Going Beyond Hyperscaler Clouds in the Sky Ecosystem with SkyPilot

Edward Zeng

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-179
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-179.html

May 17, 2023

Acknowledgement

First, I would like to thank Ion Stoica, my advisor. I would also like to thank Zongheng Yang for his constant support, encouragement, and advice. Finally, I would like to thank the wonderful SkyPilot team: Zongheng Yang, Zhanghao Wu, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, and others.

Going Beyond Hyperscaler Clouds in the Sky Ecosystem with SkyPilot

by Edward Weicheng Zeng

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Ion Stoica
Research Advisor

(Date)

* * * * * * *

Zongheng Yang
Second Reader

(Date)

Going Beyond Hyperscaler Clouds in the Sky Ecosystem with SkyPilot

Edward Zeng
edwardzeng@berkeley.edu

May 2023

Abstract

SkyPilot is an intercloud broker being developed at UC Berkeley. Until recently, SkyPilot only supported three hyperscaler clouds: AWS, Azure, and GCP. In this report, we document the addition of a new cloud, Lambda Cloud. We describe the implementation challenges of Lambda Cloud support and discuss how we overcame them. Finally, we provide a guide for adding new clouds to SkyPilot.

Contents

1 Introduction

Cloud computing is a critical component of the modern information infrastructure, providing scalable computation and storage for a wide variety of applications and services. In an era where massive amounts of data are generated, transmitted, and processed every day, cloud computing has become an indispensable tool for businesses and users alike.

There are many cloud providers (also known as clouds) in the cloud ecosystem – some prominent examples [1] include Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure (Azure), IBM Cloud, and Oracle Cloud. Cloud users have a wide range of competing options to choose from, each with their benefits and drawbacks.

However, this multitude of options comes with a downside. Many cloud providers emphasize proprietary interfaces over cross-cloud compatibility, leading to a phenomenon known as cloud lock-in, where cloud users find it difficult to migrate their workloads between clouds.

Cross-cloud compatibility, however, has become increasingly important. First, many businesses do not want their infrastructure to be tied to one particular cloud. This makes them vulnerable to large-scale outages at the cloud level, reduces their negotiating leverage with their chosen cloud, and also prevents them from taking advantage of services offered by other clouds. Keeping infrastructure on one cloud might even not be possible: if business A acquires a business that uses a different cloud, then business A's infrastructure will be inherently multi-cloud. Second, data and computational sovereignty regulations restrict enterprises from running or storing data in certain locations. As most clouds do not have datacenters in all countries, cross-cloud workload migration may be necessary to satisfy these regulations.

To mitigate the problem of cloud lock-in, Stoica and Shenker [2] proposed the concept of Sky Computing, where instead of submitting jobs to cloud providers, users submit jobs to an intercloud broker.

SkyPilot [3] is an intercloud broker being developed at UC Berkeley. For a long time, SkyPilot was an intercloud broker only supporting three hyperscaler clouds: AWS, Azure, and GCP. In this report, we document the addition of a new cloud: Lambda Cloud. The Sky Computing vision allows smaller specialized clouds to participate actively in the Sky; Lambda Cloud integration is the first step in this direction.

This report is organized as follows. In Section 2 we discuss Sky Comput-

ing and intercloud brokers. In Section 3 we describe SkyPilot's architecture. In Section 4 we introduce Lambda Cloud, and in Section 5 we describe the challenges of Lambda Cloud support. Finally, in Appendix A, we provide a guide for adding new clouds to SkyPilot.

2 Sky Computing with Intercloud Brokers

2.1 What is an intercloud broker?

In simplest terms, an intercloud broker takes a job submitted by users, chooses a cloud, and then executes the job on that cloud (Figure 1).

