Sky Computing with Intercloud Brokers

By

Zhanghao Wu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Scott Shenker
Professor Joseph E. Gonzalez
Professor Song Han

Spring 2024

Sky Computing with Intercloud Brokers

Abstract

Sky Computing with Intercloud Brokers

by

Zhanghao Wu

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

In an era where digital infrastructure increasingly relies on cloud computing, the need for flexible workload migration across clouds has become crucial. This need is particularly pressing with the recent surge in artificial intelligence (AI), which is impacted by global GPU shortages and geopolitical technology restrictions. The traditional cloud computing model, characterized by strong customer lock-in due to proprietary service interfaces and data gravity, limits the ability of businesses to adapt to these changes.

This dissertation extends and explores a recent rising concept, Sky Computing, as a transformative approach to mitigate these limitations. Sky Computing redefines the interaction between users and cloud services, proposing a unified "Sky of Computing" instead of isolated providers. This model leverages intercloud brokers to abstract the underlying cloud services, improving workload migration across clouds. We extensively explore the architectural and practical implementations of Sky Computing, leading to the development of an open-source intercloud broker, SkyPilot. SkyPilot demonstrates significant enhancements in optimizing and managing batch jobs, offering substantial cost savings, which is later extended to serving workloads, especially for AI. Further, this dissertation examines broker policy designed for deadline-sensitive jobs, implementing effective policies on SkyPilot that enable the utilization of unreliable but cost-effective spot instances while still meeting the deadline. Through real-world applications, Vicuna and SkyPilot Serving, we demonstrate how Sky Computing can support AI workloads, paving the way for further research.

Overall, we not only underscore the challenges faced by current cloud computing but also pioneer an adaptable approach through Sky Computing. This dissertation is an early step towards a more integrated and flexible cloud ecosystem, aligning technical innovation with market needs.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

I am immensely grateful to the many individuals who supported me throughout my PhD journey at Berkeley. Their guidance was pivotal in both my professional development and personal growth, making my time at the university truly enriching.

First and foremost, I extend my deepest gratitude to my advisor, Ion Stoica. Starting a PhD during the pandemic was not easy, and Ion helped me navigate through this difficult period with those late evening one-on-one meetings, accommodating time zone differences. His guidance was crucial at every stage of my PhD journey. Ion taught me to focus on important topics and strive for impact. His ability to see the bigger picture helped me approach research from a broader perspective and establish long-term goals. Each meeting with him was a profound learning experience that shaped not only my academic pursuits but also my personal development.

I am also immensely thankful to my dissertation committee members: Scott Shenker, Joseph E. Gonzalez, and Song Han. Their insights profoundly influenced the scope and direction of my research. This dissertation would not have been possible without their invaluable support. Scott's extensive knowledge of computer systems and economics always amazes me, making the research on Sky Computing a wonderful journey. From him, I also learned the importance of approaching problems fundamentally and starting with simple solutions. His ever-present humor lightened even the most challenging moments, for which I am especially grateful. Joey's energy and enthusiasm for delving into the intricacies of artificial intelligence were contagious. Working with him on projects, Graph Neural Networks and Vicuna, was not only educational but also joyful. His approach to research has left a lasting impact on how I view and tackle problems. My heartfelt thanks go to Song, with whom I had the fortune of starting my research journey on efficient machine learning during my undergraduate. His support was crucial in my early academic career. The foundational skills and confidence I gained under his mentorship were invaluable.

I have been very fortunate to work with brilliant friends throughout the four years. Zongheng Yang has influenced my PhD significantly. His mindfulness and guidance supported me in pursuing research on Sky Computing. His ability to gather efforts and move the team towards the goal has greatly inspired me and significantly helped the development and growth of our projects. Wei-Lin Chiang, with whom I worked closely throughout my PhD, has been an excellent collaborator and a great friend to discuss ideas and code together on the multiple projects we worked on. I started my collaboration with Romil Bhardwaj at a later stage of my PhD, but the countless discussions and close work on both technical and visionary problems have been really joyful and fruitful.

I am blessed to have worked with great collaborators for the Sky Computing related papers, SkyPilot and "Can't Be Late", and the SkyPilot team: Romil Bhardwaj, Wei-Lin Chiang, Eric Friedman, Tyler Griggs, Doyoung Kim, Woosuk Kwon, Frank Sifei Luan, Michael Luo, Ziming Mao, Gautam Mittal, Scott Shenker, Ion Stoica, Tian Xia, Zongheng Yang, and Siyuan Zhuang. Thanks for the great collaborations. I have learned a lot from you!

During the deployment of the open-source SkyPilot, I am very grateful to have worked with researchers from many different external organizations. Working with biologists at the Salk Institute, helping them set up their cloud infrastructure with SkyPilot, and learning from their early experience with large-scale computation on clouds was highly enriching. It was very gratifying to see the biology work our system contributed to get published in Nature. I am especially thankful to Joseph Ecker, Hanqing

Liu, Qiurui Zeng, Jingtian Zhou, and more people from the institute. Despite Salk, we also got countless wonderful users and friends who gave us excellent feedback on our system.

During the rise of large language models and our work on Vicuna, LLM Judge, and Chatbot Arena, I am honored to have worked with excellent colleagues and friends: Wei-Lin Chiang, Joseph E. Gonzalez, Dacheng Li, Tianle Li, Zhuohan Li, Zi Lin, Ying Sheng, Ion Stoica, Eric P. Xing, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, and Yonghao Zhuang. Before these projects, in my early PhD journey, I am grateful to have worked on several AI projects, including RLlib flow and GraphTrans, with wonderful collaborators: Paras Jain, Joseph Gonzalez, Eric Liang, Michael Luo, Sven Mika, Azalia Mirhoseini, Ion Stoica, and Matthew Wright.

The staff members of the lab – Kattt Atchley, Shane Knapp, Jon Kuroda, Ivan Ortega, David Schonenberg, Kailee Truong, and Boban Zarkovich – played a crucial role in enriching my graduate experience. Their assistance was invaluable. I am also fortunate to have had the opportunity to interact with and learn from a diverse group of talented individuals, including Tiemo Bang, Shiyi Cao, Audrey Cheng, Yu Gai, Paras Jain, Xiuyu Li, Kevin Lin, Xiaoxuan Liu, Simon Mo, Philipp Moritz, Robert Nishihara, Isaac Ong, Shishir Patil, Nathan Pemberton, Conor Power, Daniel Rothchild, Peter Schafhalter, Eyal Sela, Sheng Shen, Shangyin Tan, Sijun Tan, Stephanie Wang, Justin Wong, Sarah Wooders, Minkai Xu, Yi Xu, Edward Zeng, Hong Zhang, Tianjun Zhang, Wen Zhang, and Jinhao Zhu, as well as everyone in the PS2 group. A special note of appreciation for Zhuohan Li, whose roles as a superb cook, roommate, colleague, and friend have profoundly enriched my life. Our shared experiences and discussions have been a highlight of my journey.

Before joining Berkeley, I was fortunate to meet and be mentored by wonderful people at SJTU and MIT. I would like to thank Yong Yu, Kai Yu, Yanmin Qian, and Shuai Wang for introducing me to research during my undergraduate studies at SJTU. The ACM class at SJTU was a wonderful place to spend four years of my life, where I met so many wonderful faculties and friends and connected with brilliant alumni, including Xiaoyuan Liu, Zetian Jiang, Lizhen Zhu, Boyu Tian, Chen Wang, Jiaheng Zhang, Qizhe Xie, and so on. At MIT, I was lucky to have worked with Han Cai, Song Han, Ji Lin, Yujun Lin, Zhijian Liu, and Hanrui Wang at MIT HanLab. Zhijian was an incredible mentor, providing me with invaluable advice for research and my career. The interaction with the labmates at the time was also an unforgettable experience, including Tianzhe Wang, Muyang Li, Yaoyao Ding, Zhekai Zhang, Ligeng Zhu, and others.

Whenever I needed a break from research, my friends were always there to lift my spirits. Spending time in the Bay area with Yikai Li, Yunxuan Xiao, and Jiangbei Li always brought great joy and reminded me of the unforgettable moments we shared during our undergraduate days. Even though we are spread across different corners of the world, catching up online with Zhou Fan, Jinxiao Chen, Muyang Li, and Tianzhe Wang always provides a relaxing and joyful break from my routine.

Most importantly, I owe a huge debt of gratitude to my parents, Qijun Zhang and Zhengwu Wu, for their unconditional love and the numerous sacrifices they made to bring me up with the best environment for my growth. My extended family of grandparents, aunts, uncles, and cousins has always provided a network of support and love. Last but definitely not least, immense thanks to Qiurui (Rachel) Zeng, my partner and best friend, for her unwavering support and heart. Her presence makes even the toughest days bright. It is her who gives me the courage to fight against tough problems. I cherish every piece of memory we created together. And, not to forget our beloved puppy, Rocket, whose affection is a constant source of joy. This dissertation is dedicated to my family, without whom none of this would have been possible.

Chapter 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 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 has 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.

As cloud computing increasingly underpins our digital infrastructure, businesses face growing concerns about cloud lock-in. Firstly, companies seek to avoid reliance on a single cloud provider, as such dependence not only diminishes their negotiation leverage but also exposes them to risks associated with provider-specific outages. Secondly, the evolving landscape of data sovereignty regulations, such as GDPR, mandates precise controls over where data is hosted and how computational tasks are distributed across geographies. Unfortunately, not all clouds are available in every region, turning this lack of mobility across clouds into a significant barrier for businesses providing services across the globe. Thirdly, applications locked into a single cloud cannot exploit superior services from multiple providers, which might offer better pricing, availability, or performance. Given the incidents of widespread cloud resource outages, the tightening grip of governmental regulations, and the emerging new services specialized for

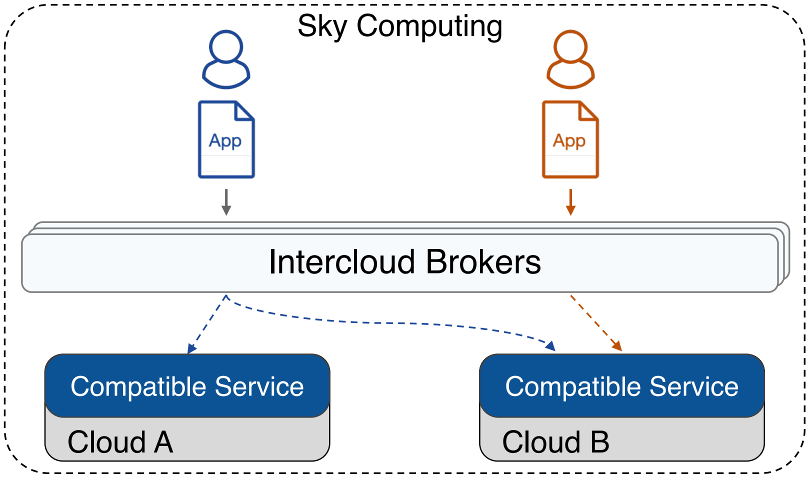


Figure 1.1: Sky Computing with intercloud brokers.

specific workloads, the ability to seamlessly transfer workloads between cloud providers has transitioned from a desirable feature to an essential requirement for businesses operating at scale. As a particular example, the rapidly growing artificial intelligence (AI) workloads are facing significant challenges due to a global shortage of GPUs [139], and geopolitical restrictions that require the AI models and data located in a specific region. To enable AI to be accessed globally, a more adaptable approach to workload migration, particularly across providers within limited regions is required.

Sky Computing, a concept first introduced in [133], was proposed to ease such migration of workloads by introducing an additional layer on top of cloud services, what we call an intercloud broker, see Figure 1.1. It changes the fundamental cloud abstraction, with users interacting with a more coherent "Sky of Computing" rather than with a collection of individual and deliberately differentiated clouds. This paradigm shift is not only a technical proposal but also represents a substantial market transformation within the cloud computing sector.

In this dissertation, we significantly expand upon and explore the concept of Sky Computing in depth. Starting with a conceptual overview (Chapter 2), we compare related concepts like standardization and multicloud with Sky Computing, which advocates for multiple application-specific intercloud brokers rather than a general standard across clouds (standardization) and abstracts away the clouds from users instead of having users interact with individual clouds (multicloud). We analyze how this vision could transform the cloud computing market and discuss the implications and economics of this model, envisioning a future of a two-sided market between users who offer jobs and clouds who offer services.

We then delve into the primary component of Sky Computing—the intercloud brokers (Chapter 3). Recent developments in cloud services, including increased interface compatibility and reduced barriers to data movement, pave the way for our design of the first intercloud broker. We outline the requirements and practical architecture design for these brokers, specialized for computational batch jobs, and contextualize our concept with existing work.

Following the conceptual and architectural groundwork, we implement a real system, **SkyPilot**

(Chapter 4), systematically constructing each component of the intercloud broker architecture. Our experiments across multiple real applications demonstrate that Sky Computing can benefit batch jobs by enabling their placement across multiple clouds, even when considering data egress costs. As an open-source project, SkyPilot has been adopted by various organizations, and we share insights and feedback from users to further the development of Sky Computing.

SkyPilot has not only proven effective for batch jobs but also shows a potential for brokers enriching the scheduling landscape by providing a broader action space with the support of workload migration. For example, with jobs managed by SkyPilot, the dynamic changes of the cloud service offerings including pricing, availability, and performance, can be utilized by a well-designed online policy that migrates jobs to better services. In Chapter 5, we explore a broker policy design for cost reduction that schedules deadline-sensitive jobs on two resource types offered by clouds – spot and on-demand instances, where the former offers cost savings at the risk of the volatile availability and the latter is more costly but reliable. We first develop a theoretical model to analyze baseline policies, which inspires the design of a simple and effective policy, **Uniform Progress**. Our empirical study demonstrates that it can outperform the baseline policy by closing the gap to the optimal policy by $2\times$. By implementing this policy on top of our intercloud broker, SkyPilot, we achieve 27%-84% cost savings across a variety of representative real-world workloads and deadlines.

The launch of SkyPilot and broker policy investigation have offered an opportunity to leverage Sky for resolving real-world problems. In Chapter 6, we present early findings from our two use cases of Sky on AI training and serving to further facilitate the future research and development of Sky Computing as well as the intercloud brokers. We first reviewed the **Vicuna** project, one of the earliest high-quality open-source chatbots. The whole workflow of Vicuna was run on SkyPilot, which yields the benefit of the Sky for AI training: packaging, high GPU availability, and cost reduction. We then switch gear to the AI serving, where we build an end-to-end serving system on top of SkyPilot, which we call **SkyPilot Serving**, to enable users to serve their AI services on the Sky. We design a simple interface with a highly modularized architecture for the system. With the preliminary adoption of the system, we have gathered multiple interesting findings and open questions that can lead to future research.

Finally, in Chapter 7, we conclude by discussing future directions for research that could propel us toward a Sky Computing future. This dissertation contributes to the ongoing evolution of cloud computing by demonstrating the potential of Sky Computing and initiating the first steps with the intercloud broker for batch jobs.

We anticipate that the proliferation of intercloud brokers, specialized for various applications, will revolutionize how users interact with cloud environments, fostering a more open, competitive, and user-centric ecosystem.

Previously published and open-source materials. This dissertation includes previously published and co-authored work as follows. Chapter 2, Chapter 3, and Chapter 4 include materials from [163, 130]. Chapter 5 includes materials from [157]. Chapter 6 includes materials from Vicuna blog post [42] and SkyPilot open-source project [130, 160].

Chapter 2

Sky Computing

Cloud Computing has become a prevalent paradigm for deploying, managing, and scaling applications. Recent years have seen the rise of many cloud providers, such as Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform, which have expanded the range of services available to users. In contrast to Internet and telephony, the other two components of modern information infrastructure, the cloud computing market has evolved away from commoditization, with cloud providers striving to differentiate themselves through proprietary services. The fragmentation has made it difficult for users to move their workloads between clouds, a capability that is increasingly important due to government regulations about data placement and processing, and to protect against major cloud outages. Sky Computing was proposed to engender a new future of cloud computing by leveraging intermediation, called intercloud brokers, to increase workload portability.

In this chapter, we summarize the history of Cloud Computing, see Section 2.1, and look into the problems caused by the siloed clouds, see Section 2.2. We then review the previous concepts related to Sky Computing, see Section 2.3, describe the vision of Sky Computing and transformational changes it could make to the future, see Section 2.4. Finally, we examine the implications of and economics of the Sky for the future, see Section 2.5.

2.1 The History of Cloud Computing

The idea of delivering computing as a utility, akin to traditional utilities like telephone systems, was pioneered by John McCarthy, a prominent figure in artificial intelligence. However, it was not until the 1980s, that the NSF's high-performance computing provided an early peek into the concept of utility computing. With the arrival of the Internet, a variety of services became globally available, such as email, bulletin board systems, and gaming. The introduction of the World Wide Web led to a surge in new services like search engines, online shopping, and later on, social media. To handle the increased demand, these services had to establish data centers and create complex distributed systems.

The early 2000s witnessed a rapid expansion in cloud computing technologies, with the launch of S3 and EC2 Amazon Web Services (AWS) in 2006 starting the cloud computing era. This movement made computing/storage resources accessible to a broader audience and introduced a "pay-as-you-go" business model. This model democratized access to computing resources, allowing startups and large enterprises alike to scale their operations without the upfront cost of traditional hardware infrastructure. This shift, coupled with the end of Moore's Law, which made it costly to develop and scale services using on-premise systems, reignited interest in what utility computing could have been. Yet, market forces have steered us towards a different path.

In the early days, Amazon led the cloud computing market, setting the unofficial standards. Over the last ten years, though, despite the leading Amazon Web Services, which controls 31% market share, numerous competitors have entered the scene, including major players, Microsoft and Google, taking 35% of the market in total, and other providers like Alibaba, Salesforce, IBM and Oracle dividing the remainder of the market [45]. This competition has driven down prices and broadened the range of products and services available. For instance, AWS now offers over 200 products and services [44].

2.2 Cloud Silos and Challenges

While hundreds of services are provided by cloud providers, a significant number of these products are exclusive to each provider, serving as a key strategy for standing out in the market. Each provider offers *unique APIs* for cluster management, specialized versions of data storage and processing services, and distinct serverless computing options, among others. As a result, applications designed for one cloud platform may require significant modifications to run on another, similar to how software designed for Windows might need alterations to operate on Mac OS. 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 has 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. Consequently, the competition in the cloud market is moving us away from the concept of utility computing, *i.e.,* clouds are being siloed.

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 its infrastructure tied to a single provider because such lock-in reduces its 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 data centers 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, such as recent shortage of high-end GPUs, essential for AI workloads, on major clouds [140], and the increasing number of government regulations, such as EU's GDPR [154], are quickly making such a solution a "must-have" for large-scale users of the cloud.

2.3 Related Concepts

The initial players in the cloud computing industry each introduced unique, proprietary interfaces, causing cloud silos. To ease the migration of workloads across clouds, we first examine two related concepts for going beyond individual clouds: adopting standards and multicloud, and discuss why they are not sufficient for achieving the goal.

2.3.1 Standardization

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? A decade ago IEEE proposed a set of Intercloud standards for portability, interoperability, and federation among cloud providers [153] 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 [119] or TensorFlow [1] in addition to low-level orchestration interfaces such as Kubernetes [83]. If the goal is to make workload migration seamless, then all of these interfaces would need to be standardized, which would impede innovation by locking the ecosystem into a set of interfaces that may not be appropriate for future uses. While this issue can arise in any tech ecosystem, it seems especially pertinent to cloud computing for having premature and/or inappropriate standardization. This is because user interaction with the cloud spans various interfaces, from low-level basic orchestration to high-level complex services, making it difficult to pinpoint which layers should be standardized. Additionally, these interfaces evolve quickly, raising questions about the right timing for standardization.

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).

2.3.2 Multicloud

Multicloud is now an industry buzzword, and there are reports [71, 138] 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.

2.4 Sky Computing

With standardization and multicloud having limited success in easing the migration of workload, this dissertation is about how to achieve it through the rise of Sky Computing, a concept first introduced in [133] but significantly extended and more deeply explored in the rest of this dissertation. In this section, we review the vision of Sky Computing from [133, 38] and argue why Sky Computing will be a transformational change in cloud computing [163].

2.4.1 The Vision of Sky Computing

Sky Computing is when users, rather than directly interacting with the cloud, submit their jobs to what is called intercloud brokers, detailed in Chapter 3, who handle the placement and oversee the execution of their jobs (see Figure 1.1). Users of an intercloud broker specify their job characteristics and constraints, along with their desired optimization metrics (such as price or performance), eliminating the need to decide which cloud services to employ for specific tasks. The intercloud broker then selects the clouds on which various parts of the job are run and then manages their execution.

In contrast to existing multi-cloud solutions, Sky abstracts away the clouds. When running a Sky application, it is transparent to the user which clouds the application runs on. The presence of intercloud brokers changes the fundamental cloud abstraction, with users interacting with a more coherent "Sky of Computing" rather than with a collection of individual and deliberately differentiated clouds. It creates a two-sided market between users who offer jobs and clouds that offer services. Many of these services (*e.g.,* Kubernetes, Apache Spark, Apache Kafka) are offered by multiple clouds, while others are cloud-specific (*e.g.,* AWS Inferentia, BigQuery). We expect that there will be multiple intercloud brokers, some possibly specialized for different workloads, and the emergence of effective intercloud brokers will foster a dynamic cloud services marketplace with many of those services being offered by more than one cloud, enhancing the portability of workloads.

This evolution in cloud services compatibility is expected to yield significant advantages, such as:

1. Reducing entry barriers to cloud adoption and thus enlarging the market.

2. Fostering rapid technological advancements through the proliferation of specialized clouds, offering users top-tier services and hardware.

3. Achieving a more cohesive integration of computing solutions, including edge and on-premise computing, along with the selection of specific cloud zones.

4. Improving compliance, security, and system robustness through diversified cloud strategies, like deploying model inference across several clouds for better reliability, or processing sensitive data in compliance with emerging legal requirements on data and operational sovereignty.

2.4.2 Why is this Transformational?

As noted in our paper [163], we are not the first to use the name "Sky Computing" as several papers, dating back to 2009, also used this term [57, 104, 98]. 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). In our vision, we take a broader view of Sky Computing, seeing it as a change in the overall ecosystem and considering how technical trends and market forces can play a critical role in the emergence of Sky Computing.

Sky Computing is a transformational change in cloud computing, not merely a tactical mechanism for workload migration. There are three reasons, each from a different perspective.

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 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, 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 Section 2.5 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 dissertation is to investigate how to start this transformation, not to define its ultimate form. As such, we start with an intercloud broker for batch jobs and AI services – 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.

2.5 Implications and Economics of the Sky

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 to provide some context for where we think this approach could take us.

2.5.1 Embracing Diversity

While there is an increase in limited interface compatibility (detailed in Section 3.1), 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 clients and clouds. Second, some tasks should be on on-premise clusters, rather than public clouds, to lower costs (see [152] 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.

2.5.2 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 consists 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 15% of the market [45]. For all of these other cloud providers, which comprise roughly half of the current cloud market, the Sky may be the preferable choice.

2.5.3 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 arise. In addition, 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 [20].

Large clouds have economies of scale. There are undeniable 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 a 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.

Chapter 3

Intercloud Brokers

In Chapter Chapter 2, we explored the concept of Sky Computing, a forward-looking paradigm in cloud computing, and its potential impacts. Sky Computing is predicated on the use of intermediation between users and cloud providers. This approach employs intercloud brokers to manage the placement and execution of user jobs, thereby eliminating the need for direct interaction between users and cloud providers.

This chapter checks the technical aspects of the journey towards implementing intercloud brokers. Initially, we will revisit recent advancements in cloud services and the emerging trend of limited interface compatibility, which forms the foundation for the functioning of intercloud brokers (see Section 3.1). Building on the premise of enhanced compatibility, we shift our focus to the design of intercloud brokers specifically tailored for computation-intensive batch jobs. We will examine the feasibility and advantages of integrating such brokers, analyze their requirements, and propose a potential architecture for their implementation (see Section 3.3). Lastly, we will discuss the literature pertaining to cross-cloud interactions and contextualize these studies in relation to intercloud brokers (see Section 3.4).

3.1 Growth in Interface Compatibility

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 [83], Apache Mesos [72]), application packaging (Docker [54]), databases (MySQL [107], Postgres [121]), big data execution engines (Apache Spark [166], Apache Hadoop [155]), streaming engines (Apache Flink [36], Apache Spark [166], Apache Kafka [16]), distributed query engines and databases (Cassandra [14], MongoDB [103], Presto [122], SparkSQL [131], Redis [125]), machine learning libraries (PyTorch [119], TensorFlow [1], MXNet [39], MLflow [102], Horovod [128], Ray RLlib [88]), and general distributed frameworks (Ray [105], Erlang [19], Akka [3]). 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]

3.2 Lower Barrier for Data Movement

In the ideal Sky Computing vision, 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.

There are signals that such data peering is occurring in clouds. Cloudflare R2 offers storage with zero-egress cost [48]. In early 2024, Google Cloud platform led the move by removing data transfer fees when users are moving off the cloud [46]. Later, Amazon and Azure followed by announcing zero-egress when leaving their clouds [59, 58].

