Realizing value in shared compute infrastructures

Andrew Chung

CMU-CS-22-151

December 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Gregory R. Ganger, Chair
Phillip B. Gibbons
George Amvrosiadis
Carlo Curino (Microsoft)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by Intel ISTC-CC. Additional support was provided by Microsoft and members and companies of the PDL Consortium (Alibaba, Amazon, Google, HPE, Hitachi, IBM, Intel, Meta, NetApp, Oracle, Pure, Salesforce, Samsung, Seagate, Two Sigma, and Western Digital).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Cluster Scheduling, Resource Management, Cloud Computing

For my family.

Abstract

As company operations become increasingly digitized, the demand to process data efficiently and cost-effectively has been ever-growing. More and more companies are therefore moving their workloads off of dedicated, silo-ed clusters in favor of more cost-efficient, shared data infrastructures, e.g., public and private clouds. These shared data infrastructures are often deployed on highly heterogeneous servers, are multi-tenant with server resources shared across multiple organizations, and serve widely diverse workloads ranging from batch analytics jobs to consumer-facing services with stringent *service level objectives (SLOs)*. Both users and operators of such shared data infrastructures strive to optimize for *value*. Users look to complete their tasks in an efficient and timely manner without having to pay large amounts of money, while operators seek to satisfy the demands of their customers to increase adoption and lower turnover, all the while without sacrificing cluster operation costs and overhead.

This dissertation presents two case studies that allows users to improve value-attainment when running their workloads in shared data infrastructures in Tributary and Stratus. Tributary is an elastic control system that embraces the uncertain nature of transient cloud resources to manage elastic long-running services with latency SLOs more robustly and more cost-effectively. Stratus is a cluster scheduler specialized for orchestrating batch job execution on virtual clusters focusing primarily on dollar cost considerations: since resources in virtual clusters are charged-for while allocated, Stratus aggressively packs tasks onto machines, guided by job run time estimates, such that allocated resources remain highly utilized.

This dissertation presents two more case studies that allow cluster operators to attain value in Wing and Talon. Inter-job dependencies pervade today's shared data infrastructures, yet are often invisible to cluster schedulers. The Wing dependency profiler analyzes job and data provenance logs to find hidden inter-job dependencies, characterizes them, and provides improved guidance to cluster schedulers and workflow managers to help users attain more value. Talon is one such workflow manager that uses information provided by Wing to load-shift batch analytics jobs to off-peak hours, thereby allowing cluster operators to save on infrastructure operation costs through reduced machines managed and usage of lower-cost, transient resources from the cloud.

Acknowledgments

I have gotten help from so many people along my 6.5 year (wow) journey at CMU, and I could not have completed my degree without each and every one of them. While there is certainly no way to fully express my gratitude, here, I wish to express my thanks, however inadequately, to those who helped me along the way.

First, I would like to thank my advisor Greg Ganger. Greg has helped me grow both as a systems researcher and as a person. I'm always surprised by how Greg can quickly cut to the core of research problems, identify potential issues and bottlenecks, and provide insights that immensely improve the quality of our research projects. Throughout my study, Greg has always been patient and supportive even when I ramble on incoherently about my research findings. Having him as my advisor is one of the best decisions I have ever made.

Next, I would like to thank Carlo Curino and Subru Krishnan, my mentors at Microsoft Gray Systems Lab (GSL). Not only are Carlo and Subru always helpful and supportive in guiding me in my research, working with them also provided me a grounded view of real systems operating at scale in industry. Even prior to starting my Ph.D. studies, resource management papers from GSL always fascinated and inspired me, and getting to work with Carlo and Subru was really a dream come true.

I'm grateful to Phil and George for joining my thesis committee, for helping me out with my Speaking and Writing skills requirements, and for giving me insightful feedback on my research throughout the years. It was also a pleasure to work with them as a TA in 15-712 and 15-719, and I will miss our grading parties.

I have been very fortunate to have opportunities to collaborate with excellent fellow students and colleagues throughout my period at CMU. Aaron Harlap has been one of my closest collaborators throughout my studies, and I am thankful to him and Alexey Tumanov for kick-starting my research career by on-boarding me to the Proteus project. Aaron's dry humor and the many nights we spent in CIC trying to make paper deadlines will sorely be missed. I also thank Jun Woo Park for introducing me to the cluster scheduling project, for helping me with job run time predictions on the Stratus project, and for the brainstorming sessions we've had on how to improve scheduling "goodness."

The Parallel Data Lab (PDL) has been a supremely nuturing environment for me as an early-career researcher, and I am grateful to all the faculty, staff, and students who contribute to keep the lab running smoothly. The feedback and inspiration we get from weekly meeting presentations and conversations with industry guests at the PDL retreat and Spring Visit days have all been invaluable to my growth as a researcher.

Specifically, I would like to thank PDL faculty members Majd Sakr, Andy Pavlo, David O'Hallaron, Garth Gibson, and David Andersen for the feedback they've provided to help me improve my research projects and presentation skills. I would also like to thank fellow PDL student members Michael Kuchnik, Rajat Kateja, Angela Jiang, Jack Kosaian, Daniel Wong, Abutalib Aghayev, Saurabh Kadekodi, Hojin Park, Sara McAllister, Lin Ma, Ziqiang Feng, Henggang Cui, Jinliang Wei, and Huanchen Zhang for the conversations (research or otherwise) and camaraderie.

To the newer PDL students (2020 onwards), I regret that I haven't had too many opportunities to interact with you due to the COVID-19 pandemic and my starting a full-time job, but I hope to cross paths with you some time in the future.

Of course, my time at CMU would not have been so great without staff members who are amazingly helpful and supportive. I'm grateful to Karen Lindenfelser and Bill Courtright for handling all the administrative tasks, organizing the many PDL events we've had throughout the years, and for making life as a PDL student much more enjoyable. I thank Debbie Cavlovich for helping me navigate through Ph.D. requirements such as scheduling my thesis proposal and defense, and for generally making life as a student in CS easier. I would also like to thank Joan Digney for making pretty posters for me at conferences and retreats, and Chad Dougherty, Jason Boles, and Mitch Franzos for dealing with technical equipment set-ups at retreats and helping me with questions on PDL clusters and running experiments for Proteus, Tributary, and Stratus.

I also want to thank the members and companies of the PDL Consortium, including Alibaba, Amazon, Datrium, Dell EMC, Facebook, Google, Hewlett Packard Labs, Hitachi, IBM Research, Intel Corporation, Micron, Microsoft Research, NetApp, Oracle Corporation, Samsung, Seagate Technology, Two Sigma, Veritas, and Western Digital for their interest, insights, feedback, and support for my research.

In 2020, I've gone the ABD route and began full-time remote work at Microsoft on the Cosmos Resource Management team. I'm especially grateful to my manager Fabio Valbuena, who I understand must have put enormous amounts of trust in me to finish my studies and to produce quality work for the team, all while allowing me to work remotely. I would also like to thank my former managers and mentors at Microsoft prior to starting my Ph.D. studies: Markus Weimer, Zhong Chen, Jaliya Ekanayake, Beysim Sezgin, and Clemens Szyperski, all of whom have encouraged and supported me to work on my Ph.D. degree.

I could not have possibly completed my studies without the support of my friends outside of PDL and CS academia: Roger Lo, Peter Chang, David Liu, Dino Wu, Danli Luo, Dian Yu, Vincent Chung, Xiaoliang Li, Quanyang Lu, Kuai-Kuai Jin, Kenneth Jeng, Anya Yu, Ruixuan Liu, Henry Zhang, Derek Liu, James Chien, Joseph Fan, Darren Chin, Cathy Cheng, and I-Ta Yang, thank you for all the great times.

More importantly, I would like to thank my wife, Yanran Yang, who has been with me through all my highs and lows, and who is always there for me when I need it most. Thank you Yanran, without you none of this would have been possible.

Finally, I'm eternally grateful to my family, who have provided me with unconditional love and support. Thank you for everything.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As business operations become increasingly digitized and as data processing tasks become more and more specialized with the proliferation of various types of data applications, companies are moving their workloads off of dedicated, siloed clusters in favor of more cost-efficient shared data infrastructures, e.g., public and private clouds. These shared data infrastructures are often deployed on highly heterogeneous servers, are multi-tenant with server resources shared across multiple organizations, and serve widely diverse workloads ranging from batch analytics jobs to consumer-facing services with stringent *service level objectives (SLOs)*.

Both users and operators of such shared data infrastructures strive to optimize for *value*. Operators seek to increase profit margins by lowering infrastructure management costs while satisfying the demands of their customers (i.e., help users maximize their value) to increase adoption and lower turnover. At the same time, users look to complete their tasks in an efficient and timely manner without having to pay large amounts of money.

But, the highly heterogeneous nature of these shared environments imposes a high barrier to value attainment for both users and cluster operators: Users are offered a wide variety of different types of compute resources, making it difficult for them to make *value-efficient* resource acquisition decisions for their applications, given application constraints. Operators, on the other hand, face difficult challenges in knowing how to assign compute resources to customers when heavily loaded. Indeed, maximizing value in shared data infrastructures necessarily requires effort from both operators and users.

This dissertation explores the problem of value attainment in shared data infrastructures from both the perspectives of users and cluster operators. On the user front, our work proposes and evaluates two resource acquisition strategies and systems for renting virtual machine (VM) instances in the public cloud: (1) Tributary [69] for running online services and (2) Stratus [34], for general batch analytics jobs. Both demonstrate significant cost savings for users for their respective application category.

On the operator front, this dissertation explores using the notions of historic inter-job dependencies and expected job value/utility to inform cluster resource managers and workflow managers about upcoming jobs, their resource requirements, and the potential value they generate to users. Historically, cluster resource managers and cluster operators are scarcely aware of how jobs are inter-dependent on one-another. Our series of work in Owl [35], Wing [36], and Talon enable cluster resource managers and workflow managers to use inter-job dependencies to effectively and

cost-efficiently allocate cluster resources to jobs to help users attain more value, while helping cluster operators reduce cluster operation costs.

1.1 Thesis statement

This dissertation describes our work in addressing the challenges of attaining value in shared data infrastructures. In particular, this dissertation examines our work done to support the following thesis statement:

Value-realized in shared data environments can be improved both by cost- and heterogeneity-aware applications from users and by value- and dependency-aware resource management systems from cluster operators.

To support this thesis, this dissertation will describe four case studies in research software systems, two of which improve value realized through user applications, and two of which improve value realized through more effective resource management by exploiting inter-job dependencies.
Realizing value through user applications. This dissertation first describes two case studies that allow users to realize value through cost savings in running their applications in the public cloud without significantly impacting their applications' performance:

(i) ***Tributary: Spot-dancing for elastic services with latency SLOs*** **(Chapter 2).** Aimed towards the management of elastic cloud services with latency SLOs, the Tributary elastic control system embraces the uncertain nature of transient cloud resources, e.g., AWS spot instances, to manage services more robustly and more cost-effectively. Such transient resources are available at lower cost, but with the proviso that they can be preempted *en masse*, making them risky to rely upon for long-running services. Tributary creates models of preemption likelihood and exploits the partial independence among different resource offerings, selecting resource allocations that satisfy SLO requirements and adjusting them over time, as client workloads change. Over a range of web service workloads, we find that Tributary reduces cost for achieving a given SLO by 81–86% compared to traditional scaling on non-preemptible resources, and by 47–62% compared to the high-risk approach of the same scaling with spot resources.

(ii) ***Stratus: Cost-aware container scheduling in the public cloud*** **(Chapter 3).** Aimed towards general batch analytics jobs, Stratus is a scheduler specialized for orchestrating job execution on *virtual clusters*, or dynamically allocated collections of virtual machine instances on public IaaS platforms. Unlike schedulers for conventional clusters, Stratus focuses on dollar cost considerations, since public clouds provide effectively unlimited, highly heterogeneous resources allocated on demand. But, since resources are charged-for while allocated, Stratus aggressively packs tasks onto machines, guided by job runtime estimates, trying to make allocated resources be either mostly full (highly utilized) or empty (so they can be released to save money). Simulation experiments based on cluster workload traces from Google and TwoSigma show that Stratus reduces cost by 17–44% compared to state-of-the-art approaches to virtual cluster scheduling.

Realizing value through dependency-aware resource management. This dissertation describes two other pieces of research in software systems that demonstrate opportunities for cluster operators and resource managers to realize value through analyses of historical inter-job dependencies, using the analysis to effectively prioritize and load-shift jobs in the setting of large, multi-tenant corporate clusters (Microsoft Cosmos).

(i) ***Wing: Unearthing inter-job dependencies for better cluster scheduling*** **(Chapter 4).** Inter-job dependencies pervade shared data analytics infrastructures (so-called "data lakes"), as jobs read output files written by previous jobs, yet are often invisible to current cluster schedulers. Jobs are submitted one-by-one, without indicating dependencies, and the scheduler considers them independently based on priority, fairness, etc. This work analyzes inter-job dependencies in a 50k+ node analytics cluster at Microsoft, based on job and data provenance logs, finding that nearly 80% of all jobs depend on at least one other job. The Wing dependency profiler analyzes job and data provenance logs to find hidden inter-job dependencies, characterizes them, and provides improved guidance to cluster schedulers or users via Owl (§4.7), a tool for visualizing Wing output. Specifically, for the 68% of jobs that exhibit their dependencies in a recurring fashion, Wing predicts the impact of a pending job on subsequent jobs and user downloads, and uses that information to refine valuation of that job by the scheduler. In simulations driven by real job logs, we find that a traditional YARN scheduler that uses Wing-provided valuations in place of user-specified priorities extracts more value (in terms of successful dependent jobs and user downloads) from a heavily-loaded cluster.

(ii) ***Talon: Reducing costs with dependency-informed load-shifting*** **(Chapter 5).** In shared data environments such as public clouds, organizations often reserve long-term, guaranteed compute resources proportional to their peak workload to ensure enough capacity to complete their jobs on time. However, such *reserved capacity* is often expensive and requires long-term commitment. Thus, careful capacity planning is warranted to lower cost.
Talon is a novel workflow management service that lowers long-term reserved resource commitment by exploiting two components prevalent in shared data environments: (1) *inter-job dependencies* derived from historical job and input and output dataflow relations and (2) intermittently-available *transient resources* that are often available at lower or no cost in shared clusters to increase cluster resource utilization, with the proviso that they can be preempted by cluster resource managers at any time with little warning.
Talon's analyses of historical job dependencies and other job properties allow it to safely load-shift jobs off-peak *and* more reliably schedule jobs on transient resources to reduce reserved resource commitment without violating job input requirements or job deadlines. In simulation experiments driven by real job logs from a production cluster at Microsoft, we find that Talon can effectively reduce reserved resource commitment by up to 38% compared to the traditional approach of reserving enough resources to handle peak workload, while incurring only minimal job deadline violations.

1.2 Contributions

This dissertation makes the following key contributions:

Tributary:

- Describes the first resource acquisition system that takes advantage of preemptible cloud resources for elastic services with latency SLOs.
- Introduces algorithms for composing resource collections of preemptible resources cost-effectively, exploiting the partial refund model of EC2's spot markets.
- Introduces a new preemption prediction approach that our experiments with EC2 spot market price traces show is significantly more accurate than previous preemption predictors.
- Shows that Tributary's approach yields significant cost savings and robustness benefits relative to other state-of-the-art approaches.

Stratus:

- Identifies the unique mix of characteristics that indicate a role for a new job scheduler specialized for virtual clusters (VCs).
- Describes how runtime-conscious packing can be used to minimize under-utilization of rented instances and techniques for making it work well in practice, including with imperfect runtime predictions.
- Exposes the inter-dependence of packing decisions and instance type selection, showing the dollar cost benefits of co-determining them.
- Describes a batch-job scheduler (Stratus) using novel packing and instance acquisition policies, and demonstrates the effectiveness of its policies with trace-driven simulations of two large-scale, real-world cluster workloads.

Wing:

- Presents the first detailed public study of hidden inter-job dependencies in a large-scale data analytics cluster, revealing important problems and opportunities.
- Describes a novel system for extracting historical inter-job dependencies from provenance data, at scale, and predicting the impact of a newly-submitted job on future jobs and users.
- Shows that use of such predictions can allow a modern scheduler, with minimal changes, to better serve the overall workload by prioritizing the highest-impact jobs.

Talon:

- Presents the first study of batch analytics job load-shiftability based on real-world job input dependencies in a large data analytics cluster, presenting significant opportunities for optimizing batch analytics job scheduling.
- Presents methods to identify jobs that are load-shiftable using inter-job dependencies and job output access logs.
- Proposes *Talon*, a novel job workflow manager that exploits job load-shiftability and low-cost-low-reliability cloud resources to reduce the workload peak on reserved resources.

1.3 Outline

The remainder of this dissertation is organized as follows. We start with describing our work in realizing value in shared compute environments. Chapter 2 describes Tributary [69], our elastic control system that robustly and cost-effectively acquires resources for latency-sensitive, SLO-aware services. Chapter 3 describes Stratus [34], our specialized scheduler that orchestrates batch

analytics job execution in public clouds by focusing on dollar-cost considerations.

This dissertation then describes our work in realizing value through dependency-aware resource management. Chapter 4 characterizes inter-job dependencies as observed in Cosmos and describes Wing and Owl, our systems for extracting historical inter-job dependencies from provenance data, and how Wing and Owl can be used to guide cluster schedulers and users to realize more value. Chapter 5 describes a workflow management system that exploits job load-shifting with inter-job dependencies in-tandem with intermittently available transient resources to minimize long-term cluster resource commitment in shared cluster capacity planning. Finally, Chapter 6 wraps up the dissertation and discusses future research directions.

Chapter 2

Tributary: Spot-dancing for elastic services with latency SLOs

One of the most straightforward ways to increase value realized of user applications run in shared compute environments is to reduce users' costs of running their applications without significantly impacting application performance. This chapter focuses on our work on Tributary [69], an elastic control system that effectively reduces cost for users running elastic web services in public clouds, with minimal impact on SLO attainment.

Elastic web services have been a cloud computing staple from the beginning, adaptively scaling the number of machines used over time based on time-varying client workloads. Generally, an adaptive scaling policy seeks to use just the number of machines required to achieve its *Service Level Objectives (SLOs)*, which are commonly focused on response latency and ensuring that a given percentage (*e.g.*, 95%) of requests are responded to in under a given amount of time [70, 82]. Too many machines results in unnecessary cost, and too few results in excess customer dissatisfaction. As such, much research and development has focused on doing this well [50, 51, 58, 88, 115].

Elastic service scaling schemes generally assume independent and infrequent failures, which is a relatively safe assumption for high-priority allocations in private clouds and *non-preemptible* allocations in public clouds (e.g., on-demand instances in AWS EC2 [10]). This assumption enables scaling schemes to focus on client workload and server responsiveness variations in determining changes to the number of machines needed to meet SLOs.

Modern clouds also offer transient, *preemptible* resources (e.g., EC2 Spot Instances [11]) at a discount of 70–80% [5], creating an opportunity for cheaper service deployments. But, simply using standard scaling schemes fails to address the risks associated with such resources. Namely, preemptions should be expected to be more frequent than failures and, more importantly, preemptions often occur in bulk. Akin to co-occurring failures, bulk preemptions can cause traditional scaling schemes to have sizable gaps in SLO attainment.

We describe Tributary, a new elastic control system that exploits transient, preemptible resources to reduce cost and increase robustness to unexpected workload bursts. Tributary explicitly recognizes the bulk preemption risk, and it exploits the fact that preemptions are often not highly correlated across different pools of resources in heterogeneous clouds. For example, in AWS EC2, there is a separate spot market for each instance type in each availability zone, and

researchers have noted that they often move independently: while preemptions within each spot market are correlated, across spot markets they are not [68]. To safely use preemptible resources, Tributary acquires collections of resources drawn from multiple pools, modified as resource prices change and preemptions occur, while endeavoring to ensure that no single bulk preemption would cause SLO violation. We refer to this dynamic use of multiple preemptible resource pools as *spot-dancing*.

AcquireMgr is Tributary's component that decides the resource collection's makeup. It works with any traditional scaling policy that determines (reactively or predictively) how many cores or machines are needed for each successive period of time, based on client load variation. AcquireMgr decides *which* instances will provide sufficient likelihood of meeting each time period's target at the lowest expected cost. Its probabilistic algorithm combines resource cost and preemption probability predictions for each pool to decide how many resources to include from each pool, and at what price to bid for any new resources (relative to the current market price). Given that a preemption occurs when a market's spot price exceeds the bid price given at resource acquisition time, AcquireMgr can affect the preemption probability via the delta between its bid price and the current price, informed by historical pricing trends. In our implementation, which is specialized to AWS EC2, the predictions use machine learning (ML) models trained on historical EC2 Spot Price data. The expected cost of the computation takes into account EC2's policy of partial refunds for preempted instances, which often results in AcquireMgr choosing high-risk instances and achieving even bigger savings than just the discount for preemptibility.

In addition to the expected cost savings, Tributary's spot-dancing provides a burst tolerance benefit. Any elastic control scheme has some reaction delay between an unexpected burst and any resulting addition of resources, which can cause SLO violations. Because Tributary's resource collection is almost always bigger than the scaling policy's most recent target in order to accommodate bulk preemptions, extra resources are often available to handle unexpected bursts. Of course, traditional elastic control schemes can also acquire extra resources as a buffer against bursts, but only at a cost, whereas the extra resources when using Tributary are a bonus side-effect of AcquireMgr's robust cost savings scheme.

Results for four real-world web request arrival traces and real AWS EC2 spot market data demonstrate Tributary's cost savings and SLO benefits. For each of three popular scaling policies (one reactive and two predictive), Tributary's exploitation of AWS spot instances reduces cost by 81–86% compared to traditional scaling with on-demand instances for achieving a given SLO (e.g., 95% of requests below 1 second). Compared to unsafely using traditional scaling with spot instances (*AWS AutoScale* [1]) instead of on-demand instances, Tributary reduces cost by 47–62% for achieving a given SLO. Compared to other recent systems' policies for exploiting spot instances to reduce cost [68, 114], Tributary provides higher SLO attainment at significantly lower cost.

Our study on Tributary makes four primary contributions. First, it describes Tributary, the first resource acquisition system that takes advantage of preemptible cloud resources for elastic services with latency SLOs. Second, it introduces AcquireMgr algorithms for composing resource collections of preemptible resources cost-effectively, exploiting the partial refund model of EC2's spot markets. Third, it introduces a new preemption prediction approach that our experiments with EC2 spot market price traces show is significantly more accurate than previous preemption predictors. Fourth, we show that Tributary's approach yields significant cost savings

and robustness benefits relative to other state-of-the-art approaches.

2.1 Background and related work

Elastic services dynamically acquire and release machine resources to adapt to time-varying client load. We distinguish two aspects of elastic control, the *scaling policy* and the *resource acquisition scheme*. The scaling policy determines, at any point in time, *how many* resources the service needs in order to satisfy a given SLO. The resource acquisition scheme determines *which* resources should be allocated and, in some cases, aspects of how (e.g., bid price or priority level). This section discusses AWS EC2 spot instances and resource acquisition strategies to put Tributary and its new approach to resource acquisition into context.

2.1.1 IaaS instance types and contracts

Cloud service providers (CSPs) offer an effectively infinite (from most customers' viewpoints) set of VM instances available for rental at fine time granularity. Each CSP offers diverse VM instance "types", primarily differentiated by their constituent hardware resources (e.g., core counts and memory sizes), and leasing contract models.

The two primary types of contract model offered by major CSPs [8, 10, 16] are *on-demand* and *transient*. Instances leased under an *on-demand* contract are non-preemptible. Instances leased under a *transient*, or *preemptible* contract are usually much cheaper, but can be unilaterally revoked by the CSP at any time. The price of on-demand instances are usually fixed for long periods of time, whereas the price of transient instances may frequently vary over time.

Here, we describe preemptible resources in AWS EC2, both to provide a concrete example and because Tributary, Stratus, and most related work specialize to EC2 behavior. In AWS EC2 [10], instances leased under transient contracts are termed *spot instances*. Prices of spot instances are dictated by a *spot market* [11], which fluctuates over time but typically remains 70–80% below the prices of corresponding on-demand instances [68]. To rent a spot instance, a user specifies a *bid price*, which is the maximum price s/he is willing to pay for that instance. The spot instance can be *revoked* at any moment, but this rarely occurs when using common bidding strategies (e.g., bidding the on-demand price) [116]. The low cost but riskier nature of spot instances presents users with a trade-off between reliability (on-demand) and cost savings (spot).

There are several properties of the AWS EC2 spot market behavior that affect customer cost savings and the likelihood of instance preemption. (1) Each instance type in each availability zone has a unique AWS-controlled spot market associated with it, and AWS's spot markets are not truly free markets [22]. (2) Price movements among spot markets are not always correlated, even for the same instance type in a given region [113]. (3) Customers specify a bid in order to acquire a spot instance. The bid is the maximum price a customer is willing to pay for an instance in a specific spot market; once a bid is accepted by AWS, it cannot be modified. (4) A customer is billed the spot market price (not the bid price) for as long as the spot market price for the instance does not exceed the bid price or until the customer releases it voluntarily. (5) As of Oct 2, 2017, AWS charges for the usage of an EC2 instance up to the second, with one exception: if the spot market price of an instance exceeds the bid price during its first hour, the customer is refunded

fully for its usage. No refund is given if the spot instance is revoked in any subsequent hour. We define the period where preemption makes the instance free as the *preemption window*.

When using EC2 spot instances, the bidding strategy plays an important role in both cost and preemption probability. Many bidding strategies for EC2 spot instances have been studied [22, 120, 134]. The most popular strategy by far is to bid the on-demand price to minimize the odds of preemption [94, 113], since AWS charges the market price rather than the bid price.

2.1.2 Cloud resource acquisition schemes

Given a target resource count from a scaling policy, a resource acquisition scheme decides *which* resources to acquire based on attributes of resources (e.g., bid price or priority level). Many elastic control systems assume that all available resources are equivalent, such as would be true in a homogeneous cluster, which makes the acquisition scheme trivial. But, some others address resource selection and bidding strategy aspects of multiple available options. Tributary's AcquireMgr employs novel resource acquisition algorithms, and we discuss related work here.

AWS AutoScale [1] is a service provided by AWS that maintains the resource footprint according to the target determined by a scaling policy. At initialization time, if using on-demand instances, the user specifies an instance type and availability zone. Whenever the scaling target changes, AutoScale acquires or releases instances to reach the new target. If using spot instances, the user can use a so-called "spot fleet"[17] consisting of multiple instance type and availability zone options. In this case, the user configures AutoScale to use one of two strategies. The *lowestPrice* strategy will always select cheapest current spot price of the specified options. The *diversified* strategy will use an equal number of instances from each option. Tributary bids aggressively and diversifies based on predicted preemption rates and observed inter-market correlation, resulting in both higher SLO attainment and lower cost than AutoScale.

Kingfisher [115] uses a cost-aware resource acquisition scheme based on using integer linear programming to determine a service's resource footprint among a heterogeneous set of non-preemptible instances with fixed prices. Tributary also selects from among heterogeneous options, but addresses the additional challenges and opportunities introduced by embracing preemptible transient resources. Several works have explored ways of selecting and using spot instances. HotSpot [116] is a resource container that allows an application to suspend and automatically migrate to the most cost-efficient spot instance. While HotSpot works for single-instance applications, it is not suitable for elastic services since its migrations are not coordinated and it does not address bulk preemptions.

SpotCheck [112] proposes two methods of selecting spot markets to acquire instances in while always bidding at a configurable multiple of the spot instance's corresponding on-demand price. The first method is *greedy cheapest-first*, which picks the cheapest spot market. The second method is *stability-first*, which chooses the most price-stable market based on past market price movement. SpotCheck relies on VM migration and hot spares (on-demand or otherwise) to address revocations, which incurs additional cost, while Tributary uses a diverse pool of spot instances to mitigate revocation risk.

BOSS [130] hosts key-value stores on spot instances by exploiting price differences across pools in different data-centers and creating an online algorithm to dynamically size pools within a constant bound of optimality. Tributary also constructs its resource footprint from different pools,

within and possibly across data-centers. Whereas BOSS assumes non-changing storage capacity requirements, Tributary dynamically scales its resource footprint to maintain the specified latency SLO while adapting to changes in client workload.

Wang et al. [128] explore strategies to decide whether, in the face of changing application behavior, it is better to reserve discounted resources over longer periods or lease resources at normal rates on a shorter term basis. Their solution combines on-demand and "reserved" (long term rental at discount price) instances, neither of which are ever preempted by Amazon.

ExoSphere [114] is a virtual cluster framework for spot instances. Its instance acquisition scheme is based on market portfolio theory, relying on a specified risk averseness parameter (α). ExoSphere formulates the *return* of a spot instance acquisition as the difference between the on-demand cost and the expected cost based on past spot market prices. It then tries to maximize the return of a set of instance allocations with respect to risk, considering market correlations and α, determining the fraction of desired resources to allocate in each spot market being considered. For a given virtual cluster size, ExoSphere will acquire the corresponding number of instances from each market at the on-demand price. Unsurprisingly, since it was created for a different usage model, ExoSphere's scheme is not a great fit for elastic services with latency SLOs. We implement ExoSphere's scheme and show in §2.4.6 that Tributary achieves lower cost, because it bids aggressively (resulting in more preemptions), and higher SLO attainment, because it explicitly predicts preemptions and selects resource sets based on sufficient tolerance of bulk preemptions.

Proteus [68] is an elastic ML system that combines on-demand resources with aggressive bidding of spot resources to complete batch ML training jobs faster and cheaper. Rather than bidding the on-demand price, it bids close to market price and aggressively selects spot markets and bid prices that it predicts will result in preemption, in hopes of getting many partial hours of free resources. The few on-demand resources are used to maintain a copy of the dynamic state as spot instances come and go, and acquisitions are made and used to scale the parallel computation whenever they would reduce the average cost per unit work. Although Tributary uses some of the same mindset (aggressive use of preemptible resources), elastic services with latency SLOs are different than batch processing jobs; elastic services have a target resource quantity for each point in time, and having fewer usually leads to SLO violations, while having more often provides no benefit. Unsurprisingly, therefore, we find that Proteus's scheme is not a great fit for such services. We implement Proteus's acquisition scheme and show in §2.4.6 that Tributary achieves much higher SLO attainment, because it understands the resource target and explicitly uses diversity to mitigate bulk preemption effects. Tributary also uses a new and much more accurate preemption predictor.

2.2 Elastic control in Tributary

AcquireMgr is Tributary's resource acquisition component, and its approach differentiates Tributary from previous elastic control systems. It is coupled with a scaling policy, any of many popular options, which provides the time-varying resource quantity target based on client load. AcquireMgr uses ML models to predict the preemption probability of resources and exploits the relative independence of AWS spot markets to account for potential bulk preemptions by

acquiring a diverse mix of preemptible resources collectively expected to satisfy the user-specified latency SLO. This section describes how AcquireMgr composes the resource mix while targeting minimal cost.

Resource acquisition. AcquireMgr interacts with AWS to request and acquire resources. To do so, AcquireMgr builds sets of request vectors. Each request vector specifies the instance type, availability zone, bid price, and number of instances to acquire. We call this an *allocation request*. An *allocation* is defined as a set of instances of the same type acquired at the same time and price. AcquireMgr's *total footprint*, denoted with the variable A, is a set of such allocations. Resource acquisition decisions are made under four conditions: (1) a periodic (one-minute) clock event fires, (2) an allocation reaches the end of its preemption window, (3) the scaling policy specifies a change in resource requirement, and/or (4) a preemption occurs. We term these conditions *decision points*.

AcquireMgr abstracts away the resource type which is being optimized for. For the workloads described in this chapter, virtual CPUs (VCPUs) are the bottleneck resource; however, it is possible to optimize for memory, network bandwidth, or other resource types instead. A service using Tributary provides its resource scaling characteristics to AcquireMgr in the form of a *utility function* $\upsilon()$. This *utility function* maps the number of resources to the percentage of requests expected to meet the target latency, given the load on the web service. The shape of a utility function is service-specific and depends on how the service scales, for the expected load, with respect to the number of resources. In the simplest case where the web service is embarrassingly parallel, the utility function is linear with respect to the number of resources offered until 100% of the requests are expected to be satisfied, at which point the function turns into a horizontal line. As a concrete example, if an embarrassingly parallel service specifies that 100 instances are required to handle 10000 requests per second without any of the requests missing the target latency, a linear utility function will assume that 50 instances will allow the system to meet the target latency on 50% of the requests. Tributary allows applications to customize the utility function so as to accommodate the resource requirements of applications with various scaling characteristics.

In addition to providing $\upsilon()$, the service also provides the application's target SLO in terms of a percentage of requests required to meet the target latency. By exposing the target SLO as a customizable input, Tributary allows the application to control the Cost-SLO tradeoff. Upon receiving this information, AcquireMgr acquires enough resources to meet SLO in expectation while optimizing for expected cost. In deciding which resources to acquire, AcquireMgr uses the prediction models described in §2.2.1 to predict the probability that each allocation would be preempted. Using these predictions, AcquireMgr can compute the expected cost and the expected utility of a set of allocations (§2.2.2). AcquireMgr greedily acquires allocations until the expected utility is greater than or equal to the SLO percentage requirement (§2.2.3).

2.2.1 Prediction models

When acquiring spot instances on AWS, there are three configurable parameters that affect preemption probability: instance type, availability zone and bid price. This section describes the models used by AcquireMgr to predict allocation preemption probabilities.

Previous work [68] proposed taking the historical median probability of preemption based on the instance type, availability zone and bid price. This approach does not consider time of

day, day of week, price fluctuations and several other factors that affect preemption probabilities. AcquireMgr trains ML models considering such features to predict resource reliability.

Training Data and Feature Engineering. The prediction models are trained ahead of time with data derived from AWS spot market price histories. Each sample in the training dataset is a *hypothetical bid*, and the *target variable*, `preempted`, of our model is whether or not an instance acquired with the hypothetical bid is preempted before the end of its preemption window (1 hr). We use the following method to generate our data set: For each instance and *bid delta* (bid price above the market price with range $[0.00001, 0.2]$) we generate a set of hypothetical bids with the bid starting at a random point in the spot market history. For each bid, we look forward in the spot market price history. If the market price of the instance rises above the bid price at any point within the hour, we mark the sample as `preempted`. For each historical bid, we also record the ten prices immediately prior to the random starting point and their time-stamps.

To increase prediction accuracy, AcquireMgr engineers features from AWS spot market price histories. Our engineered features include: (1) Spot market price; (2) Average spot market price; (3) Bid delta; (4) Frequency of spot market price changing within past hour; (5) Magnitude of spot market price fluctuations within past two, ten, and thirty minutes; (6) Day of the week; (7) Time of day; (8) Whether the time of day falls within working hours (separate feature for all three time zones). These features allow AcquireMgr to construct a more complex prediction model, leading to a higher prediction accuracy (§2.4.7).

Model design. To capture the temporal nature of the EC2 spot market, AcquireMgr uses a *Long Short-Term Memory Recurrent Neural Network (LSTM RNN)* to predict instance preemptions. The LSTM RNN is a popular model for workloads where the ordering of training examples is important to prediction accuracy [119]. Examples of such workloads include language modeling, machine translation, and stock market prediction. Unlike feed forward neural networks, LSTM models take previous inputs into account when classifying input data. Modeling the EC2 spot market as a *sequence* of events significantly improves prediction accuracy (§2.4.7). The output of the model is the probability of the resource being preempted within the hour.

2.2.2 AcquireMgr

To make decisions about which resources to acquire or release, AcquireMgr computes the expected cost and expected utility of the set of instances it is considering at each decision point. Calculations of the expected values are based on probabilities of preemption computed by AcquireMgr's trained LSTM model. This section describes how AcquireMgr computes these values.

Definitions. To aid in discussion, we first define the notion of a *resource pool*. Each instance type in each availability zone forms its own *resource pool*—in the context of the EC2 spot instances, each such resource pool has its own spot market. Given a *set of allocations* A, where A is formulated as a jagged array, let A_i be defined as the i^{th} entry of A corresponding to an array of allocations from resource pool i sorted by bid price in ascending order. We define allocation $a_{i,j}$ as an allocation from resource pool i (*i.e.*, $a_{i,j} \in A_i$) with the j^{th} lowest bid in that resource pool. We further denote $p_{i,j}$ as the bid price of allocation $a_{i,j}$, $\beta_{i,j}$ as the probability of preemption of allocation $a_{i,j}$, and $t_{i,j}$ as the time remaining in the preemption window for allocation $a_{i,j}$. Note that $p_{i,j} \geq p_{i,j-1}$, which also implies $\beta_{i,j-1} \geq \beta_{i,j}$. Finally, we define a $size(A)$ function that returns the size of A's major dimension. See Table 2.1 for symbol reference.

A	Set of allocations as jagged array
A_i	Sorted array of allocations from resource pool i
$a_{i,j}$	Set of instances allocated from resource pool i
$\beta_{i,j}$	Probability that allocation $a_{i,j}$ is preempted
$t_{i,j}$	Time left in the preemption window for $a_{i,j}$
$k_{i,j}$	Number of instances in allocation $a_{i,j}$
$P_{i,j}$	Market price of allocation $a_{i,j}$
$p_{i,j}$	Bid price of allocation $a_{i,j}$
$size(y)$	Size of the major dimension of array y
$resc(y)$	Counts the total number of resources in y
λ_i	Regularization term for diversity
$P(R = r)$	Probability that r resources remain in A
$\upsilon(r)$	The utility of having r resources remain in A
V_A	The expected utility of a set of allocations A
C_A	Expected cost of a set of allocations ($)

Table 2.1: Summary of parameters used by AcquireMgr

.

Expected cost. The total expected cost for a given footprint A is calculated as the sum over the expected cost of individual allocations $C_A\,[a_{i,j}]$:

$$C_A = \sum_{i=1}^{size(A)} \sum_{j=1}^{size(A_i)} C_A[a_{i,j}] \tag{2.1}$$

AcquireMgr calculates the *expected* cost of an allocation by considering the probability of preemption within the preemption window $\beta_{i,j}$ for a given allocation $a_{i,j}$ at a given *bid delta*. There are exactly two possibilities: an allocation will either be preempted with probability $\beta_{i,j}$ or it will reach the end of its preemption window in the remaining $t_{i,j}$ minutes with probability $1 - \beta_{i,j}$, in which case we would voluntarily release the allocation. The expected cost can then be written down as:

$$C_A[a_{i,j}] = (1 - \beta_{i,j}) * P_{i,j} * k_{i,j} * t_{i,j} + \beta_{i,j} * 0 * k_{i,j} * t_{i,j} \tag{2.2}$$

where $k_{i,j}$ is the number of instances in the allocation. and $P_{i,j}$ is the market price for instance of type i at the time of acquisition.

