time estimator* for each task, which estimates how long it will run on each specified resource. These estimates are used by the optimizer for planning the DAG. The user could determine these estimates by benchmarking the task on different configurations. If a time estimator is unspecified for a task, currently the optimizer defaults to the heuristic of choosing the resource with the lowest hourly price.[5]

Example. Listing 1 shows an application consisting of two tasks. The `train` task trains a model. It reads the input data from S3 and writes the output (the trained model) to the object store of the cloud it is assigned to run on, which is determined by the optimizer. By using `Resources`, a dictionary of resource labels, the user specifies that this training task requires either an nvidia-v100 accelerator or a google-tpu-v3-8 accelerator with 4 host vCPUs. The user also provides a `train_time_estimator_fn` lambda that estimates the task's run time on these two accelerators. For example, one can compute a rough estimate by dividing the total number of floating operations required for training the model by the accelerator's performance in FLOPS (floating point operations per second),

```
# A simple application: train -> infer.
with Dag() as dag:
  train = Task('train', run='train.py',
               arg='--data=$INPUT[0] --model=$OUTPUT[0]')
    .set_input('s3://my-data', size=150 * GB)
    # '?': saves to the cloud this op ends up running on.
    .set_output('?://my-model', size=0.1 * GB)
    # Required resources. A set ({}) means pick any Resources.
    .set_resources({
      Resources(accelerator='nvidia-v100'),
      Resources(accelerator='google-tpu-v3-8', cpu=4)})
    # A partial function: Resources -> time.
    .set_time_estimator(train_time_estimator_fn)

  infer = Task('infer', run='infer.py',
               arg='--model=$INPUT[0]')
    .set_input(train.output(0))
    .set_resources({
      Resources(accelerator='nvidia-t4'),
      Resources(accelerator='aws-inferentia', ram=16 * GB)})
    .set_time_estimator(infer_time_estimator_fn)
  # Connect the tasks.
  train >> infer
```

Listing 1: **API to express a simple application.**

or use a more accurate benchmarking-based predictor.

The `infer` task performs model serving. It takes the trained model as input (`set_input(train.output(0))`). The Airflow-like statement, `train >> infer`, enforces this dependency. These two tasks are encapsulated in a `Dag` object. The DAG is passed to the optimizer to output an execution plan, which is then passed to the provisioner and the executor.

Figure 4a visualizes the DAG. (I/O data are task attributes and not nodes in the DAG; we show them for clarity.) While simple, this basic API already exposes many degrees of freedom. For example, while train's input is on S3, the optimizer may choose to assign the task to a different cloud. In doing so, the optimizer must take into account the possible transfer costs, while satisfying the task's requirements.

For convenience, SkyPilot also offers a YAML interface to specify an application in addition to the programmatic API.

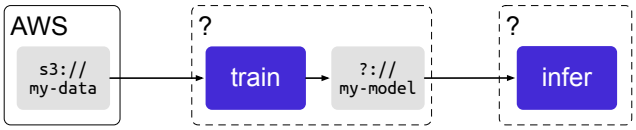
Catalog. SkyPilot implements a simple catalog to support three services (IaaS, object stores, managed analytics) on AWS, Azure, and GCP. These offerings are sufficient for our target workloads. We use the clouds' public APIs to obtain details about these offerings. Pricing is refreshed periodically.

Optimizer. The optimizer assigns each task to a cloud, location, and hardware configuration to best satisfy the user's requirements, e.g., minimize the total cost or time. It achieves this by filtering the offerings in the service catalog and solving an integer linear program (ILP) to pick an optimal assignment.

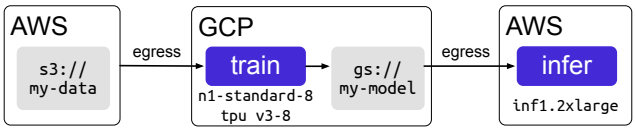
Before the actual optimization takes place, the optimizer first translates the high-level resource requirements into a set of feasible configurations, i.e., tuples of ⟨cloud, zone, instance type⟩, that can be used to run each task.[6] We call such a configuration a *cluster*. For example,

[5] Prior work [83] have considered performance prediction for analytics [84] and machine learning [78] workloads, which can also be leveraged.

[6] This also applies to most hosted analytics offerings (e.g., EMR, Dataproc) as they allow users to specify the cluster size and instance types.



(a) The declared application DAG.



(b) DAG, after optimization.

Figure 4: **An application, before and after optimization.**

`Resources(accelerator='nvidia-v100')` can be mapped to a cluster of AWS instances ⟨AWS, us-west-2a, p3.2x⟩ or Azure instances ⟨Azure, westus2-1, NC6s_v3⟩. To perform this translation, the optimizer filters the offerings in the service catalog to check if they satisfy the `Resources` required by each task. Each task is then annotated with the list of feasible clusters.

The optimizer computes execution plans at a zone level rather than a region level. This is because even in the same region, different zones can have different instance types and prices, and the data transfer between zones is not free.

ILP-based optimization. Consider a DAG with N tasks, each with C feasible clusters. Because C is typically in the 10s and can be up to 100s,[7] naively enumerating all C^N possible assignments is infeasible even for modest values of N. To solve this, we formulate the assignment problem as a 0-1 ILP.

SkyPilot supports two types of optimization objectives: either total running cost or end-to-end run time. Our ILP formulation is inspired by Alpa [94], but we additionally consider the parallelism between tasks that do not have dependency on each other. This is critical for minimizing the DAG run time.

Given a DAG (V, E) where V is the set of the tasks and E is the set of the edges representing the data dependencies between the tasks, our goal is to find an optimal mapping from each task in V to one of its annotated feasible clusters. For each task $v \in V$, we denote the set of the feasible clusters by C_v. Then we use a task time estimator to obtain a time vector $t_v \in \mathbb{R}^{|C_v|}$, where each element is the time estimate for running task v on a cluster in C_v. The time estimator can be either provided by the user or set to a default value of 1 hour. In addition, we get a cost vector $c_v \in \mathbb{R}^{|C_v|}$ by multiplying t_v by the hourly price of each cluster. To account for the data transfer overhead between two tasks $(u, v) \in E$, we define a matrix $P_{uv} \in \mathbb{R}^{|C_u| \times |C_v|}$ whose (i, j) element is the data transfer time when the parent task u is mapped to the i-th cluster of C_u and the child task v is mapped to the j-th cluster of C_v. Similarly, we define $Q_{uv} \in \mathbb{R}^{|C_u| \times |C_v|}$ for the data transfer cost between u and v.

When minimizing the total cost, we have:

$$\min_{s} \underbrace{\sum_{v \in V} s_v^T c_v}_{\text{computation cost}} + \underbrace{\sum_{(u,v) \in E} s_u^T Q_{uv} s_v}_{\text{data transfer cost}} \quad (1)$$

where $s_v \in \{0,1\}^{|C_v|}$ is a one-hot vector that selects a cluster from C_v. The objective explicitly considers the two types of cost: the first term represents the total cost spent in executing all tasks on the selected clusters, while the second term represents the total data transfer cost. After we linearize [61] the second term, we get a 0-1 ILP, which SkyPilot solves using an off-the-shelf solver, CBC [60].

Similarly, when minimizing the end-to-end time, we have:

$$\min_{s} f_{\text{sink}} \quad (2)$$

$$\text{s.t.} \quad f_v \geq \underbrace{f_u}_{\substack{\text{parent} \\ \text{finish time}}} + \underbrace{s_u^T P_{uv} s_v}_{\substack{\text{data transfer} \\ \text{time}}} + \underbrace{s_v^T t_v}_{\substack{\text{computation} \\ \text{time}}} \quad \forall (u,v) \in E \quad (3)$$

where $s_v \in \{0,1\}^{|C_v|}$ is the one-hot decision vector and $f_v \in \mathbb{R}$ is the *finish time* of the task v. The optimization constraint ensures that a task finishes no earlier than its parents, the input data arrive, and the task produces its outputs. Under these constraints, the running time of the DAG becomes the finish time of its sink.[8] Again, as we can linearize the second term, this problem can be efficiently solved by 0-1 ILP solvers.