3.3 Intercloud Brokers for Batch Jobs

Given this increasing level of limited interface compatibility, we start the journey towards Sky Computing with the intercloud brokers designed specifically for *computational batch jobs*. 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 [113] and are responsible for the recent surge of specialized hardware [47, 24, 37]. Also, computational batch jobs are normally based on a reasonable amount of data, so moving data is affordable compared to the spending on compute resources, which makes the design of the intercloud broker easier. 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.

[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."

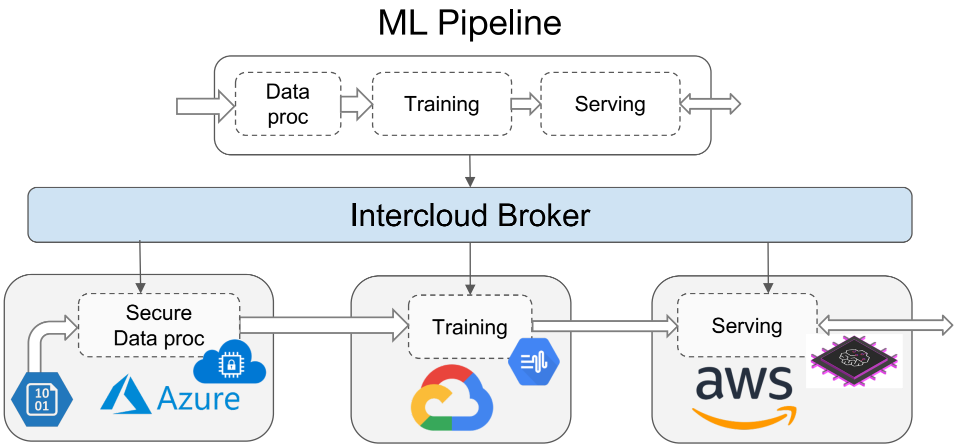


Figure 3.1: **An ML pipeline running on top of Sky.** The goal is to minimize cost while processing the input data securely.

An intercloud broker takes as input a computational request that 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.

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 3.1 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 [27] to anonymize data and thus protect data confidentiality, training on GCP to take advantage of TPUs [47], and serving on AWS to take advantage of the Inferentia accelerator [24].

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 Section 2.5.3 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.

In the rest of this section, we first review the requirements of an intercloud broker for computational batch jobs, see Section 3.3.1, and propose an architecture in Section 3.3.2.

[2] This is informed by workflow systems [13] that are now the de facto standard for orchestrating complex batch applications.

[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 [9], HDInsight [28]) and hosted Kubernetes (e.g., EKS [8], GKE [68], or AKS [29]).

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

Table 3.1: **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.

3.3.1 Requirements

Cataloging services and instances. There is a huge and growing number of services, instances, and locations[4] across clouds. As shown in Table 3.1, 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.

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 [110]. The variations are especially acute for *scarce resources* (Section 4.2.4), 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 Section 4.2.2 for more detail). When a VM is preempted, it waits for another spot VM to become available. Figure 3.2 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

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

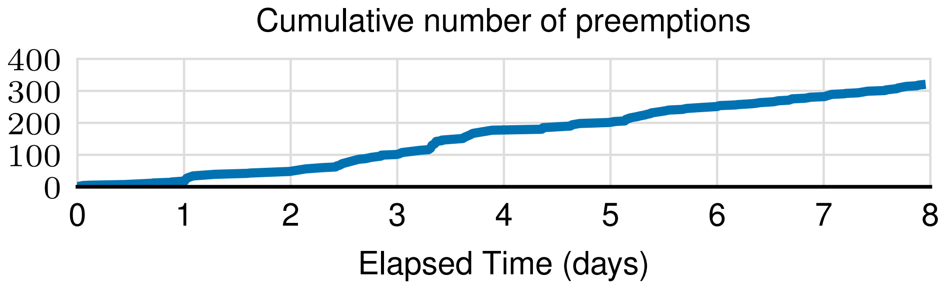


Figure 3.2: **Dynamic resource unavailability:** preemptions over time from a real-world bioinformatics workload trace. The workload ran for 8 days, using 24 large-CPU spot VMs on GCP, us-west1.

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 3.1), (2) DAG topologies becoming complex (Figure 4.8), and (3) the unpredictable resource availability and price changes during the application's provisioning or run time (Figure 3.2).

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.

3.3.2 Architecture

Given these requirements, we propose an intercloud broker architecture consisting of the following components (Figure 3.3).

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.

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.

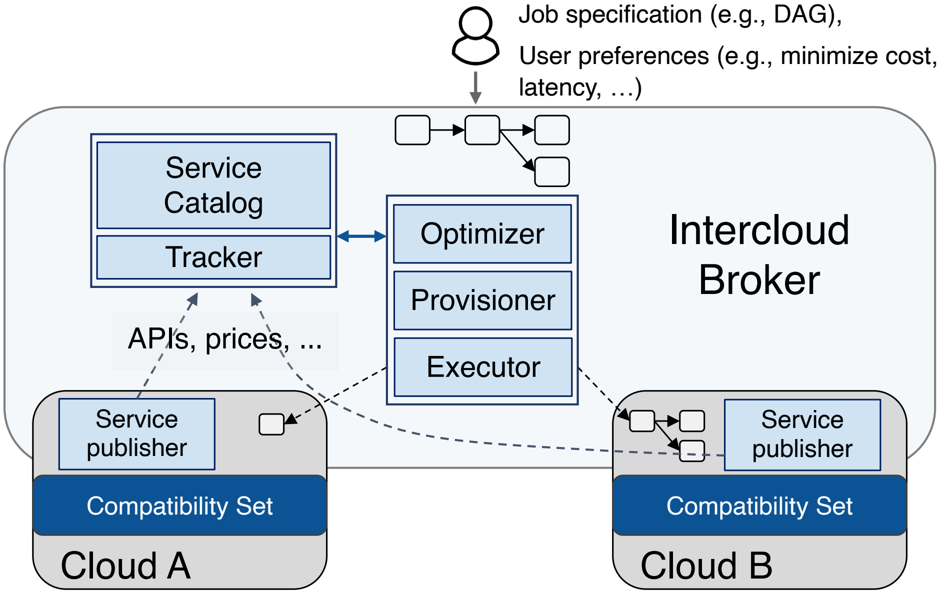


Figure 3.3: Architecture of the intercloud broker.

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 (Section 3.3.1) 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 (Section 3.3.1) 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.

3.4 Related Work

3.4.1 Cross-Cloud Compute, Storage, and Egress

Supercloud [80] 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 as well. CosTLO [159] and SPANStore [158] 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 [60]. 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 [31] is one such effort, consisting of several cloud providers who agree to reduce or even eliminate egress fees from their clouds to Cloudflare or other members. Closely related is Cloudflare R2 [48], an object store that promises to charge zero egress fees. Recently, the clouds with the majority of market share joined the efforts of reducing egress costs. Google Cloud platform led the move by removing data transfer fees when users are moving off the cloud [46], with Amazon and Azure followed with a zero-egress when leaving their clouds [59, 58]. Naturally, Sky Computing benefits from these efforts to combat data gravity, and the intercloud broker can be extended to support zero-egress storage systems.

3.4.2 Middleware

Middleware solutions (e.g., CORBA [49], Microsoft BizTalk [100], IBM WebSphere [74], 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.

3.4.3 Integration Platform-as-a-Service (iPaaS)

Like the middleware systems discussed above, iPaaS solutions [127, 106] 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.

3.4.4 Management Frameworks

In addition, there are several programming or management frameworks that support multiple clouds. JClouds [15] and Libcloud [17] 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 [136] provisions and manages resources on different clouds, but requires the usage of provider-specific APIs, and also does not handle job placement. Kubernetes [83] orchestrates containerized workloads and can be run across multiple clouds (e.g., Anthos [12]). These frameworks, while quite valuable, focus on providing more compatibility in the lower-level infrastructure interfaces offered by the clouds (see Section 3.1), and as such are nicely complementary with Sky Computing but do not obviate the need for Sky Computing.

Chapter 4

SkyPilot: an Intercloud Broker Implementation

In pursuit of the Sky Computing paradigm outlined in Chapter 2, we developed SkyPilot, an intercloud broker tailored for computational batch jobs. This tool adheres to the architecture we proposed in Chapter 3 and has been made available as an open-source project in [130]. A notable deviation from the proposed architecture is the decentralized implementation of the tracking component. Instead of a centralized tracker, SkyPilot utilizes a catalog that updates pricing daily and a provisioner that monitors and caches provisioning failures.

SkyPilot was initially written in approximately $\approx$21,000 lines of Python code, and involved several person-years of development. It initially supported major cloud platforms such as AWS, Azure, and GCP, and was utilized by users from three universities and four other organizations. As the project evolved and the development community grew, new features were added, expanding SkyPilot to over 89,000 lines of Python code and extending support to more than 12 cloud platforms, and 2 on-premise orchestration system VMWare vSphere and Kubernetes.

This chapter outlines the initial implementation of SkyPilot, focusing on its core functionality for computational batch jobs to illustrate the system's overall design (see Section 4.1). We also present experiments conducted with SkyPilot to demonstrate the tangible benefits this intercloud broker provides to real-world applications, along with microbenchmarks of the system (see Section 4.2). Lastly, we discuss the most recent deployment experiences and explore the future research and practical applications of intercloud brokers (see Section 4.3).

4.1 Implementation

In this section, we describe the implementation for the early version of SkyPilot.

4.1.1 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 [128] 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 [150], Mesos [72], and Condor [137]. 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 [4, 149].

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.[1]

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), or use a more accurate benchmarking-based predictor.

The `infer` task performs model serving by loading the model checkpoints from the previous. It takes the trained model as input, *i.e.,* `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 for further instance provisioning and execution of the job.

Figure 4.1a visualizes the computational DAG. Note that 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.

[1] Prior work [146] have considered performance prediction for analytics [149] and machine learning [124] workloads, which can also be leveraged.

```
# 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.

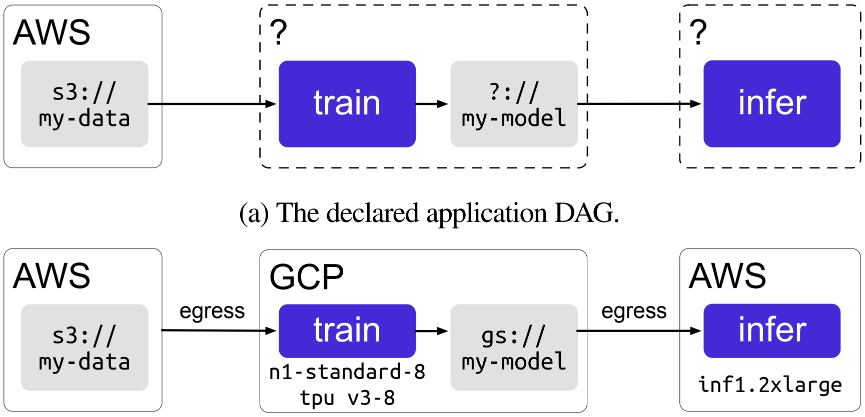
Field	Type	Description
InstanceType	string	The type of instance offering.
vCPUs	float	The number of virtual CPUs.
MemoryGiB	float	The amount of memory in GiB.
AcceleratorName	string	The name of accelerators.
AcceleratorCount	float	The number of accelerators.
Region	string	The region of the resource.
AvailabilityZone	string	The availability zone of the resource (empty if not supported).
Price	float	The price of the resource.
SpotPrice	float	The spot price of the resource.

Table 4.1: Catalog schema for IaaS in a single cloud.

4.1.2 Catalog and Tracker

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. In Table 4.1, we show an example of the catalog for the IaaS service in a single cloud with all information required by the optimizer below.

To keep tracking of the offerings and price changes, we host a broker catalog service, which contains



(a) The declared application DAG.

(b) DAG, after optimization.

Figure 4.1: An application, before and after optimization.

the offerings from multiple clouds and refreshes them periodically through clouds' public APIs mentioned above. A user of SkyPilot will fetch the latest catalog from the catalog service. The catalog service can be found at: `https://github.com/skypilot-org/skypilot-catalog`.

In the future, cloud providers may publish such information themselves and brokers may verify this information.

4.1.3 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.

Filtering. 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[2]. We call such a configuration a *cluster*. For example, when a user specifies a resource requirement: `Resources(accelerator='nvidia-v100')`, it 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.

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

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[3], 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 [170], 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:

$$\min_{s} \underbrace{\sum_{v \in V} s_v^T c_v}_{\text{computation cost}} + \underbrace{\sum_{(u,v) \in E} s_u^T Q_{uv} s_v}_{\text{data transfer cost}} \tag{4.1}$$

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 [56] the second term, we get a 0-1 ILP, which SkyPilot solves using an off-the-shelf solver, CBC [55].

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

$$\min_{s} f_{\text{sink}} \tag{4.2}$$

$$\text{s.t.} \quad f_v \geq \underbrace{f_u}_{\substack{\text{parent}\\ \text{finish time}}} + \underbrace{s_u^T P_{uv} s_v}_{\substack{\text{data transfer}\\ \text{time}}} + \underbrace{s_v^T t_v}_{\substack{\text{computation}\\ \text{time}}} \quad \forall (u,v) \in E \tag{4.3}$$

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.[4] 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

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

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

budget (or vice versa), by augmenting Equation 4.1 with the constraint in Equation 4.3 and bounding f_{sink} by the time budget. Future work can incorporate carbon footprint of cloud regions [35] into placement decisions.

4.1.4 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.

4.1.5 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 [105], 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 [83]. 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).

4.1.6 Compatibility Set

One of the distinguishing features of Sky is leveraging the already existing services and APIs across clouds (i.e., compatibility set; Section 3.1), 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".

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

Table 4.2: Evaluated workloads, cloud services used, and benefits.

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 [136].) 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[66, 126, 32] 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.

4.2 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 (Section 4.2.2).

Table 4.2 shows all workload types and their respective benefits.

4.2.1 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.

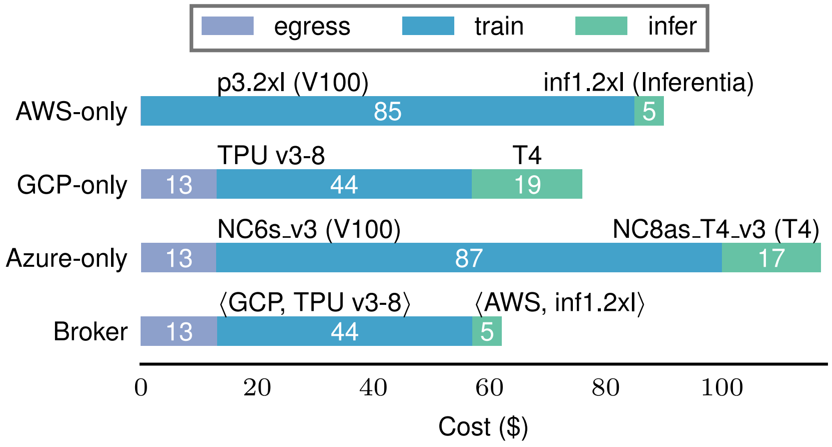


Figure 4.2: **Vision pipeline:** hardware and costs of each deployment. For simplicity, the zones chosen for the plans are omitted. For training we use mixed-precision and the XLA compiler [135] with TensorFlow Keras 2.5.0. For inference we use half-precision. On GCP, accelerators are attached to an n1-standard-8 VM.

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 (Section 4.2.1.3) an additional experiment on SkyPilot leveraging spot instances across clouds to run ML training with improved availability and cost.

4.2.1.1 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 [47].

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 [24].

The best single-cloud plans are shown in Figure 4.2, 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 required to train the model by the hardware FLOPS: [5]

```
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 4.1b and the results in Figure 4.2.

While this pipeline is simple, *its search space is already large*, with a total of 2,170 possible assignments (details in Section 4.2.4), 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.[6] SkyPilot's storage module uses GCP's storage transfer service [67] 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).

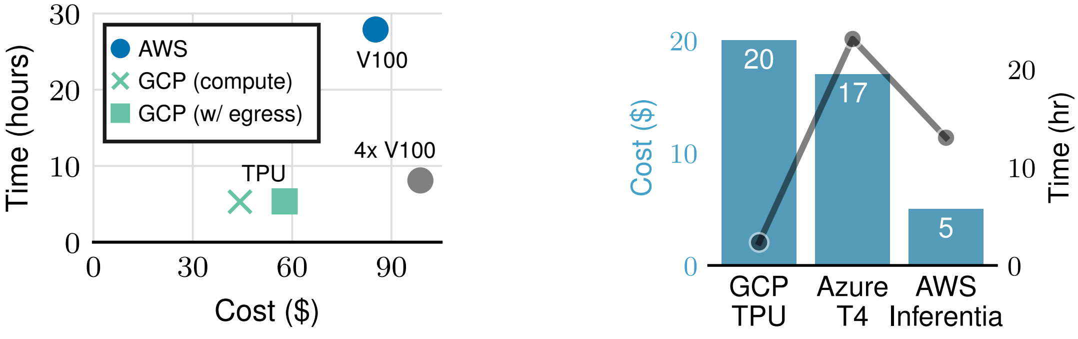
To understand the cost savings, we compare the detailed time and cost per task. For training (Figure 4.3a), SkyPilot's choice of GCP TPU takes 5.4 hours and costs $57 with egress included, which is 5.2× faster and 33% cheaper than the AWS V100 plan. (Azure V100 is similar but has $13 for egress; hence omitted.) To make the hardware more comparable, we 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 4.3b), 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 (Section 4.1), we resubmit this pipeline to SkyPilot with the time-minimizing objective. The resource

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

[6] 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 4.3: **Vision pipeline:** detailed breakdown per task.

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 4.3b. 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.

4.2.1.2 NLP Pipeline

We next run a natural language processing (NLP) pipeline that emulates an increasingly prevalent workload: fine-tuning "foundation models" [33]. It consists of three tasks (Figure 3.1):

- **Confidential data processing**: remove sensitive information from raw data using Intel SGX hardware enclaves. We use the Amazon Customer Reviews Dataset [6] and treat it as if it contained personally identifiable information (PII) and thus must be processed securely. To remove sensitive data, we run Opaque [171] 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 [53], 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 the resource: `Resources(intel_sgx=True)`, which is currently only offered by Azure [27]. For training, we consider either 4 V100s, or a TPU v3-8. For serving, we consider either a T4 GPU, or AWS's Inferentia.

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

Due to the confidential computing requirement, the only possible single-cloud plan is to run all three tasks on Azure: 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 4.3 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* (Section 4.2.4).

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.

4.2.1.3 ML Training on Spot Instances across Clouds

We evaluated SkyPilot's benefits for ML pipelines; we now 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`[7].
- *Broker*: runs on a spot instance, with SkyPilot having the freedom to choose among all US regions of AWS or GCP.

[7] 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.

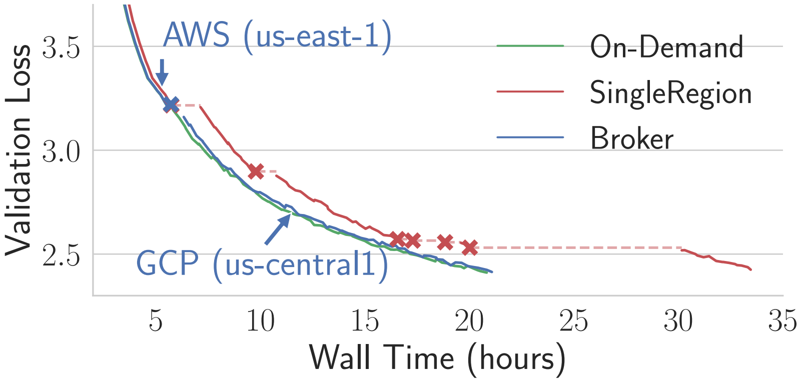


Figure 4.4: **Loss curves of training BERT with broker.** The training is on V100 with 30 epochs. Each `x` marker is a preemption event; gaps between segments are the time periods when spot instances are not available. After the first preemption event, Broker migrates the job from AWS `us-east-1` to 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 4.4: **Costs and makespan for 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 4.4 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 4.4 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

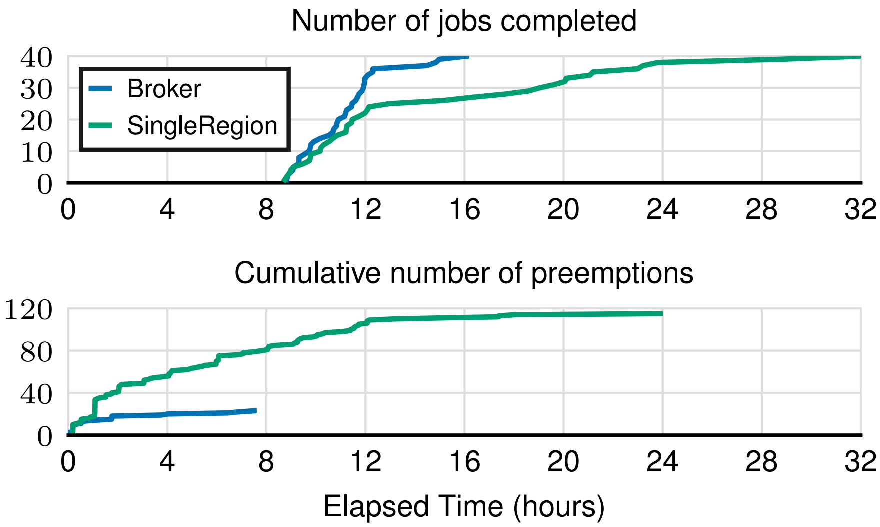


Figure 4.5: **Dynamically adjusting to availability** on a bioinformatics workload of 40 jobs on spot CPU VMs. *Broker* moves preempted jobs to a new region, while *SingleRegion* moves preempted jobs to other zones in the same region. Note the shared x-axis. Cloud: GCP.

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.

4.2.2 Bioinformatics

The intercloud broker should *dynamically* respond to the changing availability of resources (Section 3.3.1). 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 [93, 164, 91]. 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 [75]; (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 4.5 (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 4.5 (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 [132] or managed instances [75], 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.

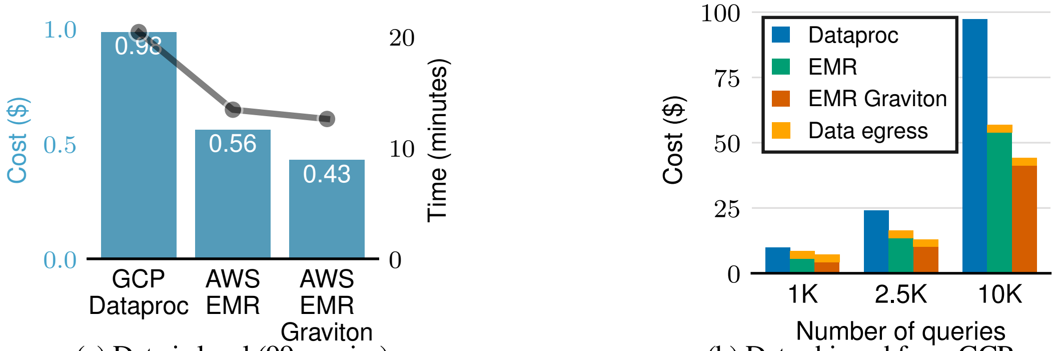
4.2.3 Managed Data Analytics

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 [9] and GCP Dataproc [63]. 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 [108] 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* [115] for 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 [23]. 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 [21].

Figure 4.6a 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 [115] 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.



(a) Data is local (99 queries)

(b) Data shipped from GCP

Figure 4.6: **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.

To understand the tradeoff between better services vs. data gravity, Figure 4.6b 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 [18, 52] and recorded in SkyPilot's service catalog.

4.2.4 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 4.5 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 (Section 4.2.2) and analytics (Section 4.2.3), can suffer from up to 2.5× price premiums if run in the

			On-demand $		**Spot $**	
Type	**Hardware**	**Zones**	Max/Min	CV	Max/Min	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 4.5: **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.

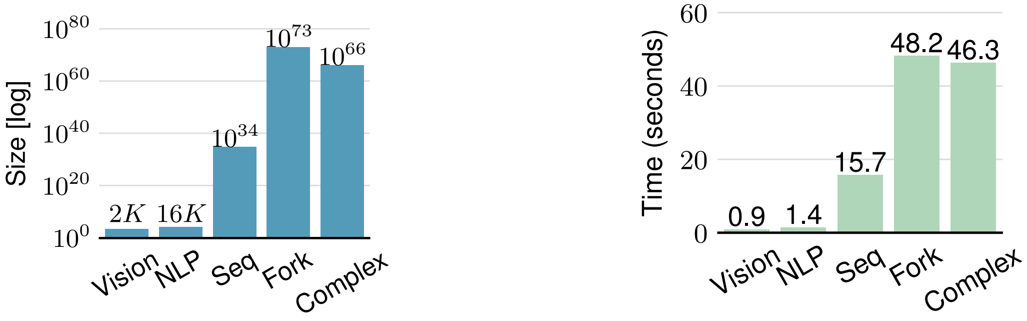
most expensive zone, which increase to 9.4× if spot instances are used. These differences are even larger for NVIDIA GPUs, which are present in just 16–50% of all zones, and their prices vary by up to 9.5× for on-demand and 10.8× for spot. Finally, despite TPUs being offered only in 4–5 (or 5%) GCP zones, 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 4.7 shows the search space sizes and the optimization time for the two ML pipelines in Section 4.2.1 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 [13]: the first two (Figure 4.8a, Figure 4.8b) are commonly used in the real world [96], while the third (Figure 4.8c) has a more complex structure.