Expected utility. In addition to computing expected cost for a set of allocations, AcquireMgr computes the *expected utility* for a set of allocations. The *expected utility* is the expected percentage of requests that will meet the latency target given the set of allocations A. Expected utility takes into account the probability of allocation preemptions, providing AcquireMgr with a metric for quantifying the expected contribution that each allocation should make to meet the resource target. The expected utility V_A of the set of allocations A is calculated as follows:

$$V_A = \sum_{r=0}^{resc(A)} P(R = r) * \upsilon(r) \tag{2.3}$$

where $P(R)$ is the probability mass function of the discrete random variable R that denotes the number of resources not preempted within the next hour, υ is the utility function provided by the service, and $resc(A)$ is the function that reports the number of resources in a set of allocations A. $resc(A)$ computes the *total amount of resources* in A, while $size(A)$ only computes the *size* of A's major dimension.

Eq. 2.3 computes the expected utility over the next hour given a workload, as though Tributary just bid for all its allocations. This works, even though there will usually be complex overlapping expiration windows across an hour, because it only needs to hold true until recomputed at the next decision point, which is never more than a minute away. To derive $P(R)$, AcquireMgr starts off with the original set of allocations A and generates all possible subsets of A. Each possible subset $S \subseteq A$, S marks some allocations in A as preempted ($\in S$) and the remaining allocations as not preempted ($\notin S$). To formalize the notion, we define the indicator variable $d_{i,j}$, where $d_{i,j} = 1$ if allocation $a_{i,j} \in S$ and $d_{i,j} = 0$ otherwise.

To compute the probability of S being the set of preempted resources ($P(S)$), AcquireMgr separates all allocations by resource pools, as each resource pool within AWS has its own spot market. We leverage the following localizing property. Within each resource pool A_i, the probability of preempting an allocation $a_{i,j}$ is only dependent on whether the allocation with the next lowest bid price, $a_{i,j-1}$, in the same resource pool is preempted. Note that $P(a_{i,1}) = \beta_{i,1}$ for allocation $a_{i,1}$ for all resource pools i. Consider two allocations $a_{i,j}, a_{i,j-1} \in A$ from resource pool A_i. We observe the following properties: (1) $a_{i,j}$ cannot be preempted unless $a_{i,j-1}$ is preempted, (2) the probability that both $a_{i,j}$ and $a_{i,j-1}$ are preempted is the probability that $a_{i,j}$ is preempted, and (3) the probability that $a_{i,j}$ is preempted without $a_{i,j-1}$ being preempted is 0. With Bayes' Rule, we observe that:

$$P(a_{i,j}|a_{i,j-1}) = \frac{P(a_{i,j} \wedge a_{i,j-1})}{P(a_{i,j-1})} = \frac{\beta_{i,j}}{\beta_{i,j-1}}. \tag{2.4}$$

Thus, for an allocation $a_{i,j}$ given subset $S \subseteq A$,

$$P(a_{i,j}|a_{i,j-1}) = \begin{cases} 0 & \text{if allocation } a_{i,j-1} \notin S, \\ \beta_{i,j}/\beta_{i,j-1} & \text{else.} \end{cases} \tag{2.5}$$

Tributary further introduces a regularization term λ_i to encourage bidding in markets with low correlation. Having instances spread across lowly correlated markets is important for avoiding high-risk footprints. If the resource footprint has too many instances from correlated resource pools, Tributary becomes exposed to having too many resources being lost to a correlated price spike, potentially causing an SLO violation. In order obtain price correlation across spot markets, we periodically keep track of fix-sized moving windows of spot markets and compute the Pearson correlation between each pair of spot markets. When computing *expected utility*, Tributary increases an allocation in A_i's probability of preemption $\beta_{i,j}$ by λ_i:

$$\lambda_i = \gamma * \sum_{l=1}^{size(A)} \rho_{i,l} * \frac{resc(A_i) + resc(A_l)}{2 * resc(A)} \tag{2.6}$$

where $\rho_{i,l}$ is the Pearson correlation between resource pools i and l, and $\gamma \in \mathbb{R} \geq 0$ is the configurable penalty multiplier. Essentially, we add a weighted penalty to an allocation based

on its Pearson correlation scores with the rest of our resources in different resource pools. In our experiments, we set $\gamma = 0.01$. The regularization term leads to Tributary creating a diversified resource pool, thus reducing the probability that a significant portion of the resources are preempted simultaneously. Having a high probability of maintaining the majority of the resource pool at any point time, allows Tributary to avoid SLO violations with a high probability.

Let's denote $P(S)$ as the probability of S being the set of resources preempted from A. AcquireMgr computes it by taking the product of the conditional probability of each allocation having the outcome specified in S. If the allocation is preempted ($d_{i,j} = 1$) the conditional probability of the allocation being preempted ($P(a_{i,j}|a_{i,j-1})$) is used, otherwise ($d_{i,j} = 0$) the product uses the conditional probability of the allocation not being preempted ($1 - P(a_{i,j}|a_{i,j-1})$).

$$P(S) = \prod_{i=1}^{size(A)} \prod_{j=1}^{size(A_i)} \Big(d_{i,j} * P(a_{i,j}|a_{i,j-1}) + (1 - d_{i,j}) * (1 - P(a_{i,j}|a_{i,j-1})) \Big) \tag{2.7}$$

Finally, AcquireMgr formulates the probability of r resources remaining after preemption $P(R = r)$ (Eq. 2.3) as the sum of the probabilities of all sets S where the number of resources not preempted in S equals to r:

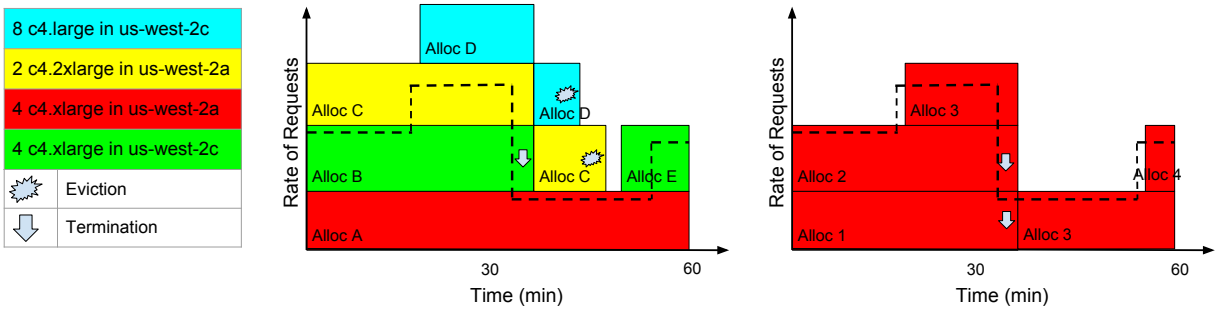
$$P(R = r) = \sum_{S \subseteq A, resc(S) = resc(A) - r} P(S) \tag{2.8}$$

which it uses to calculate the expected utility of a set of allocations A (Eq. 2.3).

Computational tractability. AcquireMgr's algorithm is exponentially computationally expensive as the number of spot markets considered increases. When considering more markets, it is possible to reduce computational complexity by grouping similar, correlated spot markets, and performing revocation analysis with a representative market. Although this would potentially decrease the precision of the preemption analysis, it would allow AcquireMgr to further improve performance by considering a larger number of markets.

2.2.3 Scaling out

Resource acquisition. When Tributary starts, the user specifies a *target SLO* in terms of percentage of requests that respond within a certain latency for Tributary to target. AcquireMgr uses this target SLO to acquire resources. At each decision point, AcquireMgr's objective is to acquire resources until the expected utility θ_A is greater than or equal to the target SLO. If the expected utility is greater than or equal to the target SLO, no action is taken; otherwise, AcquireMgr computes the expected cost (Eq. 2.2) and utility of the current set of allocations (Eq. 2.3). AcquireMgr then calculates the missing number of resources (M) required to meet the target SLO and builds a set of possible allocations (Λ) that consists of allocations from different resource pools at different bid prices (from \$0.0001 to \$0.2 above the current price). For each possible allocation Λ_i, AcquireMgr records the new expected utility divided by the new expected cost of $A \cup \Lambda_i$, choosing the allocation Λ_{chosen} that maximizes this value. AcquireMgr continues to add possible allocations until it achieves the target SLO in expectation.



(a) Legend **(b)** Tributary **(c)** AutoScale

Figure 2.1: Figures (b) and (c) show how Tributary and AutoScale handle a sample workload respectively. Figure (a) is the legend for (b) and (c), color-coding each allocation. The black dotted lines in (b) and (c) signify the request rates over time. At **minute 15**, the request rate unexpectedly spikes and AutoScale experiences "slow" requests until completing integration of additional resources with *3*. Tributary, meanwhile, had extra resources meant to address preemption risk in *C*, providing a natural buffer of resources that is able to avoid "slow" requests during the spike. At **minute 35**, when the request rate decreases, Tributary terminates *B*, since it believes that *B* has the lowest probability of getting the free partial hour. It does not terminate *D* since it has a high probability of eviction and is likely to be free; it also does not terminate *C* since it needs to maintain resources. AutoScale, on the other hand, terminates both *2* and *3*, incurring partial cost. At **minute 52**, the request rate increases and Tributary again benefits from the extra buffer while AutoScale misses its latency SLO. In this example, Tributary has less "slow" requests and achieves lower cost than AutoScale because AutoScale pays for *3* and for the partial hour for both *1* and *2* while Tributary only pays for *A* and the partial hour for *B* since *C* and *D* were preempted and incur no cost.

Buffers of transient resources. To accommodate potential resource preemptions, Tributary inherently acquires more than the required amount of resources if any of its allocations have a preemption probability greater than zero, which is always the case with spot instances. The amount of additional resources acquired depends on the target SLO and the probabilities of allocation preemptions (Eq. 2.3). While the primary goal of these additional resources is to account for preemptions, they often have the added benefit handling unexpected increases in load. Experiments with Tributary show that these resource buffers both increase the fraction of requests meeting latency targets and decrease cost (§2.4.3).

2.2.4 Scaling in

Aside from preemptions, Tributary also tries to scale in voluntarily. As described earlier, each allocation is considered only for the duration of the preemption window. When an allocation reaches the end of its preemption window, it is terminated and replaced with a new allocation if required. When resource requirements decrease, Tributary considers terminating allocations for allocations least likely to be preempted. During this process Tributary chooses the allocation with the least time remaining in the hour, computes the expected utility θ_A without this allocation, and if it is greater than the target SLO, Tributary terminates the allocation. Tributary continues to try and terminate allocations as long as θ_A remains greater than the target SLO.

2.2.5 Example and future consideration

Example. Fig. 2.1 shows how Tributary and AutoScale handle a sample workload, including how the extra resources Tributary acquires to handle preemption events can also handle an unexpected request rate increase and how aggressive allocation selection can get some resources for free due to preemptions.
Future. Tributary lowers cost and meets SLO requirements by taking advantage of low-cost spot instances and uncorrelated prices across different spot instance markets. Mass adoption of systems like Tributary could change these characteristics. While a detailed analysis of mass adoption's potential effects on EC2 spot-markets is outside the scope of this chapter, we evaluate the effects of two potential changes to the spot-market policies in §2.4.5.

2.3 Tributary Implementation

Fig. 2.2 shows Tributary's high-level system architecture. This section describes the main components, how they fit together, and how they interact with AWS.
Preemption prediction models. The *prediction models* are trained offline using TensorFlow [21] and deployed using Tensorflow Serving [18]. A separate model is used for each resource pool. To service run time predictions Tributary launches a *Prediction Serving Proxy* that receives all prediction queries from AcquireMgr, forwards them to their respective models, aggregates the results, and returns the predictions to AcquireMgr.
Resource footprint management. In Tributary, AcquireMgr takes primary responsibility for managing the resource footprint. AcquireMgr acquires instances, terminates instances, and

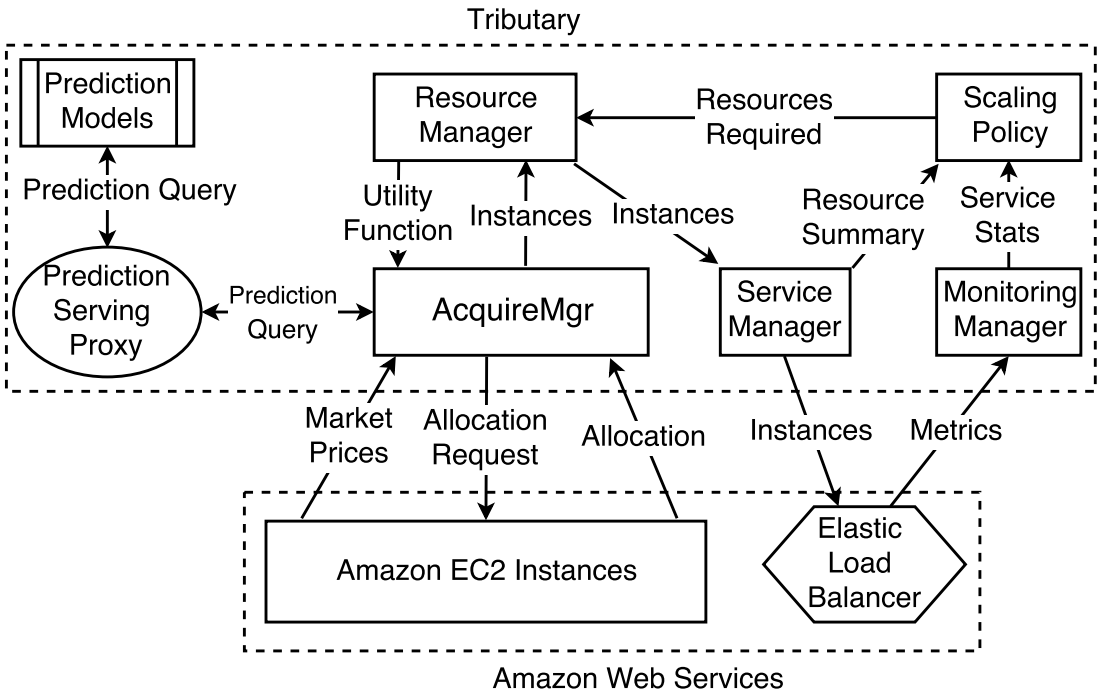


Figure 2.2: Tributary architecture.

monitors AWS for instance preemption notifications. AcquireMgr considers modifying the resource footprint at every *decision point*, and it follows the procedure described in §2.2.3 when additional resources are needed. Once AcquireMgr selects a set of instances to acquire, it sends instance requests to AWS via *boto.ec2* API calls. AWS responds with a set of spot request ids, which corresponds to the EC2 instances allocated to AcquireMgr. Once the instances are in a running state, AcquireMgr sends the instance ids associated with the new instances to Resource Manager. Instance removal follows a similar procedure.

Scaling policy. The *Scaling Policy* component determines dynamic sizing of the resource target. Through a simple event-driven API, users can implement their own scaling policies that access metrics provided by the Monitoring Manager and specify the resource target.

Monitoring Manager (MonMgr). The *Monitoring Manager* orchestrates monitoring of service system resources. The Scaling Policy can register for metrics such as total number of requests and average CPU utilization of instances. The MonMgr queries requested metrics using AWS CloudWatch each *monitoring period* and forwards them to the scaling policy.

Resource Manager (ResMgr). The *Resource Manager* is a proxy for AcquireMgr. Using resource targets provided by the Scaling Policy, the ResMgr generates the utility function used by AcquireMgr to make resource acquisition decisions.[1] The ResMgr also receives instance allocations and termination notices from AcquireMgr and forwards them to the Service Manager.

[1]Process of constructing the utility function is described in §2.4.2.

2.4 Evaluation

This section evaluates Tributary's effectiveness. The results support a number of important findings: (1) Tributary's exploitation of AWS spot market instances reduces cost by 81%–86% compared to on-demand instances and simultaneously decrease SLO latency misses; (2) Compared to standard bidding policies for spot instances, Tributary reduces cost by up to 41% and decreases SLO latency misses by 31%–65%; (3) Compared to extending those standard policies to use enough extra (buffer) resources to match Tributary's number of SLO latency misses, Tributary reduces cost by 47%–62%; (4) Tributary outperforms state-of-the-art resource managers in running elastic services; (5) Tributary's preemption prediction models improve accuracy significantly, resulting in 37% lower cost than previous prediction approaches.

2.4.1 Experimental setup

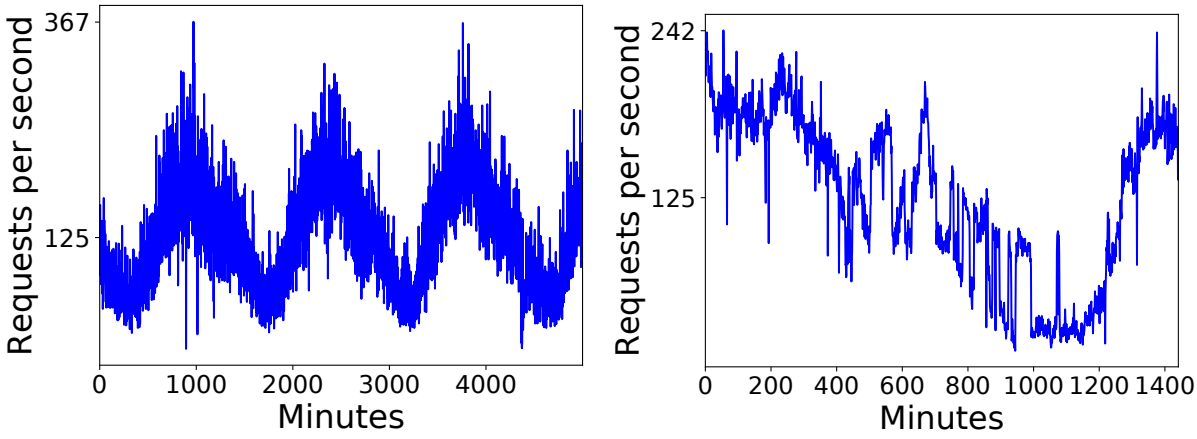
Experimental platform. We report results for use of three AWS EC2 spot instance types: *c4.large*, *c4.xlarge*, and *c4.2xlarge*. The results correspond to the *us-west-2* region, which consists of three availability zones. Using the three instance types in each availability zone, our experiments involve nine resource pools.

Workload. The simulated workload uses a real-world trace for request arrival times, with each request consisting of the derivation of the PBKDF2 [79] key of a password. The calculation of a PBKDF2 key is CPU-heavy, with no network overhead and minimal memory overhead. With the CPU performance being the bottleneck, the resource requirement can be characterized in requests-per-second-per-VCPU.

Environment. In the simulation framework, each instance is characterized with a number of VCPUs, and the request processing time is configured to the measured time for one request on an EC2 instance ($\approx$100ms). Each instance server maintains a queue of requests, and we simulate the queueing effects using the discrete event simulation library SimPy [98]. The simulation framework takes into account resource start-up time, with newly acquired instances not able to service requests for two hundred seconds following their launch.

SLO and scaling. The target service latency is set to one second, and we verified on EC2 that a VCPU can handle roughly 10 requests per second without violating the latency target. So, the requests-per-second-per-VCPU is ten, and the queue size per server instance is ten times the number of VCPUs in the instance. Tributary is not overly sensitive to the target latency setting.

Traces. We use four real-world request arrival traces with differing characteristics. *Berkeley* is from the Berkeley Home IP proxy service and *ClarkNet* is from the ClarkNet ISP's HTTP servers [41]. Both exhibit a periodic, diurnal pattern. We use the first 2000 minutes of these two traces, which covers an entire period. *WITS* is a sampled trace from the Waikato Internet Traffic Storage (WITS) [62]. The trace lasts for roughly a day, from April 6th to April 7th of the year 2000. This trace exhibits large variation of request rates throughout the day, as can be seen in Fig. 2.3b. *WorldCup98* is the arrival trace of the workload on the 1998 FIFA World Cup HTTP Servers [41] on day 75 of the World Cup. All traces are scaled to have an average of 125 requests per second in order to generate sufficient load for the experiments.



(a) ClarkNet Periodic[41] **(b)** WITS Large Variation[62]

Figure 2.3: Traces used in system evaluation.

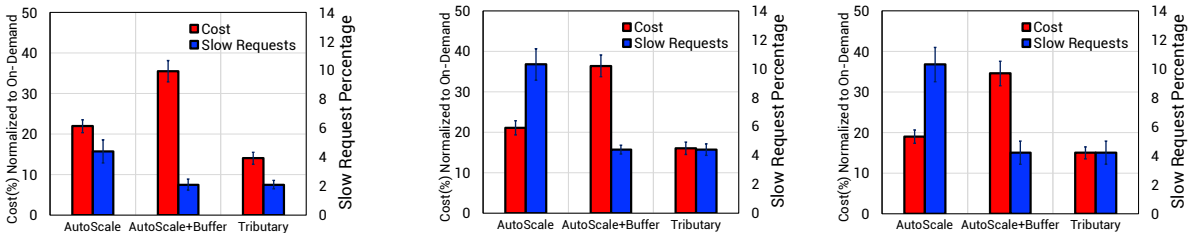
2.4.2 Scaling policies evaluated

We implement three popular scaling policies: *Reactive*, *Predictive Moving Window Average (MWA)*, and *Predictive Linear Regression (LR)* to evaluate our system. The utility function provided by the service is linear for all three policies. We make this assumption since our workload characteristic is embarrassingly parallel—if a workload exhibits different scaling characteristics, a different utility function can be employed.

The *Reactive Policy* scales out immediately when demand reported by the MonMgr is greater than what the available resources are able to handle. It scales in slowly (only after three minutes of low demand), as recommended by Gandhi et al. [51], to prevent premature scale-in in case the demand fluctuates widely in a short period of time. The *MWA Policy* maintains a sliding window of a fixed size, with each window entry consisting of the number of requests received in each monitoring period. The policy takes the average of the window entries to predict the number of requests on the next monitoring period. The policy then adjusts the utility and scaling functions according to the predicted number of requests, and reports the updated functions to the ResMgr to scale in expectation of future requests. The *LR Policy* also maintains a sliding window of a fixed size, but rather than using the average in the window for prediction, the policy performs linear regression on data points in the window to estimate the expected number of requests in the next monitoring period. Our experiments show that regardless of the scaling policy used, Tributary beats its competitors in both meeting the service latency target and cost.

2.4.3 Improvements with Tributary

Here, we evaluate Tributary's ability to reduce cost and latency target misses against AutoScale.
AWS Autoscale. AWS AutoScale (§2.1.2) as offered by Amazon only supports the simplest reactive scaling policies. To provide better comparison between approaches, we implement the AWS AutoScale resource acquisition algorithm as closely as possible according to its documentation [1] and integrate it with Tributary's SvcMgr to work with its more powerful scaling policies. From



(a) Reactive **(b)** Predictive-LR **(c)** Predictive-MWA

Figure 2.4: Cost savings (red) and percentage of "slow" requests (blue) for the *ClarkNet* trace.

here on, mentions of *AutoScale* refer to our implementation of AWS AutoScale. AutoScale is the equivalent of the AcquireMgr component of Tributary. The default AutoScale algorithm with spot instances bids for the lowest market-priced spot instance at the on-demand price upon resource requests by the scaling policy. In addition, AutoScale terminates resources as soon as the resource requirements are lowered, choosing to terminate resources that are most expensive at the moment.

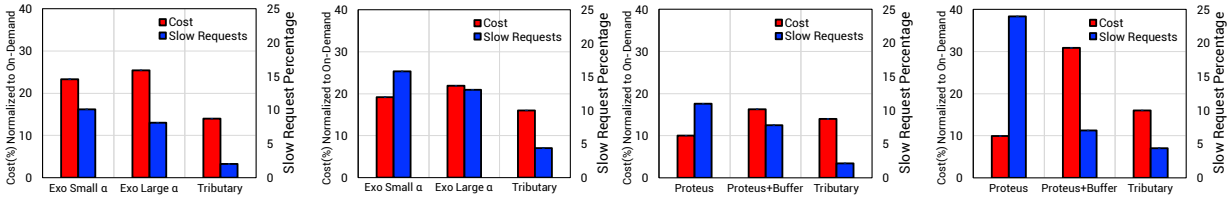
Methodology and terminology. To achieve fair comparisons across a wide range of data points, we perform cost analysis with simulations using historical spot market traces. Using traces allows us to test different approaches on the same period of market data and to get a better picture of the expected behavior of the system in a shorter amount of time. For each request arrival trace (§2.4.1) and resource acquisition approach, we present the average cost and percentage of *"slow" requests* over trace requests across ten randomly chosen day/time starting points between January 23, 2017 and March 23, 2017 in the *us-west-2* region. From here on, we define a *"slow" request* as a request that does not meet the latency target and the percentage of "slow" requests as the percentage of "slow" requests over all requests in a single trace.[2]

Cost savings and service latency improvements. Fig. 2.4 shows the cost savings and percentage of "slow" requests for the *ClarkNet* trace. The cost savings are normalized against running Tributary on on-demand resources. The results demonstrate that Tributary reduces cost and "slow" requests for all three scaling policies. Cost savings are $\approx 85\%$ compared to on-demand resources. For the *ClarkNet* trace, Tributary reduces cost by 36%, 24% and 21% compared to to AutoScale for the *Reactive*, *Predictive-LR* and *Predictive-MWA* scaling policies, respectively. Compared to AutoScale, Tributary reduces "slow" requests by 72%, 61% and 64%, respectively, for the three scaling policies.

In order to decrease the number "slow" requests, popular scaling polices are often configured to provision more resources than immediately necessary to handle unexpected increases in load. It is common to specify the resource buffer as a percentage of the expected resource requirement. For example, with a buffer of 50%, 15 resources (*e.g.*, VCPUs) would be acquired rather than the projected 10. AutoScale+Buffer shows the cost of provisioning AutoScale with a large enough buffer such that its number of "slow" requests matches that of Tributary. Tributary reduces cost by 61%, 56% and 57% compared to AutoScale+Buffer for the three scaling policies.

The cost savings for Tributary on the *Berkeley* trace relative to AutoScale are similar to those on the *ClarkNet* trace, but the reduction in percentage of "slow" requests increases. This difference in performance is due to differing characteristics of the two traces—the *ClarkNet*

[2]Prediction models were trained on data from 06/06/16 – 01/22/17.



(a) Reactive **(b)** Predictive-LR **(c)** Reactive **(d)** Predictive-LR

Figure 2.5: Comparing to ExoSphere and Proteus. Predictive-MWA results not shown but similar.

trace experiences more minute-to-minute volatility in request rate compared to the *Berkeley* trace. We observe similar levels of cost reductions and reduction in "slow" requests on the *WITS* and *WorldCup98* traces, results for WITS are shown in Tables 2.2. Compared to AutoScale+Buffer, Tributary decreased costs by 47–62% across all traces.

Scaling Policy	Cost Saving	"Slow" request Reduction
Reactive	37%	31%
Predictive-LR	33%	50%
Predictive-MWA	29%	51%

Table 2.2: Cost and "slow" request improvements for Tributary compared to AutoScale for the *WITS* trace

Attribution of benefits. Tributary's superior performance arises from several factors. Much of the reduction in cost compared to AutoScale is due to Tributary's ability to get free instance hours. *Free instance hours* occur when an allocation does useful work but is preempted by AWS before the end of a preemption window. The user receives a refund for the partial hour, which means that any work done by the allocation in the preemption window comes at no cost to the user. Tributary takes the probability of getting free instance hours into account when computing the expected cost of allocations (Eq. 2.1), often acquiring resources that provide higher opportunities for free instance hours.

Another factor in Tributary's lower cost is its ability to remove allocations that are not likely to be preempted when demand drops. When resource demand decreases, Tributary terminates instances that are least likely to be preempted, thus lowering the expected cost of its resource footprint. The reductions in "slow" requests arise from the buffer of resources acquired by Tributary (§2.2.3). When acquiring instances, AcquireMgr estimates their probability of preemption. Unless all allocations have a preemption probability of zero, which never occurs for spot instances, Tributary acquires more resources than specified by the scaling policy. The primary goal of the additional resources is to ensure that, when Tributary experiences preemption events, it still has at least the specified number of resources in expectation. The additional resources also provide a secondary benefit by handling some or all of unexpected bursts of requests that exceed the load expected by the scaling policy. The cost of these additional resources is commonly offset by free instance hours; indeed, the extra resources are acquired to cope with preemptions.

2.4.4 Risk mitigation

A key feature of Tributary is that it encourages instance diversification, *i.e.*, acquiring instances from mostly independent resource pools (§2.2.2). The default AutoScale policy is the lowest-price policy, which does not take diversification into account when acquiring instances; instead, it acquires the cheapest instance. Illustrated in Fig. 2.1, Tributary acquires different types of instances in different availability zones, while AutoScale acquires instances of the same type (all red). Diversifying across resource pools is important, because each has an independent spot market, avoiding highly correlated allocation preemptions within a single instance market. Acquiring too much from a single pool, as often occurs with AutoScale, creates a high risk of SLO violation when preemption events occur (*e.g.*, if the red allocation in Fig. 2.1c was preempted prior to minute 35).

In our experiments, we found it to be very rare for market prices to rise above on-demand prices, meaning that AutoScale rarely experiences preemption events. However, when examining past EC2 spot market traces and other availability zones, we found it to be significantly more common for the market price to rise above the on-demand price, thus preempting AutoScale instances.[3] Since Amazon charges users the market price and not the bid price, it is possible that Amazon may once again preempt instances bidding the on-demand price with regularity—a phenomenon we recently observed in the *us-east* availability zones. Thus, AutoScale's resource acquisition approach is riskier for services with latency SLOs on spot machines.

Cost of diversified AutoScale. In addition to the default AutoScale policy which acquires the lowest-priced instance, AWS also offers a diversified AutoScale policy that starts instances from a diverse set of resource pools [17]. Acquiring instances from different spot markets reduces preemption risks, but our experiments showed that it increases cost by 8%–12% compared to the lowest-price AutoScale policy. Compared to Tributary, which diversifies across spot markets intelligently, we found that a diversified AutoScale policy cost 68% more to achieve the same number of "slow" requests for the *reactive* scaling policy on the *ClarkNet* trace.

2.4.5 Pricing model discussion

Our experimental results are based on 2018 AWS EC2 billing policies, as described in §2.1.1. This section discusses how Tributary would function under two potential changes to the billing model: (1) elimination of preemption refunds, (2) institution of a free market.

Elimination of preemption refunds. If Amazon eliminates refunds when the market price exceeds bid price during the first hours of usage, Tributary would lose incentive to bid close to market price. Tributary's model would capture this change by setting β in Eq. 2.2 to zero. With higher bids, Tributary would acquire fewer resources because preemption would be less likely. The amount of resources acquired would still exceed the amount of resources required as they would still have non-zero preemption probabilities.

Although Tributary extracts significant benefit from the refunds, it still outperforms AutoScale without it. For example, in a simulation with this billing model modification, Tributary still reduces cost by 31% compared to AutoScale with sufficient buffer to match numbers of "slow"

[3]From 01/23/17–03/20/17, the market price rose above the on-demand price 0 times for the *c4.2xlarge* instance type in *us-west-2*. From 11/1/16–01/22/17, it happened 1073 times.

requests, for the *Clarknet* trace using the *reactive* scaling policy. As expected, Tributary continues to meet SLOs with high likelihood, as it continues to diversify its resource pool and acquire buffers of resources (albeit smaller ones) to account for preemption events.

Free market behavior. In its current design, the AWS EC2 spot markets do not behave as free markets [22]. Customers specify their bid prices for a given resource, but generally do not pay that amount. Instead, a customer is billed according to the EC2-determined spot price for that resource. It is possible, perhaps even likely as the spot market becomes widely popular, that AWS will transition toward a billing policy in which users are charged their *bid price*, instead of the *market price*, and prices move based on supply and demand rather than unknown seller policies. This change would render the commonly used strategy of bidding far above the market price (e.g., bidding the on-demand price) obsolete. Tributary's behavior would not change significantly, as it already often sets bid prices close to market prices and explicitly considers revocation risks, and we believe it would therefore outperform other approaches by even larger margins.

2.4.6 Comparing to state of the art

This section compares Tributary's support for elastic services to two state-of-the-art resource managers designed for preemptible instances. Since neither system was designed for elastic services with latency SLOs, Tributary unsurprisingly performs significantly better.

Exosphere. We implemented ExoSphere's allocation strategy, described in §2.1.2, with the following assumptions and modifications: (i) The ExoSphere paper did not specify whether the correlation between markets is recomputed as time moves on. In order to avoid the need to constantly reconstruct ExoSphere's resource footprint, we assumed static correlation between markets. (ii) As the ExoSphere paper does not provide guidelines as to how to choose α, we experimented with a range of α from 1 to 10^9. Higher α instructs ExoSphere to be more risk averse at the expense of higher cost.

Fig. 2.5 shows the normalized cost and percentage of "slow" requests served for Tributary and for ExoSphere with small (1) and large (10^9) values of α. These experiments were performed on a further scaled-up version of the *ClarkNet* trace (100x of already-scaled version), since ExoSphere was designed for 100s to 1000s of instances and performs poorly at a scale of 10s.[4] In our experiments, we observed that Exosphere with a small α tends to acquire mainly the cheapest resources, inducing little diversity and increasing the number of "slow" requests in the event of preemptions. Tributary's advantage in both cost and SLO attainment results from Tributary's exploitation of spot instance characteristics (§2.4.3).

Proteus. We implemented Proteus's allocation strategy, described in §2.1.2, modified to acquire only spot resources (reducing cost with no significant change in SLO attainment). Fig. 2.5 compares Tributary and Proteus for the *ClarkNet* trace, for two different scaling policies. While Proteus achieves lower cost than Tributary, it experiences a large increase in "slow" requests. This increase is due to Proteus not diversifying its resource pool, instead only acquiring resources based on reducing average per-core cost. When told by the scaling policy to acquire additional resources, similarly to AutoScale buffers (§2.4.3), Proteus is unable to match Tributary's number

[4] At small scales, ExoSphere with low α had no resource diversity. With large α, it acquired too many resources, increasing its cost.

of "slow" requests no matter how large the buffer (and, thus, how high the cost). This is once again due to the lack of diversity in the resources that Proteus acquires.

2.4.7 Prediction model evaluations

This section evaluates the accuracy of the preemption prediction models used by Tributary, which are described in §2.2.1. The recent Proteus system [68] used the historical median probability of preemption depending on the instance type, availability zone and the difference between the user bid price and the spot market price of the resource. Tributary improves prediction accuracy by using machine learning inference models trained with historical spot market data with engineered features. Fig. 2.6 shows the accuracy and F_1 scores for prediction models based on the historical median, a logistic regression classifier, a multilayer perceptron neural network (MLP NN) and a long short term memory recurrent neural network (LSTM RNN). These models were trained on spot market data from 06/06/16 – 01/22/17 and were evaluated on data from 01/23/17 – 03/20/17 for instance types *c4.large*, *c4.xlarge* and *c4.2xlarge* in *us-west-2*.

The output of the prediction models is whether the instance specified in a query will be preempted within the preemption window. Accuracy scores are calculated by the number of samples classified correctly divided by total number of samples. F_1 scores, which account for data skew, are a good accuracy measurement because the data set is skewed toward preemptions at lower *bid deltas* and non-preemptions at higher *bid deltas*. The LSTM RNN model provides the best accuracy and the best F_1 because it is able to capture the temporal nature of the AWS spot market. LSTM increases accuracy by 11% and the F_1 score by 27% compared to using the historical median. The MLP NN model performs worse than the historical median model for accuracy, but its F_1 score is higher because unlike the historical median model, the MLP model considers advanced features when predicting preemptions as described in §2.2.1. The increased accuracy of the LSTM RNN model translates to Tributary's effectiveness. When using the LSTM RNN model, Tributary runs at ≈37% less cost on the *ClarkNet* workload compared to Tributary using historical medians, because the historical median model overestimates the probability of preemption, causing Tributary to acquire more resources than necessary.

2.5 Summary

Tributary exploits AWS spot instances to meet latency SLOs for elastic services at lower cost. By predicting preemption probabilities and acquiring diverse resource footprints, Tributary can aggressively use collections of cheap spot instances to reliably meet SLOs even in the face of bulk preemptions. Our experiments show cost savings of 81–86% relative to using non-preemptible on-demand instances and 47–2% relative to traditional high-risk use of spot instances.

Tributary exploits AWS properties, such as dynamic spot markets and preemption based thereon. We believe its approach would also work for other clouds offering preemptible resources, if they expose enough information to predict preemption probabilities, probabilities, which AWS provides via the visible spot market prices. Currently, *Google Cloud Engine* [16] does not expose such a signal for its preemptible instances. For private clouds, exposing preemption logs could

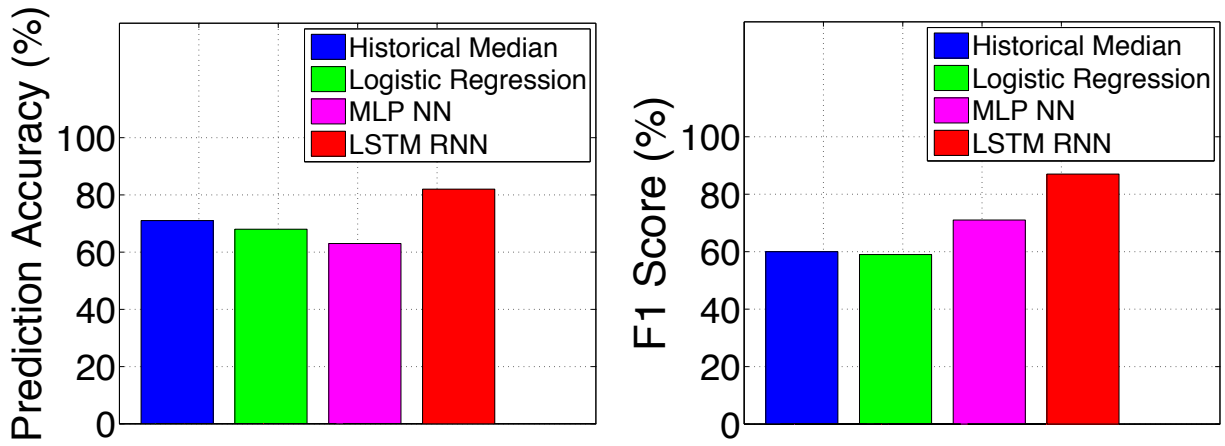


Figure 2.6: Accuracies and F_1 scores (accounts for data skew) for predicting preemption of AWS spot instances. The LSTM RNN outperforms prior techniques (blue bar) by 11% on the accuracy metric and 27% on the F_1 score metric.

provide the historical view, but even better predictions can be enabled by exposing scheduler state.