While we cover the two representative objectives above, our ILP formulation allows any combination of cost and time to be used for the optimization. For example, we can minimize the cost under a time budget (or vice versa), by augmenting Equation 1 with the constraint in Equation 3 and bounding f_{sink} by the time budget. Future work can incorporate carbon footprint of cloud regions [21] into placement decisions.

Provisioner. SkyPilot implements a *provisioner* that reads the optimized plan and allocates a cluster for the next task ready to execute. As discussed, allocations may fail due to either *insufficient capacity* in a cloud's location or *insufficient quota* of the user's account. On such failures, the provisioner kicks off *failover* as follows. First, the failed location is temporarily blocked for the current allocation request with a time-to-live. Then, the optimizer is asked to *re-optimize* the DAG with this new constraint added. The provisioner then retries in the newly optimized location (another location of the same cloud or a different cloud). If all available locations fail to provide the resource, either an error is returned to the user or the provisioner can be configured to wait and retry in a loop.

We found failover to be especially valuable for scarce resources (e.g., large CPU or GPU VMs). For example, depending on request timing, it took 3–5 and 2–7 location attempts to allocate 8 V100 and 8 T4 GPUs on AWS, respectively.

[7] For instance, the previous example that requires one V100 GPU maps to 79 feasible clusters globally across AWS, Azure, and GCP.

[8] If the DAG has multiple sinks, we create a dummy sink that has a fake dependency on the real sinks.

Executor. After a cluster is provisioned, the *executor* orchestrates a task's execution, e.g., setting up the task's dependencies on the cluster, performing cross-cloud data transfers for the task's inputs, and running the task (which can be a distributed program utilizing a multi-node cluster). We built an executor on top of Ray [71], a distributed framework that we use for intra-cluster task execution with fault tolerance support. Using Ray, rather than building a new execution engine, allowed us to focus on building the higher-level components new to the broker. For example, our executor implements a storage module that abstracts the object stores of AWS, Azure, and GCP and performs transfers. The executor also implements status tracking of task executions for resource management. On execution failures, the executor optionally exposes cluster handles to allow login and debugging.

The executor interface is modular. We envision other executors will be added in the future, e.g., for Kubernetes [36]. In addition, while our system formulation is generic enough to support arbitrary DAGs, our implementation of the executor has focused on supporting pipelines (sequential DAGs).

Compatibility set. One of the distinguishing features of Sky is leveraging the already existing services and APIs across clouds (i.e., compatibility set; §2.3), rather than building uniform services and APIs across all clouds. However, a broker still needs to develop some glue-code to handle similar but not identical services supported by different clouds. The natural question is what is the effort to implement such glue-code? The answer for our applications so far is "minimal".

To manage clusters, SkyPilot uses Ray's cluster launcher, which already supports AWS, GCP, and Azure. (Other frameworks could also be used, e.g., Terraform [51].) The main functionality we added is the control for automatic failover.

One of the most important components of any Sky application is storage. While the APIs provided by the object stores of the three major clouds are similar, they are not identical. Fortunately, all have libraries [20, 30, 46] exposing the POSIX interface, which allows us to mount different object stores as directories. Providing this functionality required only 400–500 lines of code (LoC) per object store.

Finally, for analytics applications we use high-level APIs, e.g., hosted analytics services provided by AWS (EMR) and GCP (Dataproc). Abstracting these services required us to implement just two methods: provisioning and termination. This involved only 200 LoC for EMR and Dataproc together.

5 Experiments

We conduct a series of experiments to evaluate the benefits of our intercloud broker. Overall, we found that:

- SkyPilot enables batch applications to take advantage of unique hardware, unique managed services, and improved availability across locations and clouds.
- On three applications (ML pipelines, scientific jobs, and data analytics), SkyPilot saves up to 2.7× in time, 80% in cost, and 2× in makespan, compared to using a single cloud or location.
- Even for single-cloud applications, the broker improves availability by migrating jobs across regions, a policy not supported by cloud providers' own solutions (§5.2).

Workload	Uses	Benefits from
ML	IaaS	unique hardware
Bioinformatics	IaaS (spot VMs)	improved availability
Analytics	managed analytics	unique software service & unique hardware

Table 2: **Evaluated workloads, cloud services used, and benefits.**

Table 2 shows all workload types and their respective benefits.

5.1 Machine Learning Pipelines

We start with running two ML pipelines on SkyPilot to leverage the strengths of different clouds. In both pipelines, the goal is to minimize the total cost. We consider two scenarios:

- Single-cloud: all tasks are constrained to a single cloud;
- Broker: each task runs according to the plan generated by SkyPilot's optimizer, possibly on different clouds.

Overall, both pipelines benefit from SkyPilot's flexibility to run compute-intensive tasks on clouds with unique hardware accelerators (e.g., Inferentia, TPUs) that can provide speedups which offset the cost and latency of moving the data.

Due to space limit, we show in appendix (§A.2) an additional experiment on SkyPilot leveraging spot instances across clouds to run ML training with improved availability and cost.

5.1.1 Vision Pipeline

The vision pipeline consists of two tasks: train and infer (see Listing 1). The train task trains a ResNet-50 model on the ImageNet dataset (150 GB, stored on AWS S3). The infer task runs offline inference on 10^8 images (e.g., nightly photo categorization for services like Instagram or Google Photos).

Since training deep learning models often requires iterative and heavy computations, we demonstrate a large reduction in cost and run time by moving the training data from AWS to GCP to leverage its TPU accelerators for training [23].

Setup. We specify resource candidates for each task as:

- train: `'nvidia-v100'`, `'google-tpu-v3-8'`
- infer: `'google-tpu-v3-8'`, `'nvidia-t4'`, `'aws-inferentia'`

For train, we use a V100 (common high-end GPU for training) or a TPU. For infer, we use a TPU, a T4 GPU (marketed as the most cost-effective GPU for model inference), or an Inferentia accelerator designed by AWS for cost-effective inference [15].

The best single-cloud plans are shown in Figure 5, termed {AWS, GCP, Azure}-only. The Broker plan is SkyPilot's optimizer output that minimizes the total cost. In this experiment, we used a simple time estimator that divides the total FLOPs

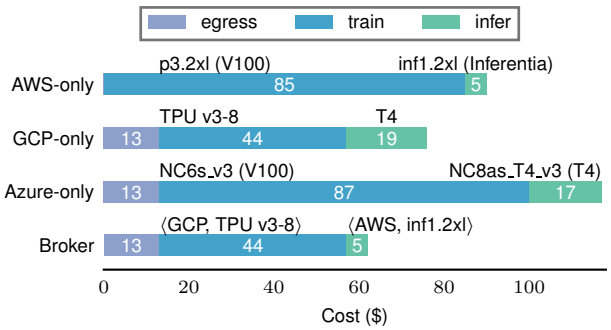


Figure 5: **Vision pipeline:** hardware and costs of each deployment. For simplicity, the zones chosen for the plans are omitted. For training we use mixed-precision and the XLA compiler [50] with TensorFlow Keras 2.5.0. For inference we use half-precision. On GCP, accelerators are attached to an n1-standard-8 VM.

required to train the model by the hardware FLOPS:[9]

```
def train_time_estimator_fn(resource):
  train_tflops = ...  # Obtained from model analysis.
  if resource.accelerator == 'nvidia-v100':
    hardware_tflops = 120
  if resource.accelerator == 'google-tpu-v3-8':
    hardware_tflops = 420
  return train_tflops / hardware_tflops
```

We used a similar FLOPs-based time estimator for infer.

Results. We show the plan generated by SkyPilot's optimizer in Figure 4b and the results in Figure 5.

While this pipeline is simple, *its search space is already large*, with a total of 2,170 possible assignments (details in §5.4), as we have multiple choices in hardware, cloud, and location. The optimizer successfully finds an optimal solution. Compared with the three single-cloud plans, the Broker plan lowers the total cost by 18%–47%, by taking advantage of the unique hardware capabilities across two clouds.