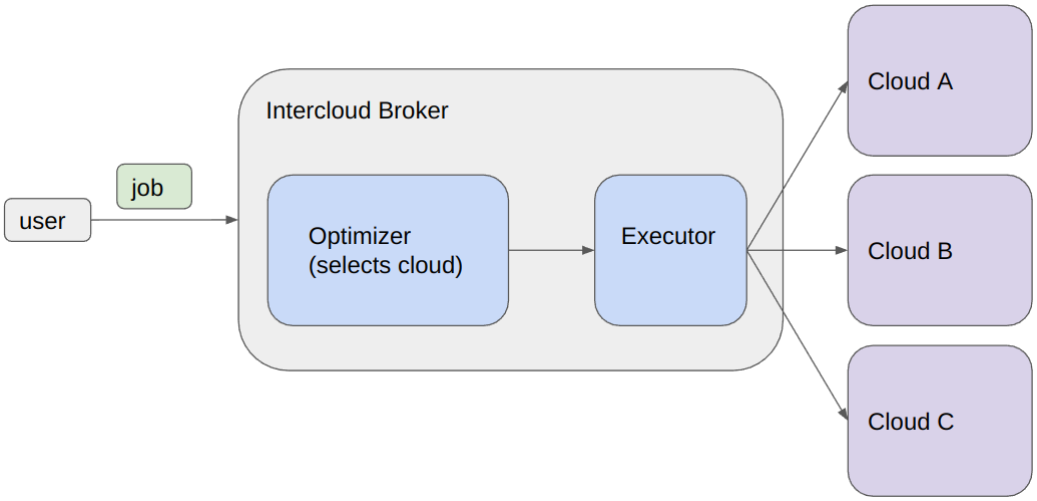

Figure 1: A Simple Intercloud Broker

Cloud selection can depend on various factors, such as estimated cost or latency.

2.2 How do intercloud brokers mitigate cloud lock-in?

Consider Cloud A in Figure 1. With an intercloud broker, users do not need to know Cloud A's interface to be able to run jobs on Cloud A. More precisely, a user can run a job on Cloud A if:

- The intercloud broker supports Cloud A, and
- Cloud A can run the job (the job does not require special functionality only found in Cloud B, for instance).

Hence, the success of Sky Computing as introduced in [2] will follow from two conditions:

1. There exists an intercloud broker that supports many clouds.
2. Most clouds can run most jobs.

This report is about achieving the first condition.

3 SkyPilot

SkyPilot is an intercloud broker being developed at UC Berkeley. In this section, we describe the architecture of SkyPilot.

3.1 Ray

SkyPilot builds on Ray [4], an open-source distributed execution framework that also originates from UC Berkeley. A core component of Ray is the Ray autoscaler (also known as the Cluster Launcher), which handles the launch and termination of virtual machines (otherwise known as nodes) on the cloud. The interface between the Ray autoscaler and a cloud is a Ray node provider, which translates internal Ray function calls into cloud-specific API calls. A brief overview of the node provider class [5] is shown below:

```
class NodeProvider:
    def non_terminated_nodes(self, tag_filters):
        # Returns a list of node ids filtered by tag_filters
    def is_running(self, node_id):
        # Returns whether the given node is running or not
    def node_tags(self, node_id):
        # Returns the tags of the given node
    def external_ip(self, node_id) -> str:
        # Returns the external ip of the given node
```

```
    def create_node(...):
        # Creates some nodes
    def terminate_node(self, node_id):
        # Terminates specified node
    ...
```

Each cloud that Ray supports must implement a node provider (Figure 2).

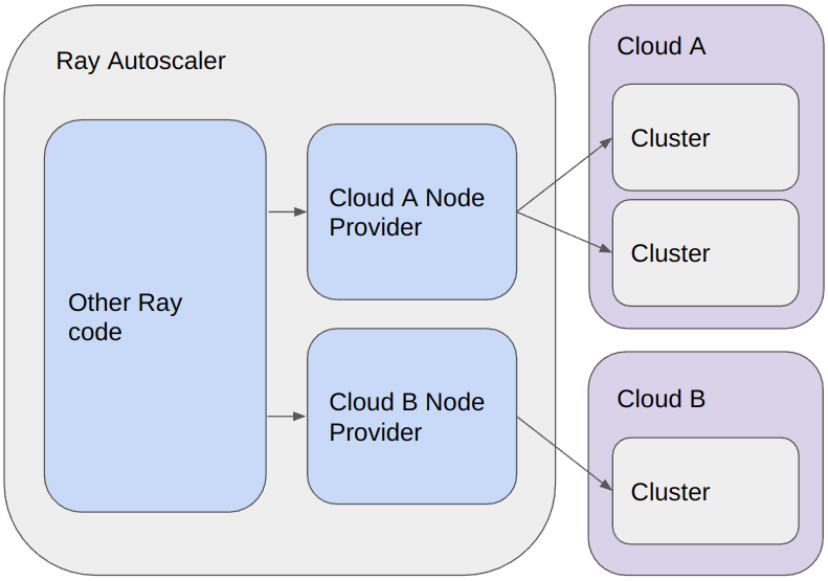

Figure 2: Ray Autoscaler Architecture

3.2 SkyPilot Architecture

Figure 3 provides an overview of SkyPilot's architecture.