Chapter 3

Stratus: Cost-aware container scheduling in the public cloud

Continuing the theme to increase value-realized of user applications in shared compute environments by reducing users' cost of running applications, this chapter focuses on our work on Stratus [34], a virtual cluster scheduler suited to cost-effectively schedule batch analytics jobs in public clouds.

Public cloud computing has matured to the point that many organizations rely on it to offload workload bursts from traditional on-premise clusters (so-called "cloud bursting") or even to replace on-premise clusters entirely. Although traditional cluster schedulers could be used to manage a mostly static allocation of public cloud virtual machine (VM) instances,[1] such an arrangement would fail to exploit the public cloud's elastic on-demand properties and thus be unnecessarily expensive.

A common approach [29, 46, 91, 93] is to allocate an instance for each submitted task and then release that instance when the task completes. Although straightforward, this new-instance-per-task approach misses significant opportunities to reduce cost by packing tasks onto fewer and perhaps larger instances. Doing so can increase utilization of rented resources and enable exploitation of varying price differences among instance types.

What is needed is a *virtual cluster (VC) scheduler* that packs work onto instances, as is done by traditional schedulers, without assuming that a fixed pool of resources is being managed. The concerns for such a scheduler are different than for traditional clusters, with resource rental costs being added and queueing delay being removed by the ability to acquire additional resources on demand rather than forcing some jobs to wait for others to finish. Minimizing cost requires good decisions regarding which tasks to pack together on instances as well as when to add more instances, which instance types to add, and when to release previously allocated instances.

Stratus is a scheduler specialized for virtual clusters on public IaaS platforms. Stratus adaptively grows and shrinks its allocated set of instances, carefully selected to minimize cost and accommodate high-utilization packing of tasks. To minimize cost over time, Stratus endeavors to get as close as possible to the ideal of having every instance be either 100% utilized by submitted work or 0% utilized so it can be immediately released (to discontinue paying for it). Via aggressive

[1] We use "instance" as a generic term to refer to a virtual machine resource rented in a public IaaS cloud.

use of a new method we call *runtime binning*, Stratus groups and packs tasks based on when they are predicted to complete. Done well, such-packed tasks will fully utilize an instance, complete around the same time, and allow release of the then-idle instance with minimal under-utilization. To avoid extended retention of low-utilization instances due to mispredicted runtimes, Stratus migrates still-running tasks to clear out such instances.

Stratus's scale-out decisions are also designed to exploit both instance type diversity and instance pricing variation (static and dynamic). When additional instances are needed in the virtual cluster in order to immediately run submitted tasks, Stratus requests instance types that cost-effectively fit sets of predicted-completion-time-similar tasks. We have found that achieving good cost savings requires considering packings of pending tasks in tandem with the cost-per-resource-used of instances on which the tasks could fit; considering either alone before the other leads to many fewer <packing, instance-type> combinations considered and thereby higher costs. Stratus co-determines how many tasks to pack onto instances and which instance types to use.

Simulation experiments of virtual clusters in AWS spot markets, driven by cluster workload traces from Google and TwoSigma, confirm Stratus's efficacy. Stratus reduces total cost by 25% (Google) and 31% (TwoSigma) compared to an aggressive state-of-the-art non-packing task-per-VM approach [116]. Compared to two state-of-the-art VC schedulers that combine dynamic virtual cluster scaling with job packing, Stratus reduces cost by 17–44%. Even with static instance pricing, such as is used for AWS's on-demand instances as well as Google Compute Engine and Microsoft Azure, Stratus reduces cost by 10–29%. Attribution of Stratus's benefits indicates that significant value comes from each of its primary elements—runtime-conscious packing, instance diversity-awareness, and under-utilization-driven migration. Further, we find that the combination is more than the sum of the parts and that failure to co-decide packing and instance type selection significantly reduces cost savings.

This chapter makes four primary contributions. **(1)** It identifies the unique mix of characteristics that indicate a role for a new job scheduler specialized for virtual clusters (VCs). **(2)** It describes how runtime-conscious packing can be used to minimize under-utilization of rented instances and techniques for making it work well in practice, including with imperfect runtime predictions. **(3)** It exposes the inter-dependence of packing decisions and instance type selection, showing the dollar cost benefits of co-determining them. **(4)** It describes a batch-job scheduler (Stratus) using novel packing and instance acquisition policies, and demonstrates the effectiveness of its policies with trace-driven simulations of two large-scale, real-world cluster workloads.

3.1 Background and related work

Job scheduling for clusters of computers has a rich history, with innovation still occurring as new systems address larger scale and emerging work patterns [30, 43, 55, 61, 78, 80, 100, 109, 125, 126]. Generally speaking, job schedulers are the resource assignment decision-making component of a cluster management system that includes support for detecting and monitoring cluster resources, initiating job execution as assigned, enforcing resource usage limits, and so on. Users submit jobs consisting of one or more tasks (single-computer programs that collectively make up a job) to the cluster management system, often together with resource requests for each task (e.g., how much CPU and memory is needed). The job scheduler will decide when and on

which cluster computer to run each task of the job. Each task is generally executed in some form of *container* for resource isolation and security purposes.

Stratus is a cluster scheduler aimed to schedule batch processing workloads (e.g., machine learning model training, parallel test-suites, and distributed ETL workloads such as MapReduce [42] and Spark [133]) on virtual clusters (a "VC scheduler"). This chapter describes how Stratus reduces cost by exploiting public clouds' effectively-unbounded virtual cluster elasticity, instance type diversity, and rental price variation.

3.1.1 Virtual clusters

This section describes virtual clusters and their properties. A *virtual cluster* (VC) is a cluster consisting of VMs rented from the public cloud. As such, VCs are highly elastic and can scale in-and-out quickly compared to traditional clusters. In this chapter, we consider VMs rented under *on-demand* and *transient* contracts, and demonstrate Stratus's efficacy in cost-savings using VMs rented from Amazon's public cloud, AWS. Properties and pricing of VMs rented under on-demand and transient contracts are described in more detail in §2.1.1.

Autoscaling of virtual clusters. There are two parts to autoscaling a virtual cluster (VC): determining the capacity to scale to and picking the right set of instances to scale to said capacity.

CSPs offer VC management frameworks (e.g., Amazon EC2 Spot Fleet [13]) for choosing and acquiring instances to scale based on a user-specified strategy, up to a target capacity. Available strategies in Spot Fleet include *lowestPrice* (always add new instances with the currently lowest spot price) and *diversified* (add new instances such that the diversity in the spot VM pool increases).

Determining the VC's target capacity can be done by the VC scheduler reactively, scaling up whenever a new job's tasks cannot be run on existing resources. Whereas forecasting the right target capacity is needed for web services to prevent violation of SLOs in which the tolerable latency is on the order of seconds or less [51] (e.g., in Chapter 2), the cluster workloads targeted by job schedulers are more forgiving. Even compared to the ideal of always being able to immediately start every new job, we observe reactive VC scaling provides reasonable job latencies (see §3.4).

Assigning containerized tasks to instances. Container services enable containerized user application tasks to be run on a public cloud. There are two dominant flavors of container management services available: *server-based* and *container-based*.

In the server-based model [12], users provide a pool of instances (e.g., via Spot Fleet) while the container service schedules tasks on to available VMs according to a configured placement policy. For example, available task placement policies in Amazon ECS [14] include *binPack* (place task on to instance with least amount of available resource), *random*, and *spread* (round-robin). Server-based container services, in the VC scheduling context, are responsible for packing containerized tasks on to VM instances.

In the container-based model [15], the container service automatically manages container placement, execution, and all underlying infrastructure. A user is billed in terms of resources consumed by the container. As explained further in §3.3, this approach is currently significantly more expensive than using a server-based model with a virtual cluster of spot instances for substantial cluster workloads.

3.1.2 Related work

Private cluster schedulers. Private clusters generally have a fixed set of machine composition, with whatever hardware heterogeneity was present at deployment time. Existing state-of-the-art schedulers [30, 38, 43, 55, 61, 72, 78, 80, 100, 109] frequently optimize scheduling decisions based on the existing set of instances. But, public clouds offer instances of many types and sizes, allowing a virtual cluster to vary over time not only in size but in composition, so the best-match instance type for a given job can usually be acquired when desired. Naturally, different instance types have different rental prices, which must be considered. Further complicating decision-making is the fact that rental costs for particular instance types can vary over time, most notably in the AWS spot markets (as discussed below). Such differences require VC schedulers to focus on different issues than traditional cluster schedulers.

Task-per-instance virtual cluster schedulers. Most previous work on scheduling jobs on public cloud resources maps each task of each job to an instance, acquired only for the duration of that task. This approach works both for *cloud bursting* configurations [29, 46, 65], in which excess load from a private cluster is offloaded onto public cloud resources, and full virtual cluster configurations.

Various policy enhancements have been explored for task-per-instance schedulers. Mao et al. [91, 93] proposed framework-aware VC scheduling techniques that balance job deadlines and budget constraints. Niu et al. [99] discuss scheduling heuristics to address AWS's previous hour-based billing model by reusing instances for new tasks with time remaining in a paid-for hour. HotSpot [116] addresses (exploits) the dynamic nature of spot markets and the diversity of instance types, always allocating the cheapest instance on which a new task will fit and migrating tasks from more expensive instances to cheaper instances as spot market prices fluctuate.

Packing VC schedulers. Compared to the common approach of assigning a single-task-per-instance in existing VC scheduling literature, schedulers that *pack* tasks (from the same or different jobs) onto instances may reduce overall cost, as they reduce the risk of lower utilization due to imperfect fit.

One reasonable approach [13] is to pack containerized tasks on an elastic VC using CSP-offered services like those discussed above. Specifically, one can use server-based container services (e.g., ECS) to place containerized tasks on to (spot) instances, while maintaining the pool of running instances with an instance management frameworks (e.g., SpotFleet). This combination essentially results in a packing VC scheduler, and it is one of the approaches to which we compare Stratus in §3.4.

SuperCloud is a system that enables application migration across different clouds, and it includes a subsystem (SuperCloud-Spot) used for acquiring and packing spot instances [74]. SuperCloud-Spot appears to be designed primarily for a fixed set of long-running jobs (e.g., services), since methods for on-line packing to address dynamic task arrival/completion and varied task CPU/memory demands were not discussed. But, it represents an important step toward effective VC scheduling, and we include it in our evaluations. We also evaluate natural extensions to it as part of understanding the incremental benefits of Stratus's individual features.

Energy-conscious scheduling. Energy-conscious schedulers attempt to reduce the energy consumption of a cluster by actively causing some machines to be idle and powering them down. To do so, they attempt to pack tasks onto machines as tightly as possible to minimize the number

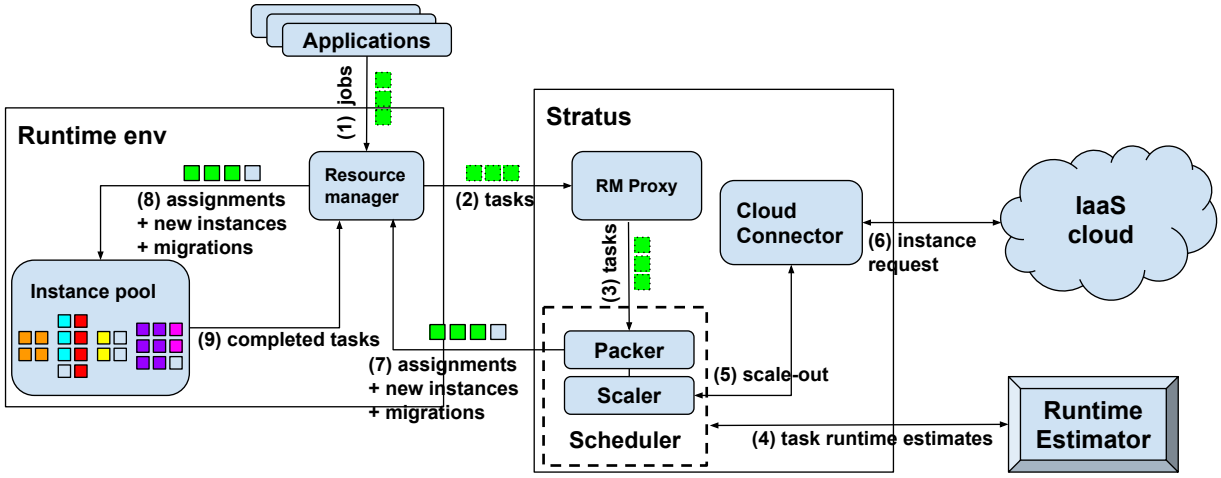


Figure 3.1: Stratus architecture

that must be kept on [26, 27, 87]. This goal draws a parallel to the goal of VC schedulers, whose primary objective is to minimize the cluster's bill typically by using less instance-time and packing instances more efficiently. Acquiring/releasing a VM instance in the cloud is akin to switching on/off a physical machine.

Although energy-conscious schedulers and VC schedulers share a goal of maximizing utilization of active machines, energy-conscious schedulers generally do not address the opportunities created by instance heterogeneity or price variation aspects of VC scheduling. Strictly focusing on task packing, however, the closest scheme to Stratus is a scheduler proposed by Knauth et al [81], which packs VMs onto physical machines based on pre-determined runtimes (rental durations). Unlike that scheduler, Stratus does not have known runtimes, but it does exploit runtime predictions to pack tasks expected to finish around the same time.

3.2 Stratus

Stratus is a VC scheduler designed to achieve cost-effective job execution on public IaaS clouds. Stratus combines a new elasticity-aware packing algorithm (§3.2.2) with a cost-aware cluster scaler (§3.2.3) that exploits instance type diversity and instance pricing variation. Stratus reduces cost in two ways: (1) by aligning task runtimes so (ideally) all tasks on an instance finish at the same time, allowing it to transition quickly from near-full utilization to being released and (2) by selecting which new instance types to acquire during scale-out in tandem with task packing decisions, allowing it to balance the cost benefits of instance utilization and time-varying instance prices. This section describes the design and implementation of Stratus.

3.2.1 Architecture

This section presents the architecture and key components of Stratus (Fig. 3.1) and walks the reader through the lifetime of a job processed by Stratus. Stratus acts as the scheduling component

of a runtime environment, such as a YARN or Kubernetes cluster. The *Resource Manager* (RM) (e.g., YARN RM/Kubernetes master) is still responsible for enforcing scheduling decisions in the environment.

Jobs submitted to the VC are processed as follows:
(1) Job requests are submitted by users and received by the Resource Manager (RM). A job request contains the number of tasks to be launched and the amount of resource required to execute each task.
(2) If a job is admitted, the RM spins off task requests from the job and dispatches them to the Stratus *RM Proxy*. The RM Proxy is responsible for receiving state events (e.g., new task request, task failure, task completion, etc.) from the RM and routing them to the scheduler.
(3) The *scheduler* consists of the *packer* (§3.2.2) and the *scaler* (§3.2.3). The *packer* decides which tasks get scheduled on which available instances. The *scaler* determines which and when VM instances should be acquired for the cluster as well as when task migrations need to be performed to handle task runtime misalignments (§3.2.4). Given a task request from the RM Proxy, the packer puts the task request into the scheduling queue. Pending tasks are scheduled in batches during a periodic *scheduling event*; the frequency of the scheduling event is configurable.
(4) The packer and scaler make scheduling and scaling decisions based on task runtime estimates provided by a *Runtime Estimator*.
(5) If there are tasks that cannot be scheduled on to any available instances in the cluster, the packer relays the tasks along with their runtime estimates to the scaler, which decides on the instances to acquire for these tasks. The scaler sends the corresponding instance requests to the *Cloud Connector*, which is the pluggable cloud-provider-specific module that acquires and terminates instances from the cloud for Stratus.
(6) The Cloud Connector translates the request and asynchronously calls IaaS cloud platform APIs to acquire new instances. When new instances are ready, the Cloud Connector notifies the packer via an asynchronous callback.
(7) The scheduler informs the RM of task placement decisions, availability of new instances, and tasks to migrate at the end of a scheduling event.
(8) The RM enforces task placements and adds new instances to its pool of managed instances.
(9) After tasks complete on instances, completion events are propagated to the RM. A job is completed when the RM receives all task completion events of the job's tasks, and its task runtimes are reported to Runtime Estimator to update job run history (§3.2.4).

3.2.2 Packer

This section describes the on-line packing component of Stratus, which places newly arriving tasks on to already-running instances. The Scaler (§3.2.3), which decides which new instances to acquire based on the packing properties of tasks that cannot be packed on to running instances, uses a compatible scheme.

Setup

The primary objective of Stratus is to minimize the cloud bill of the VC, which is driven mostly by the amount of resource-time (e.g., VCore-hours) purchased to complete the workload. Thus,

the packer aims to pack tasks tightly, aligning remaining runtimes of tasks running on an instance as closely as possible to each other; otherwise, some tasks will complete faster than others and some of the instance's capacity will be wasted.

Packer input. The inputs to the packer are:

(1) *Queue of pending task requests*, where each task request contains the task's *resource vector* (VCores and memory), estimated runtime, priority, and scheduling constraints (e.g., anti-affinity, hardware requirements, etc.).

(2) *Set of available instances*. For each instance, Stratus tracks the amount of resource available on the instance and the remaining runtimes of each task assigned to the instance (*i.e.*, time required for the task to complete).

Runtime binning. The packer maintains logical bins characterized by disjoint runtime intervals. Each bin contains tasks with remaining runtimes that fall within the interval of the bin. Similarly, an instance is assigned to a bin according to the *remaining runtime of the instance*, which is the longest remaining runtime of the tasks assigned to the instance. In both cases, the boundaries of the intervals are defined exponentially, where the interval for the i^{th} bin is $[2^{i-1}, 2^i)$. For ease of discussion, we compare runtime bins according to the upper-bound of their defined runtime intervals—*i.e.*, the smallest bins are bins with runtime intervals [0, 1), [1, 2), [2, 4), ..., and so on.

Algorithm description

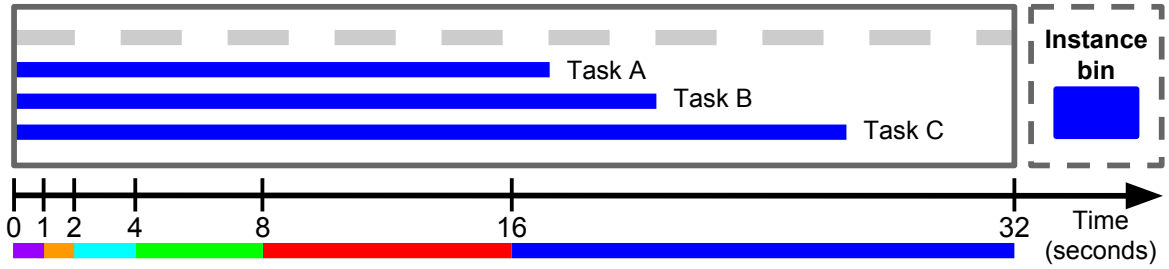
At the beginning of a scheduling event, the packer organizes tasks and instances into their appropriate bins. Tasks are then considered for placement in descending order by runtime—longest task first. For each task, the Packer attempts to assign it to an available instance in two phases: the *up-packing* phase and the *down-packing* phase.

Up-packing phase. In placing a task, the packer first looks at instances from the same bin as the task. If multiple instances are eligible for scheduling the task, the packer chooses the instance with the remaining runtime closest to the runtime of the task.

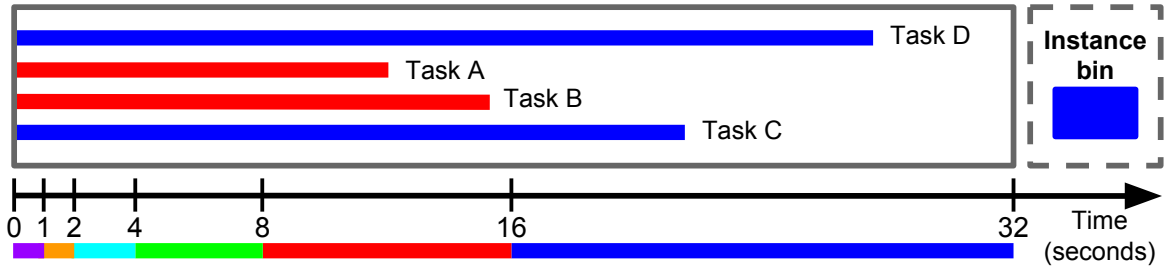
If the task cannot be scheduled on any instance in its native runtime bin, the packer considers instances in progressively *greater* bins. If there are multiple candidate instances from a greater bin, the task is assigned to the instance with the most available resources (as opposed to assigning to the instance with the closest remaining runtime). The reasoning is to leave as much room as possible in the instance, which will increase the chance of being able to schedule tasks from the same bin on to the instance when tasks arrive in the future. If the task cannot be scheduled on any instance, the packer proceeds to examine instances in the *next-greatest* bin until all instances in greater bins have been examined.

While up-packing can cause instance runtime misalignments as Stratus attempts to pack tasks with shorter runtimes on instances with greater remaining runtimes, it also increases utilization of instances in greater bins and prevents the acquisition of new instances for small and short tasks when there are enough resources to run them on already-acquired instances. Up-packing minimally disrupts the scheduling opportunities of tasks of greater bins arriving in the future, as the *up-packed* task uses only half of the remaining runtime on the instance. Fig. 3.2 shows a toy example of runtime binning in the up-packing phase on a single instance over time.

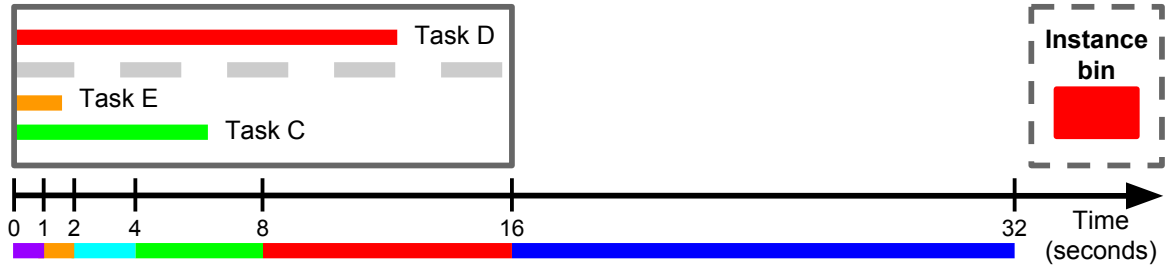
Down-packing phase. After all greater bins have been examined for VMs to schedule the task on, the Packer examines progressively *lesser* bins for a suitable VM in the *down-packing* phase.



(a) Tasks A, B, and C scheduled. All tasks in [16, 32) bin. Instance in [16, 32) bin. One empty slot.



(b) Time progresses and tasks A and B move down to [8, 16) bin. Task C remains in [16, 32) bin. Task D scheduled on instance in [16, 32) bin. Instance remains in [16, 32) bin. Instance is full.



(c) Time progresses further and tasks A and B finish. Task C moves down to [4, 8) bin. Task D moves down to [8, 16) bin. Task E is *up-packed* to the instance and placed in [1, 2) bin. Instance moves down to [8, 16) bin. One empty slot.

Figure 3.2: Toy example showing how runtime binning works with the scheduling of tasks on to an instance over time (Subfigures a–c). This simple example assumes all tasks are uniformly sized, and that the instance can hold four tasks in total. The solid gray box outlines the instance. Runtime bins are color-coded (*e.g.*, blue and red represent bins [16, 32) and [8, 16), respectively). Bars inside the instance represent tasks assigned to it. Task bars are color-coded to the bins they are assigned to. The dotted box shows the runtime bin that the instance assigned to.

If there are multiple candidate VMs from a lesser bin Stratus, like when up-packing, finds the VM with the most available resources that the task fits on. Down-packing the task *promotes* the VM to the task's native runtime bin.

While promoting an instance may cause task runtime misalignments on an instance, it is counter-intuitively beneficial in practice. Since tasks with similar runtimes and resource requests are often submitted concurrently/in close-succession for batch data processing jobs, promoting a large, poorly-packed instance may allow for more opportunities to fully utilize the instance with unscheduled tasks of such a job—especially because the need to down-pack implies that VMs that satisfy the current task's resource requirement be found neither in the task's native runtime bin nor in greater runtime bins. Promoting an instance also increases the chance of better utilizing the instance in later scheduling cycles, since tasks are always up-packed prior to being down-packed. Furthermore, if task runtimes are already inaccurate, it is likely that some of the tasks assigned to an instance in fact belong in some greater bin, especially if an instance is large. If a promoted instance remains under-utilized, instance clearing (§3.2.4) can then be used to de-allocate the instance and redistribute tasks to their rightful bins.

3.2.3 Scaler

When Stratus does not have enough instances to accommodate all tasks in a scheduling event, it scales out immediately and acquires new instances for the unscheduled tasks. Stratus's process of deciding which instances to acquire is iterative. It decides on a new instance to acquire at the end of each iteration, assigns unscheduled tasks to the instance, and continues until each unscheduled task is assigned to some new instance.

During scale-out, Stratus considers task packing options together with instance type options, seeking to achieve the most cost-efficient combination. In each iteration, it considers unscheduled tasks in each bin in descending order of runtime bins. The scaler constructs several candidate groups of tasks to be placed on the new instance. Each candidate group is assigned a *cost-efficiency score* for each possible instance type. The candidate group with the greatest cost-efficiency score is assigned to its best-scoring instance type, which is acquired and added to the virtual cluster.

Considering both in tandem is crucial to achieving high cost-efficiency. Doing either (task packing or instance type selection) in isolation, and then doing the other, results in too many missed opportunities—selecting instance types first leads to lower utilization of selected instances due to poor packing fits, whereas packing tasks first excludes opportunities to exploit dynamic price variations by limiting the instance sizes that make sense. Stratus's iterative approach balances the complexity of the potentially massive search space of combinations with the importance of exploring varied points in that space.

Candidate task groups. Candidate task groups are constructed so that the i^{th} group contains the first i tasks in the list sorted in descending runtime order. The first group contains the longest task, the second group contains the two longest tasks, and so on. The scaler continues to build candidate task groups until the aggregate resource request of the largest task group exceeds that of the largest allowed instance type.

Cost-efficiency score. The cost-efficiency score, computed as

$$score = \frac{\text{normalized used constraining resource}}{\text{instance price}},$$

for each candidate <task group, instance> pair, evaluates the resource efficiency of the placement of a candidate task group on a candidate instance relative to the cost of the instance.

To find the *normalized used constraining resource*, we first find the constraining resource type by computing the utilization for each resource type (VCores, memory) as if the task group is assigned to the instance. The resource type yielding the greatest utilization is the constraining resource type. Knowing the constraining resource type, the amount of constraining resource requested by the task group is the used constraining resource. Finally, we normalize the used constraining resource by the amount of resource of the constraining resource type available on the smallest instance type that we can acquire to obtain the normalized used constraining resource. The normalized used constraining resource is used to facilitate comparisons across <task group, instance> pairs with different constraining resource types.

For example, if a candidate task group requests 4 VCores and 1 GiB of memory and a candidate instance has 8 VCores and 16 GiB of memory, the constraining resource type would be VCores (4 VCores / 8 VCores $>$ 1 GiB / 16 GiB), the used constraining resource would be 4 VCores. Assuming that the smallest instance type that we can acquire provides 2 VCores, the normalized used constraining resource would be 2 (= 4 VCores / 2 VCores).

Intuitively, if only a single resource dimension is considered, acquiring the instance with the greatest cost-efficiency score is equivalent to acquiring the instance with the lowest *cost-per-resource-used*.

At the end of each scale-out iteration, the candidate (task group, instance) pair with the best cost-efficiency score is chosen, and the corresponding task group is scheduled on to the instance. If there remains any unscheduled tasks, the scaler begins another iteration to place the rest of the tasks and continues until all tasks are scheduled.

Runtime interval bin selection. With a sufficient amount of tasks in a runtime bin, more instance scale-out options become available—specifically for instances that are larger, potentially more cost-efficient, and less prone to resource fragmentation. As often observed [48, 106], job and task runtime distributions are frequently long-tailed. We therefore define runtime bin intervals exponentially to enable a principled way to group tasks with runtimes at the tail while not sacrificing packing efficiency for tasks that fall in lesser interval bins. Defining runtime bins exponentially also allows us to bound the number of bins without having to specify statically-sized runtime intervals or determine a particular best bin size.

Workload and scaling in. Stratus is most effective when workloads in a VC fluctuates, and it outperforms other schedulers by scaling in effectively when workloads drop. There are two opportunities when Stratus terminates instances: (1) when an instance does not have any tasks assigned to it, and (2) when it continuously experiences low utilization, in which case its tasks are migrated off of it (§3.2.4). This allows VC VM instances to be highly utilized while they are rented, achieving high cost-efficiency. Runtime binning and timely termination of low utilization instances also let VCs scale in more quickly, allowing Stratus to incur lower costs overall.

Bidding strategy and instance revocations. Stratus *does not* try to take advantage of the refund policy of spot instance revocations, where spot instances revoked within the first hour are fully refunded; rather, it focuses only on attaining cost-efficiency by exploiting cost-per-resource dynamicity[2]. Stratus uses a safe instance bidding scheme, where it always bids for an instance at

[2]In the EC2 spot market, the cost-per-resource (e.g., VCore) of instances changes frequently. For *m4* instances in

its corresponding *on-demand* price. Shastri et al. [116] found that bidding the on-demand price for Spot instances result in very long times-to-revocation (25 days on average). Our experiments confirm their observation, as only a single spot price-spike was experienced in all our experiments.

3.2.4 Runtime estimates

Runtime Estimator Runtime Estimator is the component that provides runtime estimates from a queryable task runtime estimate system for tasks submitted to Stratus. The topic of estimating job and task runtimes has been researched extensively [85, 118, 122, 123], and Stratus does not attempt to innovate on this front; instead, we obtained a copy of JVuPredict [123] and modified it to predict average task runtime rather than job runtime.

JVuPredict's algorithm works as follows: For each incoming job, JVuPredict identifies candidate groups of similar jobs in job execution history based on job attributes (e.g., submitted by same user, same job name submitted during the same hour of day,. . . , etc). For each group, several candidate estimates are produced by applying estimators (average, median,. . . , etc) to the average task runtimes of all jobs in the group. JVuPredict associates the estimate produced by the attribute-estimator pair that historically performs best (measured by normalized median absolute error) to the incoming job.

Handling runtime misestimates. The accuracy of task runtime estimates plays a large role in Stratus's packing algorithm. While Stratus's use of exponentially-sized runtime bins already tolerates some degree of task runtime misestimates, it is beneficial to incorporate more specialized methods to deal with larger misestimates. Stratus uses two heuristics to mitigate the impact of task runtime misestimates on cost:

Heuristic 1: Task runtime readjustment. In adjusting for task runtime under-estimates, Harchol-Balter et al. [67] observed that the probability that a process with age T seconds lasts for at least another T seconds is approximately $1/2$. Stratus thus readjusts task runtime underestimates by assuming that the task has already run for half of its runtime.

Heuristic 2: Instance clearing. Stratus migrates tasks away from instances that continuously (e.g., for more than three scheduling events of one minute each in our experiments) experience low resource utilization due to task runtime mis-alignments of various scales such that they can be terminated safely without losing task progress. We define such an instance as one whose resources are less than 50% utilized in each dimension, since this is often when all tasks on an instance can be migrated to a smaller instance based on how many CSPs size their VMs [8, 10, 16].

VM candidates are evaluated for clearing in decreasing order of cost-per-resource-used. For each VM candidate, either all or none of its tasks are migrated—if an instance only ends up partially-migrated, its utilization decreases while the VC operator still has to pay the same amount of money to keep the instance running; therefore, whenever an instance is selected to be migrated, it is placed on a blacklist such that no new tasks can be scheduled on to it. For each task on a VM candidate, Stratus attempts to re-pack the tasks on to currently running instances using the packing algorithm described in §3.2.2. If no suitable instance is found, Stratus may also choose to acquire a new, potentially smaller/cheaper instance on which to place all of a candidate's tasks. Stratus computes the tradeoff of clearing an instance before executing the task migrations. Stratus

us-west-2 only, the sorted order of cost-per-VCore changes up to 850 times/day (Aug. to Sept. 2017).

Instance/ Server/ VM	A bundle of resources rented from the IaaS platform, generally in the form of a virtual machine (e.g., Amazon EC2).
Container	An isolated environment deployable in an instance, e.g., a nested-VM or Linux container.
Resource	Instance hardware resources, for example, VCores and memory.
Resource util	Aggregate percent instance resources *allocated* to tasks.
Task	The smallest logical unit of a computation, typically executed in a single container.
Job	A collection of tasks that perform a computation submitted by the user of a cluster.
Runtime bin	Logical bin defined by a time interval, consisting of a set of instances whose task runtimes are estimated to continue to run for less than the upper bound of the time interval. Used by Stratus to assign tasks of similar runtimes on to instances.

Table 3.1: Summary of terms used.

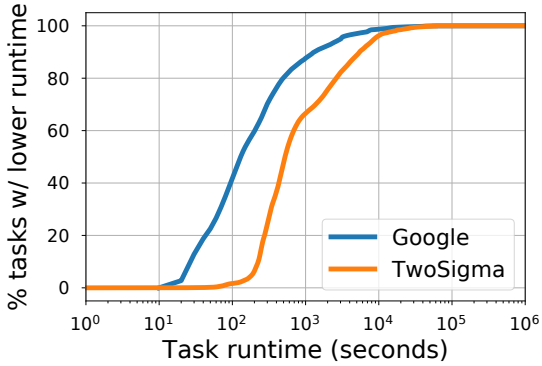


Figure 3.3(a): Task runtime distributions of the Google and TwoSigma traces. The time axis is plotted in log-scale.

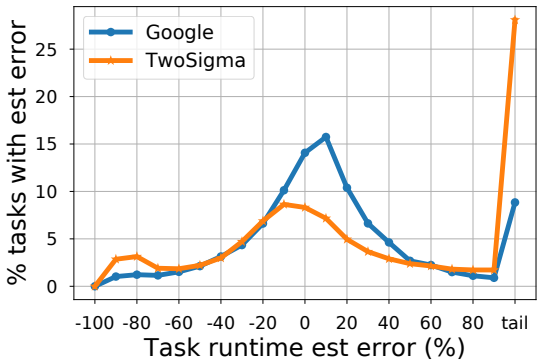


Figure 3.3(b): PMF of the task runtime estimation error of the modified JVuPredict.

only clears an instance if the predicted runtime for the instance's longest task is greater than the estimated migration time (plus spin-up time, in the case of new instance acquisitions).

3.3 Experimental setup

We use simulation-based experiments to evaluate Stratus and other VC scheduling approaches in terms of dollar cost, resource utilization, and job latency. This section describes our experimental setup.

Instance type	VCores	Memory
m4.large	2	8GiB
m4.xlarge	4	16GiB
m4.2xlarge	8	32GiB
m4.4xlarge	16	64GiB
m4.10xlarge	40	160GiB
m4.16xlarge	64	256GiB

Table 3.2: The resource capacity of each instance type.

3.3.1 Environment

Simulator. We built a high-fidelity event-based workload simulator that takes as input a job trace (§3.3.2) and a Spot market trace for each allowed instance type (discussed below). It simulates instance allocation and job placement decisions made by evaluated schedulers (§3.3.3), advancing simulation time as jobs arrive and complete. The simulator includes instance spin-up delays consistent with observations on AWS [92, 116], drawing uniformly from instance spin-up times ranging from 30 to 160 seconds. Container migration times are computed based on the container's memory footprint and a transfer rate of 160MBps for container memory [116]. To simulate the effect of spot market price movements, including the very rare spot instance revocations,[3] we use price traces provided by Amazon [11] spanning a three month period starting from June 5th, 2017.
Instance types and regions available. We limit our experiments to use instances of the same family in EC2 (*m4* instances) in order to (1) avoid unknown performance comparisons among compute resources[4] and (2) justify runtime estimates produced by JVuPredict, as JVuPredict does not consider tasks' runtime environments (e.g., underlying VM configuration) when generating runtime estimates[5]. We list the amount of resources available in each of the instance types in Table 3.2, and we assume that valid instance requests are always fulfilled. We limit instance allocations to the *us-west-2* region, because migrations and data transfers across regions incur significant cost.
VM acquisition/termination. For all evaluated schedulers, (1) instances are bid for at or above (HotSpot) the on-demand price, and (2) an instance is voluntarily released to the CSP when no more tasks are running on it.

3.3.2 Workload traces

Our experiments use two traces from production clusters. Both traces exhibit fluctuating workload, with peaks and troughs throughout the day [25]. Fig. 3.3a shows their task runtime distributions. For each trace, our evaluations use twenty 1-day ranges of the trace, starting at random points

[3]Spot instance revocation can be determined from the spot market price trace, because they occur when the market price exceeds the bid price. We use the common approach of bidding the on-demand price and, like others, observe that revocation is very infrequent [68, 116].

[4]Amazon used to report ECU as a unifying measurement to describe the CPU performance across varying instance types, but ECU measurements have since slowly disappeared from EC2's documentation, presumably due to the difficulty in summarizing the compute power of different instance types in a single number.

[5]The Runtime Estimator is a pluggable component which can be extended to use runtime estimates produced by a more sophisticated runtime estimator that is VM configuration aware [131].

within the traced period. We filter out jobs that start before the trace start time and jobs that end after the trace end time. In addition to avoiding inclusion of partial jobs, this filtering removes long-running services from the Google trace, allowing the evaluation to focus on interactive and batch jobs.

Google trace. The Google trace [105, 106], released in 2011, records jobs run on one of Google's production clusters with 12.5k machines spanning a period of 29 days. The amount of requested resources for each task has been obfuscated by Google, with each dimension re-scaled to have a value between 0 and 1 based on the largest capacity of the resource available on any machine in the trace. In our simulations, for each task resource dimension, we scale the requests to the largest corresponding resource dimension of instance types used (64 VCores and 256 GiB).

We observe the following job/task properties in the filtered Google trace: (1) Tasks are typically CPU-heavy (*i.e.*, tasks are limited by the CPU dimension when scheduled), (2) the number of tasks per job is very small—in fact, more than 75% of the jobs contain less than 10 tasks, and (3) tasks are short, with most shorter than two minutes.