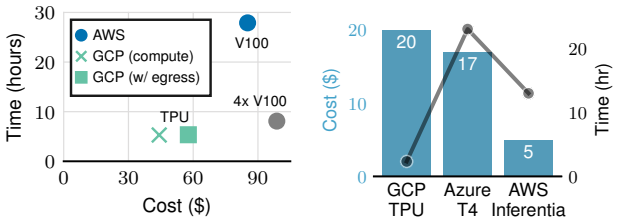
For train, the optimizer decides that, despite the input being stored on AWS, it is better to incur an egress cost and ship it to GCP to use the TPU. This choice leads to a cost of $57 ($44 compute, $13 egress) which is less than training on AWS, at $85.[10] SkyPilot's storage module uses GCP's storage transfer service [31] to copy the data in about 3 minutes.

For infer, the optimizer estimates that AWS's Inferentia is more cost-effective than the T4 GPU, after factoring in a small data egress cost (shipping the first task's output, a 0.1 GB model, from GCP to AWS with a cost of $0.01).

To understand the cost savings, we compare the detailed time and cost per task. For training (Figure 6a), SkyPilot's choice of GCP TPU takes 5.4 hours and costs $57 with egress included, which is 5.2× faster and 33% cheaper than the AWS V100 plan. (Azure V100 is similar but has $13 for egress; hence omitted.) To make the hardware more comparable, we

[9] While crude, this estimate is a reasonable approximation for throughput-bound models with intensive matrix operations, such as ResNet.

[10] If we set the input 4× as large, at 600 GB, the optimizer decides against transferring the data as the egress cost will dominate.



(a) Training (b) Serving

Figure 6: **Vision pipeline:** detailed breakdown per task.

submitted the task again requesting 4 V100s on AWS to match the FLOPS performance of a TPU v3-8: still, TPU is 1.5× faster and 42% cheaper than 4 V100s. For serving (Figure 6b), AWS's custom Inferentia chip saves both cost (71%) and time (1.8× faster) compared to the widely available T4 GPU.

Thus, clouds offer *unique hardware incentives* to different tasks, even if the data is stored on a different cloud.

Optimizing for time vs. cost. To test SkyPilot's ability to minimize the total time rather than cost (§4.3), we resubmit this pipeline to SkyPilot with the time-minimizing objective. The resource selection for train remains the same. For infer, SkyPilot now chooses GCP TPU (estimated to take 2.5 hours and cost $21, per 10^8 images) over AWS Inferentia (which was *cost-optimal*; estimated to take 8.2 hours and cost $3). The estimates reflect the actual ranking in Figure 6b. Even though the TPU costs 4× more in total than Inferentia, it reduces inference time by 5.7×. This example shows that optimal placements can change based on user preferences.

5.1.2 NLP Pipeline

We next run a natural language processing (NLP) pipeline that emulates an increasingly prevalent workload: fine-tuning "foundation models" [56]. It consists of three tasks (Figure 1):

- **Confidential data processing**: remove sensitive information from raw data using Intel SGX hardware enclaves. We use the Amazon Customer Reviews Dataset [2] and treat it as if it contained personally identifiable information (PII) and thus must be processed securely. To remove sensitive data, we run Opaque [95] on an SGX-enabled instance to filter on a column (i.e., the filtered-out information is assumed sensitive), and output only the review texts and star ratings. The size of the output dataset is 1 GB.
- **Train**: fine-tune BERT-base [59], a popular natural language understanding model, on the preprocessed and now non-sensitive data. This model predicts a rating given a review text. We fine-tune the model for 10 epochs.
- **Infer**: use the model to classify 1M new reviews.

Setup. The first task requires `Resources(intel_sgx=True)`, which is currently only offered by Azure [16]. For training, we consider either 4 V100s, or a TPU v3-8. For serving, we consider either a T4 GPU, or AWS's Inferentia.

Due to the confidential computing requirement, the only possible single-cloud plan is to run all three tasks on Azure:

		proc	train	infer	egress	**Total**
Time	**Azure**	0.6	13.3	1.5	–	15.4
(hr.)	**Broker**	0.6	3.8 -71%	1.4 -7%	0.03	5.8 -62%
Cost	**Azure**	0.8	163	1.2	–	165
($)	**Broker**	0.8	32 -80%	0.5 -58%	0.1	33.4 -80%

Table 3: **NLP pipeline:** run time and cost of each deployment plan.

a DC8 VM for SGX, an NC24s VM with 4 V100 GPUs for training, and an NC8as instance with a T4 GPU for serving.

Results. Table 3 shows the time and cost comparison between the single-cloud and Broker plans. Different from before, the Broker plan for this pipeline uses all three clouds. The search space is larger, *with over 16K possibilities* (§5.4).

As expected, the single-cloud plan restricts its choices of hardware to Azure and thus results in suboptimal cost and performance. While Azure's Intel SGX offering is unique for secure processing, SkyPilot allows this pipeline to leverage different clouds for other tasks of the same application. SkyPilot's optimizer picks the TPU (GCP) over 4 V100s for training, and the Inferentia (AWS) over the T4 GPU for serving. This considerably reduces both the total run time (by 62%) and cost (by 80%) compared with the Azure-only plan.

5.2 Bioinformatics

The intercloud broker should *dynamically* respond to the changing availability of resources (§4.1). We evaluate SkyPilot's handling of availability changes by modeling a *real user's workload*: A bioinformatic task of mapping DNA cells of sequencing data [67, 92]. The jobs are independent, have variable-sized inputs and variable run times, with each using all CPUs within one machine. Jobs are not checkpointable and failures require recomputation from scratch. Finally, these jobs are *recurring*: there are 10s to 100s of jobs to run every week based on incoming data. Due to long run times, this user exclusively uses spot VMs on GCP to save costs, and has been continuously using SkyPilot to do so for several months.

We submit 40 jobs to SkyPilot, each running on an n1-highmem-96 spot VM on GCP for 8–12 hours. We implement and compare two policies in SkyPilot: (1) *SingleRegion*, which retries each preempted job in other zones of the same region—this models providers' managed instances solutions [35]; (2) *Broker*, which retries each preempted job in the next cheapest region chosen by the optimizer. We start two sets of 40 jobs together (to minimize variance due to time) in the region with the cheapest price for this VM (us-west1). We ensure the jobs are within quotas so all job migrations are due to preemptions.

Overall, the Broker policy finishes significantly faster than the SingleRegion baseline, due to experiencing fewer preemptions. Figure 7 (top) shows that Broker completed 75% of the jobs 1.6× or 7 hours faster than SingleRegion. At around $T = 16$ hours, all Broker jobs finished, while 30% (12) of SingleRegion jobs were still running. The last SingleRegion job finished at $T = 32$ hours, yielding a 2× longer makespan.

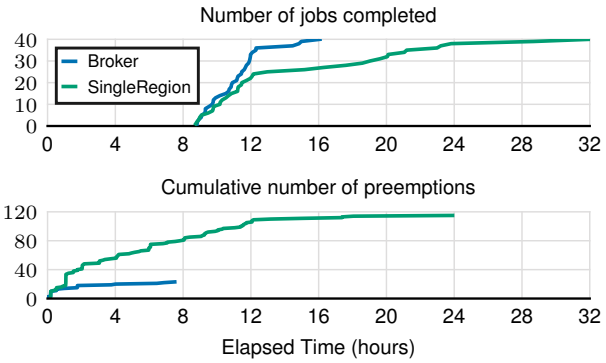


Figure 7: **Dynamically adjusting to availability** on a bioinformatics workload of 40 jobs on spot CPU VMs. *Broker* moves preempted jobs to a new region, while *SingleRegion* moves preempted jobs to other zones in the same region. Note the shared x-axis. Cloud: GCP.

Figure 7 (bottom) shows the speedup comes from Broker incurring 5× fewer preemptions. Since both policies started in the same region, the preemption curves initially overlapped. Broker swiftly moved the 22 preempted jobs to another region, which remained non-preemptive for the entire duration (e.g., last preemption occurred before $T = 8$ hours). The original region continued to experience a high preemption rate in all zones, causing SingleRegion to have far more stragglers.