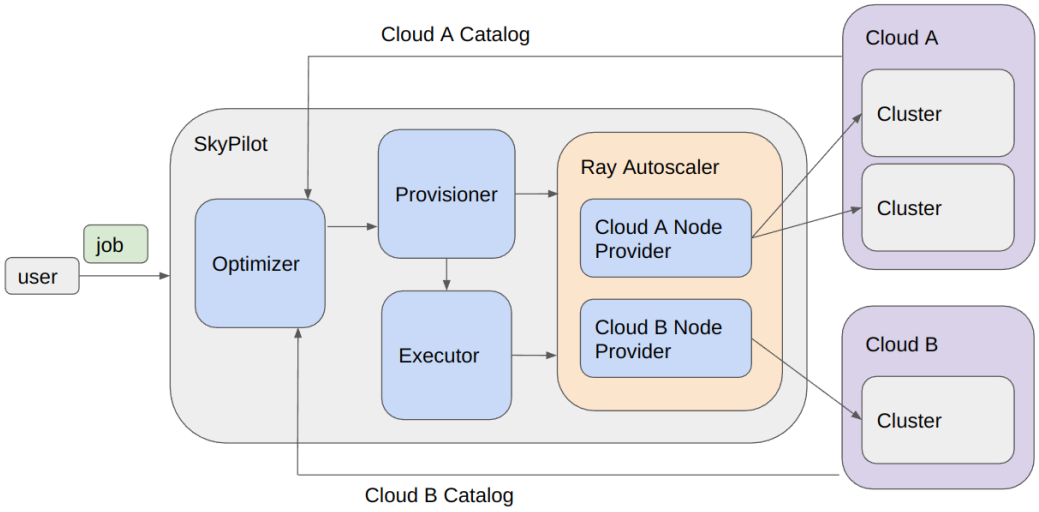

Figure 3: How SkyPilot uses Ray

A cloud **catalog** is a listing of the instance types a cloud offers, along with instance details like pricing, number of GPUs, amount of RAM, etc. When given a job, SkyPilot's **optimizer** chooses a cloud and instance type to run that job based on instance preferences and price. (There is work being done to generalize the optimizer to optimize over other variables, like estimated latency.) Once the cloud and the instance type are chosen, the **provisioner** launches a cluster (along with the SkyPilot runtime) on the specified cloud with the specified instance type. Finally, the **executor** runs the job on the launched cluster.

4 Lambda Cloud

Lambda GPU Cloud (not to be confused with AWS Lambda) is a low-cost GPU cloud [6] hosted by Lambda, a startup that specializes in deep learning compute.

Until recently, SkyPilot only supported AWS, Azure, and GCP. We chose Lambda GPU Cloud (also known as Lambda Cloud) to be the first small cloud provider to support. Our reasons were the following. First, Lambda Cloud is very cost-effective compared to its competitors, which is very important at a time where AI compute costs are skyrocketing. For instance,

an instance with 8 V100 GPUs costs 5x cheaper on Lambda Cloud than AWS or GCP, which is very important at a time where AI compute costs are skyrocketing [7].