TwoSigma trace. The TwoSigma trace [25] contains 3.2 million jobs and was collected on two private computing clusters of Two Sigma, a quantitative hedge fund, over a nine-month period from Jan. to Sept. 2016. The clusters consist of a total of 1313 machines with 24 CPU cores and 256GiB RAM each. The majority of jobs in the TwoSigma trace are batch-processing jobs that analyze financial data with home-grown data analysis applications or Spark [133] programs. The workload does not contain any long-running services.

We observe the following job/task properties in the TwoSigma trace: (1) tasks typically have substantial memory footprints, (2) number of tasks per job is greater than in the Google trace, and (3) tasks are longer on average compared to tasks from the Google trace.

Runtime predictor performance. We report the task runtime estimate error profiles of the modified JVuPredict (§3.2.4) for both traces in Fig. 3.3b. The estimates are less accurate in the TwoSigma trace compared to the Google trace because the estimate quality largely depends on (1) the ability of JVuPredict to identify similar jobs in the history and (2) the variability of runtimes within the group of similar jobs. TwoSigma trace is reported [102] to be inferior on both measures.

Assumptions We make the following assumptions about jobs and tasks in our simulation workloads: (1) tasks can be migrated without losing progress potentially using checkpoint-restore solutions such as CRIU [9], (2) tasks do not have any hard placement constraints other than (for some) anti-affinity, (3) there are no inter-task dependencies in the workloads, and (4) decisions regarding task co-location have minimal impact on task runtimes.

Of the above, (1) is recommended practice in implementing distributed applications, and (2) and (3) arise in part from the obfuscation of production data in the traces. We believe that (4) is a reasonable premise for two reasons. First, interference effects between two tasks co-located on a VM instance would likely still be present if they each ran in their own smaller instance, because smaller instances similarly share physical hardware. Second, some co-location effects are already reflected in the runtimes recorded in the traces, since both traced clusters co-locate tasks.

3.3.3 Approaches evaluated

Our experiments compare Stratus against several alternative solutions. Each solution is implemented as closely as possible to its respective source documentation. This section introduces

these approaches and modifications made to adapt them to the problem of minimizing the cost of running a workload where tasks have multi-dimensional resource requests and varying runtimes.

HotSpot: HotSpot (§3.1.2) is a single-task-per-instance VC scheduler that always chooses the cheapest instance type on which a new task will fit and will migrate the task if a different instance type becomes cheaper before it completes. We build ***HSpot***, a VC scheduler that implements HotSpot's migration and scaling policies, and enhance it with perfect runtime knowledge (unlike Stratus's imperfect predictions) so it can evaluate the tradeoff between cost added due to migration overhead vs cost reduction for running on the new cheaper instance.

Spot Fleet + ECS: A reasonable way to place containerized tasks on to Spot instances is to use one of Amazon's ECS container placement strategies in combination with EC2 Spot Fleet [13], which acquires and releases Spot instances based on the allocation policy specified by the user. We build a scheduler ***Fleet*** that uses the most cost-efficient VM acquisition policy in Spot Fleet (*lowestPrice* [17]), in tandem with the most cost-efficient packing strategy in ECS (*binpack* [14]). §3.1.2 provides more detail on Spot Fleet and ECS, along with their respective policies.

SuperCloud Spot instances: SuperCloud-Spot [74] is a packing VC scheduler specifically designed for scheduling nested VMs on Spot instances. We build ***SCloud***, implementing features as closely as possible to what was documented in the paper describing SuperCloud-Spot.

SCloud uses SuperCloud-Spot's greedy packing algorithm on the most-constrained resource type rather than its dynamic programming (DP) algorithm, which cannot be generalized to tasks of different sizes and with multiple resource request dimensions (a known NP-hard problem [101]).

SuperCloud-Spot's original migration scheme is designed for AWS's previous hour-based billing model; SuperCloud-Spot therefore makes sub-optimal decisions when computing the trade-off to re-pack tasks to new instances as it assumes no extra cost for leaving instances running as long as the instance-hour has not yet expired. So, we enhance SCloud with both HSpot's migration scheme that is suited for instances that are charged per-second and perfect task runtime knowledge. SuperCloud-Spot is described in more detail in §3.1.2.

AWS Fargate: AWS Fargate [15] is a service that allows users to run containerized workloads without having to manage VM servers. We evaluated Fargate as an alternative to manually deploying VM clusters and running tasks on top of it. As Fargate charges on-demand prices per-resource plus a premium for managing containers for users, we posit the Fargate-based solution to be much more expensive than any other VC scheduling alternatives. Indeed, simulated experiments show that at its current price-point (May 2018), Fargate costs on average $4.4\times$ more than HSpot. Our discussions thus focus on the Spot VC schedulers introduced above.

3.4 Experimental results

This section evaluates *Stratus*, yielding four key takeaways. First, Stratus is adept at reducing the VC cloud bill, such as by 25–31% compared to the non-packing VC scheduler (HSpot). Second, Stratus's runtime binning and tandem-consideration of task packing and instance selection allows it to reduce VC cost by 17–44% over the other packing-based VC schedulers (Fleet and SCloud). Third, each of Stratus's key techniques is important to achieving its cost reductions. Fourth, Stratus's instance clearing technique is beneficial and necessary in VC scheduling when runtime estimates are inaccurate.

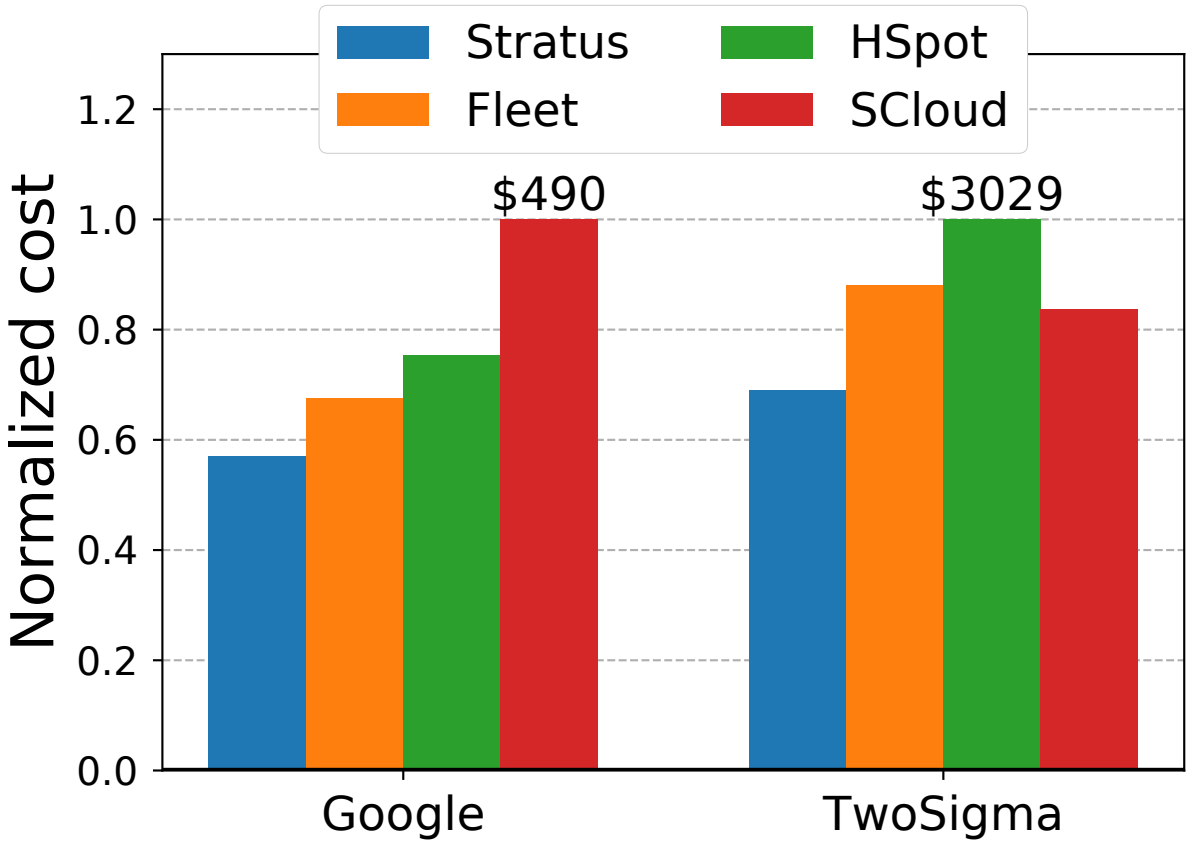


Figure 3.4: Average daily cost for each VC scheduler on the Google and TwoSigma workloads, normalized to the most costly option for the given trace. Stratus reduces the cost of other schedulers by at least 17% in both traces.

3.4.1 Stratus vs state-of-the-art

This section compares Stratus against existing VC scheduling solutions such as HSpot, SCloud, and Fleet.

Cost reduction. Fig. 3.4 shows the average daily costs of scheduling the Google and TwoSigma workloads for each VC scheduler, normalized to the most expensive case for each trace. Stratus outperforms the other VC schedulers by combination of its alignment of task runtimes and coordinated task packing and instance type selection.

Stratus outperforms HSpot by reducing the cloud bill by 25% (Google) and 31% (TwoSigma) through continuously packing newly arriving tasks on to cost-effective instances. Stratus also reduces cost by 44% (Google) and 17% (TwoSigma) compared to SCloud. While ideas from SuperCloud-Spot may have been well-suited for long-running services, it does not carry over well to workloads where task runtimes can greatly vary (Google). SCloud's scaling algorithm often bids for large VMs to reduce fragmentation and improve cost-per-resource at the time of packing. However, if task runtimes on the VM are misaligned, the large VMs acquired by SCloud will often be under-utilized as tasks on the VM complete. Because SCloud does not specify how newly arriving tasks are packed on to existing VMs, the resource holes will be unfilled until all tasks on

the VM completes. Stratus outperforms SCloud by a smaller margin on the TwoSigma workload because (1) task runtimes on the TwoSigma workload tend to be longer and more runtime-aligned (Fig. 3.3a) and because (2) task runtime estimates are significantly less accurate on the TwoSigma trace (Fig. 3.3b).

Although Fleet utilizes on-line packing, it still incurs higher cloud bills compared to Stratus. Stratus reduces the cloud bill of Fleet by 17% (Google) and 22% (TwoSigma). Aside from Stratus's runtime binning, another primary reason as to why Fleet's use of on-line packing is not as effective is due to its use of Spot Fleet's *lowestPrice* scaling algorithm. Fleet always acquires the cheapest (and frequently the tightest-fitting) instances for newly arriving tasks, leaving little room to pack more tasks on an instance and leading to greater resource fragmentation. In addition, the cheapest instance for a task may not be the most cost-efficient instance for the set of pending tasks that are available. By considering the packing of groups of tasks and their runtime alignments *while* selecting instance types, Stratus is able to achieve lower fragmentation and acquire instances with better cost-per-resource-used.

To confirm this observation, we experimented with a version of Stratus that always selects the best-fitting instance for a new task when scaling out (akin to Fleet) while using runtime binning. Consistent with our observation that schedulers that always acquire best-fitting or cheapest-fitting instances for individual tasks have little opportunity to pack, we observe the cost for Stratus increases by 17% (Google) and 25% (TwoSigma), only slightly beating Fleet in both traces.

Resource utilization. Much of Stratus's cost reduction comes from increased utilization of rented resources. Fig. 3.5 shows the utilization of the constraining resource, VCore in the case of the Google workload and memory for TwoSigma, for the four VC schedulers.

Stratus attains higher resource utilization than the other VC schedulers, achieving 86% and 79% utilization, respectively, for the two workloads. Stratus's high resource utilization results from its combination of aligning task runtimes in tasks packed onto a given instance, acquiring instances of suitable sizes, and judicious use of instance clearing to avoid retaining under-utilized instances on which most tasks already completed. Importantly, Stratus's selection of instance types during scale-out in light of different possible packing configurations, rather than only considering packing after selection, significantly increases utilization. At the same time as both, Stratus considers instance pricing differences per-resource-used, resulting in the overall cost reductions described above.

Job latency. We define *normalized job latency* as the observed job latency normalized to an idealized job runtime that incurs no scheduling or instance spin-up delays. Table 3.3 shows the 50^{th} and 95^{th} percentile normalized job latencies for each compared scheduler on each trace.

Overall, we observe that schedulers that always acquire new instances for tasks (SCloud and HSpot) incur greater normalized job latency than those that pack (Fleet and Stratus) on workloads with jobs that are mostly short and small (Google), as instance start-up delay can cause proportionally significant job slowdowns. For workloads where jobs are longer (TwoSigma), instance start-up delays are obviously less significant. And, with more memory-heavy tasks (TwoSigma), use of migration for instance clearing induces marginally higher job latencies for Stratus, because such tasks take longer to migrate.

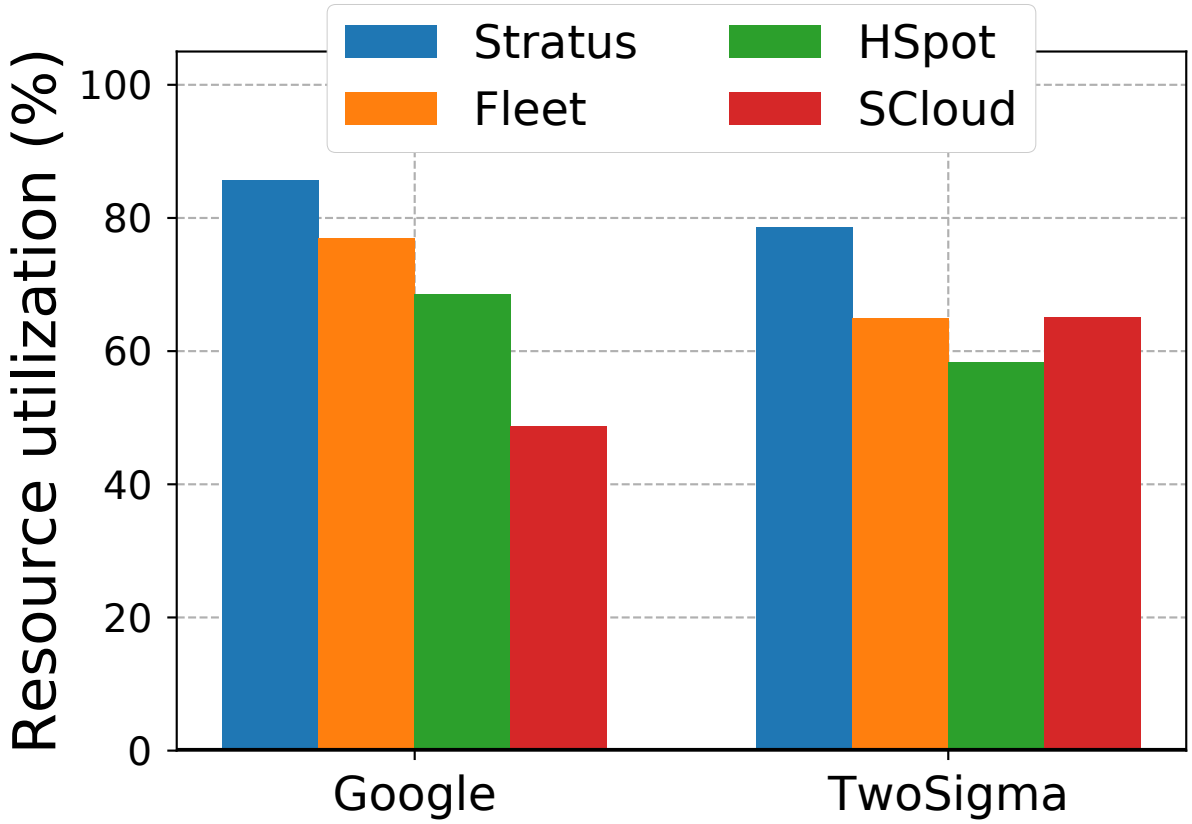


Figure 3.5: Constraining resource utilization (VCores for Google and memory for TwoSigma) with the different VC schedulers.

	Google		TwoSigma	
Scheduler	50%-ile	95%-ile	50%-ile	95%-ile
Stratus	1.3	3.2	1.1	1.6
Fleet	1.2	3.1	1.0	1.4
HSpot	2.2	7.0	1.1	1.5
SCloud	2.5	11.4	1.2	2.0

Table 3.3: The normalized job latencies for each evaluated VC scheduler. Schedulers that pack continuously (Stratus and Fleet) incur lower job latencies than those that do not (HSpot, SCloud) when jobs are short and small (Google).

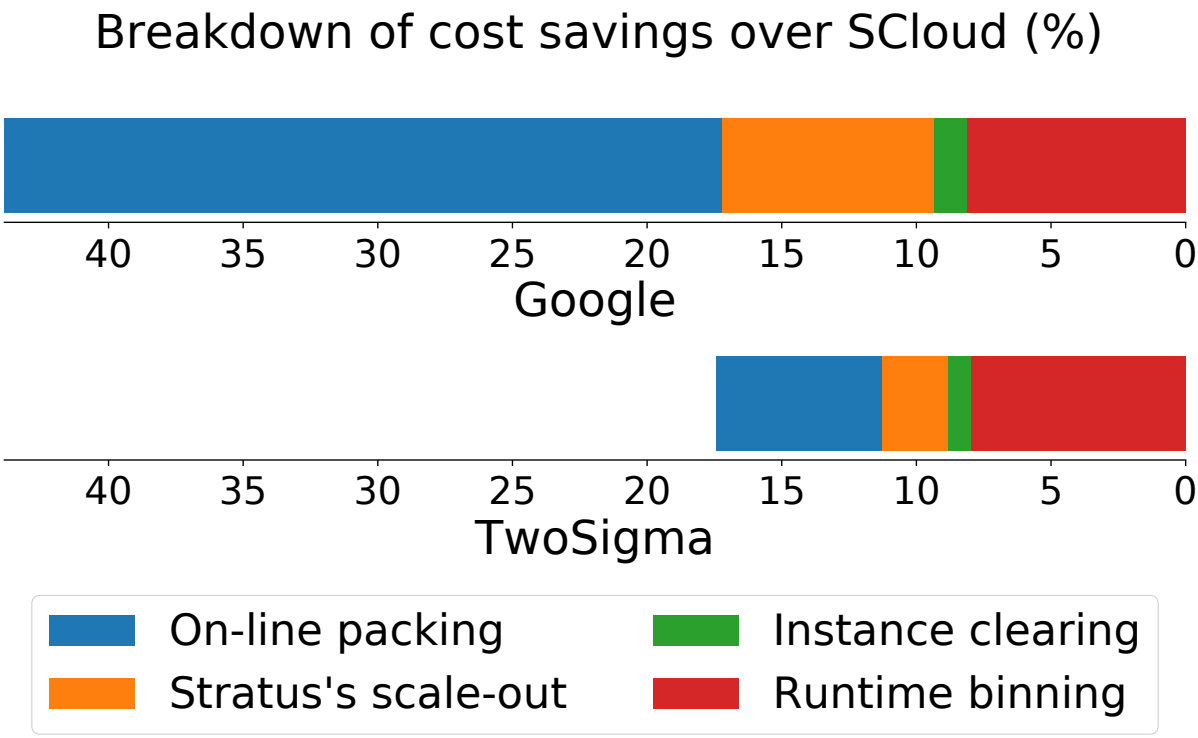


Figure 3.6: Break-down of Stratus's cost savings over SCloud (44% for Google and 17% for TwoSigma). The cost of running workloads reduces as Stratus features are added to SCloud, starting with features from left to right (on-line packing to runtime binning). The closer to zero, the smaller the cost difference between SCloud and Stratus.

3.4.2 Benefit attribution: SCloud to Stratus

Stratus uses a combination of heuristics to reduce cost. This section evaluates the incremental contributions of each by adding each to SCloud, one by one, until it matches Stratus. Fig. 3.6 shows the breakdown of how much of Stratus's cost savings is realized with each heuristic added to SCloud.

SCloud. We start the incremental build-up with SCloud, as described in §3.3.3, which only implements features as explicitly noted in the original SuperCloud-Spot paper. As discussed above (§3.4.1), Stratus reduces cost compared to SCloud by 44% on the Google trace and by 17% on the TwoSigma trace.

Adding online vector bin packing. To close the gap between SCloud and Stratus, we add support for packing new tasks on to running instances whenever possible, via the dot-product geometric heuristic for online vector bin-packing [59, 101] to SCloud. This allows released resources from completed tasks on running instances to be re-used. While this technique is effective in reducing the cost of SCloud, there remains a cost-gap of 17% (Google) and 11% (TwoSigma) between SCloud and Stratus. Indeed, both of our experimental workloads vary over time, and Stratus is more cost-effective at scaling in, as Stratus aligns task runtimes, compared to only using online vector bin packing.

Also adding Stratus's scale-out policy. While SCloud's greedy instance acquisition algorithm is

effective with tasks whose requests are uniform and tasks with resource requests only in a single dimension, it performs less well when tasks request a varying amount of resources in multiple dimensions. Using Stratus's instance acquisition scheme that considers the cost-per-resource-used of each group of tasks *assigned on to each instance* lowers the cost of SCloud with on-line packing (by 8% on the Google workload and by 3% on the TwoSigma workload). Implementing SCloud + on-line packing + Stratus's scaling heuristic closes the cost gap between SCloud and Stratus down to 9% (Google) and 8% (TwoSigma).

Also adding instance clearing. We found that incremental addition of instance clearing via migration did not help much. Interestingly, we found that instance clearing is not effective when used without taking task runtime into account. When instance clearing was used without taking task runtime into account on the enhanced SCloud, we found that cost increased on the TwoSigma trace for two reasons: (1) Although TwoSigma tasks are generally more uniform in runtime compared to Google tasks (Fig. 3.3a), task runtimes can become increasing mis-aligned with the introduction of instance clearing, as the runtimes of partially-run tasks may vary more significantly. Task runtime mis-alignment causes the number of instance under-utilizations to fluctuate, increasing the number of task migrations required. (2) TwoSigma tasks require more time to migrate, as they have larger memory footprints. Without knowing the cost-benefit tradeoff of clearing an instance, which depends on how much longer tasks will run, the number of task migrations can increase significantly.

Therefore, we enhance our enhanced SCloud with perfect runtime knowledge, such that it only migrates instances when the benefit in migration outweighs the task migration cost. Even with this unrealistic knowledge, instance clearing was not very effective in reducing cost in the enhanced SCloud. With previous features plus instance clearing, SCloud reduces the cost-gap by 1% (Google and TwoSigma) only. Stratus still reduces the cost by 8% (Google) and 7% (TwoSigma).

Side note: Stratus without instance clearing. In addition to evaluating SCloud enhanced with the previous features and instance clearing, we also evaluated Stratus with instance clearing disabled. The latter increases the cloud bill for Stratus by 28% for the Google trace and by 15% for TwoSigma. This shows that instance clearing is effective in assisting Stratus in putting tasks into their rightful bins, whereas it is less effective on other packing schemes where tasks are placed on to instances without regard to task runtimes. Unlike with the enhanced SCloud, less time is spent on task migration by Stratus (up to 23% less). Futher, tasks that have been migrated at least once in Stratus are only on average migrated 1.2 times before they reach an instance on which they terminate.

Also adding runtime binning. Adding Stratus's runtime binning to the rest of the enhancements to SCloud, we end up with Stratus.

3.4.3 Attribution: Dynamic instance pricing

Stratus's scale-out policy exploits dynamic instance pricing better than the other VC schedulers, because it considers different amounts of packing as part of selecting the most cost-efficient instance types based on current prices. Even with statically-priced instances like those offered in Google Cloud Engine and Microsoft Azure, however, Stratus's use of runtime binning and instance clearing to align co-located tasks' completion times remains beneficial.

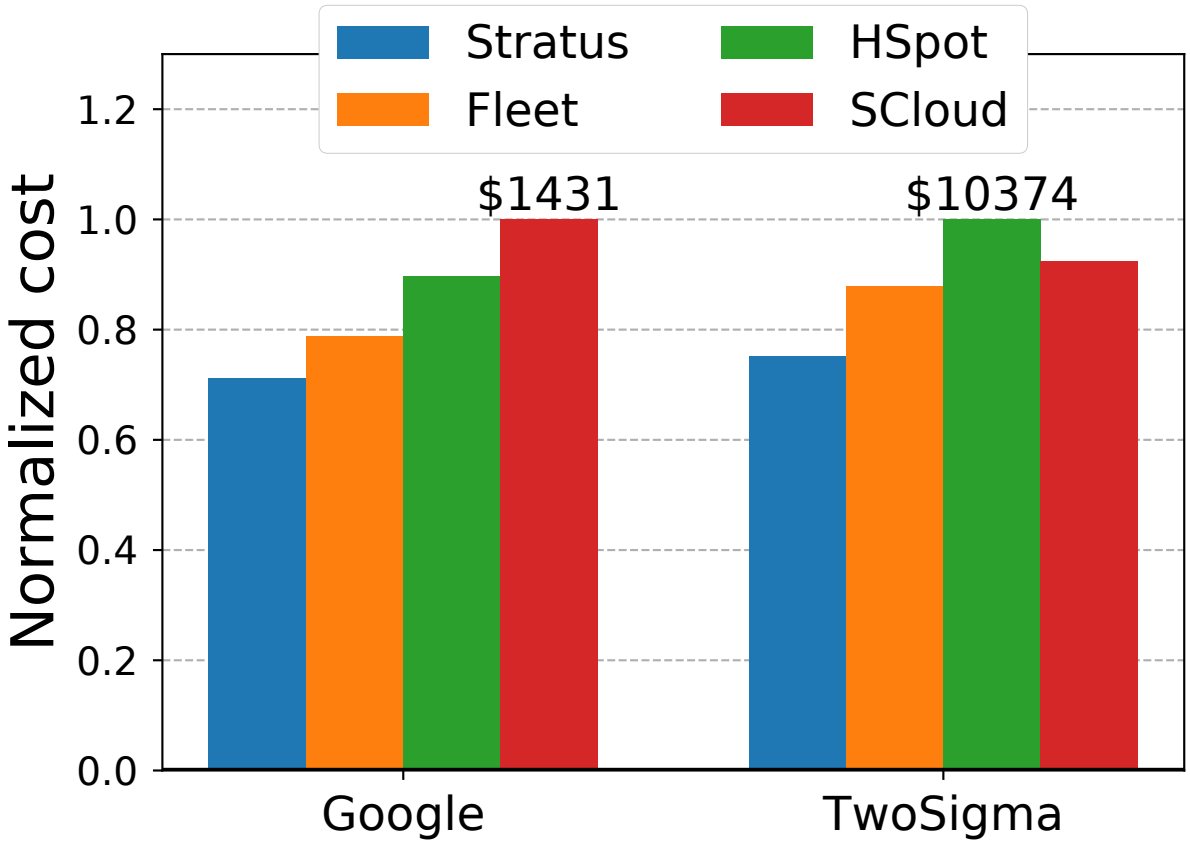


Figure 3.7: Average daily cost for each VC scheduler on the Google and TwoSigma workloads, using *only on-demand VMs*, normalized to the most costly option for each trace.

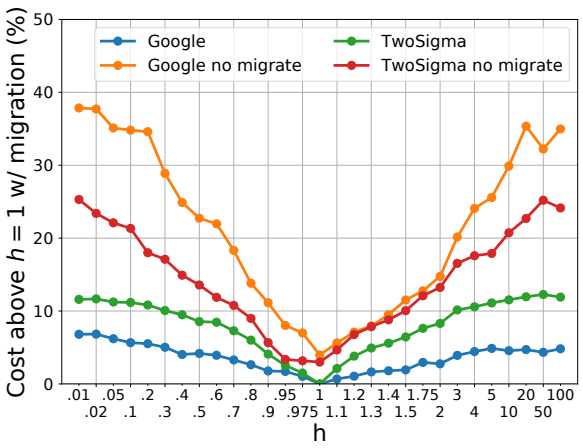
Fig. 3.7 shows the average daily costs of the VC schedulers when using only on-demand instances instead of the price-varying spot instances used in our other experiments. As expected, costs are much higher for all VC schedulers, since on-demand instances are usually more expensive than spot instances. Although the differences are somewhat smaller, the relative rankings of the VC schedulers are the same, and we find that Stratus still reduces cost compared to the others—by 10–29% for Google and by 14–25% for TwoSigma.

3.4.4 Sensitivity to runtime estimate accuracy

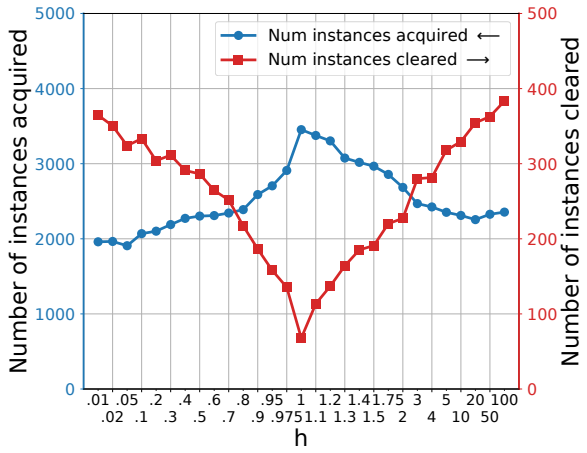
This section characterizes the effect of task runtime estimate accuracy on Stratus.

Stratus with perfect runtime knowledge

We evaluated Stratus with perfect runtime knowledge on the Google and TwoSigma workloads to see how much more Stratus could lower cost in this ideal scenario. As expected, enhancing Stratus with perfect runtime knowledge improves its cost-efficiency, further reducing cost by 5% (Google) and 9% (TwoSigma). Stratus with known runtimes reduces the cost of Stratus with JVuPredict by improving the constrained resource utilization of Stratus from 86% to 90% (Google) and from



(a) Stratus's cost savings on the Google and TwoSigma traces. Instance clearing is disabled on *no migrate* variants of Stratus. The cost increases as the range of potential estimate errors are expanded. Instance clearing is more beneficial when estimates are less accurate.



(b) Stratus's instance acquisition and clearing profiles on the Google trace. The number of instance acquisitions decreases as the range of potential estimate errors are expanded, while the number of instance clearings increases. Plot of results on the TwoSigma trace are comparable.

Figure 3.8: Experiments varying the degree of runtime estimate error in completing jobs from traces. Each experiment consists of tasks with runtime estimates set to runtime $* h_\tau$, where h_τ is uniformly sampled from $[h, 1)$ if $h < 1$ and from $[1, h]$ if $h \geq 1$.

79% to 84% (TwoSigma), and by reducing the total task migration time by 64% (Google) and by 82% (TwoSigma).

Less accurate task runtime estimates necessarily induce more task runtime misalignment for runtime-aware schedulers, leading to less effective usage of resources on instances as tasks do not complete in a coordinated manner. When task runtime estimates are inaccurate, tasks may unexpectedly complete early or late on an instance, leaving a portion of its resources idle. Although Stratus has implemented instance clearing to reduce the number of active VMs in case of under-utilization, instance clearing comes with non-trivial cost. Namely, tasks that are being migrated do not make any progress but reserve resources on both the source and the destination VMs, and during instance clearing, newly arriving tasks cannot be assigned on to the cleared VM such that the cleared VM can be terminated when its task migrations complete.

Runtime estimate accuracy sensitivity

Our previous experiments evaluate Stratus using a real state-of-the-art task runtime estimator and, in §3.4.4, a hypothetical runtime estimator providing perfect estimates. This section characterizes the effect of task runtime estimate accuracy on Stratus at a finer granularity by controlling the range of (synthetic) runtime estimate errors.

Setup. In each experiment, we generate runtime estimates for each task by scaling the actual runtime of the task by a factor of h_τ, where h_τ is uniformly sampled from a range of $[h, 1)$ if $h < 1$ and from $[1, h]$ if $h \geq 1$. Setting $h = 1$ is the same as using perfect task runtime knowledge. We perform 29 experiments on each trace, with each experiment consisting of five runs on different

	Google		TwoSigma	
h	50%-ile	95%-ile	50%-ile	95%-ile
0.01	1.3	3.2	1.1	1.7
1	1.3	3.3	1.0	1.6
100	1.3	3.2	1.1	1.8

Table 3.4: Normalized job latencies for values of h (§3.4.4).

slices of the trace, for $h \in [0.01, 100]$.

Cost trends and estimate accuracy. Fig. 3.8a shows Stratus's sensitivity with respect to cost to the accuracy of its task runtime estimates. As expected, cost increases as the quality of the runtime estimates degrades, whether under-estimates (to the left in the graph) or over-estimates (to the right). As runtime estimates become less accurate, Stratus makes less informed decisions in choosing which tasks to co-locate on instances.

Comparing variants of Stratus with and without instance clearing ("no migration"), we observe that instance clearing reduces the impact of runtime misestimates and misalignments. Stratus is efficient even without instance clearing when runtime estimates are accurate ($h = 1$), only incurring 4% (Google) and 3% (TwoSigma) more cost than with instance clearing. Instance clearing helps significantly when runtime estimates are increasingly inaccurate. Stratus achieves 31% (Google) and 14% (TwoSigma) lower cost at $h = 0.01$ with instance clearing than without. Cost savings at $h = 100$ with and without instance clearing is comparable.

The disparate behavior between the results with and without instance clearing for the Google and TwoSigma traces stems from the different characteristics of their jobs, as discussed in §3.3.2: TwoSigma jobs consist of tasks longer in duration (the time that new instances remain well-packed is longer) and often have more tasks per job (there are more tasks with similar runtimes).

Instance acquisition/clearing and estimate accuracy. The blue line in Fig. 3.8b shows Stratus's sensitivity to task runtime estimates with respect to number of instances acquired, while the red line shows Stratus's sensitivity to task runtime estimates with respect to number of instances cleared. Table 3.4 shows the normalized job latencies for polar values of h on the Google and TwoSigma traces.

As task runtime estimates become increasingly accurate, opportunities to release empty instances increase as task runtimes become better-aligned, decreasing the need to migrate tasks using the instance clearing heuristic. This, however, also means that as new tasks arrive there are less instances available on which to place the new tasks, leading to a greater number of instances acquired and a larger portion of job latency spent on waiting for instances to spin up.

Similarly, as task runtime estimates become less accurate, fewer instances are acquired since instances generally "stick around" for a longer period of time due to mis-aligned runtimes. Misaligning runtimes raises the chance to trigger instance clearing, increasing the number of instances cleared and causing the job latency to be increasingly dominated by task migration time.

Our experimental results (Table 3.4) show that for the Google and TwoSigma traces, the impact of increased instance spin-up time vs increased task migration time on job latency approximately balance out.

Applicability to other traces. Our sensitivity experiments find that the more accurate the runtime

predictions provided to Stratus, the more cost savings Stratus is able to achieve. This is consistent with our results in §3.4.1, where Stratus saves more on cost compared to other VC schedulers when provided more accurate task runtime predictions by JVuPredict in the Google trace (as opposed to less accurate task runtime predictions provided by JVuPredict in the TwoSigma trace). Stratus's scheduler can similarly help save on cost in HPC workloads run on OpenTrinity and Mustang clusters [25]. We believe that Stratus will perform even better on traces from the Mustang cluster compared to its performance on the Google traces, as JVuPredict provides more accurate predictions there. On the other hand, we believe that Stratus will perform the worst relative to other schedulers on traces from the OpenTrinity cluster, as JVuPredict provides the least accurate predictions on OpenTrinity traces.

3.5 Summary

The Stratus cluster scheduler exploits cloud properties and runtime estimates to reduce the dollar cost of cluster jobs executed on public clouds. By packing jobs that should complete around the same time, simultaneously considering possible packings and available instance types/prices, and judicious use of task migration to clear under-utilized instances, Stratus actively avoids having leased machines that are not highly utilized. We expect Stratus's approach to be a core element of future virtual cluster management for public clouds.

Chapter 4

Unearthing inter-job dependencies for better cluster scheduling

This chapter describes our work on Wing [36] and Owl [35], tools that can help both users and operators increase their value attained through understanding input and output relationships of their jobs.

Wing, which we will spend the majority of this chapter discribing, is an inter-job dependency profiler that uncovers and analyzes hidden inter-job dependencies, using historical data provenance and job logs. Our work shows that by relying completely on Wing for guidance, a batch analytics job scheduler can achieve nearly 100% of user value at constrained cluster capacities, almost $2\times$ that achieved by using the default user-provided job priorities.

On the other hand, Owl, described in §4.7, is a user-oriented visualization tool that uses Wing's analyses to help users understand how their jobs depend on other jobs, and who is consuming the output of their jobs, which in turn allows users to better determine the *importance* and *urgency* of their jobs.

Today, *data lakes* have become core elements of modern data-driven enterprises, providing required data storage and analysis infrastructure (see Fig. 4.1). Data lakes enhance data processing via a combination of two critical properties: (i) a highly consolidated, multi-tenant infrastructure that enables multiple teams of data scientists and engineers to share resources rather than each having their own, and (ii) low data access barriers that allow easy data sharing between users and various types of data analytics applications. Combined, these properties increase data re-use [28, 66] and reduce overall computational resource-hours consumed [75, 76].

This same data and resource sharing creates a new challenge: hidden inter-job dependencies. We say that Job 2 depends on Job 1 if Job 2 takes as input any output file generated and stored into the shared distributed file system by Job 1.[1] For example, in Fig. 4.1, Job 3 (from Org 3) depends on Job 2, which in turn depends on Job 1. We refer to these as *hidden* dependencies, to contrast them with explicit computation DAGs managed by schedulers within workflow managers [73, 96, 97], because there is no indication of such dependencies indicated in the job submissions—the dependencies are not expressed to the cluster scheduler.

[1] Our nomenclature and analyses focus on fundamental dataflow dependencies among batch analytics jobs, not distributed stream processing or artificial inter-relationships caused by resource contention.

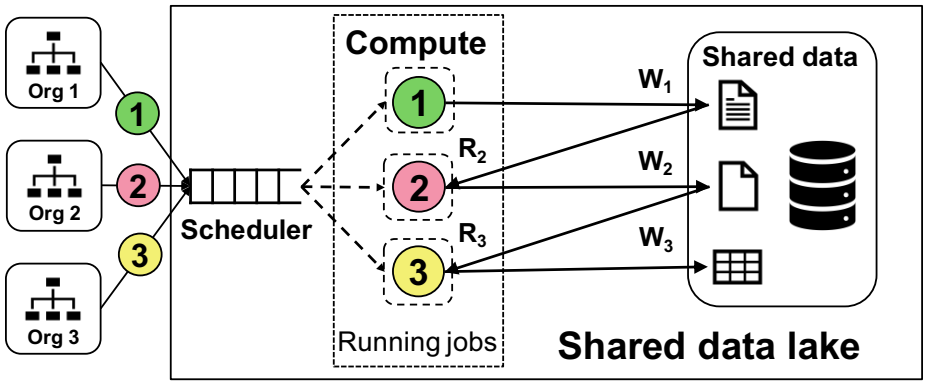


Figure 4.1: Data lake overview. Different jobs submitted by different organizations share the same compute infrastructure and read (R) and write (W) to the same storage system, thereby creating *inter-job dependencies* as jobs consume the output of other jobs. e.g., Job 2 (from Org 2) reads a file written by Job 1, so Job 2 depends on Job 1.