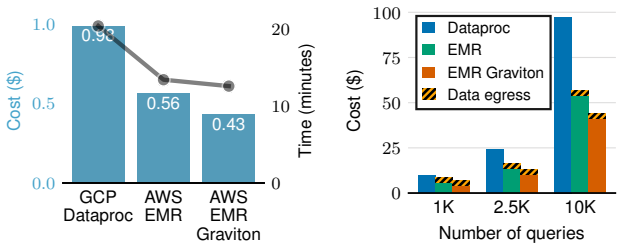
While this example represents a good case (moving from a region with a high preemption rate to a region with a low preemption rate), it shows that SkyPilot can dynamically use multiple regions to improve availability *when needed*. Managed solutions from cloud providers, e.g., spot fleets [49] or managed instances [35], are *confined within a region* and thus cannot support such a cross-region (or cross-cloud) policy.

Finally, note that this policy is not always better than SingleRegion. For example, if the jobs started in a region with a low preemption rate, some unlucky jobs could be preempted and moved to a region with a higher preemption rate, which could be worse than SingleRegion. Importantly, SkyPilot allows new policies (cross-cloud/region) to be implemented easily, and we expect this to be an area of future research.

5.3 Managed Data Analytics

So far, we demonstrated SkyPilot's ability to use IaaS (VMs) on different clouds. We now use the broker to run an analytics workload on the *managed analytics services* of two clouds: AWS EMR [4] and GCP Dataproc [29]. While VMs with the same hardware on different clouds should have mostly the same performance, we expect hosted services to exhibit more performance variations due to differences in software. We run TPC-DS [72] on the following (scale factor 100, or 33 GB of data in Parquet, generated locally on each cloud):

- GCP Dataproc: which runs vanilla Spark 3.1.2, on a 3-node n2-standard-16 cluster. Version 2.0.29-debian10.
- AWS EMR: which runs an *optimized runtime* [42] for



(a) Data is local (99 queries) (b) Data shipped from GCP

Figure 8: **Using managed analytics services with SkyPilot.** TPC-DS. **(a)** Cost (left y) and time (right y) of two hosted services in three configurations, where data is generated locally. Benefits of software and hardware offerings can combine. Mean of 3 runs. **(b)** Assuming data is stored in GCP, running more queries offsets the egress cost.

Spark 3.1.2, on a 3-node m5.4xlarge cluster. Version 6.5.0.

- AWS EMR Graviton: like above, but on a 3-node m6g.4xlarge cluster, which uses the Graviton2 ARM-based processors custom-designed by AWS [14]. Due to its cost-performance benefits, several large companies such as Netflix and Snap have moved some of their workloads to Graviton2 from traditional x86 instances [13].

Figure 8a shows AWS EMR finishes 34% faster and 43% cheaper than GCP Dataproc. We ensured that GCP's n2 cluster has the same or better hardware than AWS's m5.4x cluster. Thus, the speedup is due to EMR's optimized software runtime [42] for Spark, representing a *unique software incentive* for users with similar analytics workloads.

In addition, AWS EMR Graviton *improves* both the cost and run time over AWS EMR by 23% and 6%, respectively. Thus, this is a case of combining the unique *software* and *hardware* advantages to attract such workloads even more.

To understand the tradeoff between better services vs. data gravity, Figure 8b shows the cost of running more queries from the benchmark, assuming the data is not generated locally but initially resides in GCP and has to be copied. (Here, we simply execute the TPC-DS benchmark's 99 queries multiple times to increase the number of queries we ran.) With 1K queries, EMR's speed advantage already offsets the data transfer cost ($2.8). Running 2.5K queries yields a cost saving of 32% for EMR and 46% for EMR Graviton, while running 10K queries yields 42% and 55% savings, respectively.

To request a managed service for a task, we specify

```
task.set_managed_service(
  AnalyticsService(
    dependencies={'Spark': '3.1.2', 'Hadoop': '3.2.1', ...},
    resources=Resources(cpu=16, ram=64 * GB, num_nodes=3)))
```

where `AnalyticsService` is backed by concrete implementations such as EMR or Dataproc. The `dependencies` field specifies the desired package versions for the hosted service; such version lists are published by the cloud providers [11,26] and recorded in SkyPilot's service catalog.

Type	Hardware	Zones	On-demand $ Max/Min	On-demand $ CV	Spot $ Max/Min	Spot $ CV
CPU	AMD (8 cores)	146	2.5×	16%	7.3×	59%
	Arm (8 cores)	88	2.1×	12%	2.5×	17%
	Intel (8 cores)	248	1.6×	12%	9.4×	39%
GPU	K80 (1 chip)	56	9.5×	48%	5.9×	60%
	T4 (1 chip)	146	1.7×	12%	10.8×	29%
	V100 (1 chip)	79	1.6×	14%	1.9×	19%
	A100 (8 chips)	46	1.9×	23%	6.4×	84%
TPU	v2 (8 cores)	5	1.2×	6%	1.2×	6%
	v3 (8 cores)	4	1.1×	4%	1.1×	4%

Table 4: **Capturing the large heterogeneity of locations and pricing in the catalog.** We show for a subset of offerings, the number of zones that provide them (out of 294 zones globally across the top 3 clouds), the pricing ratios of the most costly to the cheapest zone, and the coefficients of variation (CV) of prices across zones. CPUs are the latest generation in the "general-purpose" family.

5.4 Analyzing the Broker

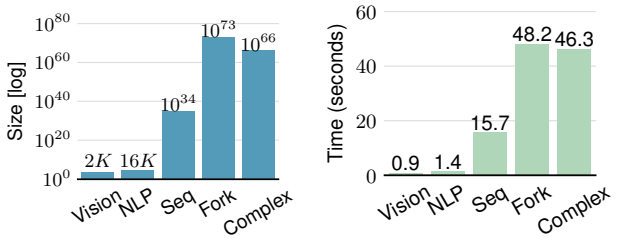
Location and pricing heterogeneity in the catalog. We analyze SkyPilot's service catalog (over 76K entries) to see how well it captures the heterogeneity in locations and prices for all three clouds. Table 4 shows the results. We see that not all offerings (VMs, accelerators) are present in all zones, and there can be large price differences across zones.

Among the 294 zones across the three clouds, the latest Intel CPUs are widely offered, but AMD is only offered in 50% of the zones, while ARM is in only 30%. CPU workloads, e.g., bioinformatics (§5.2) and analytics (§5.3), can suffer from up to 2.5× price premiums if run in the most expensive zone, which increase to 9.4× if spot instances are used. These differences are even larger for NVIDIA GPUs, which are present in just 16–50% of all zones, and their prices vary by up to 9.5× for on-demand and 10.8× for spot. Finally, despite TPUs being offered only in 4–5 (or 5%) GCP zones, there is still a 10%–20% price difference across those zones.

This significant heterogeneity in *locations* and *pricing* makes it hard for users to manually find the best placement. By capturing this heterogeneity, SkyPilot's catalog enables the optimizer to automatically exploit these differences.

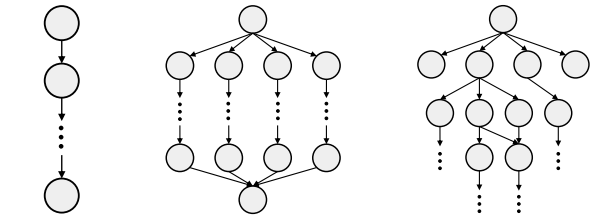
Optimizer overhead. We evaluate SkyPilot's optimizer overhead on a variety of DAGs. Figure 9 shows the search space sizes and the optimization time for the two ML pipelines in §5.1 and 3 other DAGs (see below). Despite the pipelines' simple structures (Vision, NLP), *their search spaces already have 2K–16K possible assignments*, making them non-trivial or infeasible to optimize by hand. Using the ILP, however, our optimizer can find an optimal solution in under 1.4 seconds.