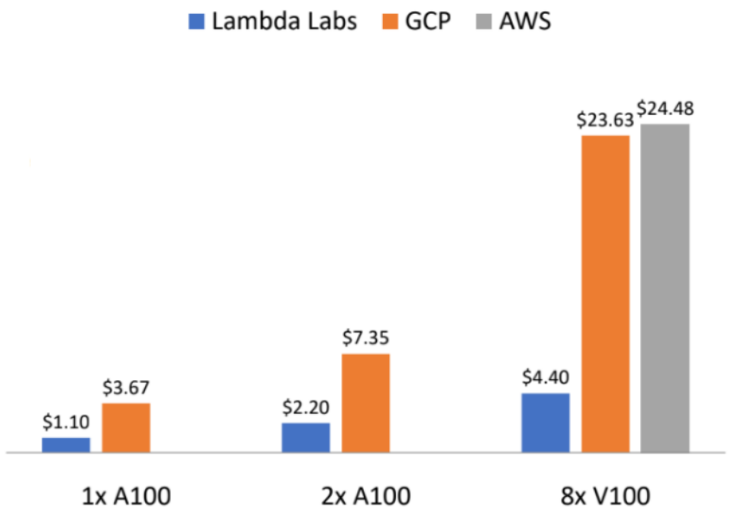

Figure 4: Cost Comparisons

Second, Lambda Cloud's interface is very different from AWS, Azure, and GCP – it is very minimalistic. Thus, Lambda Cloud integration would provide valuable insight into how to add support for a wider variety of clouds.

Third, the Sky Computing vision as laid out in [2] allows smaller, specialized clouds to participate actively in the Sky. Adding Lambda Cloud, a small cloud that specializes in deep learning compute, is the first step in this direction.

5 Integrating Lambda Cloud into SkyPilot

5.1 Overview

We implemented Lambda Cloud support for SkyPilot (merged in SkyPilot release version v0.2.5). Lambda Cloud support for single-node clusters required 1300 lines of (Python) code [8], and extending Lambda Cloud support for multi-node clusters required another 300 lines of code [9].

Of the many changes made to SkyPilot, the most notable were:

1. Lambda Cloud node provider
2. SkyPilot optimizer updates
3. Lambda Cloud catalog management
4. Lambda Cloud credential management
5. Lambda Cloud ssh setup
6. Lambda Cloud end-to-end failover

More detail about these changes is described in Appendix A. In this section, we describe the key challenges we faced: the handling of two important non-natively supported features – tagging and instance suspension.

5.2 Supporting Instance Tags on Client Side

A tag is a key-value pair. On AWS, Azure, and GCP, users can add tags to virtual machines. This allows users and applications to track virtual machine metadata. For instance, the Ray autoscaler has a tag `ray-cluster-name: xxx` to identify the cluster a node is a part of, and a tag `ray-node-type: xxx` to identify whether the node is the head node of the cluster or a worker node.

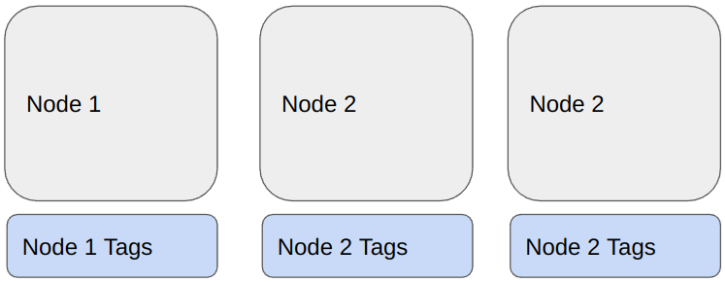

Figure 5: Tags

The necessity of instance tags for the Ray autoscaler is reflected in the node provider: every node provider must provide a function to set the

tags of a node and a function to read the tags of a node. However, because Lambda Cloud does not provide instance tags, our implementation of the Lambda Cloud node provider required a workaround. In this section, we go over different workarounds we considered and discuss their strengths and weaknesses. We start with single-node clusters and then move to multi-node clusters.

5.2.1 Single-Node Clusters

When a single-node SkyPilot cluster is launched, the client machine is the only machine that will update the tags of the single (head) node of the cluster.

Design Candidate 1: Tag File on Head Node (Not Adopted). The first design we considered was to keep a file on the head node to store its tags. Every time we needed to update or read the tags of the head node, we would update or read the tag file.