The advent of GDPR [127] forced large companies such as Microsoft to invest in infrastructures to track data provenance and data movement both within the data lake and to external components. This created an unprecedented opportunity to uncover and exploit these inter-job dependencies for scheduling: We analyze data extracted from petabytes of job and data provenance logs for 90 days of a 50k+ server cluster (part of Microsoft's Cosmos data lake [31, 39]) shared by over 1300 users from more than 150 internal organizations. In total, our analysis covers over 4 million submitted jobs and 16 million inter-job dependencies. We find that almost 80% of submitted jobs depend on output generated by at least one other job. Indicating the breadth of sharing, many dependencies are cross-organization, with 20% of jobs depending on jobs submitted by another organization.

Despite so much inter-job dependence, systems provide little support for addressing associated challenges. For example, in Cosmos, different users and organizations make their own decisions regarding when to submit jobs and how to set job priorities. Ideally, all co-dependent organizations and users would set up clear Service Level Agreements among themselves to ensure timely arrival of input data for business-critical analyses. Yet, we see signs of insufficient coordination to ensure that jobs' outputs are produced in time for consumption by dependent jobs. For example, 13% of submitted jobs depend on output files from jobs that execute at a lower priority, which can result in priority inversion since job schedulers are not dependency-aware. More broadly, 34% of recurring jobs are submitted without checking if inputs they depend on are available, failing immediately if they are not.

The Wing dependency profiler efficiently processes prior job and provenance data to predict the impact of each new job on future jobs and user downloads. Although it is inherently difficult to know what future jobs will depend on the output generated by a current job, Wing finds success by focusing on recurrence. Previous workload studies have shown that $> 60\%$ of jobs in data analytics environments are *recurrent* and suggest that dependencies of these jobs can similarly follow certain patterns [78, 121]. Our analyses in Cosmos confirm that inter-job dependencies are recurrent (79% of all inter-job dependencies are recurrent), with jobs of the same template exhibiting recurring input consumption patterns. As such, Wing uses *historically recurring*

dependencies to (i) analyze and predict relationships between common, dependent recurring jobs, and (ii) guide a cluster scheduler to value jobs in a way that accounts for hidden dependencies.

To explore Wing's efficacy, we pair Wing with stock YARN scheduling (*Wing-Agg*), replacing user-provided priorities with Wing-guided priorities. Specifically, we use number of downloads attained associated with a job's outputs as an approximation for job value,[2] and assign priorities to jobs based on *value efficiency* [33, 71, 103] (job-value divided by resource-time-used). We use trace-driven simulation to evaluate Wing-Agg, compared to using the user-provided priorities (as used in Cosmos), when the goal is to maximize the overall value attained. We find that Wing-guided scheduling achieves up to 66% more value than the Cosmos default, under cluster capacity crunch. Further, when organizational cluster resource boundaries are removed, a Wing-guided scheduler can achieve nearly 100% of value at constrained cluster capacities, almost 2× the value achieved by scheduling based on user-provided job priorities.

Contributions. This chapter makes four primary contributions: (i) It presents the first detailed public study of hidden inter-job dependencies in a large-scale data analytics cluster, revealing important problems and opportunities; (ii) it describes a novel system for extracting historical inter-job dependencies from provenance data, at scale, and predicting the impact of a newly–submitted job on future jobs and users; (iii) it shows that use of such predictions can allow a modern scheduler, with minimal changes, to better serve the overall workload by prioritizing the highest-impact jobs; and finally (iv) it demonstrates how Wing's analyses can be surfaced to users via Owl (§4.7), a tool that can help users identify critical job dependencies and visualize job impact.

4.1 Hidden inter-job dependencies in Cosmos

This section describes and analyzes hidden inter-job dependencies in a large production data lake (Cosmos), highlighting observations that affect resource scheduling decisions and opportunities. It provides an overview of Cosmos and inter-job dependencies, introduces terminology used through the rest of the chapter, and quantifies the prevalence and characteristics of hidden inter-job dependencies.

4.1.1 Cosmos

Overview. Cosmos is one of the largest big data analytics infrastructures in the world. Deployed internally within Microsoft, it is made up of multiple clusters, each with 50k+ nodes [39]. Within Cosmos, more than 80% of infrastructure capacity is dedicated to SCOPE jobs [31, 39], which are batch data analytics jobs similar in nature to Apache Spark [133] and MapReduce [42]. Our work primarily focuses on SCOPE jobs and inter-job dependencies between them.

ADLS and operations. SCOPE jobs submitted to a Cosmos cluster read input from and write output to a distributed file system known as the *ADLS*. A user can also access ADLS through a front-end service to upload or download files directly. We call actions performed on files in ADLS, either by SCOPE jobs or through the front-end, *operations*.

[2]While job output download-counts are imperfect as ground-truth for job value, a limited check (§4.3.3) against known important levels for six business-critical jobs indicates that it at least sometimes behaves reasonably.

Characteristic	Description	Heuristic
Recurring	*Recurring jobs* are jobs whose template is submitted many times over time, often to analyze fresh data. *Recurring dependencies* are dependencies occurring between jobs of two recurringly-submitted job templates.	Borrowing from Morpheus [78], jobs are identified as *recurring* if (a) jobs of a template are submitted at least three times over a period of three months, with at least one submission each month, (b) templatized job names are an exact match, and (c) source-code signatures are an approximate match. Dependencies are identified as recurring if both the upstream and the downstream jobs are recurring.
Ad-hoc	*Ad-hoc jobs/dependencies* are those not recurring.	*Ad-hoc jobs/dependencies* are those not identified as recurring.
Periodic jobs	*Periodic jobs* are recurring jobs that are submitted "on-the-clock" at a fixed cadence (e.g., submitted every hour at the start of the hour).	Jobs of a template are identified as *periodic* if they are recurring and if job submissions have near-constant inter-arrival time. To determine if inter-arrival times are near-constant, we use the coefficient of variation (CV). Jobs with small CV in their inter-arrival times are identified as *periodic*, while others are *aperiodic*.
Polling	Jobs are *polling* if they scan and wait for their inputs to become available before their submission. Input dependencies of polling jobs are similarly polling.	Jobs are identified as *polling* if they (a) are not identified as periodic, indicating that they are not submitted on a clock, (b) never fail due to missing files from their recurring upstream jobs, and if (c) they are submitted within 15 minutes of the completion of their latest-completing dependent job. Input dependencies of a polling job are *polling*.
Hard dependencies	Dependencies are *hard* if the downstream job requires the output(s) of the upstream job to be able to run successfully. If the input(s) of the downstream job is not ready by the time of its submission, the downstream job fails with a missing file exception.	Dependencies are identified as *hard* if they are (a) ad-hoc, (b) recurring *and* $> 95\%$ of jobs of the same template consume the output of only one job of the same upstream job template, or (c) if the downstream job consumes the output of the same number of upstream jobs of the same job template all the time, indicating that they expect the same number of inputs from the same number of jobs from the upstream template.

Table 4.1: Summary of and heuristics to identify and characterize job and dependency types.

Continuous logging. Cosmos continuously tracks and logs data provenance and job telemetry (e.g., compute-hours, submission/completion time, and job structure metadata) into external services: *ProvRepo* stores data provenance and and *JobRepo* stores job telemetry. Our analyses and Owl use these logs to figure out inter-job dependencies.

Job template vs job. A job template [77, 78] is a program to be executed (one or multiple times) in Cosmos, while a job is an actual execution of a job template. Each submission of a job template results in a job.

SCOPE job submission patterns. Common patterns used to submit SCOPE jobs within Microsoft include:

(i) *Manual submissions*: Where a job is manually submitted.

(ii) *Workflow managers*: Workflow managers allow users to automate SCOPE job submissions using *workflows*. Workflows consist of inter-dependent jobs that often map to a business task,

and can be triggered periodically or conditionally. Within Microsoft, there are at least five major production workflow managers, each with thousands of users.
(iii)*Custom shell scripts*: Scripts can be set up to perform automated job submissions for users. This method is more flexible, but requires specialized management.

4.1.2 Inter-job dependencies

How are inter-job dependencies formed? We say that a job A depends on a job B if A consumes *any* of B's output as input. As a concrete example of a recurring cross-organization inter-job dependency, periodic jobs deployed by the data compliance team process ADLS access logs, which are generated hourly by the ADLS team, to detect data compliance issues. There are many ways inter-job dependencies can form, and while some inter-job dependencies form through careful negotiation between users/organizations, most are formed *organically*, such as via:
(i) *Data discovery through data catalogs*: A user finds an interesting dataset while browsing through Microsoft's internal data catalog, and sets up a job to analyze the dataset.
(ii) *Script inheritance*: A user wanting to submit a SCOPE job to analyze a popular dataset often starts with a script written and shared by others, that contains logic to extract the dataset. The new script, while containing custom logic, often retains parts of the original script (e.g., priority settings).
(iii) *Logically related intra-workflow jobs*: Workflows, which can consist of multiple inter-connected jobs, are often constructed to improve job modularity and manageability. Each run of a workflow potentially creates many inter-job dependencies, as jobs within a workflow are inter-dependent. Note that, although a workflow manager may know about these inter-job dependencies, there is no interface for a workflow manager to express them to Cosmos.
Characteristics of jobs and dependencies. Our analyses uncovered a few major types of dependency and job characteristics based on job submission patterns (Table 4.1). The three most important job and inter-job dependency characteristics for our purposes are *recurring*, *ad-hoc*, and *hard*.
Challenges. Among the many ways in which inter-job dependencies can form and evolve, most promote loosely maintained (or non-existent) contracts between inter-dependent jobs in favor of developer convenience. This leads to an environment in which most users know little about upstream jobs that produce their input datasets, and even less about downstream jobs that depend on the data their jobs produce. These sub-optimal inter-job dependency configurations are often only exposed as a result of capacity impairment, unexpected job failures, or data/job audits. Indeed, inter-job dependencies are *hidden* through the availability of the many disaggregated solutions to manage and submit jobs and workflows, prompting us to develop Owl to uncover these dependencies.

4.1.3 Observations on inter-job dependencies

This section motivates our work on exploiting inter-job dependencies by describing consequential empirical observations about our inter-job dependency data, observed over three months in a single Cosmos cluster.

Observation 1 (Recurring jobs & dependencies): Most jobs and dependencies are recurring. Recurring jobs make up 68% of all submitted jobs (the other 32% of jobs are ad-hoc), while recurring dependencies make up 79% of all dependencies (the other 21% of dependencies are ad-hoc). Recurring-ness of jobs and dependencies suggest predictability, which we show to be achievable in §4.2.

Observation 2 (Priority mis-configurations): In Cosmos, jobs are assigned resources in declining priority order, where the priority of a job is assigned by the job's submitter. Here, we find that potential priority mis-configurations are frequent within Cosmos: jobs of 21% of job templates have the chance to be systematically priority-inverted—i.e., recurring jobs consuming their output have a higher priority. In addition, up to 33% of ad-hoc jobs are assigned higher priority than the average recurring job submitted within the same hierarchical queue,[3] where recurring jobs are often production jobs [78].

Observation 3 (Uncoordinated jobs): Many jobs are submitted without explicit coordination with respect to the completion of their upstream jobs—i.e., these jobs do not wait for their input to become available nor are tolerant to missing input, yet they are submitted blind with respect to the availability of their inputs. Such jobs make up 34% of recurring jobs, and can be susceptible to failure due to missing input from an upstream job not completing in time.

Observation 4 (Cross-org jobs & dependencies): Cross-org jobs and dependencies are common at Microsoft. Up to 95% of organizations have cross-org dependencies. Of all dependencies, 33% are cross-org, and 17% of template dependencies are cross-org, where a *template dependency* is a dependency between recurring jobs of two job templates. Furthermore, 28% of jobs and 23% of recurring jobs are involved in cross-org dependencies. Cross-org dependencies can be harder to manage because they require coordination between jobs across hierarchical queues and between job owners across different organizations.

Observation 5 (Jobs are highly inter-connected): Modeling jobs and their dependencies as a directed acyclic graph (DAG), where inter-job dependencies represent edges, we find that more than 50% of jobs are inter-connected in a single weakly connected component (CC), and CCs of sizes ≥ 10 cover more than 80% of all jobs. We also find that the larger a CC, the more bottom-heavy it is—the failure of certain jobs in such large CCs can cause significant amounts of cascading failure downstream.

Discussion. We have seen failure due to lacking input and priority inversions happen during manual inspection of job logs and dependency graphs, but we can not provide counts. We have also seen that: (1) users can and do fix their jobs, sometimes at the cost of sub-optimal performance and results, to work around issues, such as by by consuming stale data; and (2) some of these problematic inter-job dependencies can be masked with sufficiently available resources. A better understanding of inter-job dependencies can help us uncover problematic mis-configurations before they show up.

[3] *Hierarchical queues* designate resource shares of an organization in clusters at Microsoft. Priorities are only comparable between jobs in the same queue.

4.1.4 Limitations

While inter-job dependencies provide us with many valuable benefits, they are not without limitations. One such example happens when dependencies exist without explicit interactions with ADLS files. For example, alerts that monitor the health of a Cosmos workflow may not explicitly download expected output files, but rather may only check for the existence of the expected outputs. While the API calls to check for file existence are logged, Wing cannot tell whether these API calls are strict dependencies on outputs of ADLS operations.

Another limitation occurs due to ProvRepo's strict file versioning. In ProvRepo's logs, each "overwrite" of a file produces a new entry in ProvRepo, such that each version of each file is treated as a unique file. For example, if (1) job A writes file *F*, (2) job B reads *F* and overwrites *F* in its output, and (3) job C takes file *F* as input, Wing will output that C depends on B, and B depends on A. In certain cases, it is possible that job C can take job A's output directly as input, but Wing will believe that job C requires job B to complete in order to run successfully. This limitation imposes stricter requirements on job input dependencies, and can, for example, limit the load-shiftability of jobs (Chapter 5).

4.2 Inter-job dependency predictability

Inter-job dependencies show potential in guiding scheduling; but it is unrealistic to expect job submitters to provide all inter-job dependencies up-front due to the fragmented nature of inter-dependency knowledge (§4.1.2). While *inter-job dependency recurrence* shows promise, for Wing to effectively guide schedulers with inter-job dependencies, recurring inter-job dependencies also need to be *predictable*—i.e., it is important that past dependencies tell us something about the future. In this section, we use a simple model to predict future occurrences of recurring inter-job dependencies, and show that inter-job dependencies can be predictable.

4.2.1 Prediction model

Given a specific point in time where a job j_u of template J_u ($j_u \in J_u$, where the symbol "$\in$" is used as shorthand for "of instance") has arrived, for each recurring job template J_d that depends on the output of template J_u in a recurring fashion, our prediction model has two targets: **(T1)** *whether or not* a recurring job $\in J_d$ will arrive and depend on j_u in the future and **(T2)** *when* the first instance of such a job will arrive.

Model for (T1): Will a downstream recurring job arrive? For (T1), our model uses a configurable *prediction threshold* $tr\%$ ranging from 0 to 100 to predict whether or not a job $\in J_d$ will arrive: If $\geq tr\%$ of prior jobs $\in J_u$ have their outputs consumed by a job $\in J_d$, then the predictor predicts `true`; otherwise it predicts `false`.

Model for (T2): When will a downstream job arrive? For (T2), our model aggregates previously observed recurring dependencies where the upstream job $\in J_u$ and the downstream job $\in J_d$, and computes the median elapsed time from the *submission* of the upstream job to the submission of the first dependent downstream job.

4.2.2 Predictability evaluation

Dependencies change slowly over time. Dependency patterns of recurring jobs change slowly over time, and making predictions based on inter-job dependencies over longer periods of time presents challenges. For example, in (T1), using a month of inter-job dependency data to train our model to predict the arrival of dependent jobs occurring in the next month only allows us to capture at most 77% of upcoming jobs. Regularly training our model on a week of data to predict for the next week works comparatively well, because (1) it allows us to capture up to 95% of upcoming jobs and (2) it allows us to characterize the dependencies of 89% of job templates (covering 97% of all jobs), since jobs of most templates are submitted with an inter-arrival time of less than a week (with daily submissions being the most common).

(T1) metrics and model performance. We evaluate the prediction quality of our model on (T1) based on precision[4] and recall.[5] Fig. 4.2 examines the tradeoff between precision and recall for our model using various settings for the prediction threshold tr. As the model becomes more selective with respect to which downstream jobs will arrive ($tr \rightarrow 100\%$), it retains less relevant dependencies in total, but the dependent recurring jobs it predicts to arrive mostly do show up. The reverse is true as the model becomes less selective ($tr \rightarrow 0\%$).

We discuss the evaluation of our model based on a threshold that balances precision and recall. A common way to identify such a threhold is to select the threshold that maximizes *precision* $*$ *recall*. We find that $tr = 20\%$ yields the greatest $precision * recall$, and therefore evaluate our model by setting $tr = 20\%$. The threshold used in an online prediction service can similarly be tuned from week-to-week based on observed precision and recall, though the specific target to optimize depends on the penalties associated with making mistakes in recall or precision.

(T2) metrics and model performance. To evaluate the performance of our model on predicting *when* a downstream job $j_d \in J_d$ will arrive at the arrival time of an upstream job j_u, j_d must satisfy two conditions: our model must predict j_d to arrive based on jobs that have already arrived during a point in time in the execution trace *and* it must actually arrive. Our evaluation focuses on jobs that satisfy both above conditions.

To evaluate the performance of our model for (T2), we use the *Root Mean Squared Error (RMSE)* and the *Median Absolute Error (MAE)* metrics to measure prediction error in absolute time units. RMSE measures error by computing the root of the average of squares of errors, while MAE measures error by computing the median of absolute error $= |forecast - actual|$, over all predictions. To measure relative error, we use the *percentage error* metric: it computes $(forecast - actual)/actual$ for each prediction.

While we discuss the evaluation of our model on (T2) setting $tr = 20\%$, we find that confidence in job arrival prediction only slightly affects time-to-dependency prediction quality. This does not mean that the setting of tr is inconsequential, as tr affects the predictions of whether or not a job will arrive. Here, we evaluate the time-to-dependency predictions only for jobs that are both predicted to arrive *and* actually arrive.

[4] *Precision* is defined as the number of true positives (TPs) divided by the sum of TPs and false positives. Precision can be thought of as the percentage of positive predictions our model makes (i.e., a downstream job will arrive) that are truly relevant (i.e., such a downstream job actually arrives).

[5] *Recall* is defined as TPs divided by the sum of TPs and false negatives. Recall can be thought of as the percentage of relevant results that our model is able to correctly predict.

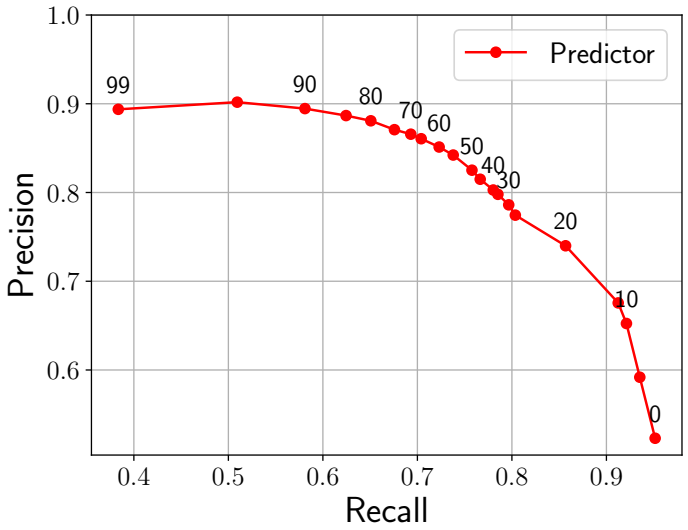


Figure 4.2: (T1) precision-recall tradeoff. *Predictor* shows the precision-recall tradeoff our dependency-based job arrival predictor makes. Each point on the curve specifies a different setting for the prediction threshold (tr). As $tr \rightarrow 100\%$ (more selective), a larger fraction of predictions are relevant (more precision), but less relevant jobs are captured in total (less recall).

We observe the RMSE and MAE of our model to be 2.5 hours and 22 minutes, respectively: MAE is smaller, as RMSE can be skewed by large mis-predictions at the tail. While our absolute errors can be improved using more sophisticated techniques, we find that our model predictions are reasonable for most jobs in our workload in terms of relative error, as shown in Fig. 4.3 in the form of a cumulative distribution function (CDF), for different settings of tr: the arrival of 50% of arrived jobs $\in J_d$ are predicted within $\pm 20\%$ of its actual arrival. But, there is also a non-trivial number of significant over-estimates: the arrival of 7% of arrived jobs $\in J_d$ are over-estimated by $2\times$ or more—i.e., the actual jobs arrive more than $2\times$ earlier than predicted. While this may not explain all mis-estimations, we have found that aperiodic recurring jobs (such as those that are manually triggered) and jobs that depend on the outputs of multiple jobs are prone to greater mis-estimates (our simple model presented here only tries to predict the arrival of a future job based on one of its directly upstream recurring jobs).

4.3 The Wing dependency profiler

This section describes Wing, an end-to-end dependency profiler meant to be run intermittently (e.g., weekly) that uncovers historical, hidden inter-job dependencies from data provenance logs. It performs a series of analyses using these inter-job dependencies in-tandem with historical job telemetry, yielding characterizations of jobs and inter-job dependencies such as signs of misconfigured priorities between recurringly-dependent jobs (§4.1.3), predictability of upcoming jobs (§4.2), and estimates of recurring jobs' aggregate value considering their impact on downstream jobs that rely on their outputs, directly or indirectly (§4.3.3). These characterizations are ultimately used to inform better scheduling decisions, where its benefits are explored in more

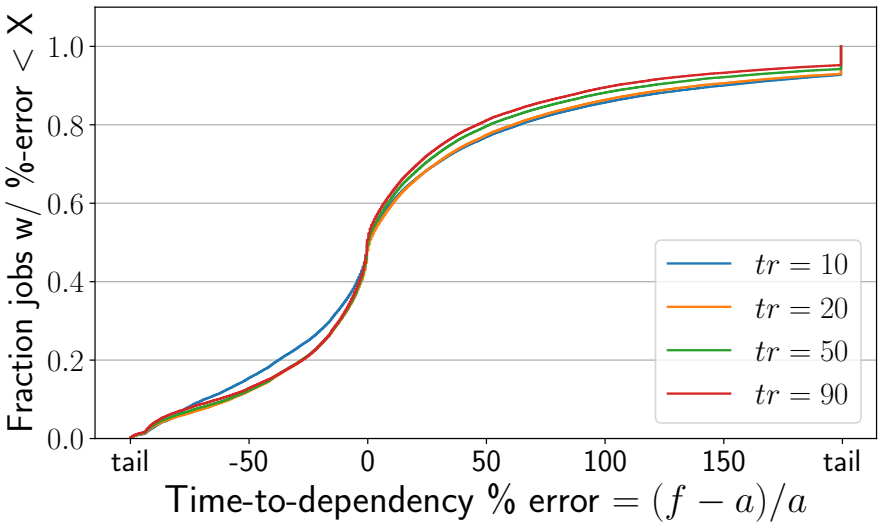


Figure 4.3: (T2) Time-to-dependency (TTD) prediction. This figure shows our predictor's performance on predicting TTD from the submission time of the upstream job, at different settings of tr in a CDF. f is the *forecasted* TTD, and a is the *actual* TTD. While being more precise ($tr \rightarrow 100$) does not yield better TTD predictions, it does affect predictions on whether or not a job will arrive.

detail in §4.6.

4.3.1 Architecture

First, we introduce related systems and data sources upon which Wing depends, and provide an overview of Wing's architecture, shown in Fig. 4.4.

Input data sources. Wing relies on the following data sources, from which we derive job dependencies and insights thereof: (i) *JobRepo* preserves job telemetry (e.g., compute-hours, submission/completion time, and job structure metadata) for submitted jobs. Wing uses JobRepo to derive recurring jobs and their historic statistics. (ii) *ProvRepo* tracks data provenance across Microsoft to support auditing and compliance applications [127]. Specifically, it stores data provenance across systems deployed within Microsoft, including but not limited to Cosmos. ProvRepo is used by Wing to uncover historic inter-job dependencies, and from there, infer recurring dependencies between recurring jobs.

Analysis pipeline. Wing's data analysis pipeline, primarily composed of a workflow of inter-dependent SCOPE jobs, is managed by a workflow manager and is periodically executed in Cosmos. The pipeline reads data from JobRepo and ProvRepo and writes its output to be consumed by WingStore.

WingStore. The *WingStore* is a service that hosts the resulting analyses of Wing's analysis pipeline, periodically renewed each time a new instance of the pipeline completes. Given a historical job or the identifier of a recurring job, one can look up relevant historical job and inter-job dependency data: Such historical job data include, but are not limited to, distributions of job runtime and compute-time used. Historical inter-job data include distributions of job fan-in/fan-out, recurring inter-job dependencies, and distributions of number of downstream jobs.

The WingStore is the interface between Wing's analysis and a Wing-guided resource manager.

4.3.2 The Wing pipeline: Single-hop analysis

Wing considers both *single-hop* and *multi-hop* dependencies in its analyses. The former occur when jobs directly consume the output(s) of another job. The latter are indirect dependencies between jobs that are connected by means of intermediate jobs. Here, we focus on the derivation of single-hop dependencies, which multi-hop dependencies are built upon, from historical provenance data stored in ProvRepo.

Single-hop dependency derivation. To derive single-hop dependencies from provenance data in ProvRepo (stored roughly in the form of <input, operation, output>, but with much more detailed context), we perform a self-join on the ProvRepo dataset with the condition of p_1.input = p_2.output. A näive self-join across multiple months of data is extremely compute intensive and can yield incorrect results, as a single file can be written multiple times by different jobs. To reduce join complexity and ensure correctness, we apply the following additional rules on the join:

(1) *R/W correctness*: The read must occur after the write. i.e., p_1.operation must occur after p_2.operation.

(2) *Last-writer wins*: If multiple writes occur on a single file, the read only depends on the latest write prior to the read.

(3) *Time windowing*: The time between the read and the write operations are at most T days, where we set $T = 30$.[6]

Time windowing can reduce join complexity and allow our analyses to account for inter-job dependencies more *fairly*—if time windows are not applied over an observation period, operations issued earlier necessarily have a higher chance to be depended-upon. In other words, for each operation between days 31–60[7] in our dataset, time windows give them equal opportunity (in wall-clock time) to be depended-upon by directly-dependent operations.

Heuristics to identify recurring jobs & dependencies. A key to the analyses that we perform is the identification of recurring jobs, for which we employ the time-tested heuristic proposed in Morpheus [78] and applied in multiple production environments [78, 121]. Through the identification of recurring jobs and uncovered single-hop dependencies, the Wing pipeline further derives recurring dependencies and uncovers dependency characteristics of jobs using similar heuristics, described in Table 4.1. While ideally, we would like the full semantics of how inter-job dependencies are formed, due to the availability of the many different ways to submit a job (§4.1.1), our usage of heuristics is necessary. Sampling 25 jobs for manual verification, we confirm that our heuristics categorize jobs and dependencies correctly for 24 of the jobs.

[6] In retrospect, we should have set $T = 31$ to capture all monthly cycles, but our results based on $T = 30$ remain valid because (1) 98% of dependencies occur within a week, and (2) jobs of 89% of templates (97% of all jobs) have mean inter-arrival times of less than a week.

[7] Operations between days 31–60 are analyzed because we observe fully over time windows of 30 days both operations they depend on (days 1–30) and those that depend on them (days 61-90).

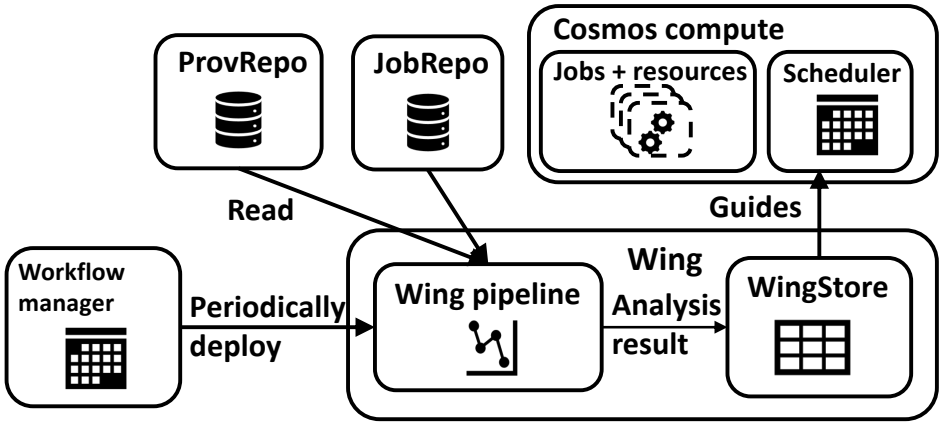


Figure 4.4: Wing architecture. A workflow manager periodically submits Wing's pipeline to Cosmos. Upon pipeline completion, results of its analyses are loaded in to WingStore, which informs Wing-guided schedulers (§4.5.2) with job and dependency characteristics.

4.3.3 Motivating multi-hop analysis: Job valuation using aggregate downloads

Companies can benefit more from their infrastructure investment through effective scheduling that prioritizes the completion of the most valuable jobs. But, often times, inter-job dependencies have not been considered when evaluating the importance of jobs—e.g., a job with high value can potentially depend on jobs with low value. In these cases, inter-job dependency awareness is key to ensure that upstream jobs do not disrupt high-value downstream jobs. Here, we look at why inter-job dependency analyses beyond direct dependencies (i.e., *multi-hop analyses*) can inform better, dependency-aware valuation of jobs to improve scheduling, and explore using the *number of downloads attained* associated with the outputs of a completed job as a proxy-metric for job value.

Priority assignments. To prioritize jobs today, schedulers in most production data analytics environments, including in Cosmos, use priority assignments to determine a job's order in its claim to resources. In this context, the notion of job value is often translated into a priority assignment on the job—the greater a job's value, the higher its priority. However, priorities in clusters are difficult to set correctly (Observation 2), and even at Microsoft, whose multi-billion dollar clusters are carefully provisioned and whose user-base is highly skilled, incidents triggered by late completion of hand-picked, closely monitored, and highly valued production jobs still occur due to mis-configured priorities.

Multi-hop value impact. The completion of a job can often be associated with some measurement of monetary value to a company. For example, jobs computing Bing's search indices directly impact the revenue of Microsoft. We term the direct value associated with the completion of a job its *job-local* value. However, the delay or failure of a job may not only affect its users and consumers of its output: through analyses of Cosmos's job DAG (Observations 3 and 5), we find that the delay or failure of certain jobs impact a lot more jobs and users than others. Hitches in the execution of these jobs are likely to cause much more financial and operational damage to users

and organizations within the company due to the ripple effects they can create downstream, yet their impact might not always be obvious. While prior work [49, 78] suggest that finishing jobs prior to the arrival of their first directly-dependent job is important, quantifying the *aggregate value* of a job necessitates inter-job dependency analyses extending beyond a single hop (i.e., *multi-hop analyses*). Fig. 4.5 showcases a toy example that computes such an aggregate value for the root job of a dependency tree.

Approximating value impact with agg. downloads. Although determining the true dollar-value of jobs is difficult, we find it promising to evaluate the importance of jobs based on their historical *aggregate user downloads*, which measures hypothetically if a job fails, how many download operations it will affect (directly or indirectly) in total. In developing Wing, we have also experimented with several alternative metrics e.g., sum of cpu-hours and number of downstream jobs. Number of downloads was preferred by our resource management team because file downloads (1) are the most direct way users interact with a job's output; (2) can be easily interpreted and understood; and (3) because file downloads can be used to quantify how soon the output(s) of a job are used upon its completion. The properties of file downloads allow aggregate downloads counts to provide a proxy-measure to how the delayed or failed outputs of jobs can impact users in and out of Microsoft. Aggregate download counts also implicitly capture the number of downstream jobs that can be impacted by the failure of a job through their associated output downloads. While further work is required to confirm that aggregate download counts represents job value and to explore how it should be combined with other signals (e.g., user-provided priorities), we use it in this chapter as our approximation of value.

Sanity-checking aggregate downloads as job value. We conducted a sanity check, using aggregate download counts for job valuation to see how it matches up with pre-existing notions of job importance. To that end, we obtained a list of six recurring job templates hand-curated by the Cosmos resource management team at Microsoft, each vetted to be significantly important to Microsoft's operation. We then look at Wing's ranks of those jobs.

Our results show that our valuation scheme mostly holds up for the most important jobs: We find that jobs of five of the six templates are consistently ranked by our scheme to be among the top 4% of all jobs submitted, with jobs of one template still ranking in the top 11%. We also measure relative rankings by user-specified priority and by our heuristic among jobs submitted to the same organizational queue, since priorities are only relevant when compared to other jobs sharing the same queue. For four out of the six hand-curated job templates, Our heuristic produces organizationally-relative rankings within 5% of priority assignment rankings. For one of the six job templates, our valuation scheme produces a ranking lower than that produced by priority assignments by up to 11%. For the last of the six job templates, however, we produce a ranking *higher* than that produced by priority assignments by 50%. This is surprising because we expected priority assignments for these six job templates, which are all verified to be highly important, to be extremely well-tuned, with highly-ranked priorities assigned to jobs of all six templates. Yet, jobs of the last template are only ranked at the 49^{th} percentile of all submitted jobs within its queue by priority assignment—this mis-configuration may lead to significant issues once the queue becomes more heavily-loaded.

Future work: Further validating agg. downloads as value. We acknowledge that accurate job valuation is a difficult problem that requires further study, and that different companies can have different notions of job value. While further efforts are ongoing at Microsoft to validate

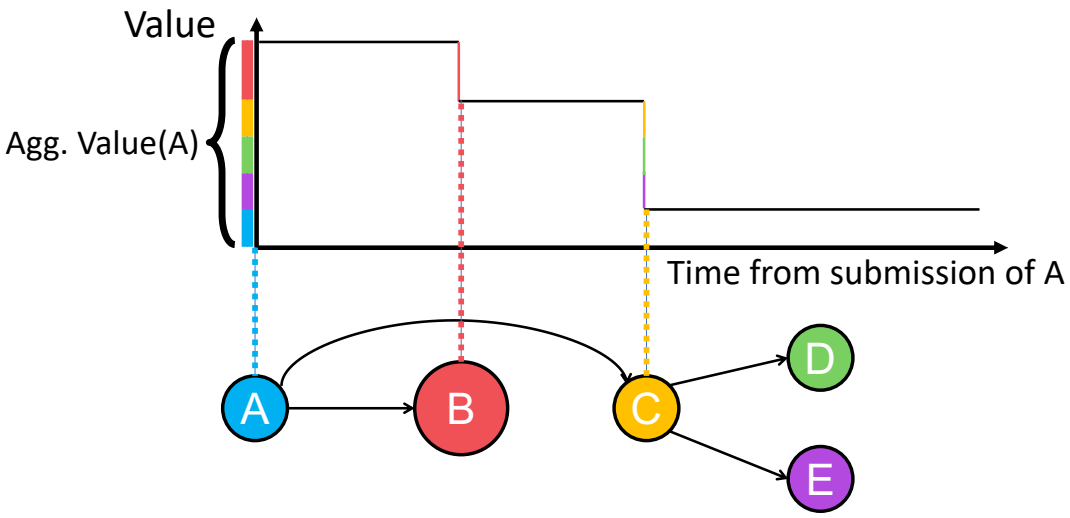


Figure 4.5: Value aggregation and value decay. In this toy example, jobs A–E are submitted at strict, absolute times, where the x-axis denotes time relative to the submission of job A. B and C have hard dependencies on A, and D and E have hard dependencies on C. The aggregate value of A is the sum of the aggregate values of B and C and A's own job-local value. With Wing, we can model how the aggregate value of A decays as it fails to complete by the time its downstream jobs arrive, losing the value of B at the time of B's submission, and collectively losing the values of C, D, and E at the time of C's submission (D and E depend *indirectly* on A through C, so if C fails, D and E will also fail). In this example, A retains its job-local value until the end.

the efficacy of our job valuation scheme (e.g., conducting surveys of Cosmos users), Cosmos's resource management team has noted that our valuation scheme is better than any of their existing heuristics used for job valuation, and are considering adopting it to aid in rolling out job upgrades and using it as a weighting function to report certain cluster performance indicators (e.g., reliability).

4.3.4 The Wing pipeline: Multi-hop analyses

Wing provides a flexible iterative solution implemented on top of SCOPE for performing *downstream multi-hop analyses*, in which for a given job, we analyze properties of its directly and indirectly-dependent jobs. Provided a set of single-hop inter-job dependencies, our framework allows the computation of both the transitive closure and aggregate statistics of all sub-DAGs rooted at each job in an inter-job dependency DAG (defined in Observation 5). Such multi-hop analyses are important to effectively guide scheduling decisions, as it can compactly characterize each job's downstream impact: i.e., if a job fails or is delayed, how will its downstream jobs and users be affected (§4.3.3)? Our framework generalizes well and allows multi-hop dependency analysis to be applied to various applications, e.g., fixing priority inversions[8] for Cosmos jobs.

Algorithm input. Our algorithm input is a single-hop job dependency DAG specified as a relational table, where the first column (`job`) holds the dependent job and the second column (`depOn`) holds the depended-upon job.

Algorithm output. Our algorithm outputs a relational table describing multi-hop dependencies. The first column (`job`) holds the downstream job, the second column (`depOn`) holds the (potentially multi-hop) upstream job, and the third column (`agg`) holds Wing-computed weights aggregated along all paths between the pair of up/downstream jobs.

Aggregation Functions (AFs). Each downstream multi-hop analysis specifies the following *Aggregation Functions* (AFs):

• *Weight function* (`wt_fn`): `wt_fn` takes in a job and its in- (or out-) edges as input, and outputs a weight `wt` for each graph edge. This operation is done once to convert the input DAG into an edge-weighted DAG.

• *Edge operation* (`e_op`): For two vertices t and v connected by an intermediate vertex u, `e_op` performs an aggregation of weights between a pair of (potentially auxiliary) in- (t, u) and out-edges (u, v) of u, constructing a new auxiliary weighted edge connecting t and v. Specifically, it computes the weight for an auxiliary edge based on new edges explored in each iteration between two indirectly connected jobs. This operation should be *distributive* over the `p_op` (defined following).