Additionally, we test on three larger and more complex DAGs, found in Airflow's repository [6]: the first two (Figure 10a, Figure 10b) are commonly used in the real world [68], while the third (Figure 10c) has a more complex structure.



(a) Search space sizes (b) Optimization time

Figure 9: **Search spaces and optimization times.** Timing is measured on an M1 MacBook Pro; mean of 3 runs. Objective is cost. Locations of feasible clusters are limited to all US zones on 3 clouds.



(a) Sequential (b) Fork-Join (c) Complex

Figure 10: **Larger DAGs found in Airflow's repository. (a)** Sequential: $|V| = 20, |E| = 19$. **(b)** Fork-Join: $|V| = 42, |E| = 44$. **(c)** Complex: $|V| = 38, |E| = 53$.

We assume each task requires an 8-vCPU Intel VM in US zones, which leads to 55 feasible clusters for each task. We assign random time estimates (sampled from $U(0,1)$ hours) to each task and a random data transfer size (sampled from $U(0,100)$ GB) to each edge. While the search spaces for the DAGs are combinatorially large (10^{34}–10^{73} possible assignments), optimization takes at most 48.2 seconds. Since each task in a DAG is coarse-grained (e.g., can take hours), this optimization time is a negligible portion of the DAG run time.

If resource availability changes during run time, the DAG may need to be re-optimized to generate a revised execution plan. As the process of re-optimization involves updating the list of feasible clusters and restarting the ILP optimization, its overhead is comparable to that of the initial optimization.

6 Deployment Experience

We have deployed SkyPilot to dozens of users from 3 universities and 4 other organizations, who have been using the broker to run both adhoc and recurring batch jobs in the clouds for many months. These users have switched to the intercloud broker from their prior solutions of manually interacting with specific clouds, either via web consoles or low-level APIs. Below, we discuss our experiences with the system so far based on user feedback.

Benefits of an intercloud broker. By surveying our users, we found that users value the broker not only for cost reduction, but also for improved availability (see §5.2) and in general for *improving their productivity*. For example, users like the broker's ability to automatically provision scarce resources across clouds or regions, the easy access to best-of-breed hardware (e.g., TPUs), and the simple packaging of existing programs. Moreover, by interacting with the broker rather than the clouds, they value the ability to run the same jobs on different clouds with no change to their code or workflow.

Cluster reuse for faster development and debugging. Users have reported that the typical provisioning time of several minutes for a new cluster is too long, especially during the iterative code development phase. To alleviate this, we added the ability to reuse existing clusters for running a new application. This also helps the debugging of Sky applications as the users can log into a cluster to inspect and troubleshoot.

Moving data is acceptable for many workloads. Data gravity can prevent workloads from being moved across clouds. However, we found that for many batch workloads, cross-cloud data transfers are not as slow or costly as we expected. In fact, moving data can be profitable even after factoring in the egress (Figure 5; Figure 8).

There are several reasons for this. First, the computation complexity of many batch jobs, such as ML training, is typically *super-linear* in the input size. Second, many datasets are not excessively large. For example, a study from Microsoft reports that most production ML datasets are between 1 GB to 1 TB [75]. Our results (§5.1.1) suggest that a 1 TB dataset can likely be moved in ∼20 minutes with a cost of ∼$90. Depending on the job, this delay and cost can be easily offset by the destination offering better hardware, software, or pricing.

On-premise clusters as part of the Sky. Users have requested the support for running jobs on on-premise clusters through the broker. There are several benefits. First, this would enable users to take advantage of idle local clusters and burst to the cloud when they are overloaded. Second, the broker would offer the same interface that hides the heterogeneity (to the extent possible), so the same Sky applications could run both in the cloud and locally. Challenges include designing spillover policies and handling compatibility and storage.

7 Related Work

Sky Computing. We are not the first to use the name "Sky Computing" as several papers, dating back to 2009, also used this term [62, 69, 70]. However, these papers focus on particular technical solutions, such as running middleware (e.g., Nimbus) on a cross-cloud Infrastructure-as-a-Service platform, and target specific workloads such as high-performance computing (HPC). This paper takes a broader view of Sky Computing, seeing it as a change in the overall ecosystem and considering how technical trends and the market forces can play a critical role in the emergence of Sky Computing.

The work most closely related to this paper is [81], but here we significantly extend that work by refining the vision, designing and building a broker, demonstrating its benefits in several applications, and reporting on early adoption.

Cross-cloud compute, storage, and egress. Supercloud [65] is a virtual cloud that can span multiple zones and clouds, using nested virtualization and live VM migration to move stateful workloads across locations. Our proposal shares the goal of easing workload migration, but supports migrating higher-level jobs (not VMs), considers a broader set of cloud services in addition to IaaS, and focuses on batch jobs by optimizing for price, performance, and availability.

There have been several proposals for cross-cloud storage solutions. CosTLO [91] and SPANStore [90] use request redundancy and replication to minimize storage access latencies. Perhaps the most comprehensive is Gaia-X, a European effort to create a federated open data infrastructure that enables data sharing with strong governance properties and respecting data and cloud sovereignty [28]. These efforts are largely orthogonal to our focus on computational tasks.

Several industry efforts have been started to reduce cross-cloud data egress fees. The Bandwidth Alliance [19] is one such effort, consisting of several cloud providers who agree to reduce or even eliminate egress fees from their clouds to Cloudfare or other members. Closely related is Cloudfare R2 [24], an object store that promises to charge zero egress fees. Naturally, Sky Computing benefits from these efforts to combat data gravity, and the intercloud broker can be extended to support zero-egress storage systems.

Middleware. Middleware solutions (e.g., CORBA [25], Microsoft BizTalk [37], IBM WebSphere [34], etc.) bear some resemblance to our work. While these solutions allow systems from different vendors to communicate and interoperate, our proposal allows an application to utilize cloud services offered by different cloud providers.

There are several differences between these efforts and the intercloud broker. First, we consider satisfying requirements such as minimizing costs which have not been a concern of these systems. Second, the intercloud broker focuses on placing the components of the same application rather than on how systems from different vendors interoperate. Finally, we are operating in a cloud setting rather than a traditional distributed system setting.

Differences aside, middleware solutions that allow cloud services to interoperate (e.g., connect an AWS S3 bucket with GCP Dataproc) could be considered as being part of the compatibility set, which the intercloud broker can leverage.

Integration Platform-as-a-Service (iPaaS). Like the middleware systems discussed above, iPaaS solutions [40, 47] also integrate distinct systems but are often run as managed services on the cloud. iPaaS solutions provide adaptors to connect APIs from different services and systems (e.g., APIs for Snowflake, Jira, or Stripe). Developers can build workflows on top (e.g., on receiving a new case in Salesforce, call Jira's API to open a ticket) and deploy them through the iPaaS.

While iPaaS can run integration workflows on the cloud, our proposal places and runs compute-intensive jobs on the most suitable cloud based on price, performance, and availability. Similar to middleware, iPaaS is complementary as we can leverage these adaptors to expound the compatibility set.

Optimization for geo-distributed analytics. A line of work has optimized the performance of geo-distributed analytics [64, 77, 86]. This setting is similar in spirit to ours: it considers running a MapReduce-style job (an analytics query) across many sites, while we consider running a DAG of coarse-grained computations potentially across several clouds.

There are three main differences. First, these techniques are system-specific optimizations, and we in general do not assume as much knowledge about the application. Second, these techniques mostly assume different sites to differ only in their WAN bandwidths and otherwise have identical hardware, while we exploit the inherent differences in hardware, software, pricing, and resource availability of several clouds or regions/zones within a cloud. Third, these solutions optimize for faster completion times, while we also consider minimizing costs and improving resource availability.

That said, we note that the intercloud broker could potentially leverage system-specific optimizations if it is told that the application is of a certain type (e.g., MapReduce).