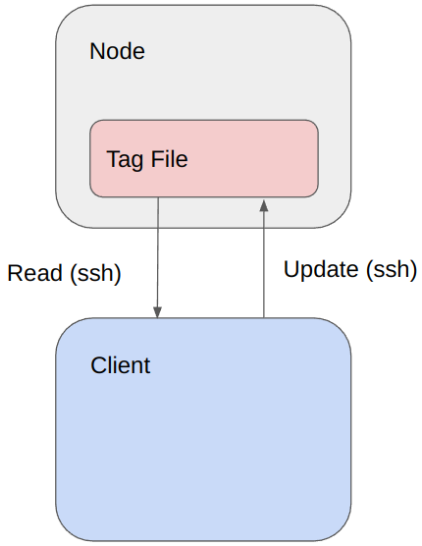

Figure 6: Tag File on Head Node

In pseudocode,

```
read_tags():
    if on client:
        over ssh:
            with file lock:
                read from tag file
    elif on head node:
        with file lock:
            read from tag file

write_tags():
    if on client:
        over ssh:
            with file lock:
                write to tag file
    elif on head node:
        with file lock:
            write to tag file
```

The correctness of this design is easy to verify because we are replicating tagging functionality in its entirety. However, this design has two large drawbacks. First, the ssh overhead turns out to be significant because of frequent tag reads. Second, if the head node crashes, then reading or writing tags will hang, making SkyPilot stall.

Design Candidate 2: Tag File on Client (Not Adopted). To improve our first design, we observed that the head node does not update its own tags in the Ray autoscaler. Thus, in our second design, we moved the tag file from the head node to the client machine. In this scheme, every time the client needed to update the head node tags, it would update its local tag file and send the updated local tag file to the head node (Figure 7).

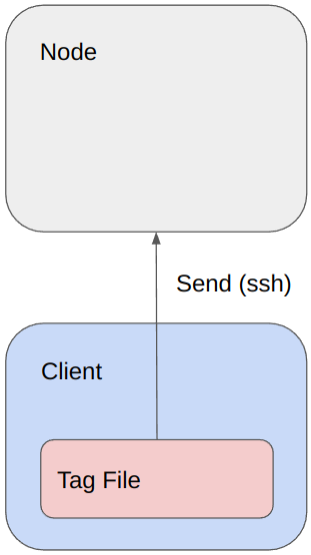

Figure 7: Tag File on Client

Sending the updated tag file is necessary because the head node needs to read its own tags. In pseudocode,

```
read_tags():
    if on client:
        read local tag file
    elif on head node:
         read most recent tag file received

write_tags():
    assert on client
    write to local tag file
    send tag file to head node over ssh
```

By only using `ssh` on tag file writes, this design mitigates the `ssh` overhead problem from the first design. However, it still uses `ssh`, which is not desirable – SkyPilot can still stall if the head node crashes.

Dynamic Tags on Client (Temporarily Adopted). This design was used when Lambda Cloud support for single-node clusters was shipped. The

key idea is that we can classify the tags that Ray uses into two types: **static tags**, which are determined at launch, and **dynamic tags**, which can change its value over its lifetime.

In Ray, dynamic tags of the head node are only read and updated by the client, not the head node. As a result, we decided to store the dynamic tags of the head node in the client's tag file and the static tags of the head node in its (statically-determined) name (Figure 8). This way, both the head node and client can access all necessary tag information without needing to send tag updates over the internet.

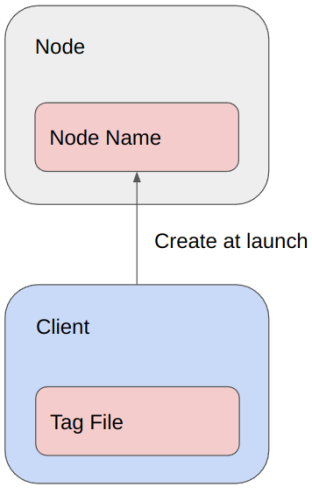

Figure 8: Dynamic Tags on Client

```
read_tags():
    if on local:
         read from local tag file
    elif on head:
        extract static tags from VM name

write_tags():
    assert on client
    write to local tag file
```

5.2.2 Multi-Node Clusters

Dynamic Tags on Launching Machine (Current Design). The current design, design 4, is a generalization of design 3 to multi-node clusters. The key idea is that the dynamic tags of any node in a cluster are only read and updated by the machine that launched it. In other words, only the client will read and update the tags of the head node, and only the head node will read and update the tags of the worker node(s). Hence, we can store static tag information in the node names and the appropriate dynamic tags on the client and the head node. Like before, this design allows all machines to access necessary tag information without needing to send updates over the internet.