• *Path operation* (`p_op`): `p_op` aggregates weights on all explored paths between two jobs. While a unique path cannot be explored multiple times, the algorithm can make multiple traversals and aggregations between the same pair of up- and downstream jobs if multiple paths between two jobs exist. This operation should therefore be *associative*.

[8] Wing can fix priority inversions by raising the upstream job's priority before its dependent high-priority job arrives. Traditional OS methods require both jobs to have arrived at the scheduler, and dependency between the two jobs is communicated through concurrency data structures (e.g., locks). There is no lock-equivalent in Cosmos's scheduler.

• *Downstream operation* (`ds_op`, optional): The downstream operation is the last step performed, after our iterative algorithm converges. For a job, it performs an aggregation on all of its downstream jobs and aggregated path weights.

Algorithm outline. We first preprocess the job dependency DAG with the AF `wt_fn` to generate the DAG edge weights `wt`. Then, for each job in parallel, our algorithm traverses the DAG and computes transitive closures along all paths, maintaining an "aggregated version" of `wt` using `e_op` and `p_op` along the way. Our algorithm completes in $O(log(diameter))$ iterations, where $diameter$ is the longest path in the DAG. In each iteration, the algorithm maintains a *frontier* and a *base* table, both with the schema (`job, depOn, wt`). The frontier table records the set of discovered furthest reachable upstream jobs by `job` in `depOn`, while the base table records the set of all discovered reachable upstream jobs by `job` in `depOn`. The `wt` column of both tables records the aggregated weights along discovered paths from `job` to `depOn`. Each iteration joins and updates the frontier and base tables, extending the "reach" of each job by a maximum of $2\times$. Our algorithm pseudocode is shown in Algorithm 1.

4.3.5 Job value aggregation with Wing

Job value aggregation properties

Fair multi-hop time windowing. Aggregating value directly on even a single-hop time-windowed job dependency graph has a critical shortcoming: when considering multiple hops, jobs at the start of the observed trace still hold an advantage over jobs toward the end of the observation window in terms of opportunities to have their multi-hop downstream dependencies also land in the observation window. To better illustrate this, suppose we are given a recurring job template X with multiple jobs in our observation window. While ideally all jobs of X should have similar amounts of downstream dependencies, jobs of X that occur earlier in the trace are more likely to have their downstream dependencies also observed in the trace, while later jobs of X in the trace are more likely to have their downstream dependencies cut off due to the limits of using a static-length trace. In the limit of using an infinitely long trace, no time windowing is necessary.

A *multi-hop time window* is therefore needed to further restrict the set of jobs eligible for value aggregation. Our multi-hop time windowing method works as follows: we first define a time window size ω smaller than the observation period. For each *valid* job j in the trace, we consider its entire set of directly and indirectly dependent jobs that are submitted by up to ω after its completion time. Here, we define valid jobs as jobs that complete at least ω prior to the end of the observation period. We set ω to *one week* for multi-hop dependency analysis, as the scale of the inter-job dependency graph bottlenecks transitive closure computation as ω increases: increasing ω exponentially increases the number of multi-hop inter-job dependencies to consider, as dependencies fan-out further into the future. ω is set to a week here to capture the majority of recurring dependencies that occur on a sub-weekly cadence (most recurring templates are submitted with inter-job arrival times of a day or less), while allowing our entire analyses pipeline to finish in approximately a day.

Value conservation. To conserve the total amount of value in the system, we employ an *equal contribution* scheme, where each job contributes value to its directly-dependent upstream jobs equally, and the aggregate value of a job in this scheme is computed as the sum of value contributed

```
   // Helper functions
1  Function preprocess(s_hop) is
2  |  gp_by_job = job 𝒢 wts=wt_fn(depOn)(s_hop);
3  |  return π job, depOn=wts.depOn, wt=wts.weights(gp_by_job);
4  end
5  Function extend_reach(t1, t2) is
6  |  e_agg = π t1.job, t2.depOn, wt=e_op(t1.wt,t2.wt)(
7  |                t1 ⋈ t1.depOn=t2.job t2);
8  |  return job, depOn 𝒢 p_op(wt)(e_agg);
9  end
   // Computation start
   Input : s_hop                                  // Single-hop dependencies
10 i = 0;                                         // Iteration
11 ftr_i = preprocess(s_hop);                     // Frontier
12 base = COPY(ftr_i) ;                           // Base
   // base at the end of iter i covers deps up to 2^i hops
13 do
14 |  i++;
15 |  base_tmp = base − ftr_{i-1};
16 |  ftr_i = extend_reach(ftr_{i-1}, ftr_{i-1});
17 |  base_tmp = extend_reach(ftr_{i-1}, base_tmp) ∪ base;
18 |  base = job, depOn 𝒢 wt=p_op(wt) base_tmp;
19 |  base = base ∪ ftr_i;
20 while COUNT(ftr_i) > 0;
21 return job, depOn 𝒢 agg=p_op(wt) base;        // Converged
```

Algorithm 1: Multi-hop downstream analysis framework. `preprocess` first assigns weights to DAG edges with `wt_fn`. In each iteration, it calls `extend_reach` to further explore the graph from each job in parallel. In `extend_reach`, auxiliary edges with edge weights specified by `e_op` are created to denote newly discovered indirect dependencies (through the `JOIN`, or $\bowtie$ operator). The auxiliary edges are deduplicated with a `GROUP BY` ($\mathcal{G}$) operator at the end of each iteration, yielding edge weights of `p_op(wt)`.

upstream by all of its downstream jobs plus the value of the job itself. In this scheme, if a job j depends directly on the output of N jobs, it contributes $1/N$ of its value to each of its jobs directly upstream. Each of the N upstream jobs in turn further propagates j's (and their own) value upstream in the same fashion; e.g., if each of the N jobs directly depend on the output of M other jobs, j contributes $1/(N * M)$ of its value to each of the $N * M$ jobs two hops upstream. This yields the following equation for computing the aggregate value of a job:

$$agg\_val(j) = \sum_{d \in \mathcal{D}_j} \Big(\sum_{p \in \mathcal{P}_{(j,d)}} \prod_{e \in p} w_e * k_d \Big) + k_j,$$

where $\mathcal{D}_j$ represents *all* downstream jobs of j, $\mathcal{P}_{(j,d)}$ represents all paths from j to d, w_e represents the weight of a directed edge e on the path p, and k_d and k_j represent the job-local values of d and

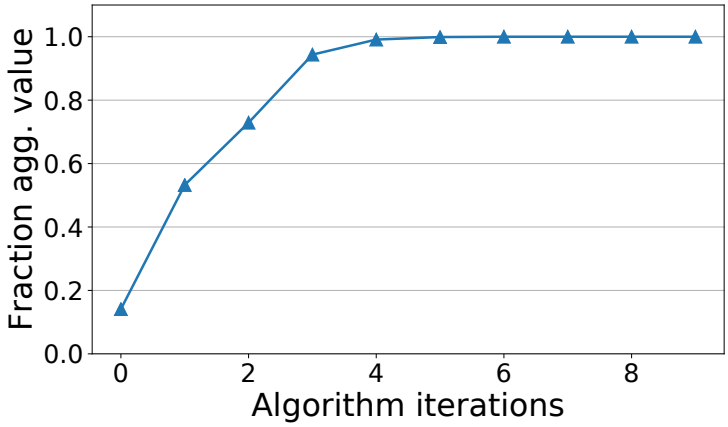


Figure 4.6: Aggregate value convergence. This figure shows the fraction of average aggregate job value uncovered downstream in each iteration of our value aggregation algorithm. 99% of aggregate value is discovered within four iterations.

j, respectively.

Wing value Aggregation Functions

We implement Owl's dependency-driven job valuation scheme with Wing's downstream multi-hop analysis framework, specifying Aggregate Functions as follows: the *weight function* `wt_fn` takes in a job j and its N upstream dependencies as input, and returns $1/N$ as the weight of each in-edge; the *edge operation* `e_op` multiplies the weights of its two operands; the *path operation* `p_op` sums the weights of its operands; and finally, for each job j, the *downstream operation* `ds_op` sums the job-local downloads of each job downstream of j multiplied by the aggregated path weights between the downstream job and j. j's job-local downloads are finally added to the downloads computed by `ds_op`, yielding j's aggregate downstream downloads.

Extensibility. While we elect to use downloads as a proxy for job value, Wing's framework is flexible enough to consider other metrics: e.g., if one day the dollar value associated with a job can be known, computing the aggregate downstream dollar value of a job is as easy as replacing a field in `ds_op`.

Aggregate value exploration and convergence

Using downloads as a proxy-metric for value, Fig. 4.6 shows the fraction of aggregate value explored in each iteration for each job on average. Considering the aggregate value of jobs with Wing allows us to uncover 83% of value that would otherwise be hidden if only job-local values (*iteration* $= 0$) were considered. In the context of value-based job scheduling (§4.4), this means that nearly $6\times$ of value can be hidden from the scheduler if jobs are independently considered. The figure also shows that 99% of average job aggregate value can be explored within four iterations of our algorithm.

4.4 Wing-Agg: Inter-job value scheduling

Value scheduling. The objective in *value scheduling* is to maximize the value achieved from executing jobs in a workload, where the completion of each job is directly associated with an amount of *job-local value* attained. Job-local value can decay over time, and this behavior is often modeled as a *value function* (VF) in scheduling literature, which expresses value attained as a function of job completion time. In value scheduling, it is therefore important to complete jobs in a timely manner to achieve the most value.

Value and priority. We make a clear distinction between the terms *value* and *priority*. In this chapter, we use the term *value* to describe a measure of "goodness" achieved associated with the completion of a job. *Priority*, on the other hand, defines the order in which pending jobs are assigned cluster resources: the higher the priority, the earlier a job receives its requested resources. Most commonly, including currently within Cosmos, the priority of a job is assigned by its submitter.

Wing-Agg. When inter-job dependencies are present, we find that it is important to consider the potential value downstream that can be lost if a job fails or is delayed. To consider the effects of inter-job dependencies, we propose a scheduling policy, *Wing-Agg*, that incorporates Wing's notion of inter-job dependencies into job priorities: *the goal of Wing-Agg is to achieve the most value for a given workload.*

As suggested in the introduction, completing the most value-impactful job may not lead to a scheduler attaining the most value, as some value-impactful jobs can also require large amounts cluster resource-time to complete. Indeed, prior work [33, 71, 103] has shown that schedulers can often benefit by considering together how much value a job provides and how much resource-time a job uses.[9]

Wing-Agg therefore considers the *aggregate value efficiency* of jobs, which measures how much aggregate value per aggregate resource-time a job impacts downstream. Essentially, Wing-Agg replaces user-assigned priorities with what Wing believes is a job's aggregate value efficiency. When a job arrives, Wing-Agg performs a look-up in the WingStore (§4.3.1). If the job is recurring, Wing-Agg computes the job's aggregate value efficiency by dividing the job's median historic aggregate value by its median historic aggregate compute-time, and assigns the quotient as the job's priority. If the job is ad-hoc, Wing-Agg estimates the job's aggregate value efficiency based on previous ad-hoc jobs that the same user has submitted. Wing-Agg assigns aggregate value efficiency rather than aggregate value as jobs' priorities to optimize for high value throughput.

4.5 Experimental setup

This section provides an overview of the Cosmos resource management infrastructure, describes our evaluated scheduling policies, and describes our experimental methodology.

Downloads attained as value. In our experiments, we use the number of downloads associated with the outputs of each job as a proxy for the value attained by a job. We model download

[9] Although Wing-Agg and shortest-job-first both use job resource-time in their decisions, Wing-Agg frequently runs longer, more value-providing jobs ahead of shorter jobs.

attainment using real-world output download traces: if a job j completes at 1PM in the real-world (from the trace) but only completes at 2PM in our experiment, j attains only the output downloads associated with its outputs that occur after 2PM, and loses the downloads that occur between 1 and 2PM. A limitation of our model of value is that it does not reward completing a job early. Further research is required to determine how much additional value the early-completion of a job yields in data lakes.

Cosmos backend: YARN and hierarchical queues. Cosmos uses a *YARN-based resource manager* [39, 125] in the backend and utilizes *hierarchical queues* (queues, for brevity) to delineate resource boundaries between organizations—users/workflow managers can only submit SCOPE jobs to queues belonging to organizations of which they are a part. Cosmos uses a scheduling policy similar to the default policy that the *CapacityScheduler* in stock YARN uses, which orders jobs in each queue based on their (often user-) assigned priorities. A key difference is that jobs are scheduled with gang semantics in Cosmos—a job is admitted only when the scheduler can ensure that a user-provided minimum number of parallel, *job-requested* resources can be granted to it.

4.5.1 Simulation setup

We evaluate the application of Wing's analyses to scheduling using simulation-based experiments due to the scale of Cosmos: the Cosmos traces we use contain $\sim$40k jobs per day, and $\sim$160k inter-job dependencies. Experiments at this scale cannot realistically be attempted on research clusters without down-sampling jobs, at which point much inter-job dependency fidelity within the original workload will have been lost. We therefore use simulations to preserve the characteristics of inter-job dependencies in our experiments.

Simulation platform: Design and implementation. Our simulation platform takes a discrete-event based approach. To ensure that our experiments retain most properties of YARN/Cosmos, our simulation platform makes minimal changes to the YARN architecture—our implementation only mocks out the real-time clock and the communication layers of the YARN servers. We also use real queue sizes for each hierarchical queue in our Cosmos cluster. The authors plan to contribute this simulator back to the open source community [6].

Simulation accuracy. To make simulation feasible given the scale of our job logs, the simulator does not model: (1) "internal" dependencies among stages of a job, but rather treat a job as a rigid collection of tasks; (2) resource-sharing through opportunistic execution [80] of job tasks, which allows jobs to use more resources than requested when those resources are otherwise idle; and (3) job sizes based on resources used rather than job-requested resources, meaning that our simulations only consider the deep blue area in Fig. 4.7.

To evaluate the fidelity of our simulator, we measure the absolute differences in job completion times between jobs in our simulations (using the baseline system policies) and the same jobs run in the real cluster. We normalize the deltas by the job's real-world latency, and observe that even at the 99^{th} percentile, jobs are shifted by only 1.3% of their latencies. Our experiments run at 100% cluster capacity also achieve average resource utilization for job-requested resources within 1.5% of what is observed in the real cluster.

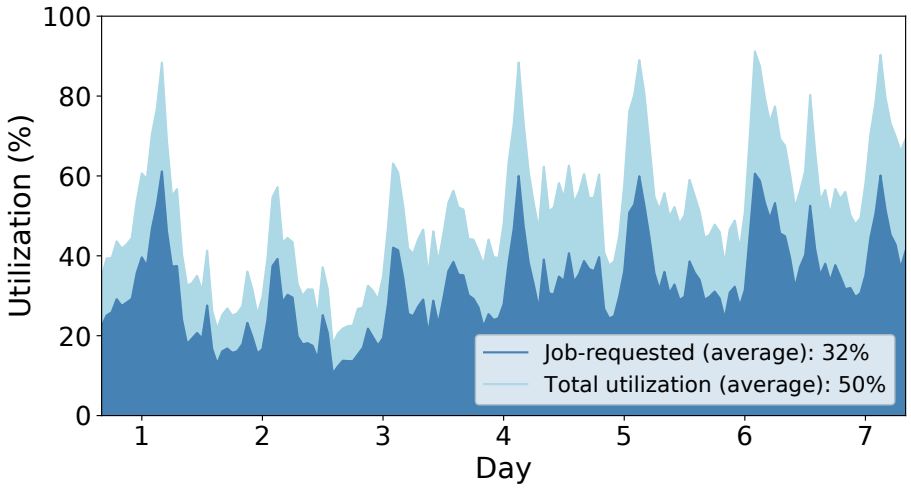


Figure 4.7: Cluster utilization. This figure shows the job-requested and total resource utilizations of our real cluster.

4.5.2 Evaluated scheduling policies

In addition to Wing-Agg (§4.4), we evaluate value-attainment on our workload traces on the following scheduling policies. All implement Cosmos's gang-scheduling semantics.

PRIO represents Cosmos's current approach, and is the default scheduling policy used by stock YARN in its CapacityScheduler. It orders jobs within each hierarchical queue based on user-specified priorities.

Wing-MIL. Millennium [33] is a VF-aware scheduler that orders jobs based on expected value attained per resource time: For each queued job it computes how much value can be gained at an estimated job completion time, divides the value by total job resource-time, and orders jobs by the resulting quotient. MIL is our implementation of Millennium on YARN, following descriptions in its design as closely as possible.

Wing-MIL is MIL using Wing-informed value functions (VFs): In addition to capturing how the job-local value of a single job decays, a Wing-informed VF captures *potential* value associated with the job lost over time by modeling a job's full decay of downloads. A job j attains all of j's aggregate downloads in the most optimistic case if it completes before or at its real-world completion time; otherwise, it loses value according to when users perform download operations *and* when downstream jobs fail due to it not completing on time (illustrated earlier in Fig. 4.5). For example, in a Wing-guided VF, if j completes at 1PM in the real-world but only completes at 2PM in our experiment, j loses all the direct downloads that occur between 1 and 2PM, *and* all the *indirect downloads* rooted in jobs that directly depend on j submitted between 1 and 2PM.

Plan-ahead based VF-aware policies. We attempted to evaluate more sophisticated plan-ahead based VF-aware policies, e.g., FirstOpportunityRate [103]. But, we found that one implementation of such a policy couldn't accommodate workloads at Cosmos scale, and efforts to mitigate bottlenecks by caching and limiting plan-ahead led to less value attainment than simpler policies (e.g., MIL). We therefore do not include our attempts with such a policy, as further work is warranted before conclusions are drawn.

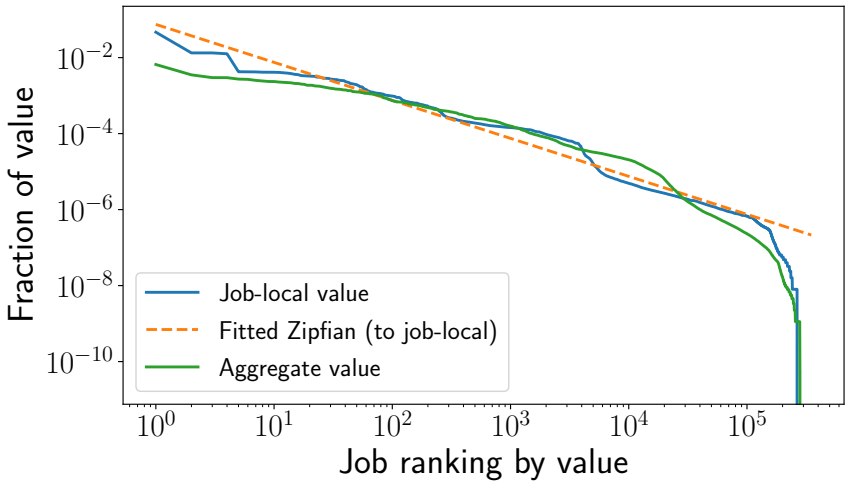


Figure 4.8: Distribution of job value. This figure shows the distributions of job-local value and aggregate job value, along with a Zipfian distribution fitted to job-local value. The distribution of job value deviates from Zipfian at lower job rankings.

4.5.3 Workload and predictor descriptions

Dataset. We use data from the final four weeks of our analysis dataset to evaluate our scheduling policies: Within the four weeks of data, Wing uses data in the first and second weeks to establish job and dependency profiles. Experiments are conducted over the third week, and downloads (value) are counted for each job up to one week (into the fourth week) from the completion of the job. Each day of traces contains ∼40k jobs and ∼160k inter-job dependencies.

Considering inter-job dependencies. Different from prior work, our experiments take characteristics of inter-job dependencies into account to realize more realistic workloads. For example, if a job holds a hard dependency on the output of an upstream job but the output is not available in time, the job fails due to missing input. Other dependency patterns, such as polling behavior (when a job waits for its inputs to become available), are also modeled faithfully. Jobs and dependencies considered in our experiments are described in Table 4.1.

Job value distribution. Job value, as measured by the number of downloads associated with the timely completion of a job in our experiments, are distributed roughly in a Zipfian fashion ($s = 1$) with deviation at the low end, as shown in Fig. 4.8. This means that the most valuable jobs are downloaded significantly more times than less valuable jobs. When scheduling for value on a workload that is inter-job dependency aware, schedulers should work to *unblock* the most valuable jobs before they arrive in order to attain their value.

Value efficiency predictor. Wing-Agg and Wing-MIL use a predictor to estimate the aggregate value efficiency associated with upcoming recurring jobs to optimize for value throughput. While §4.2 shows that direct inter-job dependencies can be predictable, it neither considers predictions on a job's subgraph of downstream dependencies, nor a job's value impact. Evaluating predictions on aggregate value efficiency therefore allows us to better understand the performance of Wing-guided schedulers. For recurring jobs in our experiments, we use a median-based predictor to predict the value efficiency associated with a job. That is, given a recurring job j of template τ, we predict j's value efficiency based on the historical median value efficiency for jobs

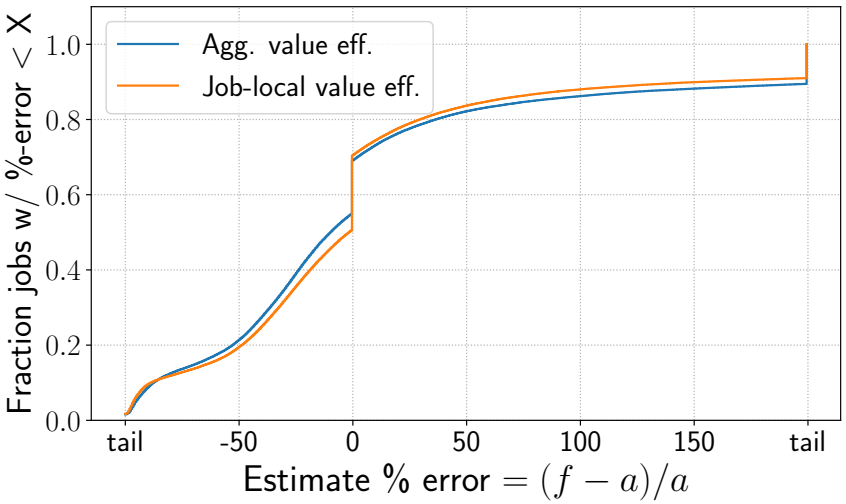


Figure 4.9: Value efficiency prediction. This figure shows the CDF of our predictor's performance on predicting the value efficiency and aggregate value efficiency of recurring jobs.

of template τ.

Fig. 4.9 shows the performance of our value efficiency predictor in a CDF. For predicting the aggregate value efficiency of a job, 39% of our predictions fall within $\pm 20\%$ of the actual value efficiency of a job, while for predicting the value efficiency of a single job, 44% of our predictions fall within $\pm 20\%$ of the actual value efficiency of a job. While we are working on further studies to improve predictor accuracy with more sophisticated methods, we find that the performance of our simple predictor enables Wing-Agg to outperform other evaluated scheduling policies in value attainment (§4.6).

4.6 Experimental results

We evaluate the efficacy of each scheduling policy for the actual full Cosmos resource capacity (100%) and for smaller capacities (at 80–20%). Value-attainment results are reported as a percentage of value achievable—i.e., if all jobs in workloads complete before any of their values are lost.

Cluster capacities & consequential policy decisions. Scheduling is most interesting when cluster capacity is constrained and schedulers need to make difficult decisions regarding which jobs to provide resources. Indeed, at 100% capacity, the baseline and more advanced schedulers perform similarly, completing $> 99\%$ of all jobs in the trace. We find that the lower cluster capacities (i.e., $\leq 40\%$) best exemplify the consequences of decisions a scheduler makes. We therefore focus the discussion of our results at these capacities to maximize observable differences.

Takeaways. Our experiments yield the following key takeaways. First, policies guided by Wing are better at achieving value when clusters are heavily-constrained. In particular, Wing-Agg outperforms all other compared policies at all capacities and improves value attained by up to 21% as capacity declines. Second, understanding the downstream impact of a job is crucial in constrained clusters, and that Wing-guided inter-job dependency predictions are accurate enough

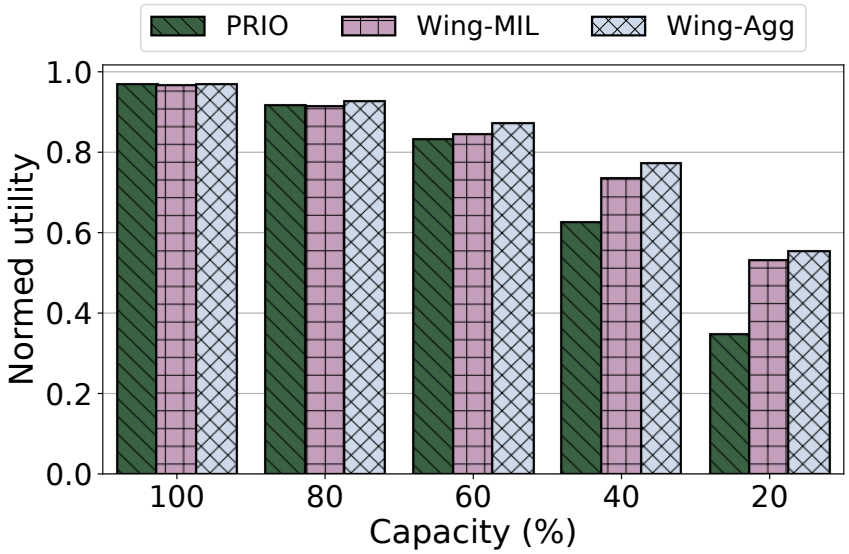


Figure 4.10: Benefits of Wing guidance. This figure shows the value attained for each scheduling policy, normalized to total value achievable. Wing-guidance (exemplified in Wing-Agg and Wing-MIL) is significantly beneficial at constrained capacities.

to be practical: Wing-Agg can effectively complete the prerequisites of the most consequential jobs. Finally, we demonstrate significant opportunity in applying inter-job dependency awareness in Wing to a cluster-wide queue and establishing a cluster-wide value metric: Wing-Agg achieves up to 93% of all value in our workload when using a single cluster-wide queue, using only 20% of cluster capacity.

4.6.1 Benefits of Wing guidance

Fig. 4.10 shows that policies guided by Wing beat PRIO at all capacities, with value attainment gaps widening as the cluster is increasingly stressed. At 60% capacity, Wing-Agg achieves 87% of value (vs PRIO's 80%). At 40% capacity, Wing-Agg achieves 77% of value (vs PRIO's 62%). Even at 20% capacity, Wing-Agg is able to capture more than half of all value (55%), while PRIO only captures 35% of value.

Considering aggregate value gives Wing-guided schedulers a two-fold benefit over PRIO. First, it naturally "fixes" priority mis-configurations, such as priority inversions, by propagating job value upstream, such that downstream jobs with high value are not blocked. Second, it guides schedulers toward sub-DAGs of high value efficiency jobs effectively, allowing schedulers to achieve more value with less resources.

Are ad-hoc jobs disadvantaged? Since Wing-Agg focuses on recurring jobs, we examine our logs to see if ad-hoc jobs are at a disadvantage when scheduled by Wing-Agg vs recurring jobs, where the priority of ad-hoc jobs are determined by the median aggregate value efficiency of previous jobs submitted by the same user. We find, from results at 20% cluster capacity, that 25% of recurring jobs fail, compared to 42% of ad-hoc jobs. However, recurring jobs also carry $9\times$ more value than ad-hoc jobs. To optimize for value, Wing-Agg necessarily needs to complete larger fractions of recurring jobs. Indeed, recurring jobs are more often production jobs [78].

Dynamic priorities (Wing-MIL). Intuitively, policies using dynamic priorities (e.g., value functions, or VFs) such as Wing-MIL should perform better than static policies such as Wing-Agg, as VFs can express both importance and urgency while priorities only allow the expression of one of the two dimensions; but, we observe that Wing-Agg outperforms Wing-MIL at all capacities, albeit only slightly.

Unlike Wing which only depends on aggregate value-efficiency predictions, Wing-MIL also depends on the time-to-dependency predictions of directly-dependent jobs (§4.2) to determine when aggregate job value decays. But, while a part of this underperformance is indeed caused by imperfect predictions of time-to-dependencies, we find that providing Wing-MIL with perfect job value and time-to-dependency information does not help much. Further analyzing our results, we find that this underperformance is mainly due to Wing-MIL's failure to consider the *properties* of inter-job dependencies. For example, a downstream job that polls for the arrival of its inputs will not fail if its upstream jobs complete late. But, VFs constructed from historical data will still reflect a drop in value at the time the polling downstream job is expected to arrive, leading Wing-MIL to believe that it should give up prematurely on scheduling the job. This shortcoming can be addressed by considering dependency properties *explicitly*, but our attempted implementation of such a policy does not significantly improve over Wing-Agg: both Wing-Agg and our attempted implementation can complete the most impactful, value-efficient jobs in a timely manner.

Practicality of Wing-Agg. The simplicity of Wing-Agg is desirable from an engineering standpoint, as Wing-Agg is both highly practical and highly scalable: Integrating Wing-Agg into a production cluster requires minimal changes to the existing resource management framework, and all the information needed for Wing-Agg to determine a job's priority can be pre-computed offline in Wing's analysis pipeline (§4.7). Adoption of Wing-Agg into production can therefore be straightforward, upon confirming job valuation schemes.

4.6.2 Sensitivity and ablation studies

Aggregate vs. job-local value. This section discusses benefits of understanding job value at an aggregate vs job-local level by comparing Wing-Agg against Wing-Direct, where *Wing-Direct considers the job-local value efficiency of a job*: i.e., Wing-Direct only considers direct-downloads associated with the outputs of and the compute-time of a single job only.

The patterned bars in Fig. 4.11 show the normalized value attained by Wing-Agg and Wing-Direct. While Wing-Direct outperforms PRIO, Wing-Agg maintains significant benefit over Wing-Direct at the tightest capacities: Wing-Agg attains 13% more overall value than Wing-Direct at 20% capacity. Our analysis finds that Wing-Direct's knowledge of job resource consumption allows it to effectively complete jobs at the head of queue, enabling it to complete a similar amount of jobs as Wing-Agg. But, with knowledge of historical aggregate value efficiency, we find that Wing-Agg completes jobs in the more value-heavy sub-DAGs of the inter-job dependency DAG, yielding significant improvements over Wing-Direct.

Wing predictions vs. Oracle knowledge. We examine how much potential benefit better predictions can provide to each Wing-aided policy. *Oracle Wing-Agg* represents Wing-Agg endowed with perfect knowledge of aggregate value efficiency, and *Oracle Wing-Direct* represents Wing-Direct provided with perfect knowledge of job-local value efficiency.

While we find that having better predictions are beneficial, the differences between the solid

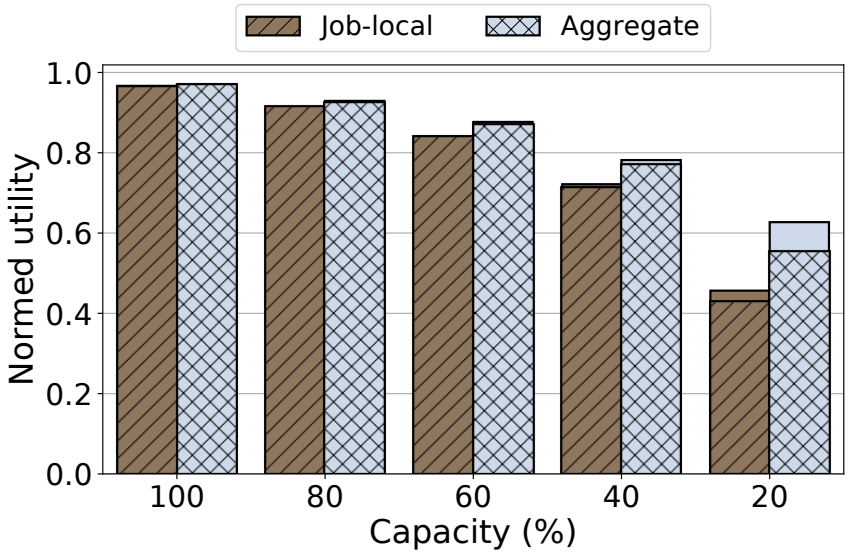


Figure 4.11: Benefits of aggregate job value. *Aggregate* (corresponding to aggregate download-aware) vs *Job-local* (corresponding to direct download aware only) bars show the benefits of aggregate value, compared to only scheduling based on job-local value. The solid portion of the bars show the benefits of Oracle knowledge.

(representing policies with Oracle knowledge) and the patterned bars (representing policies with Wing-provided predictions) in Fig. 4.10 and Fig. 4.11 show that at most capacities, Wing-guided schedulers achieve close to the value attained by their Oracle variants. However, having more accurate information presents opportunity for significant gain in value attained for Wing-Agg at 20% capacity: e.g., Oracle Wing-Agg improves value realized over Wing-Agg by 8% of overall value. Conversely, although Oracle Wing-Direct is granted exact knowledge of how value-efficient each job is, its view of the overall inter-job dependency graph leads to only incremental benefits.

Oracle benefits to aggregate value aware policies come from a more accurate knowledge of a summarized view of the inter-job dependency graph: compared to single job value-aware policies with Oracle knowledge, a policy such as Oracle Wing-Agg can efficiently complete the most consequential jobs in the job dependency graph, increasing value attained (by up to 18% of overall value vs Oracle Wing-Direct) and reducing the number of jobs failed due to missing input (by 3% of all jobs vs Oracle Wing-Direct).

Sensitivity to mis-predictions. We examine the sensitivity of Wing-Agg to aggregate value efficiency mis-predictions on our workload by running experiments that introduce artificial shifts in aggregate value efficiency provided by Oracle Wing using 20% cluster capacity. Each experimental run is associated with a maximum artificial shift s, where $s \in \{1.1, 1.25, 1.5, 2, 5, 10\}$. For each job j within each run, we scale the aggregate value efficiency e_{val} of j provided by Oracle Wing by multiplying e_{val} by a randomly sampled multiplier m between $1/s$ and s. Our results show that Wing-Agg is not sensitive to mis-predictions in value on our workload: For $s \leq 5$, value attained is only reduced by at most 4% vs Oracle Wing-Agg. For $s = 10$, value attained is only reduced by 11%. This insensitivity is because job values in our workload are distributed in a Zipfian fashion (§4.5.3), where the most valuable jobs are much more valuable than other jobs.

Reducing transitive closure computation. At 20% capacity, Wing-Direct (0 iterations of Wing's multi-hop analysis) attains 42% of all value, while Wing-Agg (9 iterations executed) attains 55% of all value. In Fig. 4.6 in §4.3.5, we find that 99% of aggregate value of most jobs can be explored in four iterations of Wing's multi-hop analysis. We therefore believe that four iterations of exploration would be sufficient to similarly attain 55% of all value, and that two iterations of exploration would allow us to attain close to 50% of all value.

4.6.3 Cluster-wide queue and value metrics

Our earlier results correspond to a simplified view of Cosmos using strictly enforced queue boundaries. Hard queue boundaries restrict placement more than in the real system, where resource-sharing (§4.5.1) softens queue boundaries, which might exaggerate Wing-Agg's benefits. To confirm that Wing-Agg's improvements are not due to hard queue boundaries, we evaluate a boundary-free alternative with experiments run using a single global, cluster-wide queue.

Evaluation. Fig. 4.12 shows the value attainment of our evaluated scheduling policies using a single cluster-wide-queue. We note that all jobs are able to complete for all scheduling policies at 60% cluster capacity. Indeed, the dark blue area in Fig. 4.7 show that these requests peak at around 60%. At 40% capacity, the cluster still has more capacity than needed most of the time: Wing-Agg achieves 99% of value, and Wing-Direct and PRIO achieve 97 and 93% of value, respectively.

Under extreme capacity crunch (e.g., 20% capacity), removing restrictions of hard queue boundaries improves value attained of all policies. But, a Wing-guided scheduler sees significantly more benefit in terms of absolute value achieved. With a cluster-wide queue at 20% capacity, Wing-Agg attains 93% of value, whereas Wing-Direct attains 84%, and PRIO only attains 47%. Furthermore, Wing-Agg fails fewer jobs compared to both Wing-Direct and PRIO (11% vs 13% and 25% of jobs, respectively).

We find that understanding inter-job dependencies is critical, as Wing-Direct with Oracle knowledge did not significantly outperform Wing-Direct with predicted values, both in terms of value attained and in terms of number of jobs failed; yet, we find that Wing-Agg with Oracle knowledge, in this setting, can achieve up to 98% of all value (comparable to performances at 100% capacity), while failing only 7% of all jobs (compared to 26% in a multi-queued setting at 20% capacity). One of the reasons why Wing-Agg is able to attain 93% of all value using only 20% of cluster capacity is due to its ability "unblock" the most valuable downstream jobs.

Recall that the simulated job sizes in our experiments are based on job-requested resources, rather than job-used resources, which may be higher because of opportunistic execution. As a result, cluster utilization is lower in our experiments. But, we believe that the rankings of the different schedulers are not affected, because the number of opportunistic resources highly correlate with that of allocated job-requested resources, both across the top 10% of most valuable jobs (Spearman correlation of 0.85) and across all jobs (Spearman correlation of 0.84). Indeed, the amount of opportunistic resources available to a job is capped with a max proportional to the number of allocated job-requested resources [110]. So, the relative differences shown for 20% cluster capacity may instead be for 30% cluster capacity in the heavier workload.