8 Conclusion

This paper describes the design, implementation, applications, and early deployment of an intercloud broker, SkyPilot. SkyPilot enables users to seamlessly run their batch jobs across clouds to minimize cost and/or delay. We see this as the first step towards a paradigm we call Sky Computing, which we hope will transform the cloud computing ecosystem to better meet user needs.

Acknowledgements. We thank the NSDI reviewers and our shepherd, Paolo Costa, for their valuable feedback. This work is in part supported by NSF CISE Expeditions Award CCF-1730628 and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware.

References

[1] Akka. https://akka.io/.

[2] Amazon customer reviews dataset. https://s3.amazonaws.com/amazon-reviews-pds/readme.html.

[3] Amazon Elastic Kubernetes Service. https://aws.amazon.com/eks/.

[4] Amazon EMR. https://aws.amazon.com/emr/.

[5] Anthos. https://cloud.google.com/anthos.

[6] Apache Airflow. https://airflow.apache.org/.

[7] Apache Cassandra. https://cassandra.apache.org/.

[8] Apache jclouds. https://jclouds.apache.org/.

[9] Apache Kafka. https://kafka.apache.org/.

[10] Apache Libcloud. https://libcloud.apache.org/.

[11] Application versions in Amazon EMR 6.x releases. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-release-app-versions-6.x.html.

[12] Artificial Intelligence: From the Public Cloud to the Device Edge. https://www.equinix.com/resources/whitepapers/nvidia-distributed-ai-cloud-infrastructure-edge.

[13] AWS and Arm. https://www.arm.com/why-arm/partner-ecosystem/aws.

[14] AWS Graviton Processor. https://aws.amazon.com/ec2/graviton/.

[15] AWS Inferentia. https://aws.amazon.com/machine-learning/inferentia/.

[16] Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[17] Azure HDInsight. https://azure.microsoft.com/en-us/services/hdinsight/.

[18] Azure Kubernetes Service. https://azure.microsoft.com/en-us/services/kubernetes-service/.

[19] Bandwidth Alliance. https://www.cloudflare.com/bandwidth-alliance/.

[20] BlobFuse - A Microsoft supported Azure Storage FUSE driver. https://github.com/Azure/azure-storage-fuse.

[21] Carbon free energy for Google Cloud regions. https://cloud.google.com/sustainability/region-carbon.

[22] Cerebras. https://cerebras.net/.

[23] Cloud TPU. https://cloud.google.com/tpu.

[24] Cloudflare R2. https://www.cloudflare.com/products/r2/.

[25] Common Object Request Broker Architecture (CORBA). https://www.omg.org/spec/CORBA.

[26] Dataproc 2.0.x release versions. https://cloud.google.com/dataproc/docs/concepts/versioning/dataproc-release-2.0.

[27] Docker. https://github.com/docker.

[28] Gaia-X: A Federated Secure Data Infrastructure. https://www.gaia-x.eu/.

[29] Google Cloud Dataproc. https://cloud.google.com/dataproc/.

[30] Google Cloud Storage FUSE. https://cloud.google.com/storage/docs/gcs-fuse.

[31] Google Cloud, Storage Transfer Service. https://cloud.google.com/storage-transfer-service.

[32] Google Kubernetes Engine. https://cloud.google.com/kubernetes-engine.

[33] HashiCorp State of Cloud Strategy Survey. https://www.hashicorp.com/state-of-the-cloud.

[34] IBM WebSphere Application Server. https://www.ibm.com/products/websphere-application-server.

[35] Instance groups, Google Compute Engine. https://cloud.google.com/compute/docs/instance-groups.

[36] Kubernetes. https://github.com/kubernetes/kubernetes.

[37] Microsoft BizTalk Server documentation. https://learn.microsoft.com/en-us/biztalk/.

[38] MLFlow. https://mlflow.org/.

[39] MongoDB. https://github.com/mongodb/mongo.

[40] MuleSoft CloudHub. https://www.mulesoft.com/platform/saas/cloudhub-ipaas-cloud-based-integration.

[41] MySQL. https://www.mysql.com/.

[42] Optimize Spark performance, Amazon EMR. https://docs.aws.amazon.com/emr/latest/ReleaseGuide/emr-spark-performance.html.

[43] PostgreSQL. https://www.postgresql.org/.

[44] Presto. https://github.com/prestodb/presto.

[45] Redis. https://github.com/redis/redis.

[46] s3fs. https://github.com/s3fs-fuse/s3fs-fuse.

[47] SAP Integration Suite. https://www.sap.com/products/technology-platform/integration-suite.html.

[48] SparkSQL. https://spark.apache.org/sql/.

[49] Spot Fleet, AWS EC2. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html.

[50] TensorFlow XLA. https://www.tensorflow.org/xla.

[51] Terraform. https://www.terraform.io/.

[52] The Cloud Imperative For Software and Platforms, Accenture. https://www.accenture.com/_acnmedia/PDF-139/Accenture-The-Cloud-Imperative-Software-Platforms-Industry.pdf.

[53] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[54] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 469–482, 2017.

[55] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Mikroelektronik och informationsteknik, 2003.

[56] Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

[57] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in Apache Flink: Consistent stateful distributed stream processing. *Proc. VLDB Endow.*, 10(12):1718–1729, August 2017.

[58] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. In *NIPS Workshop on Machine Learning Systems (LearningSys'16)*, 2016.

[59] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[60] John Forrest and Robin Lougee-Heimer. Cbc user guide. In *Emerging theory, methods, and applications*, pages 257–277. INFORMS, 2005.

[61] Richard J Forrester and Noah Hunt-Isaak. Computational comparison of exact solution methods for 0-1 quadratic programs: Recommendations for practitioners. *Journal of Applied Mathematics*, 2020, 2020.

[62] José A.B. Fortes. Sky computing: When multiple clouds become one. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 4–4, 2010.

[63] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.

[64] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–16, 2018.

[65] Qin Jia, Zhiming Shen, Weijia Song, Robbert Van Renesse, and Hakim Weatherspoon. Supercloud: Opportunities and challenges. *ACM SIGOPS Operating Systems Review*, 49(1):137–141, 2015.

[66] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning (ICML)*, 2018.

[67] Hanqing Liu, Jingtian Zhou, Wei Tian, Chongyuan Luo, Anna Bartlett, Andrew Aldridge, Jacinta Lucero, Julia K Osteen, Joseph R Nery, Huaming Chen, Angeline Rivkin, Rosa G Castanon, Ben Clock, Yang Eric Li, Xiaomeng Hou, Olivier B Poirion, Sebastian Preissl, Antonio Pinto-Duarte, Carolyn O'Connor, Lara Boggeman, Conor Fitzpatrick, Michael Nunn, Eran A Mukamel, Zhuzhu Zhang, Edward M Callaway, Bing Ren, Jesse R Dixon, M Margarita Behrens, and Joseph R Ecker. DNA methylation atlas of the mouse brain at single-cell resolution. *Nature*, 598(7879):120–128, October 2021.

[68] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, 2022.

[69] A. Matsunaga, J. Fortes, K. Keahey, and M. Tsugawa. Sky computing. *IEEE Internet Computing*, 13(05):43–51, sep 2009.

[70] André Monteiro, Joaquim S. Pinto, Cláudio J. V. Teixeira, and Tiago Batista. Sky computing: Exploring the aggregated cloud resources - part i. In *Conference: Information Systems and Technologies (CISTI)*, 2021.

[71] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[72] Raghunath Othayoth Nambiar and Meikel Poess. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1049–1058. VLDB Endowment, 2006.

[73] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, , and Matei Zaharia. Analysis and exploitation of dynamic pricing in the public cloud for ml training. *VLDB DISPA Workshop 2020*.

[74] OpenAI. AI and Compute. https://openai.com/blog/ai-and-compute/, 2018.

[75] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. End-to-end optimization of machine learning prediction queries. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 587–601, New York, NY, USA, 2022. Association for Computing Machinery.

[76] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. 2017.

[77] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. Low latency geo-distributed data analytics. *ACM SIGCOMM Computer Communication Review*, 45(4):421–434, 2015.

[78] Hang Qi, Evan R Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. *International Conference on Learning Representations (ICLR)*, 2016.