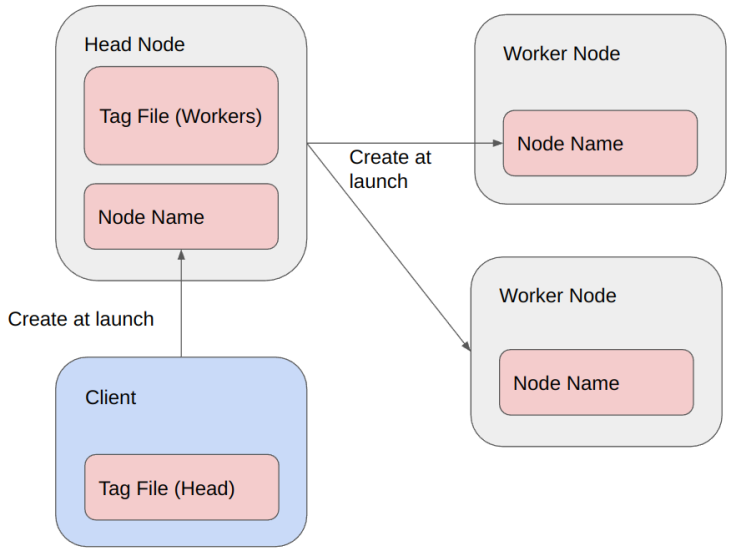

Figure 9: Dynamic Tags on Launching Machine

```
read_tags(node x):
    if on client and x is head or on head and x is worker:
        read local tag file
    else:
        extract static tags from x's VM name

write_tags(node x):
    assert on client and x is head or on head and x is worker
```

```
    write to local tag file
```

5.3 Disallowing Instance Suspension on Client Side

Instance suspension is a feature where users can shut down a VM on the cloud but keep the VM's storage. This allows users to relaunch the VM in the future by reattaching the storage. Instance suspension is useful when the user has valuable data stored on a VM but does not want to pay compute costs. With instance suspension, the user only needs to pay for storage.

SkyPilot provides a feature known as autostop, which automatically suspends idle instances to save on cost. When launching a SkyPilot cluster, SkyPilot users can specify if they want to use autostop.

AWS, Azure, and GCP support instance suspension, but Lambda Cloud does not. This means that SkyPilot cannot autostop Lambda VMs. Thus, autostop must be disabled for clusters on Lambda Cloud. The simplest way to do this is to disallow autostop when the user chooses to launch on Lambda Cloud. However, this solution misses the case where the user does not choose Lambda Cloud, but rather the optimizer chooses Lambda Cloud. Hence, we decided to make a fundamental change to the SkyPilot optimizer.

Originally, the SkyPilot optimizer only took into account price and the user's instance type preferences. However, we decided to change the optimizer to also take into account the requested features of a job (Figure 10). If the user requests a feature such as autostop, the optimizer automatically rules out clouds that do not support instance suspension, like Lambda Cloud.

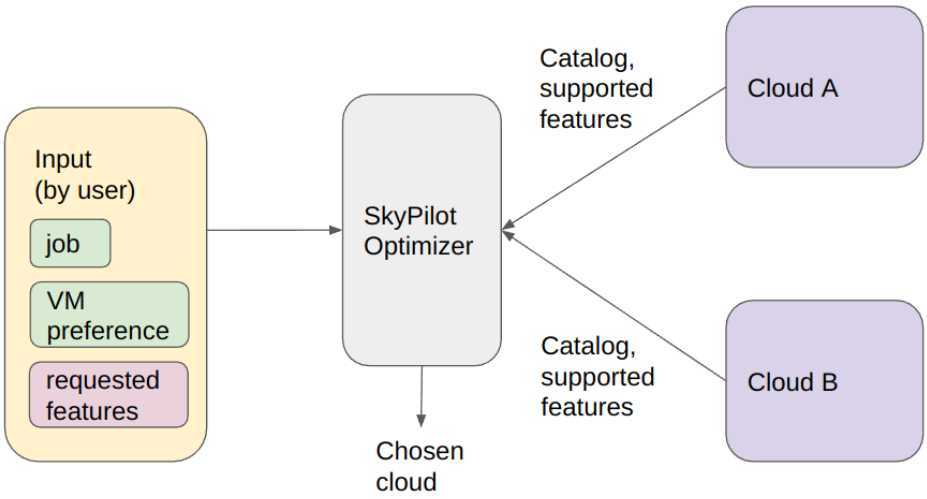

Figure 10: Skypilot Optimizer

In hindsight, this makes perfect sense – intercloud brokers should not enforce a compatibility set, and therefore the optimizer has to take into account if the user requests features that only some clouds support. Surprisingly, this issue did not come up until the implementation of Lambda Cloud support.