Toward establishing a cluster-wide value metric. Our results confirm that removing queue boundaries would be beneficial. Partitioning resources into queues naturally introduces resource

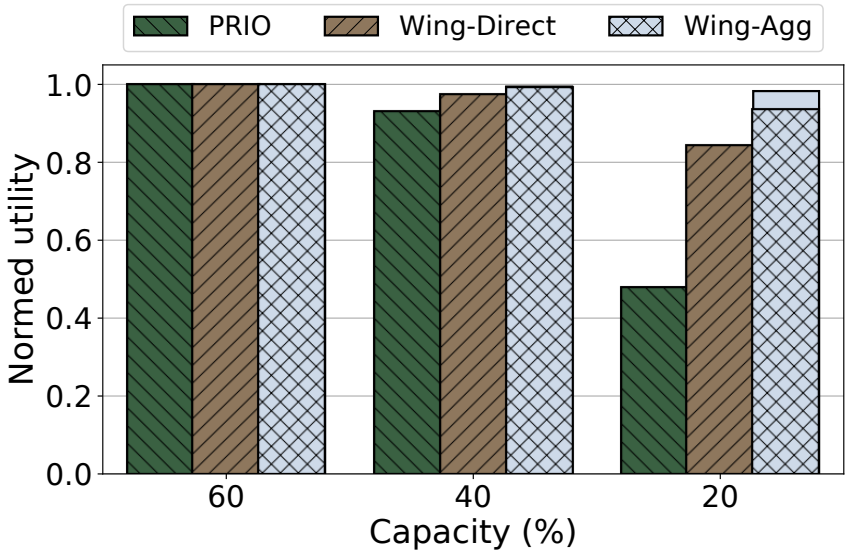


Figure 4.12: Benefits of Wing-guidance with a cluster-wide queue. This figure shows the value attained for policies from 60–20% cluster capacities in a cluster with a merged cluster-wide queue. All policies complete all jobs at 60% capacity. Wing-guidance (exemplified by Wing-Agg) is increasingly beneficial at lower capacities. The solid portion of the bars show the benefits of Oracle knowledge.

fragmentation, but usage of queues is often viewed as a "necessary evil," as certain organizations are willing to pay more to have guaranteed access to their share of compute. Yet, näively removing queue boundaries without a quota-system [126] in place may introduce resource competition, where users across different organizations assign increasingly high priorities to their jobs to acquire guaranteed resources. A cluster-wide, automated arbitrator that understands both system-internal (e.g., aware of downstream number of affected jobs and user-downloads) *and* organizational/user-defined notions of importance is therefore required. We see this as an exciting direction for further research.

State of deployment. Instead of immediately deploying Wing-Agg as described, the Microsoft Cosmos resource management team has asked us first to deploy an inter-job dependency advisory tool using analyses from Wing to aid users on better configuring their jobs (§4.7). The tool will allow us to gather user feedback on our recommendations.

4.7 Owl: Visualizing inter-job dependencies and job impact in shared clusters

As we have described previously, in companies with many users and sizable shared data repositories, job owners are seldom aware of who consumes their jobs' outputs, as their knowledge of job dependencies is often limited by their scope of business interactions and contracts. At the same time, seemingly harmless modifications to *upstream* jobs (e.g., submission schedule, priority, and output schema) can produce a ripple-effect on *downstream* jobs, interrupting chains of inter-dependent jobs and workflows and causing businesses significant amounts of damage.

Such obscure job dependencies often require co-dependent teams to establish carefully negotiated service level agreements, a counter-productive process towards the goal of a universal data lake. Indeed, we have seen in our experiments (§4.6) that schedulers guided with inter-job dependencies and value functions achieve more value compared to those that do not. This section describes Owl, a visualization of Wing's analyses that can help users identify important jobs and impactful jobs.

4.7.1 Visualizing inter-job dependencies

Analyzing data from Wing, Owl reveals job dependencies within-and-across workflows that are otherwise hidden in popular workflow visualizers [19, 20]. Owl also visualizes recurring job dependencies, enabling users to explore dependency metrics (e.g., time-to-dependency, number of priority inversions) between frequently dependent jobs and use these metrics to fine-tune job hyper-parameters (e.g., priority, resources allocated).

Representing dependencies

Job dependencies can be represented as a *weighted directed acyclic graph (WDAG)*, where each *vertex* is an instance of a job and each *edge* represents data-flow between a pair of job instances (where data is generated by the source and consumed by the target vertex). The *weight* of an edge represents the reliance of a target job on a source job, and is determined by a flexible *weighting scheme*.

Visualizing workflow dependencies

Motivation. Users often submit *workflows*, or sets of inter-dependent jobs, to complete complex business tasks. Although these business tasks might similarly be achievable with a single monolithic job, breaking the job into smaller component jobs allows for improved code reusability, manageability, and debuggability. These workflows are usually managed by automated time-based job scheduling systems.

In a large organization like Microsoft, it is virtually impossible for users to recognize all consumers of a job's outputs. Debugging workflows can therefore be a daunting task, as workflows can include numerous job dependencies across multiple workflows, each owned by a different team. The lack of awareness of dependent jobs external to a workflow can lead to job disruptions. For example, since a workflow owner often only cares about the end-result of their workflow (i.e., when their business task completes), intermediate jobs can be altered without warning as long as the pipeline completes end-to-end. These seemingly harmless modifications can potentially cause external job failures unbeknownst to the workflow owner due to inconspicuous inter-workflow dependencies. Owl's workflow view clarifies dependencies within-and-across workflows.

Visual features. Owl's *workflow graph view* allows users to browse important properties of their workflows using an interactive graph (Fig. 4.13). Graph vertices (workflow jobs) can be resized proportionally to a selected job attribute to help users identify outliers or anomalies. For instance, a user can size vertices with respect to job runtime to identify execution bottlenecks or with respect to CPU-time utilized to analyze resource budget consumption. A standard *Gantt chart*

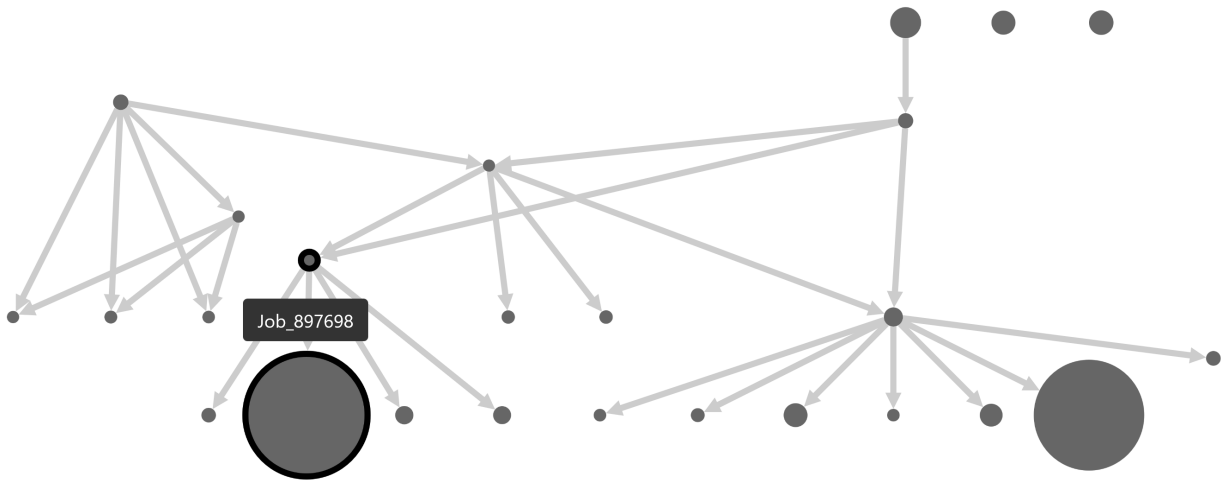


Figure 4.13: Workflow graph Shows the job dependency structure within a workflow. Allows users to identify important jobs by auto-sizing job vertices with respect to job execution attributes such as CPU-time (depicted).

shows users when each workflow job is submitted and how long it runs for during the workflow's lifetime.

The *inter-workflow dependency graph* allows users to discover dependencies across workflows. With the graph, users can easily identify owners and properties of upstream and downstream workflows, as well as pinpoint problematic cross-workflow job dependencies.

The *workflow diff utility* enables side-by-side comparison between two executions of the same recurring workflow using interactive graphs. It allows users to effectively identify anomalies in workflow structures, such as when a workflow instance executes more jobs than usual.

Visualizing recurring dependencies

Motivation. *Recurring jobs*, or jobs that are repeatedly submitted over time to analyze fresh data, make up the majority (˜60%) of CPU-time utilized in Microsoft's Cosmos clusters [78]. When recurring jobs depend on each other, a *recurring dependency* is introduced and an implicit contract is formed between the pair of jobs. Breaking the contract in any way can potentially lead to service disruption downstream. It is thus in a job owner's interest to understand characteristics of upstream recurring jobs and be aware of recurring jobs consuming the job's output.

Visual features. Aside from showing basic recurring job attributes (e.g., periodicity and distribution of job execution properties), Owl's *recurring jobs view* features a *recurring dependency graph* (Fig. 4.14) that allows users to navigate and analyze recurring dependencies and their statistics (also displayed in an interactive distribution chart), both upstream and downstream.

With the dependency graph and statistics, users will be able to discern important dependencies. For example, using the *time-to-dependency* statistic, users will be able to see which upstream job their recurring job is frequently waiting on and get a sense of their job's *deadline slack* with respect to the submission time of downstream jobs. Using the *percent-consumed* statistic, determined by the number of job instances consumed by a recurring downstream job over the total number of job instances completed, users can get a sense of which jobs upstream are critical

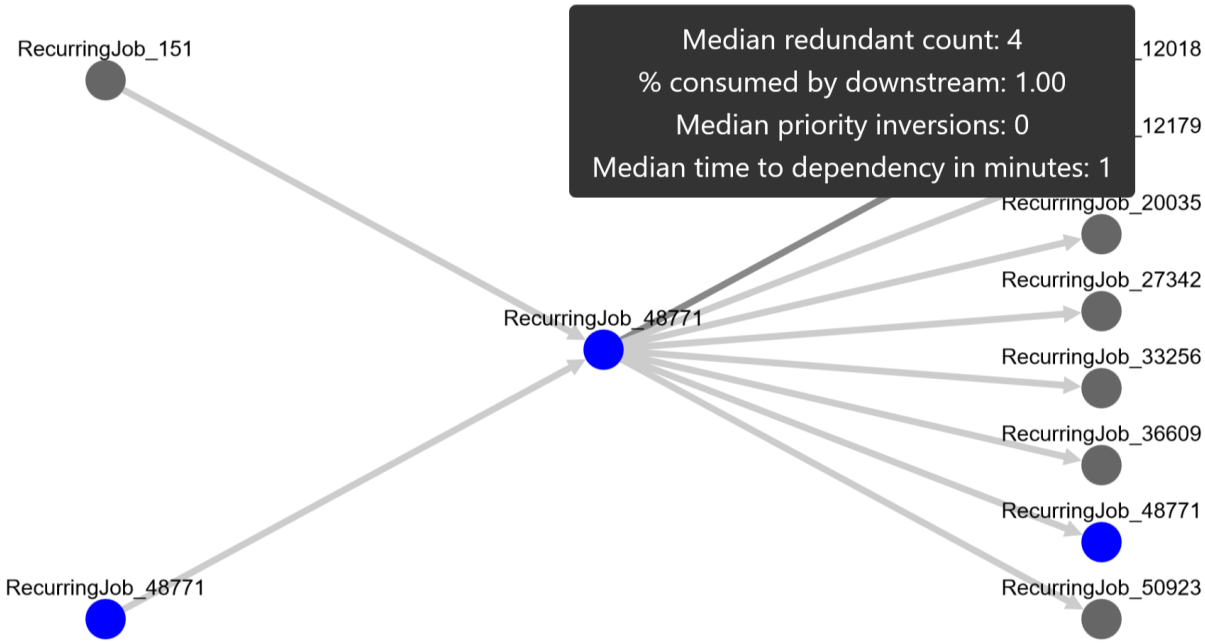


Figure 4.14: Recurring Job dependency graph Displays the target recurring job (center) and its upstream (left) and downstream (right) recurring jobs. Hovering over an upstream/downstream link shows statistics of the dependency.

to their job as well as which jobs downstream their job is critical to.

4.7.2 Visualizing job impact

Motivation. Modifying job properties (e.g., output schema) without considering job dependencies can cause a ripple effect downstream, leading to job delays or even failures. As companies move towards universal data lakes that promote heavy data sharing, altering jobs without breaking downstream dependencies becomes a difficult task. While avoiding job disruption is always preferred, in a production setting it is critical, as job failures can affect business continuity.

To provide users a sense of how important a job is, in terms of its potential effect on downstream operations, we introduce the *job impact score*, derived from historical job dependencies and telemetry logs using Wing's framework (§4.3.5). The job impact score allows users to quickly discern how many downstream operations their job affects, as well as *which* downstream operation, if affected, will lead to the most potential operation disruption further downstream.

There are two main *facets* to our job impact score: (1) the *work impact* facet measures the impact of a job on downstream jobs in terms of historical CPU-time blocked (i.e., computation-hours that cannot proceed due to the upstream job not completing by the submission time of the downstream job). (2) The *user impact* facet measures the impact of a job on users in terms of the historical number of downloads blocked (i.e., file views that cannot happen due to the upstream job failing to produce its output(s)). The remainder of the section walks the reader through our scoring methodology and how Owl visualizes historical job impact.

Methodology: Evaluating job impact. Owl uses Wing's framework proposed in §4.3.3 to quantify different facets of job importance fairly using jobs' historical downstream dependencies.

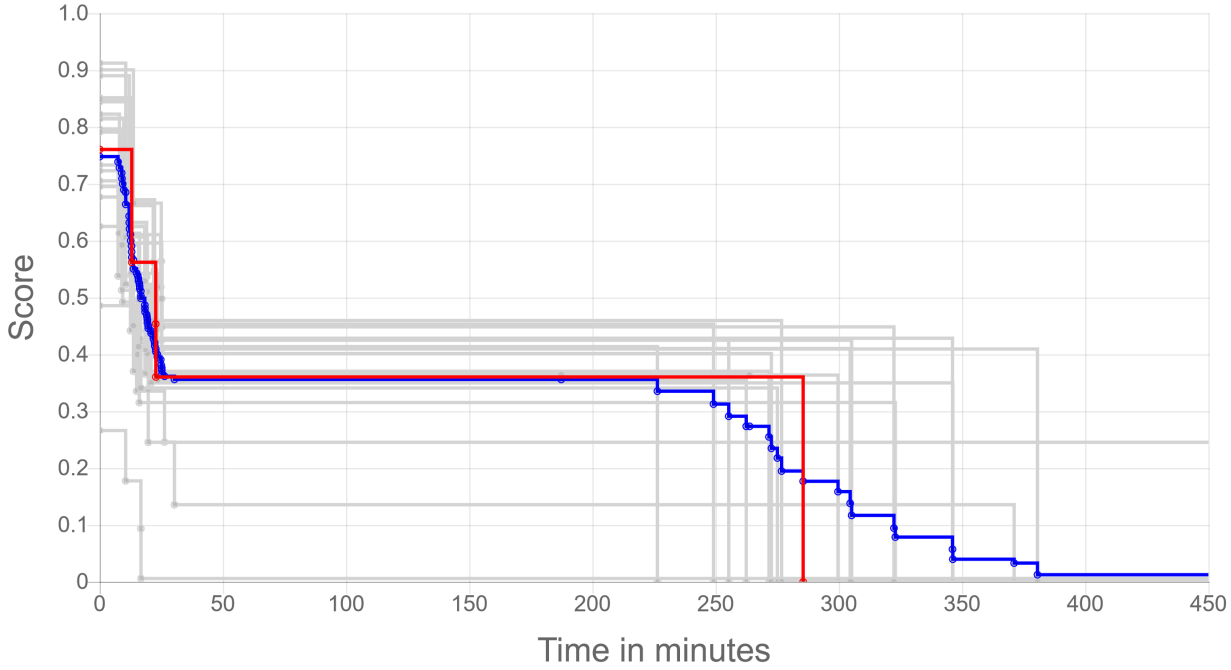


Figure 4.15: Job utility function graph Shows the value (score) of a job as a function of time-from-submission. The score displayed is normalized to the score of the most valuable job in the hierarchical queue. The red line displays the utility function of the user-selected job, while the gray lines represent utility functions of other instances of the same recurring job. The blue line sketches the average score over time of the recurring job.

Each job starts out with a *base score*. The base score corresponds to a job run statistic — for example, the statistic used to compute *work impact* is CPU-time while the statistic used to compute *user impact* is number of downloads. Other statistics, such as bytes read/written, can easily be incorporated. Downstream jobs *contribute* scores upstream, where the amount of contribution a downstream job makes to an upstream job is determined by the edges on the path(s) between the jobs and by the base score of the downstream job. The impact score[10] of a job is the sum of all its de-duplicated contributions from downstream (§4.3.5).

Methodology: Discovering dependencies. Computing the job impact score requires discovering the transitive closure of each job in the job dependency WDAG while maintaining edge weights along the way. We use the iterative bulk-synchronous-parallel algorithm introduced in Algorithm 1, §4.3.4, to compute the transitive closures.

Visual features. Job impact in Owl is visualized in two graphs. 1. The *job utility function graph* (Fig. 4.15) allows users to get a better sense of the urgency of their jobs. Each drop in score on the red/gray lines in the figure corresponds to the submission of a downstream dependent job relative to the submission time of the selected job. The magnitude of each drop corresponds to the value of the submitted downstream job. Naturally, users would want their jobs to complete before a downstream dependent job with high value is submitted, hence enabling users to infer a "deadline" for their jobs. 2. The *Sankey graph* (Fig. 4.16) allows users to quickly identify important

[10]Only deduping and summing the base scores of downstream jobs [95] is inflexible and disproportionately promotes jobs with high degrees of fan-in.

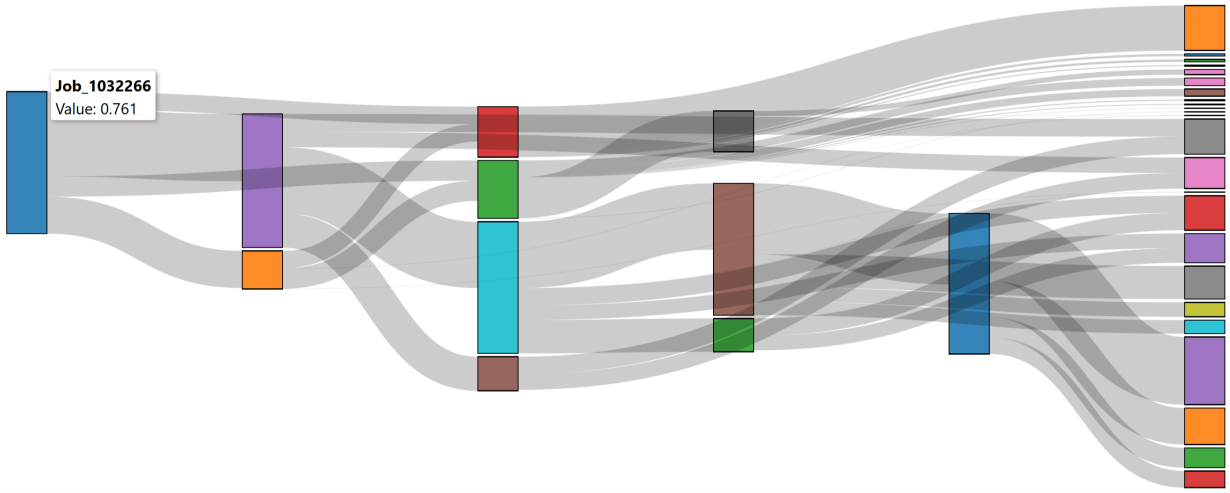


Figure 4.16: Interactive Sankey graph Shows how downstream jobs contribute value upstream. Each vertex is a job, with the height of the vertex representing relative job value. Hovering over a job displays its name and value The root job (left) represents the user-selected job. Clicking on leaf jobs (right) expands the graph further downstream.

downstream dependencies. The graph shows how value flows through job dependencies, where the height of a vertex represents the importance of a job, while the width of a flow between two vertices measures how much value a downstream job contributes upstream. For both graphs, users can select between our two facets of job impact: *work impact*, which is measured in CPU-hours, and *user impact*, which is measured in output downloads by users.

4.7.3 Summary

Shared multi-tenant infrastructures have enabled companies to consolidate workloads and data, increasing data-sharing and cross-organizational re-use of job outputs. This same resource- and work-sharing has also increased the risk of missed deadlines and diverging priorities as recurring jobs and workflows developed by different teams evolve independently. To prevent incidental business disruptions, identifying and managing job dependencies with clarity becomes increasingly important. Owl is a visualization tool that visualizes job dependencies derived from historical job telemetry and data provenance data sets through Wing, and this section has showcased Owl's features that can help users identify *critical job dependencies* and quantify *job importance* based on jobs' impact.

4.8 Related work

Workflow managers. Workflow management for batch analytics jobs is a widely studied area in the fields of databases and data management [73, 96, 97]. Our work differs in two primary ways: (1) workflow managers often assume the availability of a dependency graph up-front, while Wing infers properties of inter-job dependencies from job history; and (2) workflow managers

optimize only a single pipeline of jobs submitted by one user at a time, while Wing considers inter-dependent jobs across workflow and organization boundaries.

Cluster workload analysis. Although much work has been done on cluster workload analysis from many different perspectives (e.g., resource/workload heterogeneity [25, 37, 64, 89, 106], failure analysis [32, 47, 107], job predictability [102, 111, 123], and intra-job task dependency [60, 61, 121]), most prior work assumes (implicitly or explicitly) that each job is independent of other jobs. This chapter fills the knowledge gap with analyses of inter-job dependencies and application of this knowledge in cluster scheduling.

Cluster scheduling. Although a variety of work has been published in the area of cluster scheduling, each trying to address scheduling woes of different kinds of workloads (e.g., support for general batch analytics [30, 34, 49, 54, 55, 72, 78, 102, 124, 125], low latency scheduling [43, 44, 80, 100], and strategies to handle mixes of workloads [39, 52, 53, 109, 126]), most work in cluster scheduling similarly assume the independence of jobs. Our work shows that incorporating knowledge of inter-job dependencies can improve cluster scheduling in an environment with a lot of data and work product sharing, and we believe that considering inter-job dependencies can help future schedulers better tackle challenges, such as enabling better job task placement and learning better scheduling policies [90, 108].

Task-DAG schedulers assign resources to inter-dependent tasks within a job based on knowledge of the overall task-DAG [49, 60, 61, 90]. Such techniques and our proposed policies can be complementary, as task-DAG schedulers drill into job-level details while our schedulers (e.g., Wing-Agg) work at a higher level and treat jobs as black boxes. In particular, schedulers that predict the arrival of future jobs [78, 90] can benefit from the availability of inter-job dependency context to refine their predictions. Some task-DAG scheduling techniques could also be applied to the problem of inter-job dependency scheduling; but, these task-DAG schedulers generally assume upfront availability of task-DAGs, while full inter-job dependency graphs are rarely available ahead of time. An interesting direction for future research is in combining task-DAG scheduling techniques with some form of Wing-provided "probabilistic inter-job dependency" DAGs.

Jockey and Morpheus. Jockey [49] uses the direct dependencies of jobs to illustrate the importance of maintaining low job latency variance, but uses a step-function with value=1 until the *user-provided deadline* as each job's value function (VF). Morpheus [78] improves upon Jockey's notion of VFs by deriving deadlines based on a job's first consumer (as observed from historical instances of that job), but still considers all jobs as equal in value. In addition to our characterization of inter-job dependencies in a large analytics cluster, our work extends Morpheus and Jockey in two ways: (1) jobs no longer all have the same value—instead, Wing derives each job's value (and therefrom priority) as the sum of a chosen value metric (e.g., downloads) for all downstream dependencies, and (2) value is no longer a step-function with a single deadline based on a job's first direct consumer, but a rich decay proportional to the aggregate value of dependency sub-DAGs rooted in each direct consumer. While we do not directly compare against Morpheus, in §4.6.1, we find, in the context of Wing-MIL, that a premature drop in aggregate value can lead to the scheduler giving up early when dependency properties are not considered, leading to lower value attainment. Considering value as a step-function with a single deadline can therefore potentially be detrimental when inter-job dependencies are present in cluster workloads. While Wing-Agg uses only the initial "height" of the aggregate value VF of each job to set priorities, we believe that full aggregate value VFs can still better guide other scheduling decisions, such as

determining which jobs to load-shift.

Systems using job recurrence and data provenance. There has also been much prior work on systems that efficiently collect provenance data [40, 95] and systems that both exploit job recurrence and data provenance on other problems [63, 76], such as garbage-collecting shared computation results. Our work uses similar ideas, but focuses on facilitating better value attainment in resource scheduling.

4.9 Summary

Complex inter-job dependencies pervade modern data lakes, creating complex problems as cluster schedulers make decisions without knowing of them. The Wing dependency profiler uncovers these dependencies from provenance logs and provides improved guidance to cluster schedulers, and Owl allows users to visualize their job's dependency patterns and the importance of their jobs based on downstream impact. Evaluations with real job traces show that significantly more value, in terms of successful user downloads, can be attained by using Wing-guided priority assignments over those provided by users. Wing's effectiveness opens a new range of resource management possibilities guided by automatically-determined knowledge of the impact of jobs.

Chapter 5

Talon: Reducing costs with dependency-informed load-shifting

In Chapter 4, we studied how inter-job dependencies can be used to improve cluster scheduling to attain greater job value. In this chapter, we take a look at how we can further exploit inter-job dependencies to reduce long-term resource commitment in cluster capacity planning by load-shifting and using intermittently-available transient resources.

As a recap, as companies amass increasing amounts of data, shared compute environments such as public and private clouds are favored more and more over silo-ed clusters to eliminate the cost and complexity of cluster ownership and facilitate easier data analytics and data sharing. Indeed, the lowering of data access barriers and availability of systems that facilitate data discovery [28, 66] induced a dramatic rise in the number of *inter-job dependencies* in shared clusters (Chapter 4), where, for example, a batch analytics Job 2 *(inter-)depends on* an earlier job Job 1 if Job 2 takes as input any output generated by Job 1.

The sharing of data environments has introduced new models for users to manage compute resources for their analytics jobs. To ensure that an organization's users have enough resources to complete their data analytics tasks in shared environments, organizations can buy fixed, guaranteed share of *reserved* resources under committed, long-term contracts [2, 30, 39]. Such schemes, however, are inflexible, because they lock users in long-term and can lead to reduced cluster utilization, as reserved capacity are left unused during workload lulls.

A common strategy for clusters to increase resource utilization is to intermittently make unused resources available at lower priority and often at lower costs [7, 39]. Azure, for instance, make unused resources available at lower costs as Spot VMs [3, 24]. Workloads run on these *transient resources* run under the proviso that they can be *preempted* at any given moment, when resources are needed by higher priority workload (e.g., jobs run with reserved capacity).

In provisioning reserved capacity, whether on-premise or in shared public clouds, organizations want as little long-term resource commitment as possible to lower cost and increase flexibility, but using short-term transient resources present unique challenges in resource availability and preemption. To address these challenges, we propose *Talon*, a novel heuristics-based workflow manager. Talon minimizes long-term reserved resource commitment and reduces the cost of running workloads by exploiting transient resources, all while keeping jobs safe from violating their deadlines. It does so by exploiting the prevalence of inter-job dependencies in shared clusters

to safely *load-shift* jobs to run when transient resources are plentiful, and by co-considering job task placement and transient resource availability.

Load-shifting can be used to reduce the peak of cluster workloads, and thus reduce required cluster capacity to support the workloads, by moving jobs run during peak hours to off-peak times. However, traditionally, load-shifting is difficult without cooperation from users, as schedulers are often not provided context on whether jobs can be started early or be delayed. The advent of GDPR [127], which has necessitated many companies to track job data provenance, has changed this by enabling the analyses of (a) job output consumption patterns and (b) *inter-job dependencies*. Together, they allow Talon to determine when job outputs are consumed (i.e., *job deadlines*) and when jobs are ready to run. Talon analyzes and exploits this newly available job metadata to safely load-shift jobs to when transient resources are plentiful to reduce reserved resource capacity and cluster operation costs.

While transient resources present significant opportunities to lower reliance on reserved capacity and cluster operation costs, a challenge to effectively use transient resources is its intermittent availability: resources can be preempted in bulk [68, 69], interrupting job execution and leading to clusters not having enough transient resources to serve jobs requesting them. Talon addresses this challenge by (1) not fully utilizing all available transient resources, leaving buffers of transient resources to handle preemptions and workload spikes, and by (2) understanding jobs' deadline violation risks through data provenance analyses, providing safety for jobs with higher risk of deadline violation with task-level redundancy, while allowing jobs with lower risk to run on transient resources without redundancy, thereby more effectively using available transient resources at the same time. Although at first glance, this conservative usage of transient resources seems sub-optimal in minimizing reserved resource commitment, we find that when used in-tandem with Talon's load-shifting policy, Talon experiences minimal job deadline violations without more reliance on reserved resource capacity.

To explore Talon's efficacy to minimize reserved resource commitment while keeping jobs from violating deadlines, we evaluate Talon using workload from Cosmos, Microsoft's internal big data platform, and using transient resources from Azure, Microsoft's public cloud. Our experiments find that Talon effectively achieves its goals, allowing cluster operators to reduce reserved capacity by up to 38% compared to running the entire workload on reserved capacity. We also measure Talon's monetary cost-efficiency when running entirely in the cloud, using Azure reserved instances [2] as long-term reserved resources and using Spot instances [3] as transient resources. Our experiments find that Talon can reduce cost compared to running workloads entirely on reserved resources by up to 31%. Talon is also effective in minimizing job deadline violations, yielding a job deadline violation rate of 0.01%. We believe Talon can also help reduce cost and long-term capacity managed in other scenarios, e.g., hybrid clouds with workload bursts and systems exploiting intermittent green energy sources such as solar and wind energy.

Contributions. This chapter makes the following primary contributions: (a) it presents the first study of batch analytics job load-shiftability based on real-world job input dependencies in a large data analytics cluster, presenting significant opportunities for optimizing batch analytics job scheduling, (b) it presents methods to identify jobs that are load-shiftable using inter-job dependencies and job output access logs, and (c) it proposes *Talon*, a novel job workflow manager that exploits job load-shiftability and low-cost-low-reliability cloud resources to reduce the workload peak on reserved resources. Talon allows cluster operators to reduce reserved capacity

by up to 38% and reduce cost by up to 31% compared to running workloads entirely on reserved resources, while introducing minimal job deadline violations.

5.1 Background

This section provides a high-level overview of Azure, the public cloud on which we evaluate Talon on, and its reserved and transient resources offered. We then provide a high level overview of load-shifting, and describe properties of Cosmos that allow Talon to effectively load-shift jobs to reduce committed reserved capacity.

5.1.1 Reserved and transient resources in Azure

Azure VM is Microsoft's public IaaS (infrastructure-as-a-service) offering, where customers can rent virtual machines (VMs). VMs can be rented under *reserved* contracts, where customers will reserve VM capacity for long periods of time (in years) [2] and will be billed whether customers use the resource or not, or under *pay-as-you go* contracts, where customers can rent VMs for however long they choose, but are more expensive compared to VMs rented under reserved contracts. In addition to regular VMs, Azure VM and other IaaS providers often sell excess capacity in the form of preemptible, *transient* VMs, such as Spot [3] and Harvest VMs [24, 129]. Similar to VMs offered by AWS, as discussed earlier in §2.1.1, these VMs are intermittently available and offered at a steep discount (up to 90% off), with the proviso that they can be reclaimed by the IaaS provider with little or no warning. Talon safely exploits low-cost, intermittently available transient (Spot) resources to reduce reliance on reserved resources that require long-term rental (e.g., Azure reserved instances) or operational (e.g., bare-metal clusters) commitment.

Cosmos workloads are suited to run on transient resources. As mentioned in §4.1.1, Cosmos primarily consists of SCOPE jobs, and we specialize Talon to effectively schedule SCOPE jobs. SCOPE jobs are batch analytics jobs similar in nature to Apache Spark jobs. SCOPE jobs are fault-tolerant and massively parallel. Each SCOPE job consists of a directed acyclic graph (DAG) of stages, and each stage consists of multiple tasks. SCOPE tasks, in the majority of use-cases, are idempotent, allowing tasks to be replicated and speculatively executed [31]. This idempotency allows Talon to run tasks fault-tolerantly on transient resources, such that resource preemption causes minimal disruptions to job execution time. Furthermore, intermediate outputs are persisted between stages, either to local storage on a server in the Cosmos cluster or to ADLS, such that jobs do not need to be restarted from scratch if tasks get preempted, improving fault-tolerance.

5.1.2 Load-shifting to reduce reserved resource commitment

Load-shifting a job in time changes the time when a job is run, and can be used to reduce the peak resource demand of jobs in a cluster. In load-shifting, a job can either be *advanced* (so the job runs earlier in time) or *delayed* (so the job runs later in time). Load-shifting is practical in scenarios such as cluster capacity crunches, which can happen, for example, during scheduled

machine maintenance. In these scenarios, jobs run during periods of high resource usage can be shifted to run at *off-peak* times, when more cluster resources are available.

Load-shifting, however, has historically been difficult: delaying jobs requires us to know when a jobs' output is depended upon, and advancing jobs requires us to know when a future job will be submitted and when its inputs are available, often requiring the cooperation of users. Talon exploits newly available inter-job dependency information, described in more detail in §4.1.

Cosmos workloads are suited for load-shifting. Cosmos's workload exhibits *diurnal* behavior, with high resource usage peaks and low usage troughs. This behavior allows significant opportunity to move (load-shift) workloads off-peak and reduce the amount of committed reserved resources required to complete the workload reliably.

5.2 Talon overview

This section describes the Talon system at a high level, providing a summary of its functionalities, describing its job submission flow, and highlighting key insights.

5.2.1 Load-shifting via inter-job dependencies

Load-shifting a job in time changes the time when a job is run, and can be used to reduce the peak resource demand of jobs in a cluster. In load-shifting, a job can either be *advanced* (so the job runs earlier in time) or *delayed* (so the job runs later in time). Load-shifting is practical in scenarios such as cluster capacity crunches, which can happen, for example, during scheduled machine maintenance. Load-shifting shifts jobs run during capacity crunches off-peak, so they can run when more cluster resources are available. Talon load-shifts jobs to reduce reserved resource commitment. Load-shifting, however, has historically been difficult: delaying jobs requires us to know when a jobs' output is depended upon, and advancing jobs requires us to know when a future job will be submitted and when its inputs are available, often requiring the cooperation of users. Luckily, we can derive both the advanceability and delayability of jobs using data from JobRepo and ProvRepo.

Talon exploits the prevalence of recurring jobs and inter-job dependencies in Cosmos to determine the load-shiftability of a large fraction of jobs run in Cosmos. The delayability of a job can be estimated by analyzing output data usage patterns from historical runs of the same template[1], while the advanceability of a job can be determined using historical inter-job and data-dependency patterns of jobs of the same template. With the ability to load-shift jobs, Talon works in-tandem with resource managers to reduce peak cluster workloads safely. We examine the load-shiftability of Cosmos workloads in more detail in §5.3.

5.2.2 Architecture and job lifecycle

This section presents an overview of the architecture of Talon and walks the reader through the lifecycle of a job managed by the Talon workflow manager. Readers are welcome to follow along

[1] In theory, though predicting job output usage time can be difficult (§5.3.4).

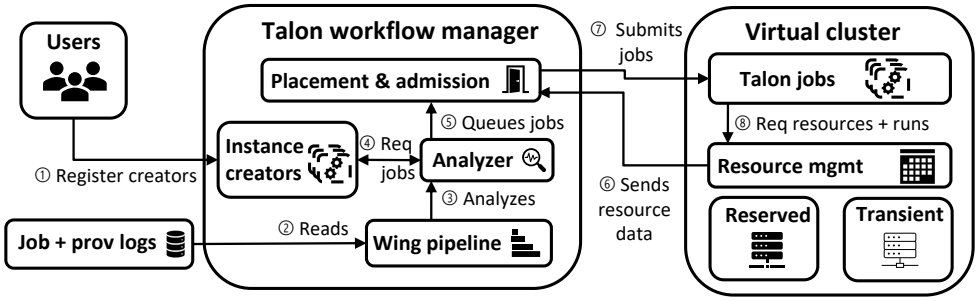


Figure 5.1: Talon architecture and job lifecycle. Talon acts as an intermediary between users and the virtual cluster consisting of reserved and transient resources. Recurring jobs are registered via job instance creators with Talon for load-shifting. As jobs become ready to run, Talon works with the virtual cluster resource manager to determine when and how best to submit queued jobs via its placement and admission policies.

with the submission process by referencing Fig. 5.1. We examine each component of the system in further detail in later sections of the chapter.

Similar to other workflow managers, Talon manages and submits jobs for users to their designated clusters. However, in addition to scheduled job submissions, Talon also works with the cluster resource manager and can choose to load-shift jobs based on cluster load. Jobs registered with Talon are submitted as follows:

(1) The user registers *job instance creators* with Talon for load-shiftable jobs. In addition to usual job submission features such as periodic job submission that many workflow managers provide, job instance creators allow users to create job instances from templates and specify if a job should be submitted when Talon deems the job ready based on historical inter-job dependencies (§5.3). It also allows users to specify Talon-specific submission parameters at the time of notification from Talon such as latest job admission time, allowing for extra flexibility. *(2)* The Wing pipeline (§5.3.1) processes job and data provenance logs from JobRepo and ProvRepo to determine job recurring-ness, recurring jobs' in-and-output dependencies, and recurring jobs' output consumption patterns. Analysis results are then passed to the Talon analyzer for evaluating job load-shiftability. *(3)* Analyzing the output of the Wing pipeline, the Talon analyzer determines which jobs are eligible for load-shifting based on the availability of their inputs, and load-shifting time bounds as determined by (§5.3). *(4)* After jobs are deemed ready to start by the Talon analyzer, the analyzer requests jobs from its registered pool of job instance creators, which supplies Talon with the actual jobs to submit, annotated with Talon job submission parameters such as latest job admission time for each instance. *(5)* The analyzer then submits jobs to the admission policy (§5.4), where jobs are queued. *(6)* The admission policy then receives cluster resource data from the cluster and works with the placement policy to determine where to place job tasks (§5.4.1) and when jobs should be admitted (§5.4.2). *(7)* Jobs are then submitted to the cluster, and will request resources from the cluster resource manager as determined by the placement policy in the previous step.

5.2.3 Operating modes

Talon operates in two modes: (1) provision analysis and (2) actual scheduling. In *provision analysis* mode, Talon plans for the next cycle of reserved resource provisioning (i.e., reserved capacity planning) and aims to reduce its peak workload run on reserved resources. In this mode, Talon will more aggressively utilize transient resources in order reduce the peak of reserved capacity. In *actual scheduling mode*, Talon is provided a pre-determined amount of reserved resources. In this mode, Talon tries to fully utilize reserved resources before using transient resources in order to not waste reserved resource-time that are already paid-for.

5.3 Talon analyzer: Finding load-shiftable jobs

The *Talon analyzer* is responsible for identifying load-shiftable jobs and sending them off to the Talon admission policy when jobs are deemed ready to start (i.e., the jobs' full set of inputs are all available), where the admission policy determines when to actually submit the job to the virtual cluster. This section describes the Talon analyzer, its input datasets, and how it identifies load-shiftable jobs using inter-job dependencies and job recurrence.