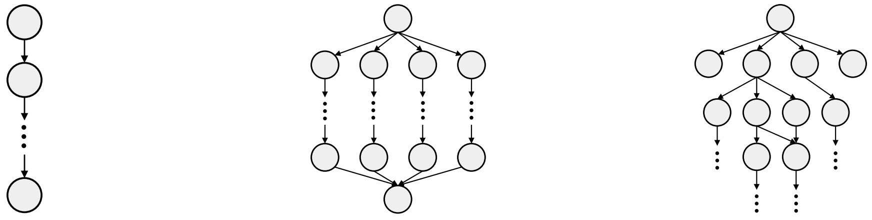
We assume each task requires an 8-vCPU Intel VM in US zones, which leads to 55 feasible clusters for each task. We assign random time estimates (sampled from $U(0,1)$ hours) to each task and a random data transfer size (sampled from $U(0,100)$ GB) to each edge. While the search spaces for the DAGs are 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.



(a) Search space sizes

(b) Optimization time

Figure 4.7: **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 4.8: **Larger DAGs found in Airflow's repository. (a)** Sequential: $|V| = 20, |E| = 19$. **(b)** Fork-Join: $|V|=42,|E|=44$. **(c)** Complex: $|V|=38,|E|=53$.

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.

4.3 Deployment Experience

When SkyPilot is firstly open-sourced, it supported three major clouds, including AWS, Azure and Google Cloud. It was deployed to dozons 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. Later on, the user base and community of open-source SkyPilot project [130] grew significantly with more than tens of organizations adopted and more than ten clouds supported.

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 and learnging with the system from the users' feedback.

4.3.1 Signals for the Two-sided Market

As more users adopted SkyPilot, we have seen a strong incentive from the cloud providers to join the Sky. Although we only implemented the support for the three clouds with the most market share initially, communities and cloud providers are excited to contribute and add the support for their own clouds, leading to the support of more than half a dozen of community-contributed clouds in SkyPilot. Later on, we observed that some AI users who had been mainly using a single major cloud with SkyPilot, started running their workloads on those smaller community-contributed clouds, due to the lower cost and better availability for the same GPU accelerators on those clouds. This shows some early signal for the two-sided market (described in Section 2.4, where smaller clouds can join the competition more easily as the Sky lower the barrier for the user to migrate workloads from a cloud to another.

4.3.2 Benefits of an Intercloud Broker for Users

By surveying our users, we found that users value the broker not only for cost reduction, but also for improved availability (see Section 4.2.2) and in general for *improving their productivity*. For example, users like the broker's ability to automatically find availability and provision scarce resources across clouds or regions, especially during the era when GPU shortage [139] is a wildly acknowledged issue for organizations willing to develop, train or serve AI models. The easy access to best-of-breed hardware (e.g., TPUs) and the fault tolerance the broker can provide for potential hardware failures is another reason people adopt SkyPilot. Moreover, by interacting with the broker rather than the clouds, they value the ability to simply package existing programs with the unified interface, run the same jobs on different clouds with no change to their code or workflow.

4.3.3 Leveraging Spot Offerings for Cost Savings

With AI being one of the core use case for SkyPilot, the ability of using spot instances for the expansive high-end GPUs became one of the most important feature SkyPilot provides to users. As a broker system, SkyPilot not only start a user's job but also keep monitoring and recovering it from any spot preemption triggered by the cloud provider. This feature could save a user more than 3× cost depending on the type of the GPUs used. During the deployment, we have observed many open questions to be solved when leveraging spot offerings, including how to handle a deadline-sensitive job that cannot tolerate too many preemptions (see Chapter 5), how to select the next region or cloud to go to when a preemption happens with both cost and reliability taken into consideration, etc. There remains a big room for the exploration of broker policies.

4.3.4 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.

4.3.5 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 4.2; Figure 4.6).

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 [118]. Our results (Section 4.2.1.1) suggest that a 1 TB dataset can likely be moved in $\sim$20 minutes with a cost of $\sim$\$90. Depending on the job, this delay and cost can be easily offset by the destination offering better hardware, software, or pricing.

4.3.6 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. The latest version of SkyPilot (v0.5.0) supports Kubernetes and vSphere, which are two popular management tools for on-premise clusters, and people found it useful to interact with the same SkyPilot interface while still utilize their on-premise resources along with clouds. Challenges include designing spillover policies and handling compatibility and storage.

4.4 Conclusion

This chapter 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. With the early deployment of SkyPilot, we have gathered interesting evidence and momentum for building the "Sky of Compute". 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.

Chapter 5

Can't Be Late: a Broker Policy with Spot

In Chapter 4, we initiated our exploration into Sky Computing by implementing an intercloud broker system for computational batch jobs, which we named SkyPilot. This system has not only proven beneficial in practice but also enriches the scheduling landscape by providing a broader action space with the support of workload migration. With jobs being managed by brokers, online policies need to be designed for brokers to take advantage of the offerings across clouds that can be dynamic during job execution.

SkyPilot incorporates a managed spot feature, enabling jobs to operate on less reliable spot instances by automatically detecting and recovering from spot preemptions. A naive online policy is adopted for handling preemptions, which simply waits until a new spot instance becomes available. This approach is cost-effective for long-running jobs, saving more than 3× of cost; however, it becomes problematic for deadline-sensitive applications due to the absence of availability guarantees. A more clever policy is required for brokers to enable the use of spot instances in deadline-sensitive applications, achieving significant cost savings while still meeting deadlines.

To allow jobs to meet deadlines while leveraging spot instances, we propose a simple idea: use on-demand instances judiciously as a backup resource, *i.e.,* having a broker unifies the offerings of both spot and on-demand instances and strategically utilize them. However, due to the unpredictable spot instance availability, determining when to switch between spot and on-demand to minimize cost requires careful policy design. In this chapter, we first provide an in-depth characterization of spot instances (e.g., availability, pricing, duration), and develop a basic theoretical model to examine the worst and average-case behaviors of baseline policies (e.g., greedy). The model serves as a foundation to motivate our design of a simple and effective policy, Uniform Progress, which is parameter-free and requires no assumptions on spot availability. Our empirical study, based on three-month-long real-spot availability traces on AWS, demonstrates that it can (1) outperform the greedy policy by closing the gap to the optimal policy by 2× in both average and bad cases, and (2) further reduce the gap when limited future knowledge is given. These results hold in a variety of conditions ranging from loose to tight deadlines, low to high spot availability, and on single or multiple instances. By implementing this policy on top of our intercloud broker, SkyPilot in Chapter 4, we achieve 27%-84% cost savings across a variety of representative real-world workloads and deadlines. The spot availability traces are open-sourced for future research.[1]

[1] See spot traces: https://github.com/skypilot-org/spot-traces

	V100 GPU	64-core CPU
AWS	3×	2–6×
Azure	3–6×	3–10×
GCP	3×	4–11×

Table 5.1: Cost savings of spot vs. on-demand instances.

5.1 Introduction

As organizations continue to migrate their workloads to clouds, the need to minimize operational costs has become a critical concern [152]. One of the top contributors to cloud spending is the cost of compute instances [147], which are typically offered in two pricing models: *on-demand* and *spot* [2].

On-demand instances are available but come at a premium cost. In contrast, *spot* instances are typically 3–10× cheaper (Table 5.1), but are less available and they can be preempted unexpectedly. As a result, more applications such as analytics [43], AI [141, 151, 95], HPC [97], and CI/CD workloads [7], are leveraging spot instances to reduce costs. To handle preemptions, these jobs either checkpoint periodically and recover from the last checkpoint on restart [167, 90], or re-execute the entire job.

However, while many applications can tolerate uncertainties introduced by spot instance preemptions, others cannot. One such category is delay-sensitive applications where a job needs to finish by a certain deadline [89]. Examples include processing new user data to keep an AI model up-to-date in a recommendation system, or analyzing the latest information to make timely decisions in a trading application. Therefore, most of deadline-sensitive applications eschew spot instances in favor of on-demand instances, thus trading off cost for predictability.

In this paper, we resolve this tradeoff by enabling an application to leverage spot instances while still meeting its deadline. For simplicity, we focus on recoverable jobs running on a single instance, and assume the running time of the job is known, as well as its deadline. A job can be in one of three states: (1) running on a spot instance, (2) running on an on-demand instance, or (3) idle, i.e., waiting for a spot instance to become available. We design scheduling policies that periodically decide whether a job should remain in the same state or switch to another state. When a job switches to a non-idle state we assume there is a delay, e.g., the overhead of provisioning/setting up a new instance, and re-starting from a previous checkpoint. Due to the high unpredictability in spot instance availability (Section 5.2.2), the key challenge lies in striking a balance between cost optimization and deadline adherence to effectively leverage the low cost of spot instances without missing the deadline.

A simple solution to this problem would be for a job to use a spot instance up to the point at which the remaining computation time equals the remaining time to deadline, and then switch to an on-demand instance until it finishes. While this "greedy" policy (Section 5.3.4) guarantees that the job will meet its deadline, we show that it is sub-optimal. We do so by developing a theoretical framework to study the worst-case behavior (e.g., competitive ratio) of the policy (Section 5.4).

To address the limitations of "greedy" policy, we propose a simple and effective policy, called *Uniform*

[2] In this paper, we do not consider "reserved" instances, whose economics involves volume contracts and is more complex.

Progress, which aims to make uniform progress towards deadline, by distributing the job computation uniformly across the time. Uniform Progress requires no assumption about spot instances' availability and is parameter-free. Using simulations on real-world traces we show that Uniform Progress outperforms greedy policy and approaches an optimal policy with limited knowledge of the future (knowing how long the next spot instance is going to last) in a variety of scenarios—from loose to tight deadlines, and from low to high spot availability. We build a prototype of Uniform Progress and evaluate it in a cloud setting on three real-world workloads: ML training, scientific batch jobs, and data analytics. Results show that Uniform Progress achieves 27–84% cost savings while meeting deadlines.

This paper is organized as follows. First, we provide an in-depth characterization of spot instances across various cloud regions, examining their availability patterns, pricing, and lifetime to inform our policy design (Section 5.2). Next, we develop a theoretical model that captures the essential dynamics of spot instances, which enables us to examine the worst-case behavior of a given policy (Section 5.3, Section 5.4). We then present our policies for jobs with both single and multiple instances (Section 5.5) and conduct a comprehensive empirical study on months-long real-world traces of spot instances (Section 5.6). We build a prototype implementation that supports the proposed policies in a cloud setting, and evaluate these policies on three real-world workloads (Section 5.7). Finally, we review related work in Section 5.8.

In summary, this paper makes the following contributions:

1. We introduce a problem of using spot instances to minimize the cost of running a job with deadline adherence.
2. We develop a theoretical framework to study the worst and average-case behavior of baseline policies, providing insights on the tradeoff between cost and deadline.
3. We propose Uniform Progress, a simple but effective policy which is parameter-free and requires no assumptions on spot availability. Empirically, we show the significant improvement of the policy in a wide variety of scenarios.
4. We implement a prototype system with Uniform Progress, and evaluate it on real-world workloads.

Finally, we open source our three-month traces of spot instance availability to encourage future research in this area.

5.2 Characterization of Spot Instances

In this section, we characterize spot instance availability and pricing over time and across availability zones. We observe high volatility in availability but a smooth pricing pattern. We use these insights to drive the design of our scheduling policy.

5.2.1 Methodology of Spot Trace Collection

We collect *spot availability traces* from public clouds. A trace is a time series showing whether a particular spot instance type is available at a given time in a zone. We collect these traces over a three-month period and in nine AWS availability zones (three in *us-west-2*, four in *us-east-1*, two in *us-east-2*).

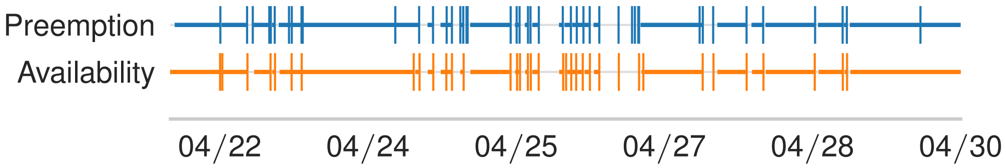


Figure 5.1: **Real spot preemptions and availability are highly correlated.** Trace is in AWS us-west-2b. **Upper**: preemptions. Horizontal lines represent a running spot instance. Vertical bars are preemptions. **Lower**: availability. Horizontal lines are spot instance available periods. Vertical bars are changes from availability to unavailability. Grey gaps are unavailability periods. Note that although some vertical bars look immediately followed by a horizontal line, there are still gaps in between.

A key challenge of trace collection is that it can be prohibitively expensive. For example, a spot V100 instance costs about $1/hour. If we collect a real *preemption* trace where an instance is kept live as much as possible modulo preemptions, collecting three-month long traces in all nine zones could cost over $10,000. Instead, we propose an approximation: we collect *availability* traces, where we try to launch a spot instance every 10 minutes to probe if it is available and then immediately terminate it. To validate this approach, Figure 5.1 shows a high correlation between the real preemption and availability signals over a week-long period. This approach reduces the cost of trace collection by about 100×. For completeness, we also include real preemption traces in our evaluation of policy performance on multi-instance jobs (Section 5.6.6) and real-world workloads (Section 5.7.2).

In this work, we focus on a few *scarce* instance types, i.e., Nvidia V100 and K80 GPU instances, which are now in high demand[111] due to the rise of Generative AI and large language models (LLMs). Focusing on these scarce instance types is thus both useful and interesting, as they are frequently preempted, providing a good testing ground for scheduling policies.

5.2.2 High Variance in Spot Availability

Our analysis reveals that spot availability varies significantly *across zones* and *over time*. Figure 5.2 (left) shows the availability traces of 9 AWS zones over 2 weeks. We observe a large difference across zones (e.g., us-west-2a vs us-east-1a). The periods of unavailability can last for hours or even days.

To understand spot availability distributions, we overlay 6-hour windows on a 2-week period (thus, $14 \times 24/6 = 56$ windows per zone) and count the fraction of availability probes that succeeded in each window. Figure 5.2 (right) plots the distributions of *spot availability fractions* in the 56 windows per zone, which approximate the fraction of time spot instances are available in each zone. We observe that each zone can go from being highly available to mostly unavailable across time (e.g., in us-east-2b, the difference between p25 and p75 is about 70%) and there is little correlation across zones. In addition, Figure 5.3 shows changes in spot availability fractions over time. We observe a highly volatile pattern: availability can change from 100% to 0% within hours.

The results above suggest that scheduling policies should be robust to highly unpredictable availability patterns. For generality, in this paper, we make no assumptions on spot availability patterns. We discuss existing prediction-based approaches in Section 5.8 and leave this direction to future work.

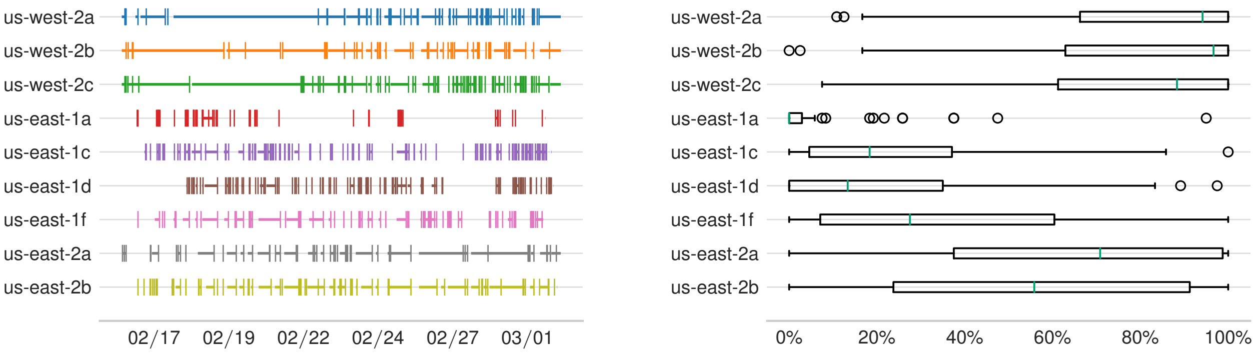


Figure 5.2: **Spot Availability is highly unpredictable and volatile.** Traces are across nine AWS zones collected. **Left**: Availability. Horizontal lines are available periods. Vertical bars are changes from available to unavailable, followed by grey gaps indicating unavailable period. **Right**: Boxplots of spot availability fraction, *i.e.,* percentage of the time an instance is available in 6-hour windows.

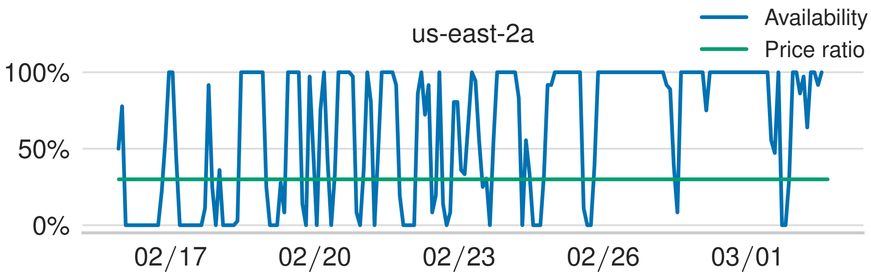


Figure 5.3: **High volatility of spot availability fraction.** Availability can jump from 100% to 0% within hours. Price ratio: spot price divided by on-demand price.

5.2.3 Relative Stability in Spot Pricing

In contrast, we observe that spot pricing is much more stable than availability. Figure 5.3 shows the price ratio of spot to on-demand for AWS stays almost constant despite significant changes in availability. In the three-month-long trace, we observe only a 5% price variation on average over any one-week period, validating the recently introduced smooth pricing model on AWS [112]. GCP's spot instance prices are even more stable as it is guaranteed to only change once every 30 days [64].

5.2.4 Correlation of Multi-Instance Availability

To understand the behavior of multiple spot instances, we analyze 2-week *preemption* traces and 2-week *availability* traces for clusters of 4 and 16 instances, respectively (see Section 5.6.1 for details). Notably, over 94% of the time, either all or none of the instances are available in each cluster. This suggests availability tends to change *simultaneously* for multiple instances (bulk preemption is also observed in [70]), up to a count of 16.

5.3 Using Spot for Deadline-Sensitive Jobs

In this section, we present a simple model to formulate the problem, discuss when a policy matters, and then give three rules for policy design followed by a basic greedy policy.

5.3.1 Problem Setup

We consider two types of instances with the same hardware: an on-demand instance, which is always available[3], and a spot instance, whose availability is unpredictable. We assume that spot availability is non-adversarial, meaning that it is independent of the job's choices and observable factors, except for Section 5.4.1, where we adopt competitive analysis for the worst case study.

We focus on long-running (hours to days) jobs where preemptions are likely. We firstly assume each job uses one instance. We will extend it to multiple instances in Section 5.5.5 and evaluate it in Section 5.6.6.

For a deadline-sensitive job, we denote *remaining computation time* at time t as $C(t)$ and *remaining time-to-deadline* as $R(t)$. This implies that the job's total computation time is $C(0)$, and deadline is $R(0)$. Based on the definition, we can derive that $R(t) = R(0) - t$ and when a job is progressing, $\partial C(t)/\partial t = -1$.

We assume that both $C(0)$ and $R(0)$ are given and the job is fault-tolerant to interruptions. For example, ML training typically has a consistent per-epoch time, indicating a predictable total runtime, and the model weights can be checkpointed and resumed for fault tolerance. Additionally, computation times for many recurring jobs (e.g., data analytics, scientific HPC, CI/CD) can be derived from past executions.

To account for overheads of starting the job on a new instance, we introduce *changeover delay*, d, which includes the time required to launch an instance, set up dependencies, and recover any potential progress loss caused by gaps between checkpoints or restarting the most recent unsaved execution. Whenever a job switches to a new spot or on-demand instance, a changeover delay occurs, meaning that $C(t)$ does not decrease for a duration of d while $R(t)$ continues to decrement. A delay d is charged at the new instance type's price. Switching from an instance to idle (i.e., termination) does not incur a delay. We will extend the model to consider variety with $C(0)$ and d in Section 5.5.6, and evaluate it in Section 5.6.7.

The goal is to minimize the cost for completing job's computation time $C(0)$ before deadline $R(0)$, *i.e.,* $C(R(0)) \leq 0$, using spot and on-demand instances. For simplicity, we define the price for an on-demand instance to be $k > 1$, and a spot instance to be 1. We assume that cloud providers charge every second when an instance is alive.[4] Based on the observation in Section 5.2.3, we assume both the on-demand and spot price are fixed throughout the time before deadline $R(0)$.

5.3.2 Scheduling Policy

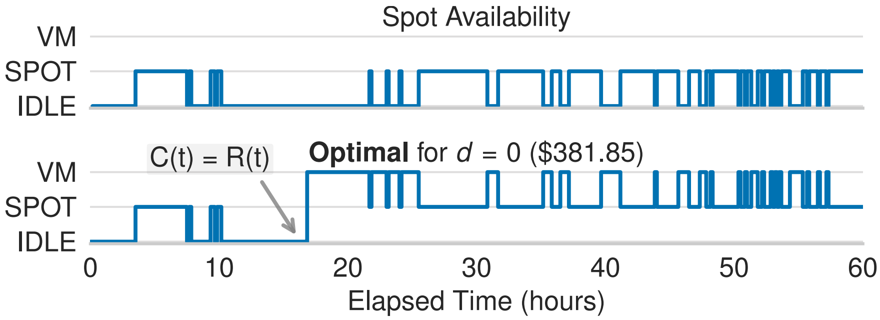
At any time t, a job can be in one of the following three states: idle, running on a spot instance, or running on an on-demand instance. While we assume that on-demand instances are always available, spot instances can be in one of two states: available or unavailable. The job's state space is the combination of any of the

[3] This is a simplifying assumption. In practice, some on-demand instance types can hit unavailability.

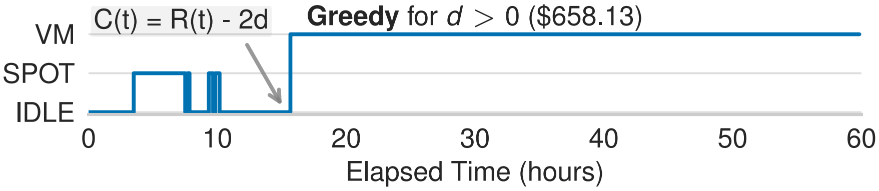
[4] Cloud providers have different billing practices, *e.g.,* AWS does not charge for spot instances preempted within the first hour, while GCP does.

Spot State \ Instance State	Idle	Spot	On-Demand
Spot Available	①	③	④
Spot Unavailable	②	-	⑤

Table 5.2: State space for a job.



(a) Without changeover delay.



(b) With changeover delay.

Figure 5.4: Example decision traces of policies on real spot availability on AWS.

instance state and the spot state, except for an impossible case where the instance state is spot with spot state unavailable (Table 5.2). A scheduling policy is invoked to decide how a job moves across instance states.

In the ideal case where changeover delay $d = 0$, the problem is simple. An optimal policy is to use a spot instance whenever it is available, *i.e.,* transition between state ② and ③, until $C(t) = R(t)$. After that, the job cannot stay idle, as it needs to utilize all the remaining time before deadline to make progress. Since there is no changeover delay, the policy can use spot whenever it is available and switch to on-demand when it is not, *i.e.,* transition between ③ and ⑤. This policy is optimal because it utilizes all available spot instance lifetimes before the deadline, without additional cost. Figure 5.4a shows an example decision trace of how this policy performs for a job with $C(0) = 48$ hours and $R(0) = 60$ hours on a real spot availability trace, where the policy utilizes every spot lifetime, and runs the remaining computation with on-demand instances.

However, when changeover delay $d > 0$, which is the practical case, the problem becomes non-trivial. The policy now has to decide whether it is worth switching to a different instance at the expense of losing time d without making progress, which increases the risk of missing the deadline. For example, applying the optimal policy above for $d > 0$ would result in missing the deadline, since every switch costs an additional time d.

In the remainder of this paper, we focus on designing policies for the more practical $d > 0$ scenario.

5.3.3 Rules for Policy Design

Based on the problem setting, we propose three basic rules that all policies without future knowledge should follow to avoid unnecessary cost or missing the deadline.

Thrifty Rule. The job should remain idle after the job complete, $C(t) = 0$.

Safety Net Rule. When a job is idle and $R(t) < C(t) + 2d$, switch to on-demand and stay on it until the job complete.

The policy is required to guarantee the job finished by the deadline. After $R(t) < C(t) + 2d$ becomes true, it is no longer safe to move from idle to spot. Otherwise, when the changeover delay of the spot finishes, the remaining time will become $R(t) < C(t) + d$, which means any preemption to the spot instance will result in missing deadline. Note that one could wait until $R(t) = C(t) + d$ then move to on-demand, but there is no gain for waiting an additional d if the job is idle.

Exploitation Rule. Once start using a spot instance, stay on it until it is preempted.

If the job is on a spot instance, any progress made will always cost the minimum price any policy could get, *i.e.,* the spot price. Voluntarily switching from spot to idle or on-demand will have no benefit, but less progress or more cost.