6 Conclusions and Future Work

Lambda Cloud is the first small specialized cloud provider to be integrated with SkyPilot, marking an important step in the road to Sky Computing.

The most difficult part of Lambda Cloud integration was Lambda Cloud's lack of tagging and instance suspension. By overcoming these challenges, we showed that clouds only need minimal functionality to be added to SkyPilot.

Other parts of Lambda Cloud support closely mirror the implementation of AWS, Azure, and GCP support (more details are found in Appendix A). We believe we can make SkyPilot integration easier for clouds by reducing code duplication in this area. This will be our future work.

7 Acknowledgements

First, I would like to thank Ion Stoica, my advisor. I would also like to thank Zongheng Yang for his constant support, encouragement, and advice. Finally, I would like to thank the wonderful SkyPilot team: Zongheng Yang, Zhanghao Wu, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, and others.

8 References

[1] Top 10 cloud service providers globally in 2023. https://dgtlinfra.com/top-10-cloud-service-providers-2022/.

[2] 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.

[3] Zongheng Yang, Zhanghao Wu, Michael Luo, Wei-Lin Chiang, Romil Bhardwaj, Woosuk Kwon, Siyuan Zhuang, Frank Sifei Luan, Gautam Mittal, Scott Shenker, and Ion Stoica. SkyPilot: An intercloud broker for sky computing. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 437–455, Boston, MA, April 2023. USENIX Association.

[4] https://github.com/ray-project/ray.

[5] https://github.com/ray-project/ray/blob/master/python/ray/autoscaler/node_provider.py.

[6] https://lambdalabs.com/service/gpu-cloud.

[7] Navigating the high cost of ai compute. https://a16z.com/2023/04/27/navigating-the-high-cost-of-ai-compute/.

[8] https://github.com/skypilot-org/skypilot/pull/1557.

[9] https://github.com/skypilot-org/skypilot/pull/1718.

[10] https://github.com/ewzeng/skypilot-new-cloud.

[11] https://lambdalabs.com/service/gpu-cloud.

A Adding a New Cloud to SkyPilot

A.1 Introduction

This appendix is an operational guide for adding a new cloud to SkyPilot, along with a new cloud support repository [10]. This guide and repository have been used by developers from Samsung and Oracle to add support for their clouds.

In this guide, we call our new cloud **FluffyCloud**.

A.1.1 What Does FluffyCloud API Need to Support?

The successful integration of Lambda Cloud shows that only a minimalistic API is necessary for SkyPilot integration. More specifically, FluffyCloud's interface only needs to be able to:

1. Launch an `ssh`-able instance in given region/zone.
2. Terminate an instance given instance id.
3. List details (e.g. id, status, ip, name) of currently running instances.

A.2 Implementation

The rest of this appendix is a step-by-step implementation guide on how to add FluffyCloud to SkyPilot.

First, we suppose that FluffyCloud has the following Python interface:

```
def launch(name:str,
           instance_type:str,
           region:str,
           api_key:str,
           ssh_key_name:str):
    """Launches an INSTANCE_TYPE instance in REGION with
    given NAME.

    API_KEY is a secret registered with FluffyCloud.
    It is per-user. SSH_KEY_NAME corresponds to a ssh key
    registered with FluffyCloud. After launching, the user
```

```
    can ssh into INSTANCE_TYPE with that ssh key.

    Returns INSTANCE_ID if successful, otherwise None.
    """

def remove(instance_id:str, api_key:str):
    """Removes instance with given INSTANCE_ID."""

def list_instances(api_key:str):
    """Lists instances associated with API_KEY.

    Returns a dictionary:
    {
        instance_id_1:
        {
            status: ...,
            tags: ...,
            name: ...,
            ip: ....
        },
        instance_id_2: {...},
        ...
    }
```

Although we do require clouds to support some (minimalistic) functionality as described in the previous section, we do not expect their interface to be the same as FluffyCloud's interface. Hence we point out parts of this implementation guide that may change depending on interface specifics.