[79] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[80] Statista. Infographic: Amazon leads $150-billion cloud market. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/.

[81] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery.

[82] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[83] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A Kozuch, and Gregory R Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. *Parallel Data Laboratory, Carnegie Mellon University, Tech. Rep.*, 2016.

[84] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[85] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[86] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. Clarinet: Wan-aware optimization for analytics queries. In *OSDI*, volume 16, pages 435–450, 2016.

[87] Sarah Wang and Martin Casado. The Cost of Cloud, a Trillion Dollar Paradox. https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization/.

[88] Joe Weinman. Intercloudonomics: Quantifying the value of the intercloud. *IEEE Cloud Computing*, 2(5):4047, September 2015.

[89] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012.

[90] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 292–308, New York, NY, USA, 2013. Association for Computing Machinery.

[91] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. CosTLO: Cost-Effective redundancy for lower latency variance on cloud storage services. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 543–557, Oakland, CA, May 2015. USENIX Association.

[92] Zizhen Yao, Hanqing Liu, Fangming Xie, Stephan Fischer, Ricky S Adkins, Andrew I Aldridge, Seth A Ament, Anna Bartlett, M Margarita Behrens, Koen Van den Berge, Darren Bertagnolli, Hector Roux de Bézieux, Tommaso Biancalani, A Sina Booeshaghi, Héctor Corrada Bravo, Tamara Casper, Carlo Colantuoni, Jonathan Crabtree, Heather Creasy, Kirsten Crichton, Megan Crow, Nick Dee, Elizabeth L Dougherty, Wayne I Doyle, Sandrine Dudoit, Rongxin Fang, Victor Felix, Olivia Fong, Michelle Giglio, Jeff Goldy, Mike Hawrylycz, Brian R Herb, Ronna Hertzano, Xiaomeng Hou, Qiwen Hu, Jayaram Kancherla, Matthew Kroll, Kanan Lathia, Yang Eric Li, Jacinta D Lucero, Chongyuan Luo, Anup Mahurkar, Delissa McMillen, Naeem M Nadaf, Joseph R Nery, Thuc Nghi Nguyen, Sheng-Yong Niu, Vasilis Ntranos, Joshua Orvis, Julia K Osteen, Thanh Pham, Antonio Pinto-Duarte, Olivier Poirion, Sebastian Preissl, Elizabeth Purdom, Christine Rimorin, Davide Risso, Angeline C Rivkin, Kimberly Smith, Kelly Street, Josef Sulc, Valentine Svensson, Michael Tieu, Amy Torkelson, Herman Tung, Eeshit Dhaval Vaishnav, Charles R Vanderburg, Cindy van Velthoven, Xinxin Wang, Owen R White, Z Josh Huang, Peter V Kharchenko, Lior Pachter, John Ngai, Aviv Regev, Bosiljka Tasic, Joshua D Welch, Jesse Gillis, Evan Z Macosko, Bing Ren, Joseph R Ecker, Hongkui Zeng, and Eran A Mukamel. A transcriptomic and epigenomic cell atlas of the mouse primary motor cortex. *Nature*, 598(7879):103–110, October 2021.

[93] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[94] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. Alpa: Automating inter- and Intra-Operator parallelism for distributed deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 559–578, Carlsbad, CA, July 2022. USENIX Association.

[95] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 283–298, USA, 2017. USENIX Association.

A Appendix

A.1 Implications and Economics of the Sky

While the body of this paper was firmly rooted in what an intercloud broker could offer now, we turn our attention to the future and ask: what are the implications of the Sky for the future cloud ecosystem? This section is inherently more speculative, so we have included it as an appendix to provide some context for where we think this approach could take us.

A.1.1 Embracing Diversity

While there is an increase in limited interface compatibility, in the overall ecosystem there is an increasing diversity in terms of location and hardware. The aforementioned regulatory concerns require greater flexibility in location; Sky Computing provides an easy way to specify the necessary location constraints. However, there are two other important location considerations. First, some tasks should be run on nearby edge clouds to lower latencies between client and cloud. Second, some tasks should be on on-premise clusters, rather than public clouds, to lower costs (see [87] for an argument as to why this is crucial). These concerns can be met by bringing edge and on-premise clouds into the Sky. The intercloud broker could then automatically send jobs to the closest edge cloud (if lowering latency is important) or to the on-premise cloud (if lowering costs is important and there is enough capacity).

In addition, by allowing users to specify specific hardware requirements in their request, one can automatically seek out clouds that have the appropriate hardware support. Or one can merely ask for high performance, and the intercloud broker will find the highest-performing cloud for that task, regardless of how they achieve it. Thus, Sky Computing turns the diversity of the current clouds from an impediment to an advantage: as long as one cloud meets a user's needs in terms of location or hardware or other constraints, the intercloud broker will find it.

A.1.2 Economic Analysis

For analytical convenience here we assume that in the future clouds will fall into two categories. Some clouds will remain *proprietary*, offering their own APIs for some tasks and charging for data egress in an attempt to keep customers tied to their cloud. However, others will join the Sky and become a *commodity* cloud in that they fully embrace the open source interfaces and do reciprocal data peering with other clouds that have joined the Sky. The economic choice facing clouds is which of these alternatives they choose. Note that even proprietary clouds can be used by the intercloud broker, but doing so may entail data egress charges.

The choice facing consumers is which of these two types of clouds they choose to use: do they send their workloads to a single proprietary cloud, or do they let the intercloud broker find which clouds to run on? In what follows, we assume that users attempt to optimize some measure of price and

performance of each task; we will denote this metric by P2, and define it so that smaller values are better. The relative importance of price and performance will differ between users, but we do not address that here as it overcomplicates the analysis without adding much insight; instead, we assume all users attempt to minimize the same measure P2. We now analyze, in a vastly oversimplified model, how the ecosystem of clouds might evolve given this consumer behavior.

Denote by R the set of proprietary clouds and denote by S the set of commodity clouds (i.e., the Sky). Assume that the workload from user α consists of a set of tasks j, with a weight or frequency w_j^α that represents the fraction of their workload that consist of task j. Note that this analysis can either apply to individual applications (which involve a DAG of tasks), or an overall workload.

The P2 of task j on cloud c is denoted by P_j^c. If a cloud does not support that task, P_j^c is set to be infinite. Let $\tilde{P}_j^c$ be the P2 taking into account the delays (and perhaps egress charges, if a proprietary cloud is used) in sending data between different clouds. We then define P_j and $\tilde{P}_j$ as the minimal P2's achievable (the latter taking into account the extra inter-cloud delays and cost, and the former not): $P_j = \min_{c \in S \cup R}[P_j^c]$ and $\tilde{P}_j = \min_{c \in S \cup R}[\tilde{P}_j^c]$.

Assume for simplicity that these workloads are either sent to the Sky (i.e., placement determined by the intercloud broker), or to a single proprietary cloud. Given these assumptions, if the workload is sent to a proprietary cloud, the user α will choose the cloud $c \in R$ that minimizes $\sum_j w_j^\alpha P_j^c$; call this cloud $c(\alpha)$. If sent to the Sky, then the overall P2 is $\sum_j w_j^\alpha \tilde{P}_j$. Given our assumptions, a user will pick between $c(\alpha)$ and the Sky, depending on whether the sum $\sum_j w_j^\alpha [P_j^{c(\alpha)} - \tilde{P}_j]$ is positive (Sky) or negative (proprietary cloud $c(\alpha)$). Note that since by definition $P_j \leq P_j^{c(\alpha)}$ this can only be negative if the inter-cloud delays or costs are significant.

The question a cloud faces is whether to join the Sky or not. If it remains a proprietary cloud, the only customers it gains are those for whom its overall average P2 is best: i.e., for those users for whom it is $c(\alpha)$. If it joins the Sky, it gains revenue for each task j where its performance is best among the clouds (taking into account the inter-cloud delays).