This rule will not violate the deadline because the Safety Net Rule guarantees that $R(t) \geq C(t) + 2d$ holds at the time t when the job is moved to the current spot instance. After the changeover delay is incurred and the job starts progressing, $R(t) - C(t)$ will not change, *i.e.,* $R(t) \geq C(t) + d$ holds, meaning the remaining time is enough for at least one changeover even if the current spot is preempted. The job will be able to switch to on-demand when Safety Net Rule kicks in.

5.3.4 Greedy Policy

Based on the three rules, we propose a straightforward greedy policy. The greedy policy behaves as follows:

1. Stay on any available spot instance until it is preempted (Exploitation Rule), and keep waiting if no spot instance is available, *i.e.,* transition between ② and ③ in Table 5.2.

2. (Safety Net Rule) When $R(t) < C(t) + 2d$ holds and the job is idle, move to on-demand and stay there until the end.

In Figure 5.4b, we show the decision trace of the greedy policy on the same spot availability trace as before (Figure 5.4a). The greedy policy acts much more conservatively than the previous optimal policy without changeover delay. That is because greedy can no longer afford frequent switches between

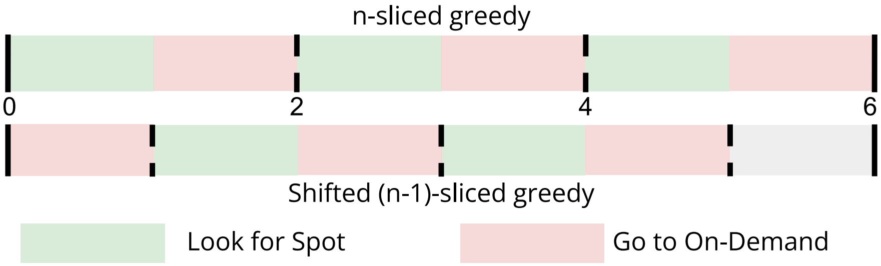


Figure 5.5: **Example slicing for randomized shifted greedy (RSF) policy.** The deadline $R(0) = 6$, computation time $C(0) = 3$, and slices $n = 3$. Dashed lines indicate boundaries of slices.

on-demand and spot instances as before without missing the deadline. Thus, we now turn our attention to: can we do better than greedy while not assuming future knowledge?

5.4 Theoretical Analysis

In this section, we delve into theoretical aspects of the problem and prove the existence of a policy that is better than greedy in both worst and average cases.

5.4.1 Worst Case with Competitive Analysis

We first look into the worst case by investigating the competitive ratio c of a policy without knowledge of future spot availability, which is the ratio of the cost of the policy to the best omniscient policy with full knowledge of future spot availability. By "worst" case, we assume that spot instances are chosen by an oblivious adversary, who can base their decisions on complete knowledge of the job's policy but not on random coin flips used by the policy. Our goal is to prove that there is a policy with lower competitive ratio c than greedy, *i.e.,* performs better in the worst case.

To simplify the presentation, we assume changeover delay d is small and ignore the term $O(d)$. Also, we use $R(t) = C(t) + d$ as Safety Net Rule's condition, instead of $R(t) = C(t) + 2d$, which will not affect the conclusion, due to negligible $O(d)$.

A natural bound for c is $1 \leq c \leq k$, where k can be reached when the oblivious adversary choose a case that a given policy have to use all on-demand, and the omniscient policy could use all spot instances. We can prove that for any $R(0)$, $C(0)$, a deterministic policy cannot perform better than greedy.

Theorem 1. *For any deterministic policy P, $c \geq k - O(d)$.*

Proof. Since the policy P is deterministic, the adversary can choose spot availability as follows. It makes the spot available only when P starts using on-demand or $R(t) = C(t) + d$. If P switches to the spot, the adversary waits for d units of time, then preempts the spot, so P makes no progress on any spot instances, *i.e.,* must use at least $C(0)$ units of on-demand.

With that adversary, we examine P have to use all on-demand while omniscient policy can finish the job with all spots. Consider the first time t', where $R(t') = C(t') + d$. Over $t' \leq t \leq R(0)$, P cannot switch to spot, but the omniscient policy could as it knows the spot will remain available, *i.e.,* P makes $C(t')$ progress on on-demand, while omniscient is on spot. Next, P must have accumulated the $C(0)-C(t')$ before t'. Due to the adversary, any work accumulated before t' should be on on-demand when a spot is available. Thus, the omniscient policy can make $C(0)-C(t')$ of progress on those spots before t'. □

With Theorem 1, we can conclude that a policy has to be randomized to beat greedy, whose competitive ratio $c=k$, as an adversary can simply make spot available from t', where $R(t')=C(t')+d$. We now construct a better policy on top of greedy. We first extend greedy to an *n-sliced greedy* policy, in which we divide the time into n even slices with length $\frac{R(0)}{n}$ and apply greedy in each of these slices with $\frac{C(0)}{n}$ progress to make. The upper figure in Figure 5.5 is an example of n-sliced greedy, with a deadline $R(0)=6$ and 3 slices. In each slice, the policy enforces the job to make $\geq \frac{C(0)}{n}=1$ units of progress within $\frac{R(0)}{n}=2$.

We then shift the n-sliced greedy policy by $\frac{C(0)}{n}$ to get *shifted* $(n-1)$*-sliced greedy* policy, which uses on-demand for time $\frac{C(0)}{n}$ from start (1 in the example Figure 5.5) and then applies $(n-1)$-sliced greedy from $t=\frac{C(0)}{n}$ until $t=R(0)-\frac{R(0)-C(0)}{n}$.

Although both policies have $c=k$, we can define a *randomized shifted greedy (RSF)* policy by using either the n-sliced or the shifted $(n-1)$-sliced greedy with equal probability at any time t. We can prove that the competitive ratio for RSF is bounded and lower than greedy.

Theorem 2. *If* $R(0)\geq 2C(0)$*, then for RSF policy has* $c\leq\frac{k+1}{2}+\frac{k-1}{2n}+O(d)<k$.

Proof. By ignoring the terms of $O(d)$, at any time t before the last split, at least one of the policies is looking to use a spot (as shown in Figure 5.5), so any available spot is used for at least half of the time. Thus, except for the last $C(0)/n$ progress, at least half of the remaining progress is done on spot instance:

$$c\leq(\frac{1}{n}+\frac{1-1/n}{2})k+\frac{1-1/n}{2}+O(d)=\frac{k+1}{2}+\frac{k-1}{2n}+O(d)<k$$ □

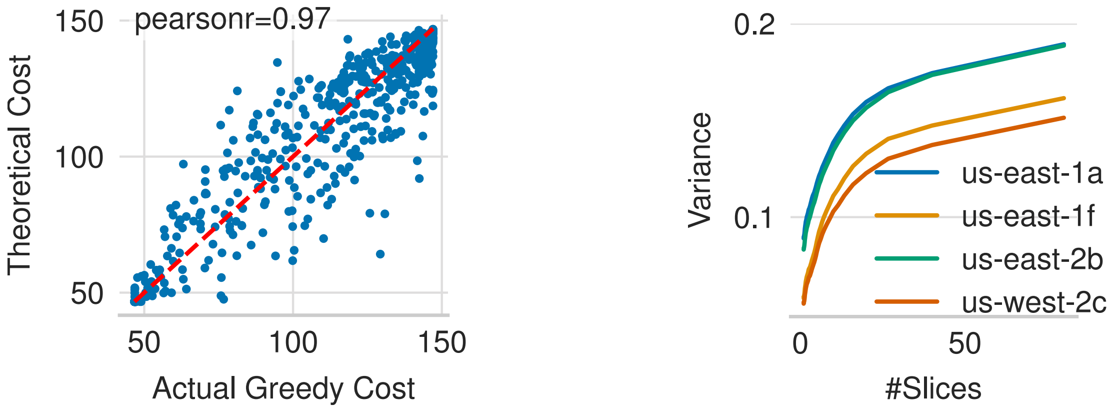
When deadline $R(0)$ is more than 2× longer than computation time $C(0)$, the worst case (the largest gap to omniscient policy) for RSF policy is bounded, *i.e.,* provably better than greedy.

For $R(0)\leq 2C(0)$, we can simply use on-demand until $R(t)=2C(t)$ then start using RSF policy. We denote this *modified RSF (MRSF)* policy.

Corollary 1. *Let* $a=\frac{R(0)}{C(0)}-1$ *for* $0<a\leq 1$*. MRSF policy has:*

$$c\leq k-ak+a(\frac{k+1}{2}+\frac{k-1}{2n})+O(d)=k-\frac{a(k-1)(n-1)}{2n}+O(d)<k$$

With MRSF policy, we shown that there exists a policy that performs better than greedy for any $R(0)$, $C(0)$ in worst cases by randomization and distributing job progress.



(a) Theoretical vs actual cost of greedy policy with a changeover delay $d = 0.01h$.

(b) Variance vs number of slices with an 80-hour deadline on multiple availability zones.

Figure 5.6: **Numerical results for validating the theoretical greedy cost and the assumption for increasing variance in the stochastic model.** Both analysis are conducted on sampling sub-traces from 2-month AWS spot availability traces.

5.4.2 Average Case with Stochastic Model

Since spot availability is a complex stochastic process, we propose a simpler model that is analytically tractable for the development of practical policies. With that model, we will show that n-sliced greedy is better than greedy in the average case.

In order to model the spot process, we consider a smoothed version where we assume that a fractional spot is always available, with a ratio $r < 1$, *i.e.,* a job running on the fractional spot makes r amount of progress per unit of time. For example, if spots have 4-hour average lifetimes and 1-hour average wait times after preemption. Then, the fractional spot has a ratio, $r = 4/(4+1) = 0.8$, and a job using it makes 0.8 amount of progress per unit of time.

Similar as Section 5.4.1, for simplicity, we assume that d is relatively small and ignore terms of $O(d)$. We first consider greedy policy. It will use the fractional spot until $R(t') = C(t') + O(d)$ and then switch to on-demand. At time t', the job progress on the fractional spot would be $C(0) - C(t') = rt' - O(d)$, *i.e.,* $C(t') = C(0) - rt' + O(d)$, and the remaining time would be $R(t') = R(0) - t'$. We can derive t' and expected payment (total cost) p:

$$R(t') = C(t') \implies R(0) - t' = C(0) - rt' + O(d) \tag{5.1}$$

$$t' = \frac{R(0) - C(0) + O(d)}{1 - r} \tag{5.2}$$

$$p = rt' + (R(0) - t')k + O(d) = (r - k)t' + kR(0) + O(d) \tag{5.3}$$

We can observe that the payment depends on the fractional spot ratio r. For simplicity, we will drop $O(d)$ in following formulas. Since $r - k < 0$, payment p reduces when the time t' spent on the fractional spot increases.

In Figure 5.6a, we calculate both actual and theoretical costs, p, for greedy policy on real availability traces for a 48-hour job with various deadlines (52 to 92 hours) and small changeover delays. It illustrates that theoretical costs with the significant simplified stochastic modeling matches well with actual costs.

We now consider the n-sliced greedy policy from Section 5.4.1. For a fixed r, the n-sliced greedy has the same expected payment as original greedy. However, when we started considering the expected payment across difference traces, variance for fractional spot involves. We show that n-sliced greedy works better than original greedy in average.

Consider spot fraction $\mathcal{R}$ as a random variable with mean r and variance v. We can prove that the expected time on the fractional spot $E[t']$ increases with the variance v, *i.e.,* larger v indicates lower expected cost:

Proof. Let $\mathcal{R} = r + \delta$. Based on Equation 5.2, we have $E[t']$:

$$E[t'] = E[\frac{R(0) - C(0)}{1 - (r + \delta)}] = \frac{R(0) - C(0)}{1 - r} E[1 + \frac{\delta}{1 - r} + \frac{\delta^2}{(1 - r)^2} + \cdots]$$

where the second equation is derived from Taylor expansion for $\delta \to 0$. By construction, $E[\delta] = 0$ and $E[\delta^2] = v > 0$. When we take the first three terms, we get an approximation:

$$E[t'] = \frac{R(0) - C(0)}{1 - r}(1 + \frac{v}{(1 - r)^2})$$

We calculate the difference of the expected time on the fractional spot $E[t']$ for policies with variance v_1 and v_2

$$\Delta = \frac{R(0) - C(0)}{(1 - r)^3}(v_1 - v_2)$$

Since $\Delta > 0$ when $v_1 > v_2$, we can conclude that $E[t']$ increases with the variance v. □

With the formula above, we calculate the difference of n-sliced (with variance $\hat{v}$) to original greedy (with variance v):

$$\Delta = \frac{R(0) - C(0)}{(1 - r)^3}(\hat{v} - v)$$

where $\hat{v}$ is the variance over slices with length $\frac{R(0)}{n}$ and v is the variance for traces with length $R(0)$. Since $\mathcal{R}$ is averaged over time, we expect $\hat{v} > v$ (shown in Figure 5.6b), *i.e.,* $\Delta > 0$. We can conclude that n-sliced greedy has larger $E[t']$, leading to a lower expected cost p than original greedy in average case. Also, as v increases with n, n-sliced policy can achieve better performance with more slices, when d is relatively small.

5.5 Methodology

Building on our theoretical analysis, we now propose policies for real-world cloud settings. In this section, we will examine the performance of a Time Sliced policy derived from the theoretical analysis, and extend it to a parameter-free Uniform Progress policy. Additionally, we present a upper bound of cost savings through the Omniscient policy, which has the knowledge of future spot availability, and a Partial

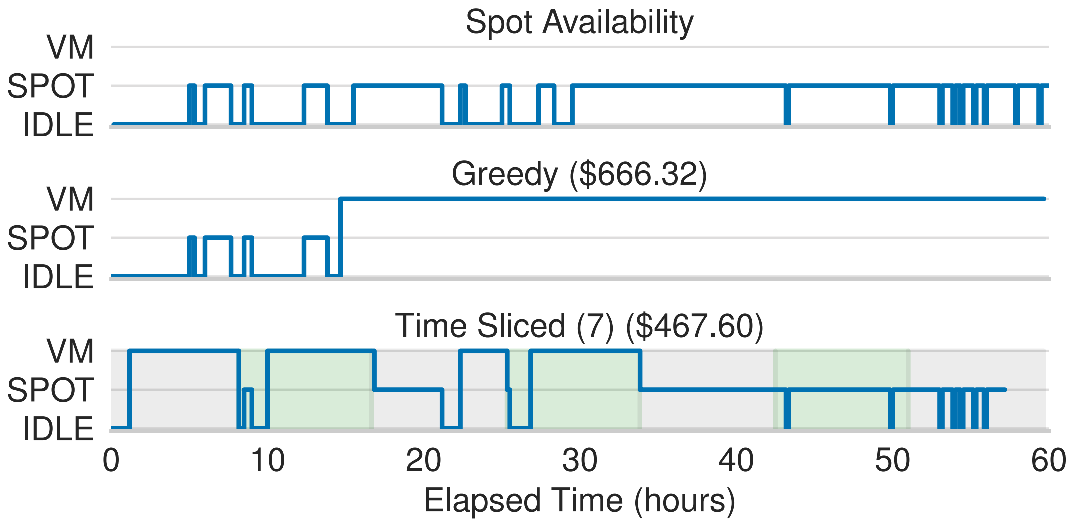


Figure 5.7: **Example decision traces comparing Time Sliced and greedy policy.** Time Sliced policy cuts costs by better utilization of available spot near deadline.

Lookahead Omniscient policy that only has a shorter lookahead of the future (*e.g.,* 6 hours). Then, we will discuss an interesting scenario when the next spot lifetime is given, and propose an extension that combines Uniform Progress with a Next Spot Lifetime Oracle. Lastly, we extend the policies to multiple instances, and relaxed job computation times and changeover delays for generality.

5.5.1 Time Sliced

Based on the n-sliced greedy policy in Section 5.4.1, we propose the Time Sliced policy. We divide the time before deadline, $R(0)$, into slices, and assign each slice a proportionate computation time $C(0)/n$ and deadline $R(0)/n$, denoted as C_i and R_i for slice i. In each time slice, we apply greedy policy – switching to on-demand instances when $R_i(t) < C_i(t) + 2d$. We make two changes compared to the n-sliced greedy policy: (1) jobs can continue on spot instances whenever available after $C_i(t) \leq 0$, and (2) if a slice makes more progress than required, we reduce the required computation in the succeeding slice, C_{i+1}. We do not apply randomness as in the competitive analysis for simplicity based on the assumption that clouds are non-adversarial.

Figure 5.7 presents example decision traces for both greedy and Time Sliced. The spot availability trace shows when spot is available on cloud. The greedy policy utilizes all available spot until $R(t) < C(t) + 2d$. At this point, the job cannot tolerate another changeover delay and must stay on on-demand until the end, rendering available spots close to deadline unusable. In contrast, Time Sliced policy's decision is divided into seven slices (with alternating colors), with greedy applied in each slice. Due to the progress made in earlier slices, Time Sliced allows more slacks to switch between spot and on-demand instances when the deadline is close. This enables better utilization of spot instances, reducing total cost. In this specific example, Time Sliced reduces 30% cost compared to greedy.

In Figure 5.8, we evaluate Time Sliced by comparing it to greedy in terms of average cost savings

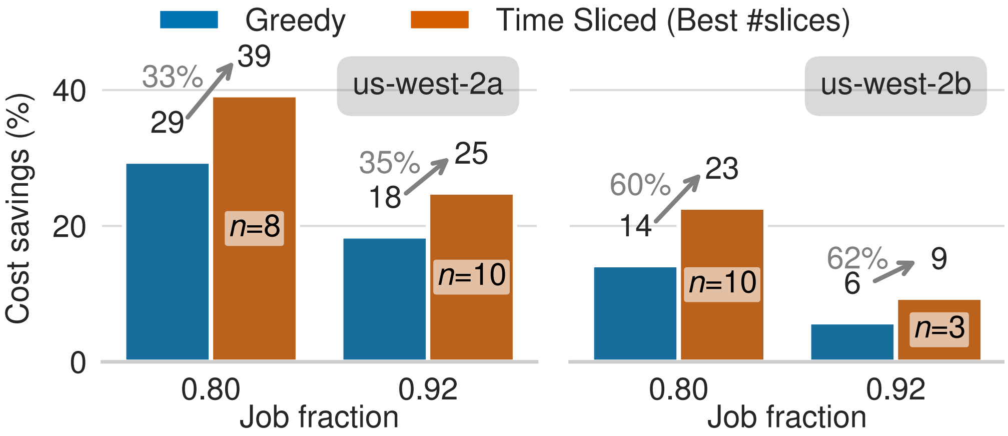


Figure 5.8: **Cost savings (*higher is better*) vs. on-demand with Greedy and Time Sliced policies.** Job fraction is $\frac{C(0)}{R(0)}$, and n is the best number of slices chosen for the Time Sliced policy.

across 600 random p3.2xlarge availability traces on AWS. Picking the optimal number of slices enables Time Sliced to achieve 33-62% additional cost savings for relatively tight deadlines. These results suggest that *ensuring uniform progress throughout a job's lifetime* leads to better utilization of spot availability in expectation. We apply this idea in the design of Uniform Progress below.

5.5.2 Uniform Progress

Although Time Sliced policy with the best slice number n outperforms greedy, selecting the optimal n for different cases is not practical. We take the uniform progress idea from Time Sliced policy and design a parameter-free policy, denoted as Uniform Progress.

5.5.2.1 Pushing the Slices to the Extreme

Time Sliced policy guarantees uniform progress by enforcing it in discrete slices. While progress can be left behind within a slice, it is ensured by the end of each slice. At the end of a slice i, $t_i = i\frac{R(0)}{n}$, *i.e.,* $i = t_i\frac{n}{R(0)}$. The *current progress*, $cp(t_i) = C(0) - C(t_i)$, is guaranteed to meet the *expected progress*, $ep(t_i)$:

$$cp(t_i) \geq ep(t_i) = i\frac{C(0)}{n} = t_i\frac{C(0)}{R(0)} \tag{5.4}$$

Note that when the slice number $n = 1$, there is only one t_i, *i.e.,* $t_1 = R(0)$, and Time Sliced becomes greedy policy and only enforces progress $C(0)$ at deadline $R(0)$. When more slices involve, with larger n, Equation 5.4 applies to more time steps $t \in \{\frac{R(0)}{n}, \frac{2R(0)}{n}, \ldots, \frac{nR(0)}{n}\}$. According to the stochastic model in Section 5.4.2, n-sliced greedy will perform better when n increases, given small changeover delays. Intuitively, this is due to a more aggressive enforcement of progress. For instance, increasing n from 2 to 10 within a 50-hour deadline ensures expected progress made every 5 hours instead of every 25 hours.

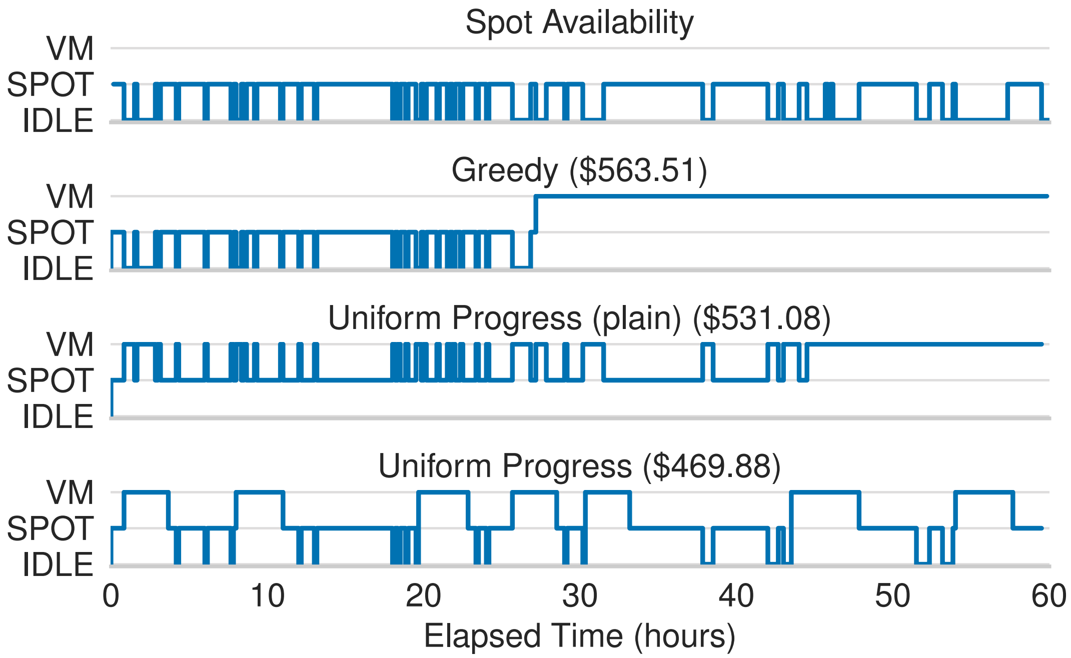


Figure 5.9: **An example decision trace for Uniform Progress.**

We adapt this idea into Time Sliced by pushing $n \to \infty$, making each slice infinitesimal. That enforces Equation 5.4 at any $t \leq R(0)$, *i.e.,* fully distributing progress within the deadline:

$$cp(t) \geq ep(t) = t\frac{C(0)}{R(0)}, \forall t \leq R(0) \tag{5.5}$$

5.5.2.2 Uniform Progress Policy

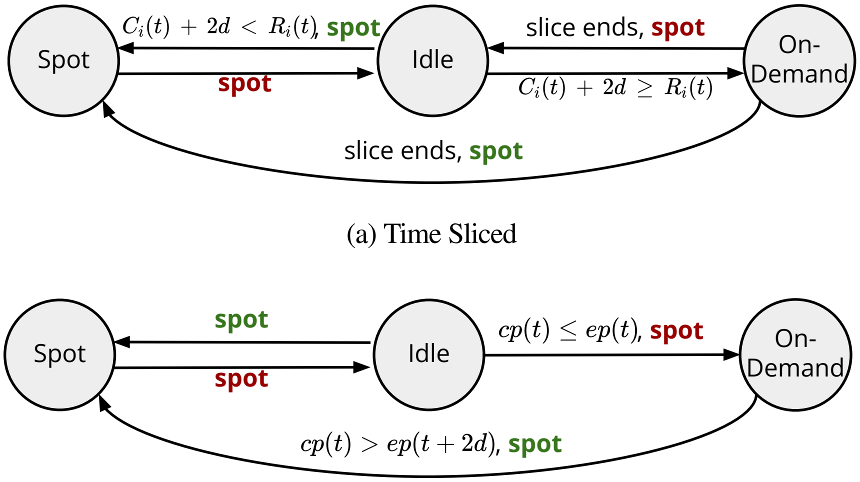
We propose a parameter-free policy, called *Uniform Progress (plain)*, that switches among three instance states: idle, spot, and on-demand. The policy, based on Equation 5.5 and the rules in Section 5.3, has the following rules:

1. **Uniform Progress:** When the job is idle and $cp(t) < ep(t)$, switch to on-demand and stay on it to catch up progress.

2. **Taking Risks:** Switch to spot whenever it is available (even when $cp(t) < ep(t)$). Stay on the spot until it is preempted (Exploitation Rule).