A.2.1 Setup

1. Clone SkyPilot [11] and install from source.
2. Clone the new cloud support repository [10].

Notation. In this guide, the path `sky/` means `skypilot-repo/sky/` and the path `fluffycloud/` means `new-cloud-support-repo/fluffycloud/`.

A.2.2 Implement Node Provider

1. Read through the node provider class definition [5].
2. `fluffycloud/node_provider.py` provides some template code for implementing the node provider. Use it to implement FluffyCloud's node provider.

   Note: If your cloud has a different API than FluffyCloud, then you may need to make substantial tweaks to the template code.

To add FluffyCloud's node provider to SkyPilot:

1. Create the directory `sky/skylet/providers/fluffycloud/`
2. Create a copy of `fluffycloud/__init__.py` and place it at

   `sky/skylet/providers/fluffycloud/__init__.py`

   This is a one-line file, no changes are needed. (Of course, you may need to rename FluffyCloud to the name of the cloud you are adding.)
3. Place FluffyCloud's node provider at

   `sky/skylet/providers/fluffycloud/node_provider.py`

A.2.3 Add Catalog

A Skypilot catalog is a `csv` file with the following columns:

```
InstanceType,AcceleratorName,AcceleratorCount,vCPUs,
MemoryGiB,Price,Region,GpuInfo,SpotPrice
```

For instance, a catalog entry for Lambda Cloud would look like:

```
gpu_8x_v100,v100,8.0,92.0,448.0,4.4,us-east-1,
"{'gpus': [{'name': 'v100', 'manufacturer': 'nvidia',
'count': 8.0, 'memoryinfo': {'sizeinmib': 16384}}],
'totalgpumemoryinmib': 16384}",
```

Your task:

1. Create a catalog at

   `~/.sky/catalogs/v5/fluffycloud/vms.csv`

   and add one catalog entry (more can be added later).

A.2.4 Catalog parsers

Now let's implement some functions that parse and read the catalog. This is very easy since most of this functionality has been implemented; we are simply calling the appropriate functions.

1. Implement `sky/clouds/service_catalog/fluffycloud_catalog.py` by copying `sky/clouds/service_catalog/lambda_catalog.py` and making appropriate changes.

A.2.5 Implement FluffyCloud Class

Now let's create the FluffyCloud class. This class calls some service catalog functions and contains code that checks FluffyCloud credentials. Many of the functions in this class are also straightforward to implement and can be copied from other clouds.

1. Create a copy of `fluffycloud/fluffycloud.py` and place it at

   `sky/clouds/fluffycloud.py`

   Most of code can be used as is, with the exception of a few marked locations.

A.2.6 Implement ssh setup

This code is executed (via `ssh`) after a cluster is launched. Most of this code is very similar to the existing setup code for other clouds, and can be copied from them.

1. Implement `sky/templates/fluffycloud-ray.yml.j2` by copying

   `fluffycloud/fluffycloud-ray.yml.j2`

and making appropriate changes.

2. Update `_get_cluster_config_template` in

   `sky/backends/cloud_vm_ray_backend.py`

A.2.7 Implement end-to-end failover

Now let's add support for end-to-end failover for FluffyCloud. Like the previous step, most of this code is very similar to the existing code for other clouds.

1. Copy `_update_blocklist_on_fluffycloud_error` from

   `fluffycloud/failover.py`

   and place it in `sky/backends/cloud_vm_ray_backend.py`.

2. Update `_update_blocklist_on_error` in

   `sky/backends/cloud_vm_ray_backend.py`

A.2.8 Miscellaneous updates

Finally, there are some miscellaneous places that we need to make changes to. They are described in `fluffycloud/misc.py`.

A.3 Instance Suspension (if FluffyCloud supports it)

SkyPilot supports instance suspension by setting `cached_stopped_nodes: true` in the generated Ray `yaml`. Hence, to support instance suspension, it suffices to make the node provider stop instances in `terminate_node` instead of terminating them if `cached_stopped_nodes` is set to `true`.