Assuming most users have a broad workload including many tasks, this analysis suggests that a cloud should only remain proprietary if it can compete across a broad collection of tasks. Joining the Sky becomes the rational choice for clouds who realize they cannot compete broadly, but can find narrower market niches (i.e., sets of tasks) where they excel.

Note that two proprietary clouds compete in a zero-sum manner: for users sending their workloads to proprietary clouds, either one gets the business or the other. Sky clouds compete in a much different way. Of course, they all compete to provide the best P2 implementations for each task. However, a cloud providing a superior solution for one type of task *helps* a cloud focusing on other types of tasks, because users will only use the Sky if the overall service they get is better than that on proprietary clouds. Thus, the ecosystem of Sky clouds combines *competition* on each task type with *collaboration* to provide high-quality support across a broad spectrum of tasks. This is the interdependence in the Sky.

This analysis is obviously oversimplified in many dimensions. For instance, users make different tradeoffs between cost and delay, and workloads are more complicated than just a linear combination of tasks. However, none of these considerations undercut the general observation above that proprietary clouds must be prepared to compete across a wider range of tasks (since their egress charges and proprietary interfaces *purposely* reduce the likelihood of users offloading to other clouds).

For a fledging cloud provider, it seems clear that joining the Sky is the preferable choice. These new clouds can concentrate on narrow sets of tasks where they can compete favorably with existing commodity and proprietary clouds, and they need not worry about marketing as the intercloud brokers will seek out the best P2 available.

None of these results are surprising, as the intercloud broker effectively sets up a two-sided market. Two-sided markets are common, and they are typically opposed by market actors who have high margins and want to preserve them, but are welcomed by those struggling to get a foothold in the market and who cannot otherwise overcome the inherent advantages of the dominant market players (such as much better name recognition, much larger sales forces, etc.). In the current cloud market only Amazon and perhaps Azure can be seen as having dominant market positions; all other cloud providers have less than 10% of the market [80]. For all of these other cloud providers, which comprise roughly half of the current cloud market, the Sky may be the preferable choice.

A.1.3 Speculation

In many ways, the intercloud broker is merely a mechanism that turns cloud computing into a more competitive market. However, efforts to create the Sky will be for naught if the currently dominant clouds remain dominant and proprietary even after the intercloud broker is put in place. Here we speculate briefly on the factors that will play a critical role in how the competition plays out. We start with four basic assumptions:

Sky-based clouds may innovate faster: Sky clouds need not market their technologies; they merely need to post faster speeds and/or lower prices for various workloads. Thus, the intercloud broker itself speeds innovation because workloads will automatically follow the better P2s, no matter how they arose. In addition, Sky clouds can focus their innovative energies on narrow classes of tasks where they might have special expertise (e.g., Oracle for databases) or special hardware (e.g., Samsung for storage, Google for TPUs, NVIDIA for GPUs). In fact, this is already happening; see the recent announcements by Nvidia, Equinix, and Cirrascale [12].

Large clouds have economies of scale: There are undeni-

able advantages to operating a cloud at scale, such as greater leverage with suppliers and the ability to amortize various infrastructure costs over larger deployments. These advantages may be the single biggest barrier to the success of Sky.

Infrastructure providers might provide smaller clouds with better economies of scale: Infrastructure providers, such as Equinix, who have experience in building out clouds and who can amortize infrastructure costs, can help smaller clouds with deployment. This will not match the economies of scale of the largest clouds, but will allow small clouds to be deployed with reasonable efficiency.

Small clouds are not necessarily small companies: One worry is that the proprietary clouds would engage in predatory pricing to prevent the Sky from emerging. However, many companies that will deploy Sky-based clouds will be using them as showcases for their technology (Samsung for storage, Oracle for database workloads, etc.), and they have very deep pockets. So predatory pricing will actually hurt the large clouds more than the smaller ones (because they have smaller market share, their losses are smaller).

Based on these assumptions, the crucial question is whether the rate of innovation of the smaller clouds (which can be more narrowly targeted) is sufficient to compensate for their disadvantage in economies of scale (which is mitigated by infrastructure providers). We have no wisdom to offer on this central but speculative question. However, with innovative companies like Google, IBM, and Alibaba counted as "small clouds" likely to join the Sky rather than remain proprietary, we believe that there is a significant chance that the Sky could emerge as an economically viable alternative to the current cloud ecosystem.

A.2 ML Training on Spot Instances Across Clouds

In §5.1 we evaluated SkyPilot's benefits for ML pipelines; here, we show an additional experiment to demonstrate that SkyPilot can run a single ML training job on spot instances across clouds, improving resource availability and reducing costs. In the event of spot instance preemptions, SkyPilot supports migrating a job to another zone, region, or cloud where spot instances are available. We consider training a BERT model with a V100 GPU on a subset of Wikipedia, WikiText-103 (0.5 GB), for 30 epochs. For failure recovery, we save the current model checkpoint (1.5 GB) periodically to a persistent storage. Each epoch runs for around 40 minutes and each checkpointing incurs an overhead of 0.5 minutes.

We evaluate three different strategies to run the job:

- *On-Demand*: runs on an on-demand instance on AWS.
- *SingleRegion*: runs on a spot instance in a single AWS region, `us-east-1`.[11]
- *Broker*: runs on a spot instance, with SkyPilot having the freedom to choose among all US regions of AWS or GCP.

[11] We chose it as it had the lowest preemption rate at the time of experiment among all US regions. Spot hourly price was $0.91, vs. on-demand's $3.06.

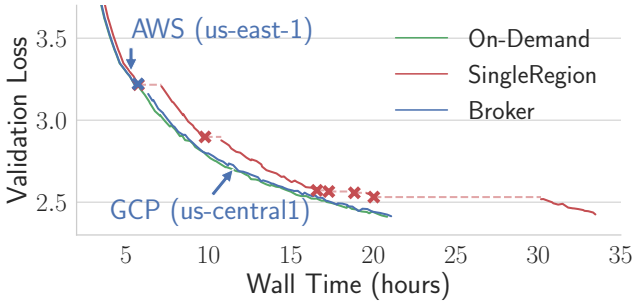


Figure 11: Loss curves of training BERT on V100 for 30 epochs. Each **x** marker is a preemption event; gaps between segments are the time periods when spot instances are not available. After the first preemption event, Broker migrates the job from AWS `us-east-1` to GCP `us-central1`, while SingleRegion waits in the same region.

	Cost	Makespan
On-Demand	$61.2	**20 hrs**
SingleRegion	$21.8	34 hrs
Broker	**$18.4**	21 hrs

Table 5: Costs and makespan for the three strategies to finish BERT training. Data transfer and checkpointing overheads are included.

For a fair comparison, we launch all strategies at the same time and in the same starting region. With SingleRegion, if no spot instances are available in the region when a preemption happens, it waits until they become available again and then resumes the job from the latest checkpoint. With Broker, if no spot instances are available it immediately triggers re-optimization and searches for availability in other regions and clouds; if found, SkyPilot transfers the data/model checkpoint to the new location and resumes the job there. The cost of each data and checkpoint egress across clouds is $0.2.

Figure 11 plots the validation loss curve for each strategy. Around hour 6, the spot instances used by both the SingleRegion and Broker strategies get preempted. SingleRegion sticks with the same region (`us-east-1`), but needs to wait for 3 hours (dashed line) to get a new spot instance. In contrast, Broker searches for spot instances in other AWS regions, which fail to provide capacity, before finding availability in GCP's `us-central1` region. After hour 6, the SingleRegion job experiences several more preemptions which cause further delays. Overall, the delays from using a single region adds more than 10 hours to the completion time.

Table 5 shows the total cost and makespan for the three strategies. Broker finishes ∼40% faster than SingleRegion because it can leverage spot instance availability across regions and clouds. Moreover, Broker is 10% cheaper than SingleRegion: despite the cross-cloud data egress costs incurred by Broker, the faster recovery time and fewer preemptions (thus, less lost progress) reduce the overall cost compared to SingleRegion. Compared to On-Demand, Broker saves 70% cost due to lower spot prices, while incurring a minimal overhead in makespan (∼5%) due to job recovery and checkpointing.