To avoid missing deadline, we also apply Safety Net Rule on top. The first rule asks the policy to maintain steady progress, while *Taking Risks* rule allows the policy to utilize any available spot instances by taking the risk of changeover delays.

In Figure 5.9, we show an example decision trace. Similar to Time Sliced, Uniform Progress (plain) can achieve better cost savings compared to the greedy policy by evenly distributing progress within the deadline. However, during periods when spot life/wait time are relatively short, the policy suffers from



(a) Time Sliced

(b) Uniform Progress

Figure 5.10: **State machine diagram for Time Sliced and Uniform Progress.** spot means spot unavailable and spot means spot available. The Safety Net Rule is left out for simplicity.

frequent switches between spot and on-demand instances. When the job is on on-demand, and a spot becomes available, our policy will immediately switch to spot. If the spot is preempted by the cloud shortly, the job may make little progress. When that happens, $cp(t) < ep(t)$ can still hold and the job will be scheduled to on-demand again, wasting two changeover delays, $2d$ (one for spot and one for on-demand).

To address that, we propose adding hysteresis to the policy. Although the policy does not know or control the lifetime of a spot instance, it can ensure that the progress made on on-demand instances is sufficient to compensate for potential losses in the worst-case scenario. We thus add another rule:

3. **Hysteresis:** When the job is on on-demand, stay on it until $cp(t) \geq ep(t+2d)$.

We call the resulting policy *Uniform Progress*. Figure 5.9 shows that the hysteresis mitigates frequent switching by enforcing more progress on on-demand, and improves cost savings.

Figure 5.10 compares the state transitions of Uniform Progress and Time Sliced. Both policies share the uniform progress idea, but Time Sliced is discretized, relying on Safety Net Rule within each slice and slice boundaries to jump off an on-demand instance. In comparison, Uniform Progress replaces slice parameters with a global uniform progress checker, $cp(t) \geq ep(t)$, and a hysteresis, $cp(t) \geq ep(t+2d)$.

We will evaluate the policies above in Section 5.6. In order to properly assess a policy's performance relative to the best cost savings, we next discuss several policies, which have access to future knowledge, and use them as cost saving upper bounds.

5.5.3 Omniscient

First, we propose the Omniscient policy, which assumes full future knowledge and generates the theoretically optimal plan.

5.5.3.1 Omniscient Policy

The Omniscient policy minimizes cost for a given availability trace and deadline $R(0)$. We define some binary variables:

- $a(t)$ whether a spot instance is available at time t.
- $s(t)$,$v(t)$ indicate the policy choose to use a spot/on-demand instance at time t.
- $x(t)$,$y(t)$ represent changeover delays happen to a spot/on-demand instance at time t.

By discretizing time, we can represent the policy as a cost minimization problem:

$$\min_{s(t),v(t)} \sum_{t=0}^{R(0)} [s(t)+v(t)k] \tag{5.6}$$

$$\forall t,\, s(t)+v(t) \leq 1,\, s(t) \leq a(t) \tag{5.7}$$

$$\sum_{t=0}^{R(0)} [s(t)+v(t)] \geq d \sum_{t=1}^{R(0)} (x(t)+y(t)) + C(0) \tag{5.8}$$

$$\forall t,\, x(t) \leq s(t),\, x(t) \leq 1-s(t-1),\, x(t) \geq s(t)-s(t-1) \tag{5.9}$$

$$\forall t,\, y(t) \leq v(t),\, y(t) \leq 1-v(t-1),\, y(t) \geq v(t)-v(t-1) \tag{5.10}$$

Equation 5.7 ensures the policy to choose only one instance at a time and only use spot when it is available; Equation 5.8 requires the total time on spot and on-demand instances to be larger than sum of the time spent on changeover delays and the job runtime; Equation 5.9 and Equation 5.10 set variables $x(t)$ and $y(t)$ to 1 when a changeover occurs for spot and on-demand instances, respectively. The resulting formula is an integer linear programming (ILP) problem and can be solved using ILP solvers [101, 55].

5.5.3.2 Partial Lookahead Omniscient Policy

Omniscient, with complete knowledge of future spot availability, produces an unachievable bound. To better understand the impact of partial knowledge, we propose Partial Lookahead Omniscient, which has limited foresight into future spot availability. By partitioning the deadline into n slices, it can only see complete availability within each slice. To incorporate that knowledge, we modify Omniscient formula to minimize the average cost of progress made in a slice i while ensuring the job progress at the end to be at least $iC(0)/n$.

Instead of minimizing the total cost for the progress, in Partial Lookahead Omniscient policy, a job can make more progress than it is assigned in each slice and reduce the computation time required in the next slice by the additional progress made. Therefore, we modify the ILP formula for the Omniscient policy with the following formula for a slice i to minimize the average cost of the progress made in that slice:

$$\min_{s(t>t_{i-1}),v(t>t_{i-1})} \sum_{t=t_{i-1}}^{t_i} [s(t)+v(t)k]/P_i \tag{5.11}$$

$$\forall t,\, s(t)+v(t)\leq 1,\, s(t)\leq a(t) \tag{5.12}$$

$$P_i = \sum_{t=t_{i-1}}^{t_i} [s(t)+v(t)] - d \sum_{t=t_{i-1}+1}^{t_i} (x(t)+y(t)) \tag{5.13}$$

$$P_i \geq \frac{iC(0)}{n} - \sum_{j=1}^{i-1} P_j \tag{5.14}$$

$$\sum_{j=1}^{i} P_i \leq C(0) \tag{5.15}$$

$$\forall t,\, x(t)\leq s(t),\, x(t)\leq 1-s(t-1),\, x(t)\geq s(t)-s(t-1) \tag{5.16}$$

$$\forall t,\, y(t)\leq v(t),\, y(t)\leq 1-v(t-1),\, y(t)\geq v(t)-v(t-1) \tag{5.17}$$

where the Equation 5.13 ensures the total progress at the end of the slice is at least $cp(t_i)\geq iC(0)/n$, and the Equation 5.15 avoids making more total progress than the job computation time.

5.5.4 Next Spot Lifetime Oracle

Both Omniscient and Partial Lookahead Omniscient policies assume complete knowledge of future availability with different lookahead windows. We propose a more realistic scenario where cloud providers offer an oracle $o(t)$ that returns the lifetime of the next spot instance a job can acquire at the current time t. This assumption is reasonable as providers can determine when to reclaim a spot instance.

Uniform Progress can be extended to leverage this oracle. We introduce two new conditions to replace the hysteresis:

1. If the job is idle, we only switch to spot when the average cost per unit of progress is lower than on-demand cost $\frac{o(t)}{o(t)-d} < k$, *i.e.,* $o(t) > \frac{kd}{k-1}$.

2. If the job is on on-demand instance, we switch to spot only when the average cost per unit of progress, considering switching to spot and back to on-demand, is less than staying on the current on-demand: $\frac{o(t)+kd}{o(t)-d} < k$, *i.e.,* $o(t) > \frac{2kd}{k-1}$.

5.5.5 Extending to Multiple Instances

All the discussions above are based on single-instance scenario. We now extend the policies to multiple instances. We assume gang-scheduling is required, *i.e.,* all instances must be running for a job to progress. This is typical in distributed ML training [86, 129, 87] and HPC workloads [34]. A cluster may consist solely of spot instances, on-demand instances, or a mix of both. We call clusters with an identical resource type *homogeneous* and those with a mix *heterogeneous*. Changeover delays are incurred when a cluster is reconfigured, *i.e.,* the number of spot/on-demand instances in it changes, unless it has no instance after reconfiguration.

We introduce a new rule for all multi-instance policies:

Polarization Rule. For a job requiring $N > 1$ instances, a policy should either use no instance or N instances at any time.

Since gang-scheduling is required, a cluster with fewer than N instances incurs unnecessary costs without job progress. Thus, once any instance is preempted, a policy should immediately reconfigure the cluster to either 0 or N instances.

We now extend previous policies to multiple instances.

5.5.5.1 Extending Greedy and Uniform Progress.

First, observing that spot availability tends to change simultaneously for multiple instances (Section 5.2.4), we propose each policy should produce homogeneous clusters. We will show that this assumption does not harm performance on reasonably large clusters (Section 5.6.6). Combining this with *Polarization Rule*, the action space for a policy is simplified to either: N spot, N on-demand, or no instances at any time t.

The problem for multiple instances is now equivalent to the single instance, with the one-to-one mapping of states (Section 5.3.2):

- **Cluster state**: N spot, N on-demand, or no instances map to spot, on-demand or idle states for single-instance jobs.
- **Spot state**: If available spot instances $a(t) < N$, it is equivalent to a spot being unavailable in the single-instance scenario, and $a(t) = N$ maps to a spot being available.

Thus, for multi-instance jobs, we directly execute greedy and Uniform Progress using the mappings above.

5.5.5.2 Extending Omniscient.

Omniscient (Homogeneous). For Omniscient, we can also restrict it to produce homogeneous clusters and get Omniscient (Homogeneous), where all instances in a cluster with N instances should be the same type (all spot, all on-demand, or none). We revise the semantics of the original variables:

- $a(t)$: the number of spot instances available at time t.
- $s(t)$, $v(t)$ indicate the policy chooses to use all spot or all on-demand for the cluster at time t.
- $x(t)$, $y(t)$ represent the changeover delay that happens to the spot/on-demand cluster at time t.

The Omniscient policy with the same instance type can be represented as:

$$\min_{s(t),v(t)} \sum_{t=0}^{R(0)} N[s(t)+v(t)k] \tag{5.18}$$

$$\forall t,\ s(t)+v(t)\leq 1,\ s(t)\leq a(t)/N \tag{5.19}$$

$$\sum_{t=0}^{R(0)} [s(t)+v(t)] \geq d \sum_{t=1}^{R(0)} (x(t)+y(t))+C(0) \tag{5.20}$$

$$\forall t,\ x(t)\leq s(t),\ x(t)\leq 1-s(t-1),\ x(t)\geq s(t)-s(t-1) \tag{5.21}$$

$$\forall t,\ y(t)\leq v(t),\ y(t)\leq 1-v(t-1),\ y(t)\geq v(t)-v(t-1) \tag{5.22}$$

Omniscient (Heterogeneous). To obtain a better theoretical upper bound for cost savings, however, we further adapt Omniscient to support heterogeneous clusters, denoted as Omniscient (Heterogeneous), by modifying the ILP Equation 5.6 to factor in a mixed cluster configuration.We first update the definition of variables as follows:

- $s(t)$, $v(t)$: the number of spot and on-demand instances in the cluster at time t.
- $p(t)$: whether the cluster is UP at time t.
- $z(t)$: whether changeover delay is triggered at time t.
- $m(t)$, $n(t)$, $j(t)$, $k(t)$: intermediate binary variables.

The following is the Omniscient policy for multi-nodes with gang scheduling.

$$\min_{s(t),v(t)} \sum_{t=0}^{R(0)} [s(t)+v(t)k] \tag{5.23}$$

$$\forall t,\ s(t)+v(t)-N\cdot p(t)=0,\ s(t)\leq a(t) \tag{5.24}$$

$$\sum_{t=0}^{R(0)} [s(t)+v(t)] \geq d \sum_{t=1}^{R(0)} z(t)+C(0) \tag{5.25}$$

$$\forall t,\ s(t)-s(t-1)\leq N\cdot z(t) \tag{5.26}$$

$$\forall t,\ v(t)-v(t-1)\leq N\cdot z(t) \tag{5.27}$$

$$\forall t,\ m(t)\leq s(t)-s(t-1)+(N+1)\cdot j(t) \tag{5.28}$$

$$\forall t,\ m(t)\leq s(t-1)-s(t)+(N+1)\cdot(1-j(t)) \tag{5.29}$$

$$\forall t,\ n(t)\leq v(t)-v(t-1)+(N+1)\cdot k(t) \tag{5.30}$$

$$\forall t,\ n(t)\leq v(t-1)-v(t)+(N+1)\cdot(1-k(t)) \tag{5.31}$$

$$\forall t,\ z(t)\leq m(t)+n(t) \tag{5.32}$$

Equation 5.26 and Equation 5.27 set $z(t)=1$, when either the number of spot or on-demand increases in the cluster; Equation 5.28 to Equation 5.32 enforces $z(t)=0$ when $s(t)=s(t-1)\wedge v(t)=v(t-1)$, *i.e.,* the number of spot or on-demand used by the job does not change. That said, Equation 5.26 to Equation 5.32 make sure $z(t) = 1$ iff changeover delay happens at time t.

5.5.6 Relaxing Computation Time and Changeover Delay

In real-world scenarios, exact computation times and changeover delays may be uncertain. We generalize our model to accommodate such variability.

Computation time. To account for the inaccuracies of a user-provided job computation time $\bar{C}(0)$, we denote the difference to the actual job computation time as $\delta = C(0) - \bar{C}(0)$. Given that no policy can predict $C(0)$ precisely beforehand, we adjust the deadline guarantee of the policies to be best effort, ensuring a finish time within the original deadline plus the difference, $R(0) + \delta$. This is guaranteed by having all policies stay on the current instance and switch to on-demand[5], after the job does not finish but has already made $\bar{C}(0)$ progress, *i.e.,* $\bar{C}(t) \leq 0$. When a user overestimates a job's computation time $\bar{C}(0) > C(0)$, it should finish before the original deadline. Otherwise, if the job computation time is underestimated $\bar{C}(0) < C(0)$, the job should finish within the original deadline plus the difference. Note that there is no additional d, as Safety Net Rule guarantees that when $\bar{C}(t) \to 0$, the job should either be on on-demand already or $R(t) \geq \bar{C}(t) + d$, *i.e.,* there is a spare d for the job to switch to on-demand.

Changeover delay. We now adjust the model to factor in system stragglers and variations in changeover delay. We assume that no policies can foresee the exact changeover delay until its occurrence, though the average changeover delay is given. If the maximum possible changeover delay, $\hat{d}$, is also given (*e.g.,* the most significant possible progress loss is triggered), we can prove that policies should be able to ensure a deadline of $R(0) + 2(\hat{d} - d)$.

Proof. With the assumptions, all variance of changeover delay will be directly reflected in $C(t)$.

We consider the last moment a job is idle before it finishes (at time t), there are three cases:

1. $R(t) > C(t) + 2d$: Safety Net Rule should never be triggered, causing the maximum time the job finishes to be $R(0) - R(t) + C(t) + \hat{d} < R(0) + \hat{d} - 2d$.

2. $R(t) = C(t) + 2d$: Safety Net Rule kicks in at $t + \varepsilon$, and the guaranteed deadline should be $R(0) - R(t) + C(t) + \hat{d}$, *i.e.,* $R(0) + \hat{d} - 2d$.

3. $R(t) < C(t) + 2d$: It means the job was on a spot instance and got preempted. In this case, at the time t' the job jumped onto the spot instance, we have $R(t') \geq C(t') + 2d$. Thus, the worst case for the guaranteed deadline would be the job experience two maximum changeover delays, once for jumping onto a spot, and once for jumping onto an on-demand instance. That said, the bound for the finish time would be $R(0) - R(t') + C(t') + 2\hat{d} \leq R(0) + 2(\hat{d} - d)$.

Combining the three cases, the bound for the deadline guaranteed should be $R(0) + 2(\hat{d} - d)$ □

[5] If the job was on a spot instance, it should switch to on-demand after the spot instance is preempted (Exploitation Rule).

Policy	**On-Demand** (hours)	**Spot** (hours)	**Spot Util.**
On-Demand	48.0 ± 0.0	0.0 ± 0.0	0%
Greedy	30.8±17.7	17.2 ± 17.7	63%
Uniform Progress	25.1±15.3	22.9 ± 15.4	84%
Omniscient	20.7±15.5	27.4 ± 15.5	100%

Table 5.3: **Compute time spent on on-demand and spot instances**, averaged across 8 scenarios for a job fraction of 0.8. "Spot Util." indicates the fraction of compute time on spot leveraged by a policy vs. the Omniscient policy.

If a user would like to ensure the original deadline with a given maximum changeover delay, they can specify a new deadline $R(0)-2(\hat{d}-d)$.

With the new model, policies can account for the variety, by guaranteeing a bounded relaxed deadline $R(0)+\delta+2(\hat{d}-d)$.

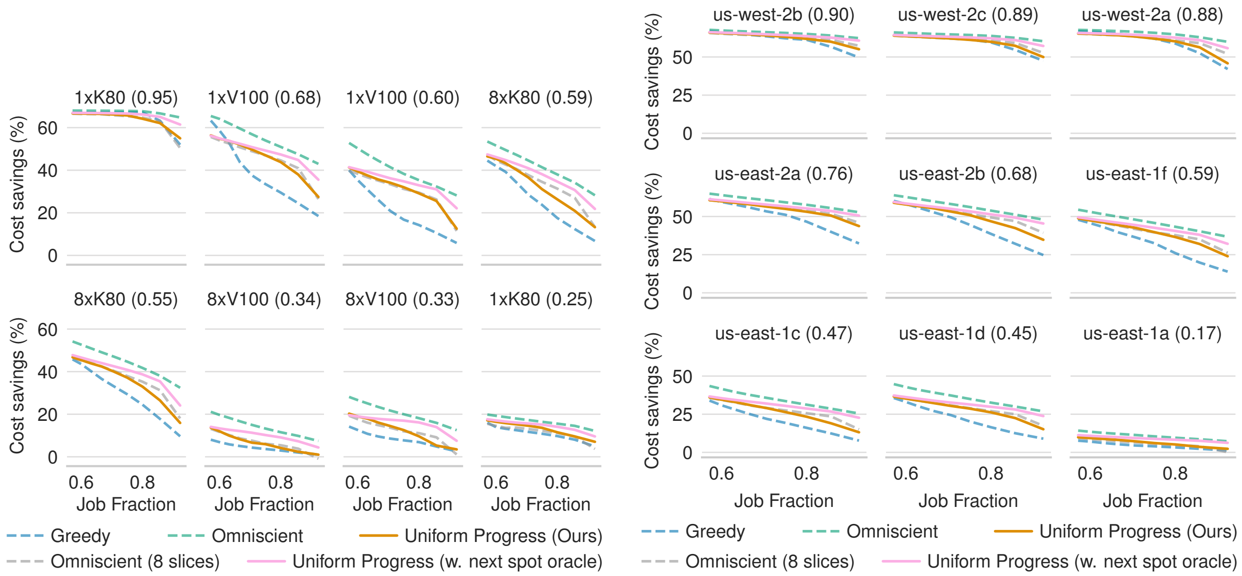
5.6 Evaluation

In this section, we conduct experiments to assess the performance of the proposed policies using real spot instance traces collected from the cloud.

5.6.1 Datasets and Setup

We collected spot availability traces on AWS (Section 5.2.1). These traces include a 2-week availability trace started on 10/26/2022, with four instance types: p3.2xlarge/p3.16xlarge (1/8 V100), p2.2xlarge/p2.16xlarge (1/8 K80), and two availability zones: us-west-2a and us-west-2b. Moreover, we collect a 2-month long availability trace started on 02/15/2023 for p3.2xlarge instances across nine zones from regions, us-east-1, us-east-2, and us-west-2. For multiple instances, we collect 2-week *preemption* traces for 4 p3.2xlarge in 3 AWS zones (*us-east-1f*, *us-east-2a*, *us-west-2c*), and 2-week *availability* traces for 16 p3.2xlarge in 3 zones (*us-east-2a*, *us-west-2b*, *us-west-2c*). All the availability traces were collected with a 10-minute probe interval. As demonstrated in Section 5.2.1, availability and preemption traces are highly correlated, indicating that the performance of the policies on availability traces should reflect their real-world performance. We will use preemption traces in Section 5.6.6 for multiple instances benchmark and Section 5.7.2 for real system evaluation.

We evaluate the policies on both 2-week traces, and 2-month traces. For all experiments, we randomly sample 300 starting points for each trace, considering each pair of instance type and zone. We consider cases where the *job fraction* $\frac{C(0)}{R(0)} > 0.6$, *i.e.,* the deadline is relatively tight, as the problem becomes less interesting when deadlines are loose and available spot instances within deadline are sufficient to complete the job. For loose deadlines, jobs can utilize spot instances whenever they are available until the remaining time-to-deadline $R(t)$ is relatively tight compared to the remaining computation time $C(t)$, and then start



(a) 2-week dataset.

(b) 2-month dataset.

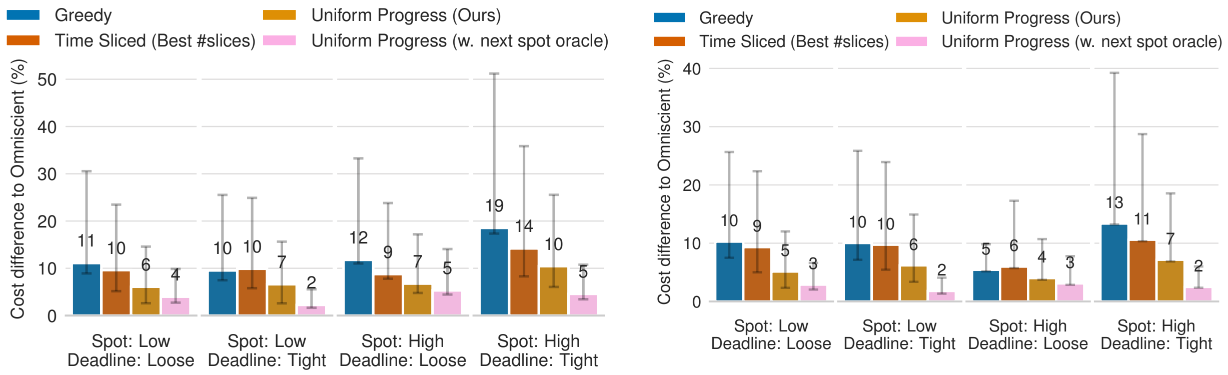
Figure 5.11: **Cost savings (*higher is better*) against on-demand instances on real spot availability traces.** Omniscient (8 slices) is Partial Lookahead Omniscient. Larger job fraction means tighter deadline. Each sub-plot is on a (instance type, zone) trace. Values in '(x)' are average spot fractions (percentages of time a spot instance is available) across all samples in the trace.

applying policies, see Section 5.6.8. The computation time is set to 48 hours for consistent comparison across different settings (experiments for different computation times can be found in Section 5.6.8). Unless noted, changeover delays d are set to 0.2 hours and costs are normalized by on-demand costs in all experiments.

Baselines. To our knowledge, existing methods in literature (Section 5.8) do not consider switching between spot and on-demand in a cost optimization and deadline adherence setting for batch jobs. Thus, we compare our results against policies with future knowledge, which serve as strong upper bounds.

5.6.2 Time Spent on On-demand and Spot Instances

We first show different policies' overall compute times on on-demand vs. spot instances, which exclude changeover delays. Such breakdowns examine how well spot instances are utilized. Table 5.3 shows the results with a fixed job fraction $\frac{C(0)}{R(0)} = 0.8$ on the 2-week traces, averaging across eight (instance type, availability zone) pairs, each with 300 randomly sampled traces. We observe that our Uniform Progress runs on spot instances 21% longer than greedy policy on average, reducing the gap to Omniscient's spot usage by 57%.



(a) 2-week dataset, aggregated on 300×8=2400 sampled traces.

(b) 2-month dataset, starting from 2/15/2023, aggregated on 300×9=2700 sampled traces.

Figure 5.12: **Impact of Spot Fraction and Deadline.** Cost difference compared to Omniscient policy (normalized by on-demand cost, *lower is better*), measuring a policy's proximity to Omniscient. Error bars range from p25 to p75. "Spot" represents spot fraction.

5.6.3 Various Deadlines

Figure 5.11a evaluates the cost savings achieved by the policies across various deadlines (represented as job fractions) on the 2-week availability traces. Our Uniform Progress consistently surpasses the greedy in cost savings in all cases, while approaching savings of Omniscient policy.

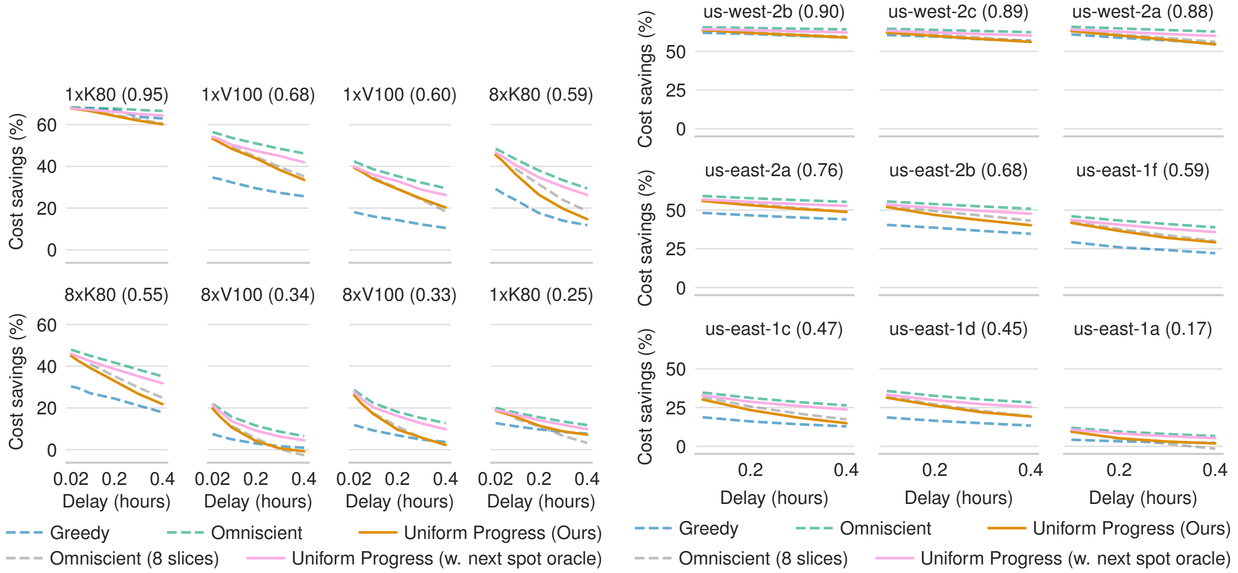
While Uniform Progress excels, there is still a gap to Omniscient. We compare Uniform Progress with Partial Lookahead Omniscient policy with 8 slices, which assumes strong knowledge of the future (around 6 hours of lookahead): Uniform Progress achieves similar performance in most cases, despite lacking future knowledge. This suggests any other policy without future knowledge may not yield much higher savings.

We also investigate the potential improvement of Uniform Progress policy by assuming cloud providers offering an oracle for the lifetime of the next spot instance (Section 5.5.4). With such knowledge, cost savings improve significantly, nearing the theoretical optimum when deadlines are tighter.

The conclusions also hold on the 2-month traces, see Figure 5.11b.

5.6.4 Impact of Spot Fraction and Deadline

To better understand the influence of *spot fractions* (the percentage of time a spot instance is available) and deadlines on policy performance, we categorize them into two dimensions: low or high spot fraction, and loose or tight deadline. Tight deadline represents job fraction $\frac{C(0)}{R(0)} > 75\%$, while high spot fractions are defined as those exceeding 50%. Our 2-week traces have an even distribution between high and low spot fractions, while the 2-month traces show a dominance of high spot fractions, which forms 72% of all cases. This aligns with our earlier observation of the volatile nature of spot instance availability (Section 5.2.2).



(a) 2-week dataset.

(b) 2-month dataset.

Figure 5.13: **Impact of changeover delays (d).** Values in '(x)' are average spot fractions over all samples in the trace.

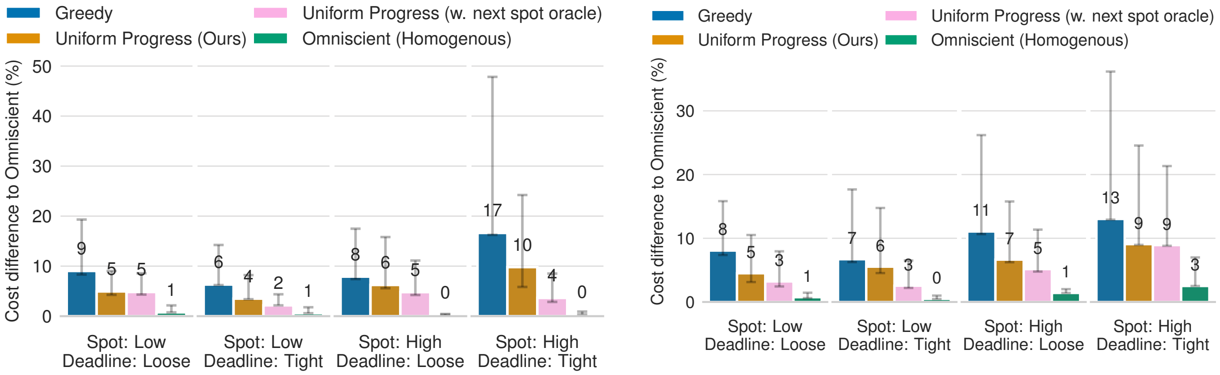
Figure 5.12a and Figure 5.12b present the performance of the policies compared to Omniscient policy (theoretical upper bounds for cost savings) in the four categories, on 2-week and 2-month datasets respectively.

For tight deadlines, the number of feasible instance switches is limited to at most $\frac{R(0)-C(0)}{d}$, demanding strategic planning of each changeover. When spot availability is high and deadline tight (the rightmost group of bars), all policies lacking future knowledge exhibit a relatively large gap to the optimal. Nevertheless, Uniform Progress still reduces the gap by $\sim 2\times$ compared to the greedy policy. This efficiency arises from its uniform progress guarantee and hysteresis, which optimize spot utilization within the deadline while avoiding frequent changeovers. The small gap between Uniform Progress with the next spot lifetime oracle and Omniscient policy confirms that the ability to skip short spot lifetimes and strategically switching from on-demand to spot with the opportunity cost in mind is crucial to achieve close to optimal performance.

As deadlines loosen and spot availability increases, all policies perform closer to Omniscient policy, as jobs have greater flexibility to wait for spot instances and switch between resource types, *i.e.,* judicious planning becomes less important.

Additionally, we show the performance of Time Sliced policy with the best number of slices (within 50 slices). Time Sliced policy outperforms Greedy because of uniform progress it guarantees, but worse than Uniform Progress, potentially due to a higher overhead between slice switches.

Regardless of the different categories, our Uniform Progress policy reduces the gap to optimal by nearly 2× compared to greedy policy for both average and tail (p75) cases.



(a) Cost savings for jobs on 4 instances.

(b) Cost savings for jobs on 16 instances

Figure 5.14: **Cost savings for jobs on multiple instances.** It is compared to Omniscient with heterogeneous clusters.

5.6.5 Different Changeover Delays

In Figure 5.13, we evaluate the performance of policies across various changeover delays. Our Uniform Progress performs consistently similar to the Partial Lookahead Omniscient policy. As changeover delays increase, cost savings compared to on-demand instances are reduced. This is because for each spot instance being used, a larger changeover delay means we pay the same price for less actual progress, so that switching to spot instances becomes less economical. Both Uniform Progress and Partial Lookahead Omniscient approach the greedy policy as d increases. However, Uniform Progress combined with the next spot lifetime oracle consistently remains close to the upper bound, due to its ability to skip short spot lifetimes and judiciously calculate the opportunity cost of switching from an on-demand to a spot instance.

5.6.6 Multiple Instances

We now evaluate the policies on multi-instance jobs. Figure 5.14a shows the cost savings on 4-instance clusters for various policies compared to the theoretical upper bound set by our Omniscient (Heterogeneous) policy (Section 5.5.5). The difference between Omniscient (Homogeneous) and Omniscient (Heterogeneous) is negligible (at most 1%), which validates the use of homogeneous clusters in our policy formulation. Our Uniform Progress consistently outperforms the greedy policy, especially in high-spot-availability, tight-deadline conditions, which agrees with the conclusion on single-instance jobs (Section 5.6.4). We observe a similar win for clusters with 16 instances (Figure 5.14b). Due to monetary budget limits, we leave the extension to larger clusters ($N > 16$) to future work.

5.6.7 Relaxed Computation Time and Changeover Delay

We show that the variations for computation time and changeover delays introduced in Section 5.5.6, marginally influence the cost savings. In Figure 5.15, we apply a uniformly distributed variance to the computation time and changeover delays, and compare all policies with Omniscient policy, which

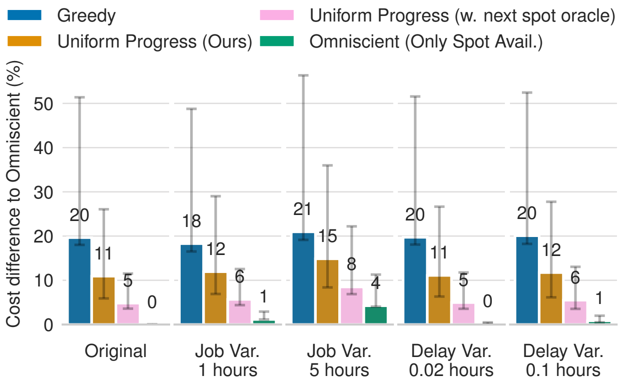


Figure 5.15: **Cost savings with relaxed job computation time or changeover delays.** All policies are compared against Omniscient knowing exact spot availability, computation time, and changeover delays in advance. Omniscient (Only Spot Avail.) only has the information of spot availability.

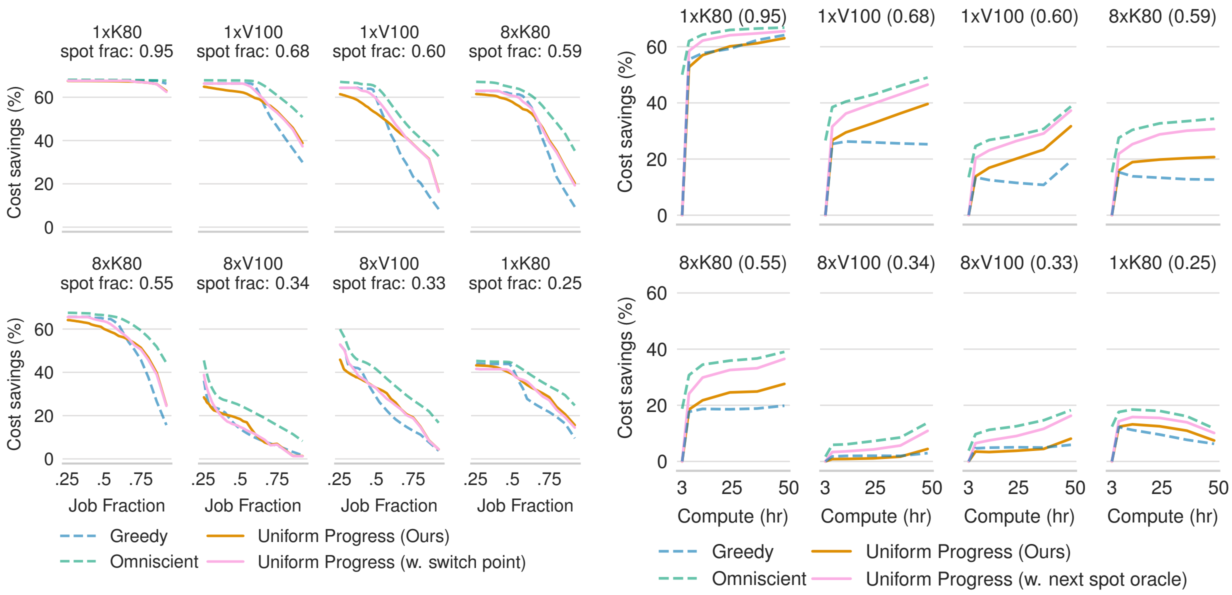
possesses exact knowledge of the job and delays. The experiments are conducted in the same settings as Section 5.6.4, with a single instance, high spot fraction, and tight deadline. All policies can guarantee deadlines in the new model. The performance of Omniscient with only spot availability information degrades when the variance of computation time increases. When a user-provided job computation time is larger than the actual one $\bar{C}(0) > C(0)$ (overestimate), it cannot fully utilize spot instances close to the deadline, while, for $\bar{C}(0) < C(0)$ (underestimate), it has to use on-demand after exceeding the original deadline $R(0)$. Similarly, it performs worse when the variance of changeover delay increases, due to sub-optimal decisions made with partial information. However, in all cases, we observe Uniform Progress outperforms greedy with a relatively stable gap, indicating its robustness.

5.6.8 Ablation Study for Experiment Setup

We now examine the ablation study of the basic setups for our experiments to further show the generality of Uniform Progress.

Loose Deadline In this chapter, we mainly discuss policy design for jobs with relatively tight deadlines, as very loose deadlines will likely lead to jobs able to finish on spot instances only. As mentioned in Section 5.6.1, when a loose deadline is given, a job can utilize spot instances whenever available until timestamp t_0, when the remaining time to deadline $R(t_0)$ becomes tight compared to the job progress $C(t_0)$, and apply the policy. We conduct experiments for loose deadlines, by setting the switch point at $\frac{C(t_0)}{R(t_0)} = 0.7$.

In Figure 5.16a, we can see that greedy policy gets close to the upper bound of the cost savings for very loose deadlines, as jobs are likely to be able to finish on spot instances only. It is worth noticing that job fraction 0.25 represents the deadline is $4x$ longer than the job duration. By allowing jobs to utilize as many spot instances as possible when the remaining time is abundant, all policies perform similarly in cases where loose deadlines are given.



(a) Loose deadline.

(b) Various job compuation time. Job fraction (computation time/deadline) is set to 85%.

Figure 5.16: **Ablation study for experiment setup.** Cost savings for loose deadline and various job computation time.

Various Job Computation Time. We compare the cost savings for the policies across different job computation times with the same job fraction 85% in Figure 5.16b. When job computation time is very small (comparable to the changeover delay), all policies' cost savings drop quickly, as switching between spot and on-demand instances is not worth the cost caused by the changeover delay. However, when the job computation time increases, there is more optimization opportunity for the policies, as more changeover delay can be tolerated, leading to a larger gap between Uniform Progress and greedy policy.

5.7 Practical Usage

In this section, we discuss our implementation of the prototype and evaluate it with three real-workload: machine learning training, bioinformatics (HPC), and data analytics.

5.7.1 Implementation

We implemented the policies on top of the intercloud broker system in Chapter 4, SkyPilot [163]. Given an availability zone and an instance type to use, our policies drive a job's resource provisioning and switching decisions.

In the system, a controller is in charge of monitoring spot availability and managing the job with heartbeats. All policies are invoked by the controller behind a simple interface as follows. Periodically,

Workload	**Location**	**Instance**	**SpotPrice (Saving)**	**Compute**	**Deadlines**	**Changeover Delay**
ML Training	AWS(west-2b)	p3.2xlarge	\$0.92/hr (-67%)	72hrs	84/100hrs	$4+5+9$mins $\approx$0.3hrs
Bioinformatics	GCP(east1-b)	c3-highcpu-88	\$0.34/hr (-91%)	22.5hrs	24/28hrs	$2+1+8$mins $\approx$0.2hrs
Data Analytics	AWS(east-1c)	r5.16xlarge	\$1.85/hr (-55%)	27hrs	30/36hrs	$4+1+7$mins $\approx$0.2hrs

Table 5.4: **Detailed characteristics of real workloads.** All locations are in US. Deadlines are derived from job fractions 90% and 75%, and changeover delays are the sum of VM provisioning, environment setup, and job recovery progress loss time.

		Uniform Progress	
Workload	**On-demand**	Tight DDL (0.9)	Loose DDL (0.75)
ML	\$233.5	\$138.2 (**-41%**)	\$122.0 (**-48%**)
Bioinfo	\$140.5	\$51.9 (**-63%**)	\$22.8 (**-84%**)
Analytics	\$109.6	\$80.0 (**-27%**)	\$74.1 (**-32%**)

Table 5.5: **Cost savings for real workloads.** Results of two deadlines are shown (job fractions 0.9 and 0.75).

the policy observes current_instance_state (in {idle, spot, on-demand}) and a boolean is_spot_available through the controller, and then uses them to compute a decision (in the same state set). If the decided state differs from the current instance state, the decision is executed by the system's provisioner module (*e.g.,* switch from on-demand to spot). To obtain the boolean is_spot_available, the controller invokes cloud-specific capacity reservation APIs (*e.g.,* AWS EC2 offers a create_capacity_reservation API) which return whether a zone has capacity for a spot instance type.

5.7.2 Real Workloads

We validate our policy across AWS and GCP platforms using real-world preemption traces with spot availabilities ranging from 70% to 90%. Metrics like changeover delays and other system lags are measured directly from the implementation and included in the evaluation.

5.7.2.1 Workload Setup

We benchmark all policies on real workloads, including Machine Learning (ML) Training, Bioinformatics, and Data Analytics. We summarize the settings of the three workloads in Table 5.4.

Machine Learning Training. We consider pre-training a RoBERTa [94] model on a subset of Wikipedia, WikiText-103, with a V100 GPU instance (p3.2xlarge) on AWS. We follow the configuration of FairSeq's reproduction [117, 123] to train the model for around 110 epochs (each takes about 40 minutes). To be fault-tolerant, we checkpoint the model weights twice per epoch to a cloud object

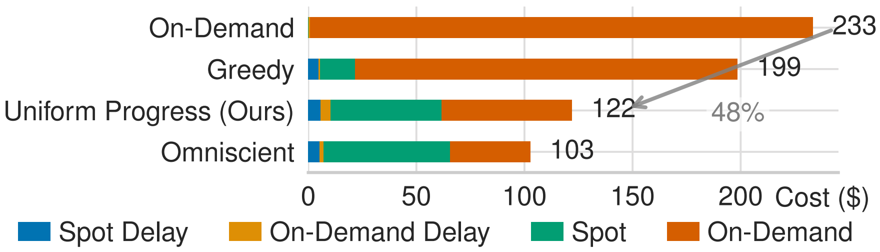


Figure 5.17: **Cost breakdown of each policy for ML workload.**

store (AWS S3). The average progress loss and the time to reload the model into GPU are included in the changeover delay.

Bioinformatics. We run a bioinformatics workload of mapping DNA cells of sequencing data [92] on GCP. The workload has 90 independent tasks, each with a relatively short duration (15 minutes). Each task requires a powerful multi-core CPUs for parallelization. We use GCP's latest C3 generation of compute instance, c3-highcpu-88. In this workload, interrupted tasks need to be recomputed entirely after recovery. We use the average task duration as the changeover delay (see Table 5.4).

Data Analytics. We run Apache Spark [167] (v3.2.0) on a widely-used benchmark, TPC-DS [108]. We use scale factor 1000 to generate 300 GB of data on a 64-core CPU instance (r5.16xlarge). The data is stored on a persistent disk, which is attachable for future instances. We run all queries 10 times. Similar to the bioinformatics workload, each query needs to start over if interrupted. We add a weighted average of query runtimes (7 mins) as progress loss into the changeover delay.

5.7.2.2 Results

We consider two different deadlines (job fractions 90% and 75%) for each workload. We first present detailed cost breakdowns for the ML workload with loose deadlines in Figure 5.17. Uniform Progress achieves 48% cost savings compared to only using on-demand. It outperforms Greedy (15%) and approaches the optimal (55%). Similar patterns are observed in the other two workloads. We show the cost savings in Table 5.5. For Bioinfomatics' c3-highcpu-88, the spot price is 91% cheaper than on-demand. This allows Uniform Progress to achieve 63% cost savings even when the deadline is tight and 84% savings when the deadline is loose. For the analytics workload, the spot price discount is much smaller (55%). In this case, Uniform Progress achieved 27% and 32% savings for tight and loose deadlines, respectively. Note, however, these savings still approach those achieved by the Omniscient (32% and 46%, for tight and loose deadlines).

5.8 Related Work

Spot pricing and availability modeling. AWS pioneered spot instances in 2009, using a bidding mechanism to monetize unused cloud capacity [112]. The pricing model has evolved to offer more stability, diminishing bidding, with other cloud providers adopting similar strategies, such as GCP's constant 30-day spot price [64], Oracle Cloud's fixed 50% discount for preemptible instances [116], and Azure's stable regional pricing [85]. While spot pricing is relatively stable, modeling spot availability remains challenging due to its black box nature. While prior work attempted to model preemption patterns [82] and employed ML prediction methods [162, 161, 70], we design our policy to be robust against potential changes in spot eviction strategies of the cloud providers.

Applications using spot instances. The cost-effective nature of spot instances has driven their adoption for savings. Frameworks like Bamboo [142], Spotnik [151], and Srifty [95], was developed for machine learning on spot instances. Narayanan *et al.*[109] showed significant reductions in machine learning training costs using spot instances across multiple clouds. CompuCache[168] leverages spot instances for in-memory data caching. However, preemptions can negatively impact application performance [11, 162], and deadline-constrained applications may struggle to effectively utilize spot instances.

Job scheduling with preemptions. Running jobs on preemptive devices is investigated on intermittent systems, where jobs can be interrupted due to sporadic harvestable energy. Many studies [61, 41, 77] focus on scheduling multiple real-time IoT tasks, due to the limited computation resources on these devices. Spot instances introduce preemption to resource-demanding batch jobs on clouds. From the cloud providers' perspective, existing work [76, 5, 78] investigates how to maximize revenue, or enhance runtime guarantees. For end-users, earlier studies explored bidding-based policies for bag of tasks with deadlines [165, 148, 99, 120], but these approaches are less applicable to current spot markets due to changes in pricing model. Recently, Snape [162] investigates using a mix of spot and on-demand instances for long-running services. It optimizes for SLO which require the number of instances available to be close to the target one at any time. It is different from deadline-sensitive batch jobs studied in this paper, where the job can stay idle for long periods, as long as it can meet the deadline.

5.9 Conclusion

Spot instances are economically appealing, but unreliable due to the preemptions. In this paper, we resolve a critical challenge of minimizing the cost for delay-sensitive jobs by utilizing a mix of spot and on-demand instances. Our work features a comprehensive analysis of spot instances and presents a theoretical framework to assess policies in both worst and average cases. This inspires the development of our proposed policy, Uniform Progress, which is simple, parameter-free, and effective without relying on assumptions of spot availability. Our empirical study using 3-month real-world traces demonstrates a significant improvement in cost savings compared to the greedy policy, closing gaps with the optimal policy by approximately 2× on both single or multiple instances. We also find that if cloud providers were willing to offer an oracle for the next spot instance's lifetime, it could further improve applications' cost efficiency, by enabling our Uniform Progress to approach the upper bound of cost savings. We implemented a prototype on top of SkyPilot and showcased the effectiveness of Uniform Progress on three real workloads, reducing the cost by 27%-84%. We open source the spot traces for future research.

Chapter 6

Use Case: AI Training and Serving on the Sky

The launch of SkyPilot and the strategic deployment of spot instances with "Can't Be Late" present a compelling opportunity to leverage the Sky for addressing real-world challenges. Rather than detailing complete research projects, this chapter highlights the preliminary outcomes of two initiatives that utilize the Sky. Our goal is to support the ongoing research and development in Sky Computing and intercloud brokers. In Section 6.1, we explore the development of a high-quality large language model (LLM) chatbot, Vicuna, on the Sky; in Section 6.2, we discuss the serving system we have developed for AI workloads using SkyPilot.

6.1 Vicuna: an Open-Source Chatbot

The release of GPT-4 [114] in March 2023 marked a significant event in AI, showcasing unprecedented capabilities in natural language processing. While GPT-4 is proprietary, it spurred interest in the methodologies and capabilities of such advanced systems. Meta's Llama [143, 144], a large publicly available model, did not have its chat capabilities explored until initial initiatives began to explore these capabilities by instruction tuning it.

Fine-tuning and evaluating such models require extensive computational resources. For instance, a 13B model typically needs 8 Nvidia A100 GPUs. Given the high cost and limited availability of these resources [139, 111], Sky Computing and SkyPilot offer a viable solution by enabling access to multi-cloud resources and cost-effective spot instances.

With SkyPilot, we were able to fine-tune the Llama model on user-shared conversations to create an open chatbot, Vicuna [42], which has become one of the earliest attempts to develop open chatbots that achieve impressive quality, based on both automatic evaluation with GPT-4 and human evaluations [169].

The workflow overview of Vicuna is shown in Figure 6.1. The detailed data processing, training, and serving code can be found in our open-source project, FastChat: `https://github.com/lm-sys/FastChat`. In the subsequent sections, we look at Vicuna's workflow and the substantial benefits brought by the integration with the Sky.

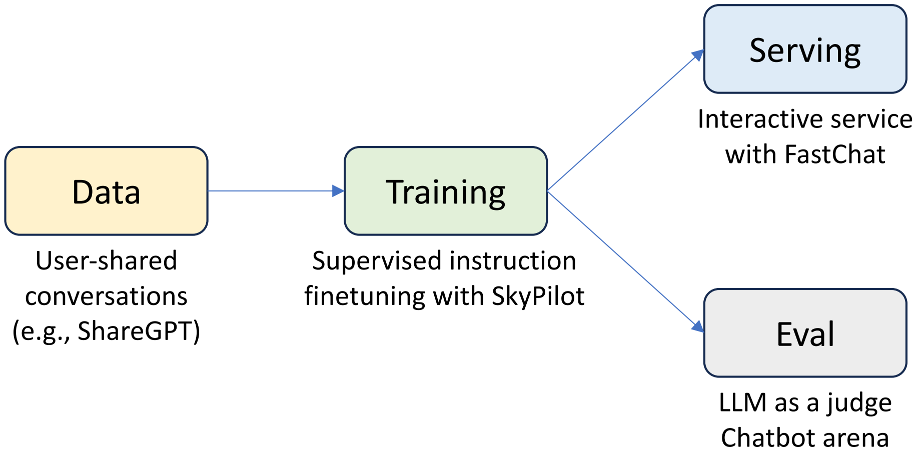

Data
User-shared conversations (e.g., ShareGPT)
Training
Supervised instruction finetuning with SkyPilot
Serving
Interactive service with FastChat
Eval
LLM as a judge
Chatbot arena

Figure 6.1: Workflow overview of Vicuna.

6.1.1 Dataset

As high-quality conversations were increasingly shared by users on ShareGPT.com, we recognized the potential for fine-tuning the foundational Llama model to develop a chatbot. We processed approximately 70K user-shared conversations, converting HTML back to markdown and filtering out inappropriate or low-quality samples. We also segmented lengthy conversations to fit the model's maximum context length.

6.1.2 Finetuning

Our training recipe builds on the previous instruction tuning technique, Alpaca [134]. In addition to Alpaca, we added several improvements:

1. **Multi-turn conversations.** The quality of the chatbot could be affected by the number of conversation turns it trains on. To incorporate multi-turn conversations, we customized the fine-tuning loss by masking the prompt tokens from users with zero, leaving only the responses.

2. **Memory optimizations.** To enable Vicuna to handle long contexts, we expanded the max context length from 512 in Alpaca to 2048, which substantially increases GPU memory requirements. We adopted gradient checkpointing [40] and flash attention [51, 50] to reduce the memory footprint.

The 40× larger dataset and 4× sequence length for training pose a considerable challenge in training expenses. To access the available GPUs across regions/clouds and reduce the cost of fine-tuning the Vicuna model, we launched the training of Vicuna with SkyPilot. The job specification is defined in the YAML interface mentioned in Section 4.1, see the example YAML in Figure 6.2. In the YAML, we use `resources` to specify high-level resource requirements for the job, including accelerators, using spot instances, and disk size required to cache the models. We also include a `file_mounts` section that

```
resources:
    accelerators: A100-80GB:8
    disk_size: 1024
    use_spot: true

file_mounts:
  /artifacts:
    name: MY_BUCKET_NAME
    mode: MOUNT

setup: |
  # Installation of dependencies for finetuning, such as
  # Huggingface, flash attention, etc.

run: |
  torchrun \
    --nnodes=$NUM_NODES \
    --nproc_per_node=$SKYPILOT_NUM_GPUS_PER_NODE \
    --master_port=12375 \
    --master_addr=$HOST_ADDR \
    --node_rank=${SKYPILOT_NODE_RANK} \
    train.py \
  # Additional arguments for training
```

Figure 6.2: YAML specification of Vicuna finetuning job.

mounts a cloud storage to the machine, so that the Vicuna training script can periodically save checkpoints to the cloud storage, and whenever recovery happens, the training program loads the latest checkpoint from the same path the bucket is mounted to and continues the training process.

Despite the optimization SkyPilot provides to launch GPU resources on the clouds, we also utilize the managed spot feature[1], with which SkyPilot will start a cheap spot controller that keeps monitoring the spot instances and recover the job from any preemption, see Section 4.2.2. It slashes costs for training the 7B model from $500 to around $140, and the 13B model from around $1K to $300.

During the training of Vicuna on spot instances, we realized the importance of scheduling deadline-sensitive jobs on spot instances, due to the tight release timeline for the model. It further motivated the in-progress "Can't Be Late" research at the time in Chapter 5 for cost savings under given deadlines, by mixing spot and on-demand instances for training.

6.1.3 Serving

After Vicuna is trained, we hosted it on the cloud, to ease the evaluation of the model by directly interacting with it. We built an interactive service in our FastChat project based on Gradio [2] and hosted the model with SkyPilot on the cloud. During the serving, we observed that serving LLMs has very different

[1] Detailed docs can be found at: https://skypilot.readthedocs.io/en/v0.5.0/examples/spot-jobs.html

requirements than existing web services, including longer computational times, more compute-intensive demands, and harder scalability due to the availability of accelerators, which led to our later development of the SkyPilot Serve in Section 6.2.

6.1.4 Evaluation

While largely overlooked by existing LLM benchmarks, human preferences serve as a direct measure of a chatbot's utility in open-ended, multi-turn human-AI interactions. To bridge this gap, we introduce two novel benchmarks expressly tailored to assess human preferences:

1. **LLM as a judge.** We created 80 benchmark questions for 8 common categories and asked GPT-4 to rate the preference of each pair of models.
2. **Chatbot arena:** We created a crowdsourcing benchmark platform[2] featuring anonymous battles, where users directly interact with two anonymous models and rate them.

The details of the evaluations are described and analyzed in our previous paper [169].

6.1.5 Benefits of the Sky

With the entire Vicuna workflow launched with SkyPilot, we identified three major benefits for running AI workloads on the Sky:

Packaging and running on any cloud. SkyPilot's user-friendly interface simplifies the configuration and deployment process, ensuring reproducibility and ease of use across different cloud environments. This feature significantly enhances the accessibility and scalability of AI projects, reducing the barrier to entry for utilizing advanced computational resources.

High GPU availability. SkyPilot's ability to integrate with multiple clouds enables it to automatically locate the best available GPU resources, thereby minimizing the need for manual intervention and ensuring efficient resource utilization.

Cost reduction. By automatically selecting the least expensive options for computational resources across different zones and regions, and facilitating the use of spot instances with robust recovery mechanisms, SkyPilot reduces training costs by approximately threefold. This economic efficiency makes large-scale AI training more accessible and sustainable.

[2] The chatbot arena can be found at `https://chat.lmsys.org`

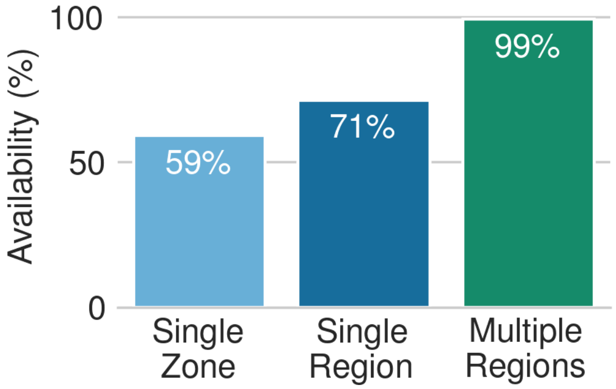


Figure 6.3: Availability comparison for a service hosted on a single zone, a single region, and multiple regions.

6.2 SkyPilot Serve: a System for Serving AI across Clouds

With the rapid expansion of large-scale AI models, the demand for serving AI models has significantly increased, introducing several unprecedented challenges such as GPU shortages and escalating resource costs. Existing systems, originally designed for normal web services, often fall short in addressing these issues effectively.

For instance, hosted Kubernetes clusters on cloud platforms like Amazon Elastic Kubernetes Service (EKS) [8] and Google Kubernetes Engine (GKE) [68], are typically confined to a single region, if not a single zone, within the respective cloud infrastructure. This limitation restricts the accessible resource pool, severely impacting system performance and scalability.

Figure 6.3 illustrates the variations in service availability when a system has access to resources from a single zone, a single region, or multiple regions. Our observations indicate that the more extensive the resource pool a serving system can tap into, the higher the service availability it achieves. This observation underscores the need for the development of a serving system on the Sky.

We built the serving system on top of SkyPilot and open sourced it in the same project[3]. Due to the across-cloud nature of the broker, SkyPilot, the serving system can easily support scaling the services across multiple locations, including different regions and clouds.

6.2.1 Architecture

SkyPilot offers a good starting point for building the serving system on the Sky as it already has the abstraction to run workloads on any cloud. Figure 6.4 shows the architecture of the system. Whenever a service is launched with SkyPilot Serve, a lightweight controller is started to monitor and manage all the service replicas. The controller contains three major components: a load balancer, a replica manager, and an autoscaler.

[3] The detailed docs for SkyPilot Serve can be found at: `https://skypilot.readthedocs.io/en/v0.5.0/serving/sky-serve.html`.

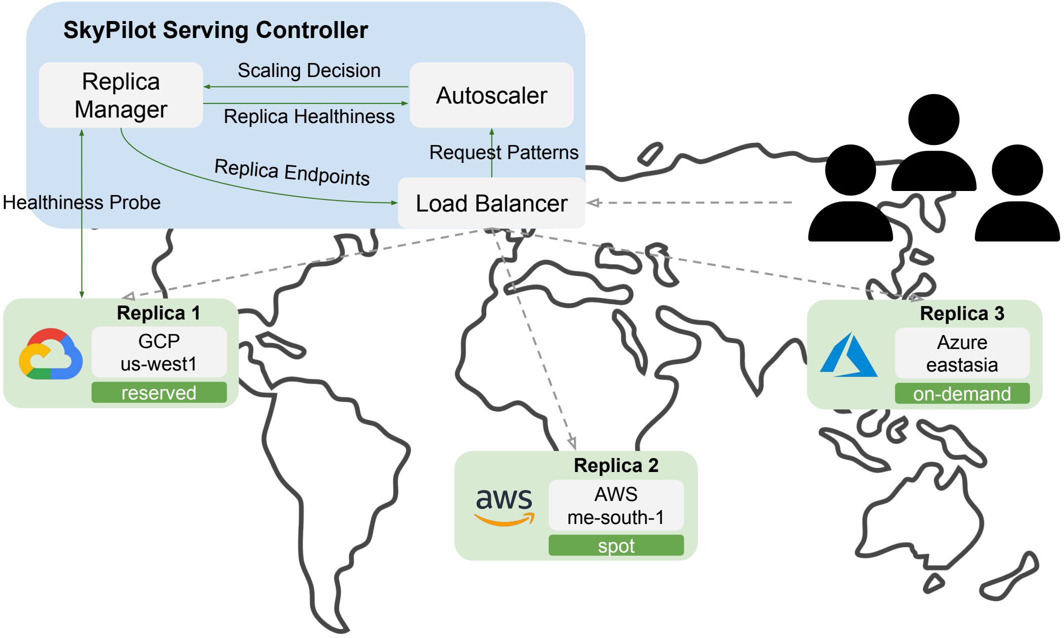


Figure 6.4: Architecture of SkyPilot Serve.

Load balancer. The load balancer exposes a single endpoint to the end-users and directs users' requests to one of the healthy replica endpoints. The information on the healthy replicas is retrieved from the replica manager with a REST API. We collocate the load balancer with the serving controller for convenience, but the load balancer is very modularized, so it can also be started outside a serving controller.

Autoscaler. The autoscaler component is the brain of the controller, which makes decisions about when to scale up and down the service replicas. It collects information from both the replica manager about the current replicas and the load balancer about the request patterns. With this information, the autoscaler generates plans for scaling to satisfy the service level objective (SLO).

Replica manager. The replica manager is the component that directly handles the life cycle of service replicas. It is in charge of monitoring and managing all the service replicas.

For monitoring, the replica manager continuously probes any existing service replicas associated with the current serving job to check their health and detect abnormalities.

For managing, when a new request for scale up is received from the autoscaler, the replica manager will submit a service job to the inter-cloud broker, SkyPilot. With the optimization of SkyPilot, the service replica will be launched on the cheapest available location from any region or cloud; when scale down is triggered by the autoscaler, the replica manager can also invoke SkyPilot's API to terminate the replica. This broker significantly simplifies the life cycle management of the service replicas in the implementation of the serving controller.

```
service:
  readiness_probe: /v1/health
  replica_policy:
    min_replicas: 0
    max_replicas: 10
    target_qps_per_replica: 2.5

# Fields below describe each replica.
resources:
  ports: 8080
  accelerators: {L4:8, A10g:8, A100:4}

setup: |
  # Commands for setting up dependencies

run: |
  # Commands for starting the services
```

Figure 6.5: YAML interface for a service.

6.2.2 Interface

To provide a simple interface, we built it directly on top of the SkyPilot job specification, as a service on each replica is equivalent to a job with an endpoint exposed. We show an example YAML interface for a service, in Figure 6.5.

In addition to the original SkyPilot job specification, the service YAML includes an additional `service` field, where a user can specify the requirements for the service. It mainly contains two sections:

1. `readiness_probe` defines the path that the service controller should use to access the replicas through HTTP requests for checking their healthiness. The path depends on the API a user's program exposes for healthiness checks.

2. `replica_policy` contains configurations for the autoscaler about how it should scale the replicas based on the current replica statuses and request patterns. In the example, we show several basic configurations, including the minimum/maximum number of replicas and the target queries per second per replica, but it can have more advanced configurations when more customized autoscaler policies are added.

In the example, `ports` and `accelerators` in the `resources` field are also specified. `Ports` requires each replica to expose port 8080 as the endpoint for the service running on it so that each healthiness probe and user request are sent to the endpoint. For `accelerators`, multiple candidate accelerators (8 L4, 8 A10g, or 4 A100) are allowed for each replica, so that the service can utilize the multiple resource pools in different locations as well as the resource pools for different accelerator types.

6.2.3 Deployment Experience

SkyPilot Serve has garnered much attention since its first release, due to its capability to maximize resource availability with the Sky and potential cost savings it can bring by using spot instances for serving AI. Below, we discuss key aspects of the deployment experience and learnings in more detail:

Resource availability. Unlike traditional web services, AI services are compute-intensive and require high-end GPUs. A critical challenge is effectively leveraging available resources during periods of GPU shortages, making serving both scalable and cost-efficient. Sky Computing fits well into this scenario, as it unifies resources from different providers. With the support of Kubernetes, SkyPilot can further utilize in-house resources, integrating resource pools from both on-premise facilities and the clouds.

Flexibility and customizability. The world of AI is fast-evolving, necessitating a serving infrastructure that can quickly adapt to changing technologies and requirements. Designed with modularity at its core, SkyPilot Serve enables seamless integration with various inference engines, such as vLLM [84], Nvidia Triton [145], and Huggingface TGI [73], as well as models like Vicuna [42], Llama [143, 144], or Mistral [81]. This flexibility allows users to deploy a wide range of AI applications without being tied to specific technologies. Additionally, the system includes a customizable load balancer and autoscaler, which can be tailored to meet specific operational demands or performance metrics. For instance, one organization using SkyPilot Serve implemented a load-balancing policy that distributes loads across multiple replicas effectively. Users can also define custom rules for scaling up or down based on actual load, ensuring efficient resource use by prioritizing prepaid or reserved resources.

Cold start time. Minimizing the cold start time is critical for serving AI workloads, as it directly impacts how quickly the system can scale service replicas in response to traffic spikes. For example, if a replica takes 20 minutes to initialize, the system will fail to meet the service level objective (SLO) during that time. Cold start time typically involves provisioning the replica, downloading or reading the model weights, and setting up and initializing the inference engine. The challenge intensifies with larger models; for instance, a 70B large language model might occupy 140GB of storage, significantly prolonging the time to download and read model weights. SkyPilot Serve has made preliminary attempts to reduce the cold start time through containerization and the use of custom cloud images, but there remains substantial room for optimization.

Leveraging spot replicas. In Chapter 5, we explored the potential of leveraging spot instances for deadline-sensitive batch jobs, devising a policy to optimize costs relative to specific deadlines. In a serving scenario, spot instances also offer significant benefits. Unlike batch jobs, serving is stateless, making it easier to migrate and recover a service replica during spot preemption. However, serving presents unique challenges, such as a stringent need for availability and service level guarantees. A serving system must provide some redundancy to accommodate potential spot preemptions. The policies for the load balancer and autoscaler merit further investigation when incorporating spot instances.

Heterogeneous accelerators and optimization. Despite the management of heterogeneous resource types, spot and on-demand instances, supporting different accelerators within a single service also presents a compelling direction for system design. While a service may run on various accelerators, it must account for performance differences among the devices. For instance, A100 GPUs may be more efficient for processing a large volume of batched requests, whereas L4 GPUs might be more cost-effective during periods of low traffic. An optimizer that can automatically select the most appropriate GPU type for a given request load could significantly enhance the cost efficiency of the serving system.

6.3 Conclusion

This chapter has explored the use of SkyPilot and the concept of Sky Computing in facilitating the development and deployment of large language models (LLMs), particularly focusing on the Vicuna chatbot and AI workload serving systems. Through detailed discussions on the use cases of training and serving AI models, we have highlighted several key advancements and lessons learned that showcase the potential of Sky Computing and SkyPilot.

The development of Vicuna demonstrated the practical benefits of leveraging resources across clouds to address the high computational demands of training AI models. By utilizing spot instances and optimizing resource allocation, SkyPilot enabled significant cost reductions while maintaining high availability and scalability. This approach not only democratizes access to cutting-edge AI technologies by lowering financial barriers but also enhances the efficiency and sustainability of AI research and development.

In the serving domain, we build SkyPilot Serve as a framework capable of managing AI services across multiple clouds. It mitigates critical challenges such as resource shortages, and cost efficiency. The flexibility of an intercloud broker in handling various infrastructural challenges and its adaptability to a range of AI applications underscore its potential as a cornerstone in the future landscape of AI serving systems. Future directions for research may focus on further reducing cold start times, enhancing the efficiency of resource utilization, and refining the integration of spot instances in serving scenarios. Additionally, exploring the potential of heterogeneous accelerators and optimizing the balance between performance and cost will be pivotal in advancing the state-of-the-art in AI serving infrastructure.

In summary, this chapter underscores the relevance and potential of Sky Computing in advancing AI technologies. AI serves as an ideal starting point for realizing the potential of Sky Computing due to its intensive computational requirements and rapid innovation cycles. There is a promising path ahead for researchers exploring this intersection, with opportunities to extend the learnings from AI workloads to other domains. We believe that the insights gained from deploying intercloud brokers for AI can be generalized to other workloads, potentially catalyzing the growth and evolution of Sky Computing.

Chapter 7

Conclusion

This dissertation has significantly extended and explored the rising concept of Sky Computing over clouds. Through the introduction of intermediate layers, intercloud brokers, we have explored how these innovations could fundamentally transform user interactions with cloud services and influence the economics of the cloud market.

As a preliminary step in exploring the feasibility and implications of Sky Computing, this dissertation unveiled novel insights into cloud computing introduced by the Sky:

- The development of intercloud brokers, facilitating emerging compatible services, has proven to be plausible. These brokers ease workload migration across clouds.
- By abstracting away the underlying clouds, Sky Computing introduces new dimensions for job optimizations, leading to enhanced cost efficiency.
- Sky Computing is not merely a long-term vision that transforms the cloud market; it already can deliver tangible benefits to real-world applications and cloud users.

We conclude this dissertation by reflecting on lessons learned and identifying potential areas for future research.

7.1 Lessons Learned

Early signals of the Sky. As described in the Sky Computing, we expect that the intercloud brokers will build a two-sided market where an intercloud broker helps match jobs offered by users and services offered by clouds. During the deployment of the system, we observed some early indicators from both sides:

- **Clouds' perspective:** Initially supporting only AWS, Azure, and Google Cloud, SkyPilot's user base expanded, prompting specialized GPU cloud providers to integrate with our system. The system now supports 12 public clouds, with 8 of them contributed by the cloud providers or the community, which underscores the growing incentive for providers to participate in Sky Computing.

- **Users' perspective:** An organization that initially used a single cloud provider expanded its infrastructure to include multiple clouds managed by SkyPilot. This transition was seamless, requiring no modifications to existing job setups, and facilitated access to specialized AI clouds, which do not offer a significant amount of services but just a service offering GPU resources and compatible with the major clouds.

Both perspectives offer early validations of our vision for Sky Computing. Firstly, the broker enables users to easily migrate their workloads across clouds, so that a user can easily choose services from multiple clouds. Secondly, a cloud with a smaller market share can still join the competition by focusing on specialized services instead of having to implement all services required by users.

Intercloud brokers can adopt online optimization. Originally, SkyPilot employed an optimizer that produced execution plans prior to job commencement. However, our "Can't Be Late" research shows significant potential for online optimization – adjusting strategies in response to dynamic market conditions and resource availability. Additionally, as more providers and services join the ecosystem increasing the diversity and volatility of the services, online optimization becomes more crucial.

Sky computing can be more decentralized with on-premises clusters involved. The rapid growth of AI and the GPU shortage lead to organizations starting to reserve or buy high-end GPUs to build on-premises clusters or in-house data centers. While Sky Computing is mainly about clouds, we should not limit the definition of "cloud" to a provider whose core business is to offer computing services. Instead, on-premise clusters or in-house data centers can also join the Sky as a special "cloud" that is owned by one or multiple users. That said, Sky Computing abstracts away not only the centralized cloud providers but also decentralized on-premises resources that are not intentionally built as a cloud. By adding support for job scheduling systems for on-premises clusters, Kubernetes, and VMWare vSphere, SkyPilot can create a Sky with both public clouds and on-premises clusters involved, where a user job can firstly utilize the on-premise resources before spilling over to clouds.

7.2 Future Work

Sky Computing presents numerous research opportunities that could further refine and expand its capabilities. Here are several key areas for future exploration:

7.2.1 Brokers Specialized for Various Workloads and Services

As an early step towards Sky Computing, SkyPilot initially focuses on compute-intensive batch jobs, supporting virtual machine services across different clouds. These virtual machines, while basic, have proved highly useful for users, particularly for AI workloads. However, SkyPilot only covers a limited range of compatible services. Below, we outline several services that could be further investigated and supported within Sky Computing:

Job submission platform. Many clouds offer job submission platforms, alongside the virtual machines, such as AWS Batch [22], Google Cloud Batch [62], etc. These systems are slightly higher level than virtual machines, as a user does not need to worry about the underlying clusters used for running their jobs. A broker could be developed to take jobs from users and submit them to any available job submission system. Several open questions remain: (1) how to construct a hierarchical job queue on top of each system's own queue, and (2) how to integrate job submission systems with virtual machine support, allowing the broker to interact at different levels of the cloud stack for the same purpose.

ML platforms and endpoints. Job submission platforms handle general job submissions, while many clouds also offer even more workload-specific services. Recently, ML-targeted platforms arose, such as AWS SageMaker [25], Google Vertex AI [69], and Azure OpenAI service [30]. These ML platforms can further simplify the finetuning and serving of AI workloads. Brokers can also be designed to interact with those ML platforms as the underlying compatible services as well.

Data storage. SkyPilot supports migrating job data using cloud services, such as Google Storage Transfer Service. Previous research, including SkyPlane, found that adding intermediate virtual machines can reduce cross-cloud data transfer times by 5x. CloudCast introduced optimization for replicating data across multiple cloud storages. A broker system could help optimize fine-grained data placement, preventing users from having to interact directly with numerous storage services. Cloud storage, such as AWS S3 [10], Google Cloud Storage [65], and Azure Blob Storage [26], are widely used for storing data. SkyPilot supports migrating job data using cloud services, such as Google Storage Transfer Service [67]. Previous research, including SkyPlane [79], found that adding intermediate virtual machines can reduce cross-cloud data transfer times by 5x. CloudCast [156] introduced optimization for replicating data across multiple cloud storages. A broker system for data could help optimize fine-grained data placement, preventing users from having to interact directly with numerous storage services.

7.2.2 Optimization for Jobs on Brokers

Various optimization metrics. Thus far, cost and time have been the primary optimization metrics explored in this dissertation's investigation of Sky Computing. However, real-world scenarios often demand more nuanced considerations. Below are several additional optimization metrics:

- **Reserved resources utilization.** During the deployment of SkyPilot, it was observed that many enterprise users have specific contracts with one or more cloud providers. These contracts often reserve a set number of resources that can be accessed at any time, immune to the fluctuations in demand on the clouds. These reservations, typically prepaid, directly impact cost efficiency. The optimizer should account for these factors and prioritize using these resources over potentially cheaper alternatives.

- **Carbon footprints.** Given the varying operational practices of cloud providers, who manage data centers in different regions using distinct energy sources, the carbon footprint of each service can significantly differ. With global warming as a critical concern, developing metrics to measure the carbon footprints of cloud services is essential. Brokers should proactively measure this metric, or cloud providers should reliably report these numbers.

- **Job-specific performance.** The performance of compatible services can vary significantly depending on the implementation and solutions adopted by each cloud. As demonstrated in SkyPilot experiments, even similar services from different providers can show varied job-specific performances, such as managed data analytics services and different accelerators. To automatically generate cost-effective plans for jobs, the optimizer should consider each service's performance along with cost. The challenge lies in benchmarking these performances economically or trusting the performance metrics provided directly by the clouds.

Trade-offs among different metrics. In SkyPilot, optimization is currently based on a single metric. However, in many real-world cases, a single metric cannot adequately reflect the trade-offs among multiple factors, such as cost versus time spent on a job. The policy design in "Can't Be Late" offers one approach to formulating this problem, setting a deadline constraint on a job while optimizing cost. As more metrics are introduced, more sophisticated methods for combining them should be explored.

Online optimization. With the deployment of SkyPilot, new dimensions for resource scheduling have emerged for jobs running on clouds. A broker can do more than static optimization to find the best locations and resources for launching a job. It can also apply online optimization during the job's execution by monitoring the resource market for better offerings, re-optimizing with migration overhead included, and migrating the job to new offerings if the new plan provides better cost-efficiency.

The policy design for deadline-sensitive jobs on spot and on-demand instances in "Can't Be Late" (Chapter 5) is one use case for online optimization. Potential topics for further exploration include:

- **Spot instances across multiple locations.** Our investigation initially focused on utilizing spot instances within a single availability zone. In a broader Sky scenario, leveraging spot availability across multiple zones, regions, or clouds is possible. Preliminary experiments in SkyPilot suggest that resource pools are likely isolated across multiple locations, leading to higher availability with more locations. Extending "Can't Be Late" to multiple locations adds a new dimension to the action space for the policy – migrating across locations. Given that spot availability across multiple locations is a black box to the user, this problem could also be considered as a multi-arm bandit problem, where each location has a hidden and dynamic availability ratio. We believe this extension could further enhance cost savings for a deadline-sensitive job.

- **Very large scale elastic job.** "Can't Be Late" was extended to multiple instances but with a relatively small number. While useful for many compute-intensive jobs, extremely large-scale jobs, such as the pretraining of large language models (LLMs), involving hundreds to thousands of instances, may require further policy design to reconfigure the proportion of spot and on-demand instances. Moreover, if the job is elastic, the total number of instances in a cluster can also dynamically change to better balance cost, computation time, and availability.

- **Live migration during execution.** Despite differences in pricing models and instance types, SkyPilot experiments have shown that using performance differences produced by compatible services across clouds (e.g., different accelerators like Nvidia GPUs and Google TPUs) can vary in cost-efficiency. However, due to demand spikes, a broker might not initially secure a resource with the best optimization metric, leading to the selection of a sub-optimal resource to start the job. During execution, if a better resource becomes available, an interesting direction would be to allow the broker to monitor these resources and determine if checkpointing and migrating the job to newly available resources is worthwhile, considering the overheads involved.

7.2.3 AI Serving on the Sky and Optimizations

While compute-intensive batch jobs serve as a good starting point, AI serving is another promising area of focus, particularly due to its stateless nature and ease of migration. AI typically requires longer response times due to extensive computational demands, making it more tolerant of network latency across different regions and clouds. These characteristics make AI serving well-suited for Sky Computing scenarios.

In developing SkyPilot Serve – a prototype for deploying AI serving – we explored several intriguing topics beyond the online optimization previously mentioned for general brokers:

Spot instances as service replicas with service level guarantee. In "Can't Be Late," we discussed policy design using spot instances for deadline-sensitive batch jobs to save costs. Serving workloads could similarly utilize spot instances, albeit with different performance metrics to maintain. In batch jobs, total job completion time is paramount, and periods without available instances are acceptable. Conversely, in serving, the need to guarantee a service level objective makes availability a critical concern. There should always be a minimum number of instances available to keep the service operational. This necessitates a distinct policy approach for handling spot preemptions and leveraging on-demand instances to ensure availability.

Leveraging heterogeneous accelerators. AI workloads require accelerators, such as Nvidia GPUs. While AI training typically needs a set of homogeneous GPUs, AI service replicas can be run on different GPUs. As these accelerators come from varied resource pools in the cloud, their availability and pricing can significantly differ, as can task performance. For example, a higher-end GPU may offer better throughput but is significantly more expensive and much less available on the cloud. To ensure service level objectives are met cost-effectively, a policy must be crafted to optimize the trade-off among performance, availability, and cost.

Connecting to existing serving systems and hierarchical optimizations. Serving is an established domain, supported by numerous existing systems capable of handling AI workloads, such as KServe and RayServe. However, these solutions are generally confined to a single region on a cloud. With SkyPilot Serve, we have developed a serving system capable of scaling across different regions and clouds based on bare virtual machine services, utilizing SkyPilot to provision and manage all service replicas. This necessitates a broker system that reimplements many features already supported by existing serving systems, such as observability. Exploring the possibility of connecting multiple serving systems across different regions and clouds through a single broker, which would present a unified endpoint to users, is a valuable avenue. This approach would require the broker to balance loads across multiple serving systems and manage service queues and metrics from them, involving two-level scheduling where the broker not only decides which serving system to route the load to but also implicitly controls the autoscaling of the underlying systems by sending different loads to them.

Bibliography

[1] 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.

[2] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. "Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild". In: *arXiv preprint arXiv:1906.02569* (2019).

[3] *Akka*. https://akka.io/.

[4] 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)*. 2017, pp. 469–482.

[5] Fadi Alzhouri, Anjali Agarwal, and Yan Liu. "Maximizing cloud revenue using dynamic pricing of multiple class virtual machines". In: *IEEE Transactions on Cloud Computing* 9.2 (2018), pp. 682–695.

[6] *Amazon Customer Reviews Dataset*. https://s3.amazonaws.com/amazon-reviews-pds/readme.html.

[7] *Amazon EC2 Spot customers*. https://aws.amazon.com/ec2/spot/customers/.

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

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

[10] *Amazon S3*. https://aws.amazon.com/s3/.

[11] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 735–751. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ambati.

[12] *Anthos*. https://cloud.google.com/anthos.

[13] *Apache Airflow*. https://airflow.apache.org/.

[14] *Apache Cassandra*. https://cassandra.apache.org/.

[15] *Apache jclouds*. https://jclouds.apache.org/.

[16] *Apache Kafka*. https://kafka.apache.org/.

[17] *Apache Libcloud*. https://libcloud.apache.org/.

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

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

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

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

[22] *AWS Batch*. https://aws.amazon.com/batch/.

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

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

[25] *AWS SageMaker*. https://aws.amazon.com/sagemaker/.

[26] *Azure Blob Storage*. https://azure.microsoft.com/en-us/services/storage/blobs/.

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

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

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

[30] *Azure OpenAI Service*. https://azure.microsoft.com/en-us/products/ai-services/openai-service.

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

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

[33] 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". In: *arXiv preprint arXiv:2108.07258* (2021).

[34] I. Buch, M. J. Harvey, T. Giorgino, D. P. Anderson, and G. De Fabritiis. "High-Throughput All-Atom Molecular Dynamics Simulations Using Distributed Computing". In: *Journal of Chemical Information and Modeling* 50.3 (2010), pp. 397–403. DOI: 10.1021/ci900455r.

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

[36] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. "State Management in Apache Flink: Consistent Stateful Distributed Stream Processing". In: *Proc. VLDB Endow.* 10.12 (Aug. 2017), pp. 1718–1729. ISSN: 2150-8097.

[37] *Cerebras*. https://cerebras.net/.

[38] Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E Gonzalez, Joseph M Hellerstein, Michael I Jordan, Anthony D Joseph, Michael W Mahoney, et al. "The sky above the clouds". In: *arXiv preprint arXiv:2205.07147* (2022).

[39] 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.

[40] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. "Training deep nets with sublinear memory cost". In: *arXiv preprint arXiv:1604.06174* (2016).

[41] Maryline Chetto. "Optimal scheduling for real-time jobs in energy harvesting computing systems". In: *IEEE Transactions on Emerging Topics in Computing* 2.2 (2014), pp. 122–133.

[42] Wei-Lin Chiang*, Zhuohan Li*, Zi Lin*, Ying Sheng*, Zhanghao Wu*, Hao Zhang*, Lianmin Zheng*, Siyuan Zhuang*, Yonghao Zhuang*, Joseph E. Gonzalez*, Ion Stoica*, and Eric P. Xing*. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. Mar. 2023. URL: https://lmsys.org/blog/2023-03-30-vicuna/.

[43] Navraj Chohan, Claris Castillo, Mike Spreitzer, Malgorzata Steinder, Asser N Tantawi, and Chandra Krintz. "See spot run: using spot instances for mapreduce workflows." In: *HotCloud* 10 (2010), pp. 7–7.

[44] *Cloud Computing with AWS*. https://aws.amazon.com/what-is-aws/.

[45] *Cloud Market Gets its Mojo Back; AI Helps Push Q4 Increase in Cloud Spending to New Highs*. https://www.srgresearch.com/articles/cloud-market-gets-its-mojo-back-q4-increase-in-cloud-spending-reaches-new-highs.

[46] *Cloud switching just got easier: Removing data transfer fees when moving off Google Cloud*. https://techcrunch.com/2024/01/11/google-says-itll-stop-charging-fees-to-transfer-data-out-of-google-cloud/.

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

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

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

[50] Tri Dao. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning". In: (2023).

[51] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness". In: *Advances in Neural Information Processing Systems*. 2022.

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

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

[54] *Docker*. https://github.com/docker.

[55] John Forrest and Robin Lougee-Heimer. "CBC user guide". In: *Emerging theory, methods, and applications*. INFORMS, 2005, pp. 257–277.

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

[57] José A.B. Fortes. "Sky Computing: When Multiple Clouds Become One". In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 2010, pp. 4–4. DOI: 10.1109/CCGRID.2010.136.

[58] *Free data transfer out to internet when leaving Azure*. https://azure.microsoft.com/en-us/updates/now-available-free-data-transfer-out-to-internet-when-leaving-azure/.

[59] *Free data transfer out to internet when moving out of AWS*. https://aws.amazon.com/blogs/aws/free-data-transfer-out-to-internet-when-moving-out-of-aws/.

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

[61] Hussein EL Ghor, Maryline Chetto, and Rafic Hage Chehade. "A real-time scheduling framework for embedded systems with environmental energy harvesting". In: *Computers & Electrical Engineering* 37.4 (2011), pp. 498–510.

[62] *Google Cloud Batch*. https://cloud.google.com/batch?hl=en.

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

[64] *Google Cloud Spot VM Pricing*. https://cloud.google.com/compute/docs/instances/spot#pricing.

[65] *Google Cloud Storage*. https://cloud.google.com/storage.

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

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

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

[69] *Google Vertex AI*. https://cloud.google.com/vertex-ai/docs/start/introduction-unified-platform.

[70] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons.
"Tributary: spot-dancing for elastic services with latency SLOs".
In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*.
Boston, MA: USENIX Association, July 2018, pp. 1–14. ISBN: 978-1-931971-44-7.
URL: https://www.usenix.org/conference/atc18/presentation/harlap.

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

[72] 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.
Boston, MA: USENIX Association, 2011, pp. 295–308.
URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[73] *Huggingface TGI*.
https://huggingface.co/docs/text-generation-inference/en/index.

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

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

[76] Syed M. Iqbal, Haley Li, Shane Bergsma, Ivan Beschastnikh, and Alan J. Hu.
"CoSpot: A Cooperative VM Allocation Framework for Increased Revenue from Spot Instances".
In: *Proceedings of the 13th Symposium on Cloud Computing*. SoCC '22.
San Francisco, California: Association for Computing Machinery, 2022, pp. 540–556.
ISBN: 9781450394147. DOI: 10.1145/3542929.3563499.
URL: https://doi-org.libproxy.berkeley.edu/10.1145/3542929.3563499.

[77] Bashima Islam and Shahriar Nirjon.
"Scheduling computational and energy harvesting tasks in deadline-aware intermittent systems".
In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*.
IEEE. 2020, pp. 95–109.

[78] Navendu Jain, Ishai Menache, Joseph Naor, and Jonathan Yaniv.
"Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters".
In: *ACM Transactions on Parallel Computing (TOPC)* 2.1 (2015), pp. 1–29.

[79] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. "Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays". In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1375–1389. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/jain.

[80] Qin Jia, Zhiming Shen, Weijia Song, Robbert Van Renesse, and Hakim Weatherspoon. "Supercloud: Opportunities and challenges". In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 137–141.

[81] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. *Mistral 7B*. 2023. arXiv: 2310.06825 [cs.CL].

[82] JCS Kadupitige, Vikram Jadhao, and Prateek Sharma. "Modeling The Temporally Constrained Preemptions of Transient Cloud VMs". In: *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*. HPDC '20. Stockholm, Sweden: Association for Computing Machinery, 2020, pp. 41–52. ISBN: 9781450370523. DOI: 10.1145/3369583.3392671. URL: https://doi-org.libproxy.berkeley.edu/10.1145/3369583.3392671.

[83] *Kubernetes*. https://github.com/kubernetes/kubernetes.

[84] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023.

[85] Sungjae Lee, Jaeil Hwang, and Kyungyong Lee. "SpotLake: Diverse Spot Instance Dataset Archive Service". In: *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 2022, pp. 242–255. DOI: 10.1109/IISWC55918.2022.00029.

[86] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. "Scaling Distributed Machine Learning with the Parameter Server". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 583–598. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[87] Shen Li, Yanli Zhao, Rohan Varma, Omkar Salpekar, Pieter Noordhuis, Teng Li, Adam Paszke, Jeff Smith, Brian Vaughan, Pritam Damania, et al. "PyTorch Distributed: Experiences on Accelerating Data Parallel Training". In: *Proceedings of the VLDB Endowment* 13.12 (2019).

[88] 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.

[89] Richard Liaw, Romil Bhardwaj, Lisa Dunlap, Yitian Zou, Joseph E Gonzalez, Ion Stoica, and Alexey Tumanov. "Hypersched: Dynamic resource reallocation for model development on a deadline". In: *Proceedings of the ACM Symposium on Cloud Computing*. 2019, pp. 61–73.

[90] Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. "Tune: A Research Platform for Distributed Model Selection and Training". In: *arXiv preprint arXiv:1807.05118* (2018).

[91] Hanqing Liu, Qiurui Zeng, Jingtian Zhou, Anna Bartlett, Bang-An Wang, Peter Berube, Wei Tian, Mia Kenworthy, Jordan Altshul, Joseph R Nery, et al. "Single-cell DNA methylome and 3D multi-omic atlas of the adult mouse brain". In: *Nature* 624.7991 (2023), pp. 366–377.

[92] Hanqing Liu, Qiurui Zeng, Jingtian Zhou, Anna Bartlett, Bang-An Wang, Peter Berube, Wei Tian, Mia Kenworthy, Jordan Altshul, Joseph R. Nery, Huaming Chen, Rosa G. Castanon, Songpeng Zu, Yang Eric Li, Jacinta Lucero, Julia K. Osteen, Antonio Pinto-Duarte, Jasper Lee, Jon Rink, Silvia Cho, Nora Emerson, Michael Nunn, Carolyn O'Connor, Zizhen Yao, Kimberly A. Smith, Bosiljka Tasic, Hongkui Zeng, Chongyuan Luo, Jesse R. Dixon, Bing Ren, M. Margarita Behrens, and Joseph R Ecker. "Single-cell DNA Methylome and 3D Multi-omic Atlas of the Adult Mouse Brain". In: *bioRxiv* (2023). DOI: 10.1101/2023.04.16.536509. eprint: https://www.biorxiv.org/content/early/2023/04/18/2023.04.16.536509.full.pdf. URL: %5Curl%7Bhttps://www.biorxiv.org/content/early/2023/04/18/2023.04.16.536509%7D.

[93] 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". en. In: *Nature* 598.7879 (Oct. 2021), pp. 120–128.

[94] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

[95] Liang Luo, Peter West, Pratyush Patel, Arvind Krishnamurthy, and Luis Ceze. "SRIFTY: Swift and Thrifty Distributed Neural Network Training on the Cloud". In: *Proceedings of Machine Learning and Systems* 4 (2022), pp. 833–847.

[96] 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)*. 2022, pp. 303–320.

[97] Aniruddha Marathe, Rachel Harris, David K Lowenthal, Bronis R de Supinski, Barry Rountree, and Martin Schulz. "Exploiting redundancy and application scalability for cost-effective, time-constrained execution of hpc applications on amazon ec2". In: *IEEE Transactions on Parallel and Distributed Systems* 27.9 (2015), pp. 2574–2588.

[98] A. Matsunaga, J. Fortes, K. Keahey, and M. Tsugawa. "Sky Computing". In: *IEEE Internet Computing* 13.05 (Sept. 2009), pp. 43–51. ISSN: 1941-0131. DOI: 10.1109/MIC.2009.94.

[99] Ishai Menache, Ohad Shamir, and Navendu Jain. "On-demand, Spot, or Both: Dynamic Resource Allocation for Executing Batch Jobs in the Cloud". In: *11th International Conference on Autonomic Computing (ICAC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 177–187. ISBN: 978-1-931971-11-9. URL: https://www.usenix.org/conference/icac14/technical-sessions/presentation/menache.

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

[101] Stuart Mitchell, Michael OSullivan, and Iain Dunning. "PuLP: a linear programming toolkit for python". In: *The University of Auckland, Auckland, New Zealand* 65 (2011).

[102] *MLFlow*. https://mlflow.org/.

[103] *MongoDB*. https://github.com/mongodb/mongo.

[104] 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.

[105] 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: USENIX Association, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/nishihara.

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

[107] *MySQL*. https://www.mysql.com/.

[108] 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. Seoul, Korea: VLDB Endowment, 2006, pp. 1049–1058.

[109] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. "Analysis and Exploitation of Dynamic Pricing in the Public Cloud for ML Training". In: *Workshop on Distributed Infrastructure, Systems, Programming, and AI*. Aug. 2020. URL: https://www.microsoft.com/en-us/research/publication/analysis-and-exploitation-of-dynamic-pricing-in-the-public-cloud-for-ml-training/.

[110] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. "Analysis and Exploitation of Dynamic Pricing in the Public Cloud for ML Training". In: *VLDB DISPA Workshop 2020* (). URL: https://par.nsf.gov/biblio/10213411.

[111] *Navigating the High Cost of AI Compute*. https://a16z.com/2023/04/27/navigating-the-high-cost-of-ai-compute/.

[112] *New Amazon EC2 Spot pricing model: Simplified purchasing without bidding and fewer interruptions*. https://aws.amazon.com/blogs/compute/new-amazon-ec2-spot-pricing/.

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

[114] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL].

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

[116] *Oracle Computing Pricing*. https://www.oracle.com/cloud/compute/pricing/.

[117] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. "fairseq: A Fast, Extensible Toolkit for Sequence Modeling". In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*. Minneapolis, Minnesota: Association for Computational Linguistics, June 2019, pp. 48–53. DOI: 10.18653/v1/N19-4009. URL: %5Curl%7Bhttps://aclanthology.org/N19-4009%7D.

[118] 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. Philadelphia, PA, USA: Association for Computing Machinery, 2022, pp. 587–601. ISBN: 9781450392495.

[119] 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". In: (2017).

[120] Deepak Poola, Kotagiri Ramamohanarao, and Rajkumar Buyya. "Fault-tolerant Workflow Scheduling using Spot Instances on Clouds". In: *Procedia Computer Science* 29 (2014). 2014 International Conference on Computational Science, pp. 523–533. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2014.05.047. URL: https://www.sciencedirect.com/science/article/pii/S1877050914002245.

[121] *PostgreSQL*. https://www.postgresql.org/.

[122] *Presto*. https://github.com/prestodb/presto.

[123] *Pretraining RoBERTa using your own data*. https://github.com/facebookresearch/fairseq/blob/main/examples/roberta/README.pretraining.md.

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

[125] *Redis*. https://github.com/redis/redis.

[126] *s3fs*. https://github.com/s3fs-fuse/s3fs-fuse.

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

[128] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[129] Alexander Sergeev and Mike Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv preprint arXiv:1802.05799* (2018).

[130] *SkyPilot*. https://github.com/skypilot-org/skypilot.

[131] *SparkSQL*. https://spark.apache.org/sql/.

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

[133] Ion Stoica and Scott Shenker. "From Cloud Computing to Sky Computing". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '21. Ann Arbor, Michigan: Association for Computing Machinery, 2021, pp. 26–32. ISBN: 9781450384384. DOI: 10.1145/3458336.3465301. URL: https://doi.org/10.1145/3458336.3465301.

[134] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. *Stanford Alpaca: An Instruction-following LLaMA model*. https://github.com/tatsu-lab/stanford_alpaca. 2023.

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

[136] *Terraform*. https://www.terraform.io/.

[137] Douglas Thain, Todd Tannenbaum, and Miron Livny.
"Distributed computing in practice: the Condor experience".
In: *Concurrency and computation: practice and experience* 17.2-4 (2005), pp. 323–356.

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

[139] *The Desperate Hunt for the A.I. Boom's Most Indispensable Prize*.
https://www.nytimes.com/2023/08/16/technology/ai-gpu-chips-shortage.html.

[140] *The Desperate Hunt for the A.I. Boom's Most Indispensable Prize*.
https://www.nytimes.com/2023/08/16/technology/ai-gpu-chips-shortage.html.

[141] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu.
"Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs".
In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.
Boston, MA: USENIX Association, Apr. 2023, pp. 497–513. ISBN: 978-1-939133-33-5.
URL: https://www.usenix.org/conference/nsdi23/presentation/thorpe.

[142] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu.
"Bamboo: Making Preemptible Instances Resilient for Affordable Training of Large DNNs".
In: *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*.
Boston, MA: USENIX Association, Apr. 2023, pp. 497–513. ISBN: 978-1-939133-33-5.
URL: https://www.usenix.org/conference/nsdi23/presentation/thorpe.

[143] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample.
LLaMA: Open and Efficient Foundation Language Models. 2023.
arXiv: 2302.13971 [cs.CL].

[144] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al.
"Llama 2: Open foundation and fine-tuned chat models".
In: *arXiv preprint arXiv:2307.09288* (2023).

[145] *Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution.*
https://github.com/triton-inference-server.

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

[147] *Vantage Cloud Cost Breakdown Report*. https://www.vantage.sh/cloud-cost-report/2023-q1.

[148] Prateeksha Varshney and Yogesh Simmhan. "AutoBoT: Resilient and Cost-Effective Scheduling of a Bag of Tasks on Spot VMs". In: *IEEE Transactions on Parallel and Distributed Systems* 30.7 (2019), pp. 1512–1527. DOI: 10.1109/TPDS.2018.2889851.

[149] 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)*. 2016, pp. 363–378.

[150] 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*. 2015, pp. 1–17.

[151] Marcel Wagenländer, Luo Mai, Guo Li, and Peter Pietzuch. "Spotnik: Designing distributed machine learning for transient cloud resources". In: *Proceedings of the 12th USENIX Conference on Hot Topics in Cloud Computing*. 2020, pp. 4–4.

[152] 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/.

[153] Joe Weinman. "Intercloudonomics: Quantifying the Value of the Intercloud". In: *IEEE Cloud Computing* 2.5 (Sept. 2015), p. 4047.

[154] *What is GDPR, the EU's new data protection law?* https://gdpr.eu/what-is-gdpr/.

[155] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2012. ISBN: 1449311520.

[156] Sarah Wooders, Shu Liu, Paras Jain, Xiangxi Mo, Joseph E. Gonzalez, Vincent Liu, and Ion Stoica. "Cloudcast: High-Throughput, Cost-Aware Overlay Multicast in the Cloud". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 281–296. ISBN: 978-1-939133-39-7. URL: https://www.usenix.org/conference/nsdi24/presentation/wooders.

[157] Zhanghao Wu, Wei-Lin Chiang, Ziming Mao, Zongheng Yang, Eric Friedman, Scott Shenker, and Ion Stoica. "Can't Be Late: Optimizing Spot Instance Savings under Deadlines". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024. URL: https://www.usenix.org/conference/nsdi24/presentation/wu-zhanghao.

[158] 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. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 292–308.

ISBN: 9781450323888. DOI: 10.1145/2517349.2522730.
URL: https://doi.org/10.1145/2517349.2522730.

[159] 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)*. Oakland, CA: USENIX Association, May 2015, pp. 543–557. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/wu.

[160] Tian Xia, Zhanghao Wu, Ziming Mao, and Zongheng Yang. *Introducing SkyServe: 50% Cheaper AI Serving on Any Cloud with High Availability*. https://blog.skypilot.co/introducing-sky-serve/. Feb. 2024.

[161] Fangkai Yang, Bowen Pang, Jue Zhang, Bo Qiao, Lu Wang, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, Íñigo Goiri, Eli Cortez, Senthil Baladhandayutham, Victor Rühle, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. "Spot Virtual Machine Eviction Prediction in Microsoft Cloud". In: *Companion Proceedings of the Web Conference 2022*. WWW '22. Virtual Event, Lyon, France: Association for Computing Machinery, 2022, pp. 152–156. ISBN: 9781450391306. DOI: 10.1145/3487553.3524229. URL: https://doi-org.libproxy.berkeley.edu/10.1145/3487553.3524229.

[162] Fangkai Yang, Lu Wang, Zhenyu Xu, Jue Zhang, Liqun Li, Bo Qiao, Camille Couturier, Chetan Bansal, Soumya Ram, Si Qin, Zhen Ma, Íñigo Goiri, Eli Cortez, Terry Yang, Victor Rühle, Saravan Rajmohan, Qingwei Lin, and Dongmei Zhang. "Snape: Reliable and Low-Cost Computing with Mixture of Spot and On-Demand VMs". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 631–643. ISBN: 9781450399180. DOI: 10.1145/3582016.3582028. URL: https://doi-org.libproxy.berkeley.edu/10.1145/3582016.3582028.

[163] 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)*. Boston, MA: USENIX Association, Apr. 2023, pp. 437–455. ISBN: 978-1-939133-33-5. URL: https://www.usenix.org/conference/nsdi23/presentation/yang-zongheng.

[164] 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”. en. In: *Nature* 598.7879 (Oct. 2021), pp. 103–110.

[165] Murtaza Zafer, Yang Song, and Kang-Won Lee. “Optimal Bids for Spot VMs in a Cloud for Deadline Constrained Jobs”. In: *2012 IEEE Fifth International Conference on Cloud Computing*. 2012, pp. 75–82. DOI: 10.1109/CLOUD.2012.59.

[166] 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*. USENIX Association. 2012, pp. 2–2.

[167] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Commun. ACM* 59.11 (Oct. 2016), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/2934664. URL: https://doi-org.libproxy.berkeley.edu/10.1145/2934664.

[168] Qizhen Zhang, Phil Bernstein, Daniel S. Berger, Badrish Chandramouli, Boon Thao Loo, and Vincent Liu. “CompuCache: Remote Computable Caching using Spot VMs”. In: *Conference on Innovative Data Systems Research (CIDR 2022)*. Jan. 2022. URL: https://www.microsoft.com/en-us/research/publication/compucache-remote-computable-caching-using-spot-vms/.

[169] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. “Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena”. In: *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. 2023. URL: https://openreview.net/forum?id=uccHPGDlao.

[170] 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)*. Carlsbad, CA: USENIX Association, July 2022, pp. 559–578. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

[171] 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. Boston, MA, USA: USENIX Association, 2017, pp. 283–298. ISBN: 9781931971379.