5.3.1 The Wing pipeline: Identifying recurring job candidates for load-shifting

For jobs to be load-shiftable, they must (1) be *recurring*, and (2) exhibit *recurring in/output dependency patterns*. Specifically, we cannot advance jobs that we do not know will arrive in the future, nor can we safely delay jobs without knowing when their output files will be needed.

To identify candidates for load-shifting, Talon utilizes output from the Wing pipeline (Chapter 4), an inter-job dependency profiler implemented as a pipeline of batch-analytics jobs that identifies recurring jobs and characterizes their inter-job dependencies based on historical job telemetry and data provenance from JobRepo and ProvRepo, respectively. Using definitions from Wing, jobs are identified as *recurring* if (a) jobs of a recurring template are submitted at least three times over a period of three months, with at least one submission each month, (b) templatized job names are an exact match, and (c) source-code signatures are an approximate match. Inter-job dependencies are identified as recurring if both the upstream and the downstream jobs are recurring.

The Talon analyzer takes as input from Wing a set of historically recurring jobs, their characteristics (e.g., job arrival rate, recurrence cadence, and run time distributions), and jobs' full sets of input/output dependencies, recurring or otherwise.

5.3.2 Advanceable jobs

For a job to be advanceable, its arrival must be predictable. Job arrival predictability can either be provided to Talon directly by job submitters when registering job instance creators, or can be inferred via historical job arrival and input consumption patterns. Talon can also work with existing workflow managers to obtain more job metadata to aid in load-shifting. This section will focus on *inferring future job arrivals*.

Heuristic to identify advanceable jobs. Talon identifies a job as *advanceable* if it is recurring and if its future arrival is predictable; that is, if the recurring job satisfies at least one of the following conditions:
(1) The job's arrival is inferrable via input dependencies. Based on historical inter-job dependencies, Talon will identify jobs of recurring template J_A as advanceable if the completion of one of J_A's upstream recurring jobs is highly indicative that a job of template J_A will arrive in the future. Recurring jobs of template J_A are identified as advanceable based on upstream recurring jobs of template J_B if jobs of J_A has a recurring input dependency on jobs of J_B, *and* for $> 90\%$ of J_B's historical job instances, there is a job instance of J_A that depends on their outputs. Here, the completion of a job of J_B is a strong signal that a job of J_A will arrive in the future.
(2) The job is periodic. A recurring job is predictable if it is periodic. *Periodic jobs* are recurring jobs that are submitted on a fixed schedule periodically (e.g., submitted every hour at the start of the hour). We obtain the set of periodic job templates from Wing's analyses of historical data, where Wing identifies periodic job templates as templates whose jobs have near-constant inter-arrival times (i.e., their inter-arrival times have a low coefficient of variation) (§4.1).

5.3.3 Workload advanceability

We conduct a study in Cosmos to see how much opportunity job advanceability provides Talon in load-shifting. Namely, we want to find out (1) the precision our heuristic yields to identify advanceable recurring jobs (i.e., the fraction of actual future-arriving jobs out of all that were predicted) and (2) how much resource-time in Cosmos is advanceable by how much. This study does not aim to identify the full set of advanceable jobs, but rather only aims to identify enough advanceble job resource-time to demonstrate sufficient opportunity for load-shifting.

Methodology. Here, we use a window of two weeks of jobs and inter-job dependencies as the set of historical data upon which we "train" a predictor (the *training window*) to predict the arrival of future jobs in the next day (the *testing window*). We slide the windows over two months of data, advancing the windows one day at a time.

Advanceability precision. Our identification of future-arriving jobs achieves a precision of 87%, close to the 90% threshold (§5.3.2) we use in our policy to identify advanceable jobs via inter-job dependencies.

Advanceable resource-time. For each advanceable job we correctly predict to arrive, we measure how much time we can advance the job by. The advanceability of all recurring jobs is upper-bounded by their latest-arriving input, from which we identify using data provenance logs. If the recurring job is also a periodic job, we further upper-bound its advanceability by half of its periodicity. The advancebility of a daily job, for example, is upper-bounded by *max*(12 hours, time from latest arriving input). Our study finds that, while most jobs are not advanceable by a large amount (Fig. 5.2a), up to 42% and 24% of job *resource-time* can be advanced by greater than 15 minutes and one hour, respectively (Fig. 5.2b). This indicates that large jobs are often more advanceable and yield significant opportunities for load-shifting.

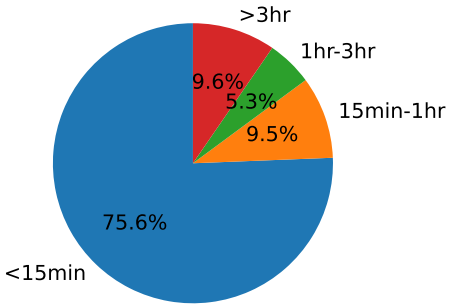


Figure 5.2(a): **Cluster job count advanceability.** This figure breaks jobs submitted to the cluster down by job advanceability. While only approximately 24% of jobs are advanceable by $>$ 15 minutes, we find that large resource-time consuming jobs are more advanceable, such that $>$ 42% of resource-time are advanceable by $>$ 15 minutes.

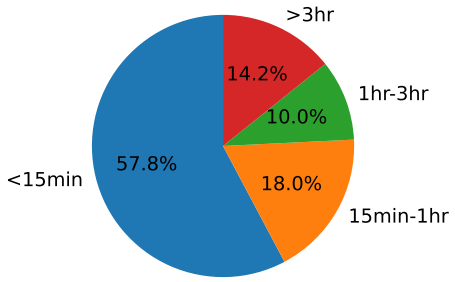


Figure 5.2(b): **Cluster job resource-time advanceability.** This figure breaks cluster resource-time (job-utilized-resources $\times$ time) down by advanceability. More than 42% of resource-time is advanceable by $>$ 15 minutes, and more than 24% of resource-time is advanceable by $>$ one hour, demonstrating opportunities to load-shift cluster compute.

5.3.4 Delayable jobs and workload delayability

To safely delay a job (i.e., start a job later), we want to ensure that its outputs are generated before outputs are first used, either by a downstream job or by an operation through the front end. This requires us to predict two things about a job: its run time and when its outputs are first used. A recurring job with low predictability in either its runtime or when its output is used cannot be safely delayed.

Job run time predictions. We use a sliding-window model that uses the median run time of recurring jobs in the past week to predict the run time of an upcoming recurring job of the same template. The window of our model slides by a day at a time (i.e., model is re-computed daily). We find that 42%, 60%, and 83% of our model's predictions yield run time prediction errors, as measured by $(predicted - actual)/actual * 100$, of 10%, 20%, and 50% or less, respectively. Fig. 5.3 shows the prediction error distribution of our model.

Job time to output usage (TTOU). While we have access to significant job metadata to categorize recurring jobs, it is still difficult to predict when the output of a job will first be accessed because the distribution of recurring jobs' *time to output usage (TTOU)* is multi-modal. Fig. 5.4 shows the distribution of recurring jobs' TTOU, normalized over each recurring template's median TTOU. While the distribution peaks at the median, many jobs fall far to both ends of the distribution, making prediction difficult. Indeed, we find that even using the most conservative prediction for TTOU to prevent deadline violations, where a predictor predicts the shortest seen historical TTOU each time, the predictor still over-predicts for 2% of jobs.

Talon does not delay jobs. When ignoring unpredictable jobs with low run time or low time to output usage predictability, our workload yields only 9% of job resource-time that can load-shifted by more than an hour. Considering that delaying jobs is both high-risk and low-reward, *Talon does not explicitly consider delaying jobs*. We leave the design of a load-shifter that safely delays

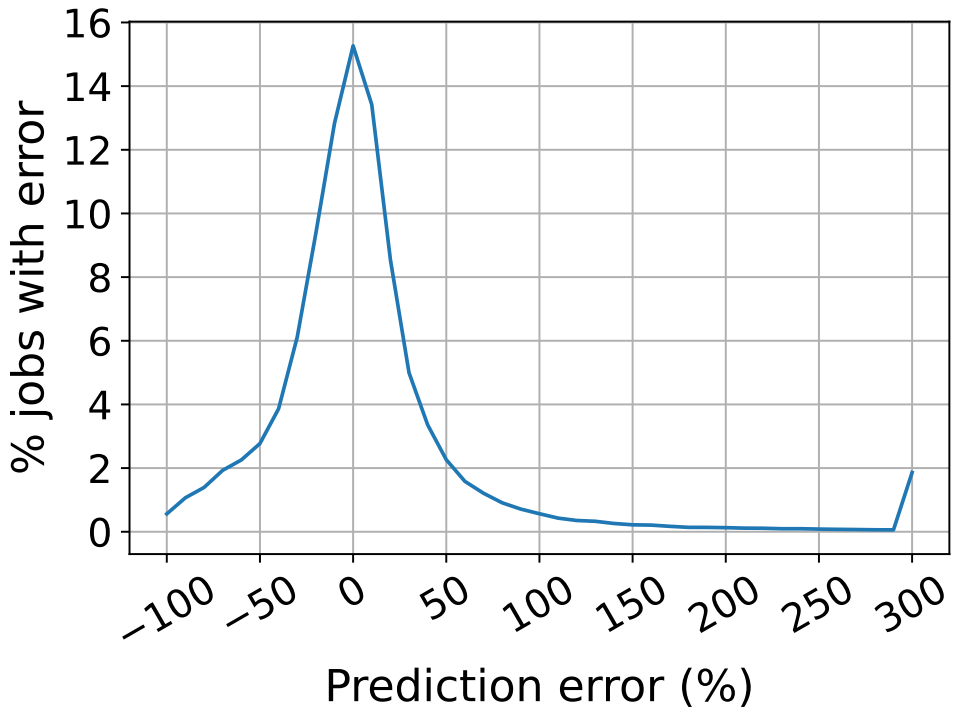


Figure 5.3: Job run time prediction error. This figure shows the probability mass function of the prediction error of our median based recurring job run time predictor. Here, $error = (predicted - actual)/actual * 100$. 60% of predictions fall within $\pm 20\%$.

jobs as future work.

5.3.5 Load-shifting correctness and flexibility

Talon uses inter-job dependencies derived based on historical job data provenance recorded in ProvRepo. As described in §4.1.4, these logs capture file versioning, such that each modification to an existing file produces a new file entry in ProvRepo, i.e., each version of each file is treated as a unique file. Talon therefore also respects file versioning. While this provides guarantees for output correctness for jobs that depend on specific versions of files, Talon misses out on opportunities to load-shift jobs even more for jobs that are less strict with respect to file versioning.

5.4 Talon's load-shifting approach: Job placement, admission, and scheduling

This section first describes Talon's policies under its provision analysis mode (§5.2.3), detailing how it decides on what kind of resources (reserved or transient) to run jobs to minimize job deadline violations with its *job placement policy* (§5.4.1) and how it load-shifts jobs using information from the Talon analyzer and the cluster resource manager in-tandem with its *job admission policy* (§5.4.2). Talon's provision analysis mode uses transient resources more aggressively

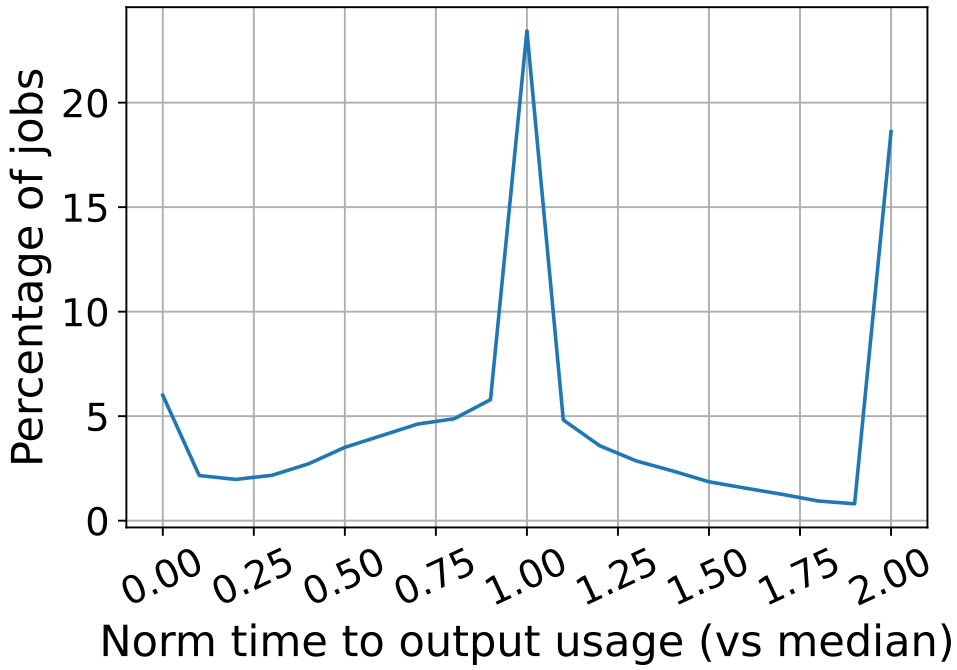


Figure 5.4: Normalized job time to output usage distribution. This figure shows the distribution of time to output usage (TTOU) of recurring jobs normalized against the median TTOU of jobs of the same template. Much of the distribution lies toward the two tails, making accurate predictions of TTOU difficult.

compared to its actual scheduling mode in order to reduce the reserved resource peak. We then describe how Talon performs actual job scheduling in a virtual cluster with a mix of reserved and transient resources (§5.4.3).

5.4.1 Job placement policy

This section describes Talon's *job placement policy* under its provision analysis mode. Talon's job placement policy instructs jobs on what kind of resources its tasks should run and whether or not jobs should employ fault tolerance strategies if running on transient resources. Talon's job placement policy considers resource and job properties (e.g., recurrence, predicted run time, and job load-shifted time) in-tandem, such that jobs can more effectively use available cluster resources to minimize reserved resource peaks, while allowing jobs to meet their deadlines.

Keeping track of cluster resource type usage. Talon's job placement policy determines how a job should request resources for its tasks. To reduce peak reserved resource workload, the placement policy can instruct jobs to use transient resources instead of reserved resources. However, due to the intermittently available nature of transient resources, there can be times when there is not enough transient capacity to handle queued transient resource requests. The job placement policy therefore receives periodic resource snapshots from the resource manager to keep track of available transient resources in the cluster.

Job strategies to handle resource preemptions. For jobs whose tasks run on transient resources, tasks have a chance of failing due to resource preemption. In batch analytics jobs, a common

strategy to recover from task failure is to retry the failed task, but this can increase job run time. An effective way to handle task failures in batch analytics jobs without increasing job run time is to *replicate* tasks, i.e., to run multiple copies of a given task. As long as one replica of the task succeeds, the task is deemed successful. The Talon job placement policy can instruct jobs to run tasks with replicas if jobs run on transient resources. In our experiments using workload from Cosmos and Harvest VMs from Azure, we find that using a task replication factor of two (creating two copies of each task) works well to handle transient resource (Harvest VM) preemptions. Other more advanced fault-tolerance strategies can also be applied at the job placement policy.

Handling transient resource bulk preemptions. Transient resources are *intermittently available*, and large amounts of transient resources can be preempted in bulk at any given time, as observed in prior work [68, 69] and in our workload traces. Bulk resource preemptions can lead to *retry storms* where failed tasks running on preempted resources can all submit retry requests at once, which in turn can lead to long queue times of requests if transient resources are used too aggressively. Talon robustly handles bulk transient resource preemptions by using transient resources judiciously, rather than fully packing transient resources whenever available, described in more detail next.

Determining the type of resource a job should use. Talon splits jobs into two cases when constructing jobs' task resource request policies: (1) ad-hoc and shorter-running recurring jobs and (2) longer-running recurring jobs.

(Case 1) Ad-hoc and shorter-running recurring jobs. If a job is categorized as ad-hoc (non-recurring) or as a predicted shorter-running recurring job (predicted to run less than three minutes), Talon uses transient resources if there is an *abundance* of available transient resources, otherwise it uses reserved resources. Ad-hoc and shorter-running recurring jobs are always run *reliably* by Talon, i.e., they either run on reserved resources or on transient resources with replicas.

Ad-hoc jobs need to be run reliably because they are less predictable, as they have no tracked historical instances. We find that ad-hoc jobs are not correlated between any of job run time, TTOU, and job size. We therefore need to run ad-hoc jobs carefully, with high reliability, to meet job deadlines.

Shorter-running recurring jobs are run reliably, on the other hand, for two primary reasons: (i) predicted shorter-running recurring jobs use less resource time. Although these jobs account for nearly half of all jobs in our workload, they only account for roughly 3% of cluster resource-time. Scheduling them reliably therefore present minimal impact to both reserved and transient resource availability, while allowing a large fraction of jobs to meet their deadlines; and (ii) preemptions cause greater disruptions for shorter-running jobs. When resources are preempted in these jobs, they will need to detect task failures, re-request preempted resources, and set up retried tasks again to restart the tasks. For short jobs, the process can take a long time relative to their run times, and is more likely to cause job deadline violations.

To make sure that these jobs run reliably, Talon's placement policy runs ad-hoc and predicted shorter-running jobs on transient resources with replicas only if there is an *abundance* of transient resources to prevent retry storms on bulk resource preemption. Otherwise, Talon runs these jobs on reserved resources. To determine if there is an abundance of available transient resources, Talon uses the following heuristic: $available > allocated/t$, where t is the *placement resource slack* and is configurable (experiments set $t = 2$).

(Case 2) Predicted longer-running recurring jobs. These jobs include all other jobs not covered in the previous case, and account for roughly 80% of cluster resource-time while only accounting

for 35% of all jobs. There are therefore significant opportunities to reduce reserved resource pressure by placing these jobs on transient resources. Talon's placement policy decides that if there is an abundance of transient resources or if such an incoming job is advanced by at least half of its predicted run time, the job will use transient resources for its tasks; otherwise it will use reserved resources.

Running too many of these jobs on transient resources with replicas, however, can run the risk of triggering retry storms upon bulk preemption. As these jobs take the bulk of cluster resource-time, to reduce transient resource pressure, Talon's placement policy will choose to take a risk to run these jobs without replicas if they are advanced by $> p\times$ their predicted run times, where p is a configurable *placement job run time slack*. Since jobs advanced by $> p\times$ their predicted run times have a long slack until their deadlines (experiments set $p = 1$), they will also have a long period of time to retry their tasks should any of them fail. By running these jobs with a single replica, Talon reduces resource competition for transient resources, reducing risk of retry storms and allowing more jobs to run on transient resources instead of reserved resources. Our experiments show that this strategy to run only a single replica reduces transient resource-time usage by 10%, while only leading to minimal job deadline violations.

5.4.2 Job admission policy

Here, we describe Talon's job admission policy under its provision analysis mode. After a Talon job is ready to run as determined by Talon's analyzer (§5.3), it is not immediately submitted to the cluster. Rather, it is placed in the queue of Talon's *job admission policy* to control for cluster load and resource availability. Advancing jobs immediately without consideration for resource availability can cause jobs to run during periods of higher workload, introducing additional resource contention among jobs. We observe that advancing jobs immediately both increases reserved resource capacity needed and increases the number of job deadline violations. To determine when a job should be admitted, the admission policy needs to work with both the resource manager and Talon's job placement policy.

Admitting jobs. When a job is queued at the admission policy, the policy first checks the job's type. If the job is ad-hoc or a predicted-short recurring job, the job is immediately admitted. As mentioned in §5.4.1, these jobs are often low resource-time impact or are unpredictable. If the job is a predicted-long recurring job, the admission policy checks if the job is run on transient resources. The job will be admitted if it is run on reserved resources, indicating urgency, or if there is an abundance (§5.4.1) of transient resources.

If the job is not yet admitted, the admission policy will check if the job is advanced by more than $m\times$ its predicted run time, where m is a configurable *admission job run time slack*. Note that m is a separate setting from p, as defined in the placement policy. Setting $m > p$ allows predicted-longer recurring jobs to be admitted earlier and run on transient without replicas, allowing advancement of jobs while introducing less resource contention. Sensitivity experiments found that setting $m \leq p$ increased reserved peak workload. Our experiments found setting $m = 3$ and $p = 1$ to be effective.

Latest admission time. Each job can be associated with a *latest admission time*, such that Talon admits a job no later than its latest admission time. This time can either be specified by the job instance creator or be determined by how much time a job is advanced by. Advanced jobs' latest

```
// Global parameters
Data: resources: Current state of cluster resource availability
      p: Tunable, placement run time slack
      t: Tunable, placement resource slack
      m: Tunable, admission run time slack
Function is_abundant(resource_type) is
    Input  : resource_type: Reserved/transient
    Output : True if abundant
    rot = resources where type == resource_type;
    return rot.idle > rot.alloc/t
end
Function is_long_reJob(job) is
    Input  : job: The incoming job
    Output : If job is a predicted-longer recurring job
    if !is_recurring(job) then return False;
    return predicted_runtime(job) > 3 minutes
end
Function placement(job) is
    Input  : job: The job for placement
    Output : Pair of <resource type, replica factor>
    if is_long_reJob(job) then
        adv_mul = (time job is advanced by) / job.pred_runtime;
        if adv_mul > p then return <transient, 1>;
        if adv_mul > 1/2 —— is_abundant(transient) then
            return <transient, 2>
        end
        return <reserved, 1>;
    else if is_abundant(transient) then
        return <transient, 2>;
    else
        return <reserved, 1>;
    end
end
Function admission(job) is
    Input  : job: The job for admission
    Output : Whether the job should be admitted
    if job is ad-hoc || !is_long_reJob(job) then return True;
    job_pl = placement(job);
    if job_pl is reserved then return True;
    if is_abundant(transient) then return True;
    adv_mul = (time job is advanced by) / predicted_runtime(job);
    return adv_mul ≤ m
end
```

Algorithm 2: Placement and admission policies.

admission times are their original submission times before advancement. Algorithm 2 shows the heuristics used for Talon's placement and admission policies in its provision planning phase.

5.4.3 Actual scheduling mode

§5.4.1 and §5.4.2 describe how Talon plans for future capacity, and more-aggressively uses transient resources to reduce the peak of workload run on reserved resources—inviting a tendency to under-utilize reserved resources. When performing actual scheduling, however, schedulers should fully use reserved resources, which are already paid-for ahead of time.

Talon's actual scheduling mode allows for better utilization of reserved resources. Here, Talon's policy placement *decisions* are made as-is using a shim resource management layer, but the *actual placement* of jobs' tasks use reserved resources when possible, assuming a planned capacity limit on reserved resources. In other words, when performing actual scheduling, Talon's reserved capacities are limited based on reserved capacity provisioning results determined by decisions in §5.4, but will utilize available reserved capacity whenever possible.

5.5 Experimental setup

We run simulation experiments using workload from a Cosmos production cluster and transient resources from Azure to evaluate Talon's efficacy in minimizing reserved resource commitment. This section describes how our experiments are set up. We start with providing an overview of Cosmos workloads and Azure resources and describing workload and resource traces on which we use to evaluate on results. We then evaluate the accuracy of our simulator, describe load-shifting approaches that we compare Talon against, and detail our evaluation metrics and cost models.

5.5.1 Cosmos, Azure, and our workload

Cosmos, YARN, and gang-scheduling. Cosmos uses YARN [39, 125] to allocate tasks on to its compute resources, and uses a simple priority-based load-shifting approach. Different from stock YARN, batch analytics (SCOPE) jobs in Cosmos are scheduled with *gang semantics*, such that a job is only started when it is able to acquire a user-provided minimum number of parallel running containers. We extend our application logic and resource manager to gang-schedule resources. In addition, each SCOPE job consists of a directed acyclic graph (DAG) of stages, with each stage consisting of tasks that can run in parallel. We simplify SCOPE jobs used in our experiments to consist only of a set of linear critical path stages. Should tasks of a stage fail, only the failed tasks within the stage need to be retried.

Workload description. We use four weeks of Cosmos job and inter-job dependency data to evaluate our approaches. Talon uses data in the first three weeks to establish job and inter-job dependency profiles. Experiments are conducted over jobs submitted in the final week. Each day of traces contains 40k jobs and 160k inter-job dependencies on average.

We evaluate on our results on an entire week's worth of workload at once rather than as multiple experiments over multiple days or hours because (1) load-shifting of jobs can potentially span across days and because (2) capacity-planning requires planning for the peak usage across longer periods of times. We note that Talon achieves similar results and out-performs compared approaches in peak minimization over multiple experiments by splitting the workload into multiple days (2% standard deviation across days).

Inter-job dependencies. While our evaluation considers inter-job dependencies to determine whether a job is able to start running and whether or not a job is load-shiftable, to allow all jobs to run to completion in our experiments, we do not fully model inter-job dependency characteristics. That is, if job j_b has a required dependency on upstream job j_a, j_b will not fail at submission time even if j_a has not completed by then. j_b will be delayed until all of its upstream jobs complete and all of its inputs are available.

Job output consumption and deadline-sensitive jobs. 85% of workload jobs are accessed, either by a downstream job or by a Cosmos-external user or process, within a week of the job's completion. The figure drops to 80% when only looking at output consumption external to Cosmos where consumption manifests as a *download operation* in Cosmos's front end logs. In our experiments, we primarily focus on SLO attainment from the perspective Cosmos-external systems and users. We therefore define a *deadline-sensitive job* as a job whose output has associated download operations.

Virtual cluster setup. Our experiments assume jobs run in a hybrid "virtual cluster", where reserved capacity assume no failure and transient capacity manifest as Harvest VMs rented from Azure that can be preempted, or reclaimed by Azure at any given moment. The two types of resources are managed as separate pools by YARN using node labels.

Harvest VM and eviction rates. Talon exploits Harvest VMs, a type of low-cost transient (Spot [3]) VM contract from Azure, to reduce usage of reserved resources and overall reserved capacity. These VMs are less reliable compared to reserved VMs and can be reclaimed (preempted) by Azure with little warning. Different from regular Spot VMs, machine resources (compute and memory) allocated to a Harvest VM can also grow and shrink dynamically over time. Fig. 5.5 shows the time-to-preemption distribution for the Harvest VMs that we use in our workload, while Fig. 5.6 shows the fluctuation of total available transient resources over time in our experiments. Nearly 90% of all VMs have an uptime of greater than one hour, and more than 50% of machines stay up for more than a day. For our experiments, we apply an average of 84% discount to transient VMs used compared to on-demand reliable VMs, using prices for Spot VMs reported by Azure [4].

Transient/Harvest VM resource availability. We apply resource availability constraints on transient resources in our experiments, as transient resources are only intermittently available. For our experiments, we use a scaled version of real Harvest VM availability traces from Azure for our experiments such that the peak of available Harvest VM resources match 80% that of resources consumed by Cosmos. Considering $2\times$ task replication, load-shifting approaches will be able to use 40% of effective workload resource-time on average opportunistically on transient resources during stable workload periods (non ramp up/down). This is similar to what we observe in real Cosmos cluster workloads. Fig. 5.6 shows the Cosmos workload resource usage, overlapped with the scaled Harvest VM resource availability traces from Azure, that we use for our experiments. For the period of traces collected, we did not observe correlation between Harvest resource availability and Spot market pricing of various instance types. This corroborates observations by Ben-Yehuda et al. [22], who noted that Spot market prices were not market-driven, but were rather generated via a dynamic hidden price reserve mechanism[2].

Azure Reserved VM instances. In addition to pay-as-you-go on-demand and Harvest VM [24]

[2]Note that pricing mechanisms may change at any time in the future.

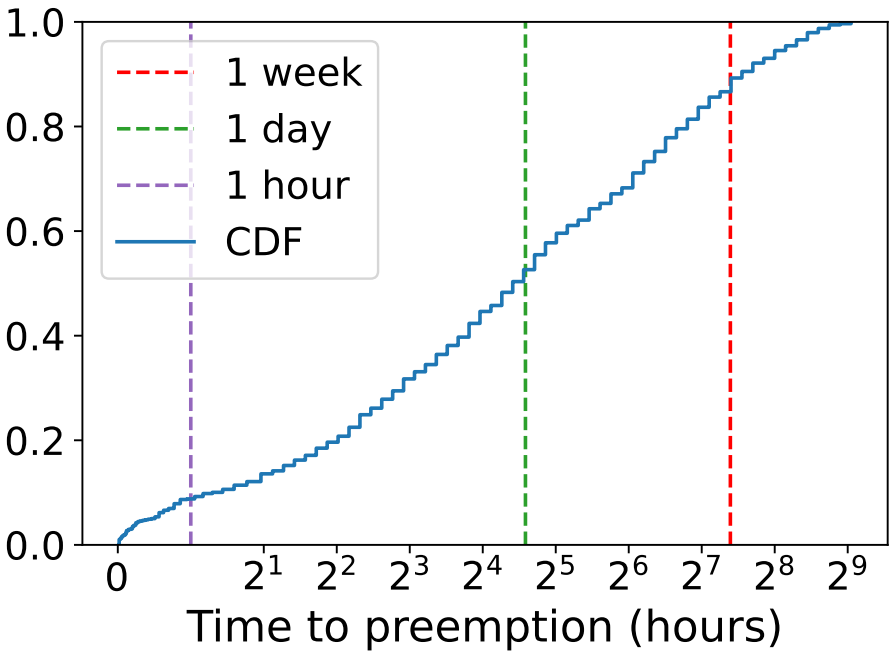


Figure 5.5: Harvest VM time to preemption. This figure shows the time to preemption of Harvest VMs. Nearly half of all Harvest VMs live for $>$ a day, and more than 10% of Harvest VMs live for $>$ a week.

instances, many public clouds today offer *Reserved VM instances* [2], where users can pay a discounted price for cloud VMs, provided that users enter a long-term contract with the cloud provider. The user is charged for the entirety of the contract regardless of rented VM usage. Our experiments use 3-year contracts, which provide a 61% [4] discount compared to acquiring reliable VMs on-demand.

5.5.2 Simulation setup

Simulation and preserving workload characteristics. The evaluation of the effectiveness of Talon to minimize reserved capacity with its load-shifting policies cannot realistically be attempted on research clusters without significantly changing the characteristics of Cosmos workloads by either sampling a small subset of jobs submitted, reducing the size of jobs run, or both. To preserve original workload jobs and their inter-job dependencies, we use and extend the trace-driven simulator used in the evaluation of Wing (§4.5.1) [6].

Simulator fidelity. Our simulator preserves fidelity of the original workload, along with inter-job dependency characteristics, by running real YARN resource management logic locally, only mocking away the running of real containers, the network communication layer, and calls to wall clock time. We find that in our simulation running a week of Cosmos workload, the difference between simulated and real-world job completion times differ within 1.3% of a job's run time at the 99^{th} percentile.

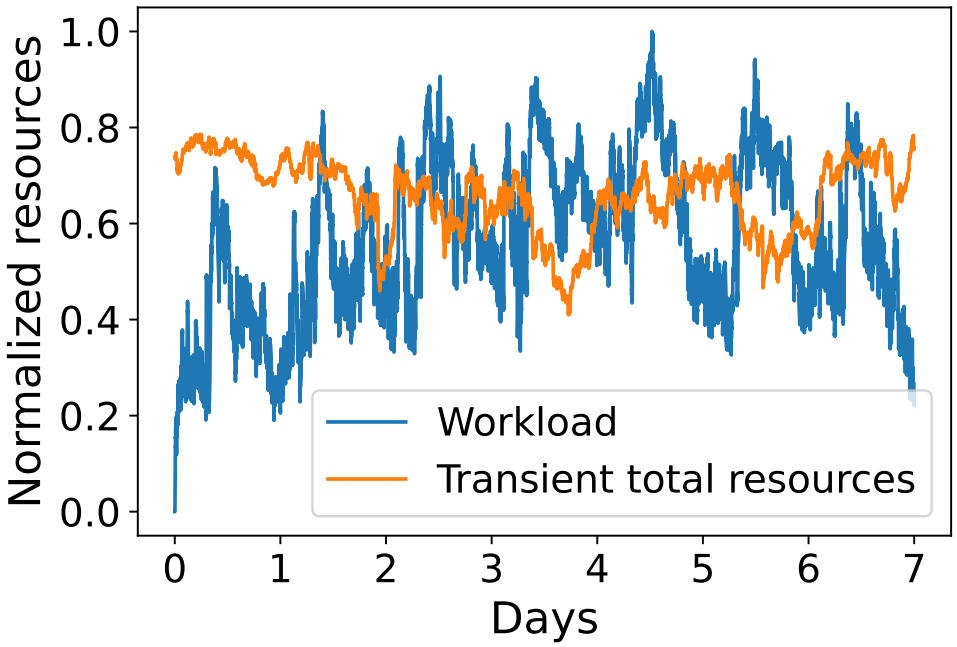


Figure 5.6: Workload traces. This figure shows the normalized resource usage of our workload, along side a scaled resource availability of Harvest VMs.

5.5.3 Compared load-shifting approaches

This section describes load-shifting approaches that we compare Talon against in our experiments.

TRADITIONAL: Reserved resources only. TRADITIONAL is the normal approach that most use, using enough reserved capacity to handle the overall peak workload. With TRADITIONAL, there is no load-shifting, and all submitted jobs are admitted immediately and run on reserved resources. Workloads scheduled with TRADITIONAL incur no deadline violations, but also requires reserved capacity that matches the workload peak.

GHDP: Adapting GreenHadoop. GreenHadoop [56] is a state-of-the-art green-energy-aware scheduler that looks to match batch-analytics workload to exploit available *green energy*. Similar to transient resources, the availability of green energy fluctuates. Green energy can often also be more cost-effective than its alternative in *brown energy*. We therefore find that, with adaptations, deadline and green energy aware schedulers can serve as appropriate comparisons for Talon.

GreenHadoop, at the highest level, performs its scheduling in two steps: (1) it selects machines to turn on/off based on estimated green energy availability, machine-local job data availability, and estimated energy demands of running and waiting jobs, and (2) it runs waiting jobs based on green energy availability in ascending *latest job start time* order to run on active machines. Latest job start time here is estimated based on job deadlines and predicted job run times. Jobs that will violate latest job start time are immediately admitted, potentially running on brown energy if necessary.

GHDP is our compared implementation of GreenHadoop, adapted to work with reserved and transient resources, as opposed to brown and green energy. We make the following adaptations to GreenHadoop: (a) We substitute "energy" with "machine resources" (i.e., virtual cores and

memory), such that green energy corresponds to transient resources, and brown energy corresponds to reserved resources. To account for GreenHadoop's estimated energy availability, we use the "current" transient resource availability, updated every scheduling interval (5 seconds), as an estimation for transient resource availability in the next interval. Our estimation yields a prediction error of 7.3% on average predicting an hour ahead, whereas GreenHadoop's green energy prediction yields an error of 12.6% on average [56]. (b) We loosen GreenHadoop restrictions by not considering machine-local data availability, as a job's task data is localized from a distributed file system only upon job start. (c) We provide GreenHadoop with estimates of latest job start times derived by the Wing pipeline (§5.3.1).

GHDP-R: GHDP with task replication. We also implement GHDP-R as a variation of GHDP with job task replication. GHDP-R works the same as GHDP, but tasks run on transient resources run with a replication factor of 2 to improve reliability and account for resource preemption.

GreenHadoop with Talon-guided job advancement. Finally, to see how GreenHadoop performs with even more opportunities to load-shift jobs, we provide our implementations of GreenHadoop with load-shifting information and slack provided by Talon's Analyzer (§5.3), allowing it to advance recurring job instances. We implement GHDP-R+TAdv and GHDP-R+TAdv as variations of GHDP and GHDP-R with Talon-guided job advancement, respectively.

Actual scheduling: Effectively using reserved resources. Like Talon, GreenHadoop tries to schedule aggressively on transient resources because transient resources ("green energy") cost less, which yields reserved capacity savings during provisioning phase, but will incur more cost during actual scheduling, as reserved resources are already paid-for. We therefore also enhance variants of GreenHadoop with actual scheduling modes similar to that described in §5.4.3 that fully use reserved resources when available. In our evaluations, approaches' reserved capacities are limited based on that determined in its provisioning phase, while deadline violations are determined based on actual scheduling results (§5.2.3).

5.5.4 Evaluation metrics and cost models

The goal of Talon is to minimize the peak workload run on reserved resources while minimizing job deadline violations. We consider three cost models (CMs) to evaluate the effect of Talon's reserved workload peak minimization on overall cost. The amount of job deadline violations incurred is an evaluation metric for all CMs.

CM1: Reserved peak workload minimization. *CM1* takes a straightforward look at reserved peak workload minimization. In CM1, we consider the scenario in which reserved capacity is much more expensive and requires much more commitment compared to alternatives, such that minimizing reserved resource capacity is of primary concern to the user. Here, reserved capacity can be interpreted as on-premise locally managed machines (e.g., Cosmos today), guaranteed workload capacity in a shared cluster (e.g., hierarchical queues in YARN [7]), or higher cost reserved VM instances in the cloud. We consider the amount of reserved capacity committed as the target for "cost reduction."

CM2: Reserved contracts in public cloud. *CM2* considers a public cloud scenario, where reserved capacity represents VMs in the cloud rented as reserved VM instances [2] (§5.5.1) under long, three-year contracts. The cost to minimize here is the monetary cost required to complete the workload.

CM3: Pay-as-you-go contracts in public cloud. *CM3* similarly considers minimizing monetary cost in a public cloud scenario, but in CM3, reserved resources are rented under pay-as-you-go contracts. i.e., in CM3, resources, both reserved and transient, are rented only for the period of time during which they are being utilized by job tasks, such that Talon's peak reduction is not as important as in CM1 and CM2.

5.6 Experimental results

This section describes our experimental results evaluating the efficacy of each compared load-shifting approach, using a week's worth of Cosmos workload, on the three cost models described in §5.5.4.

Takeaway. Talon effectively minimizes reserved resource peak without sacrificing job SLO attainment, Talon requires only 62% of reserved capacity relative to TRADITIONAL, and only incurs deadline violations on 0.004% of deadline-sensitive jobs in our workload (CM1). Talon is also adept at lowering the cost of executing jobs when only using resources rented from the public cloud, reducing overall cost by 31% compared to running the workload exclusively on reserved VMs (CM2). Finally, Talon reduces cost by 33% when scheduling entirely using VMs under pay-as-you-go contracts, where its judicious use of transient resources allows it to incur minimal number of deadline violations, violating the deadlines of only 0.005% of deadline-sensitive jobs in our workload, incurring the least job deadline violations out of all evaluated load-shifting approaches (CM3).

5.6.1 Talon vs state-of-the-art

This section compares the performance of Talon under each cost model (§5.5.4) against TRADITIONAL, and variations of GHDP (§5.5.3), on our week-long Cosmos workload.

CM1: Reserved peak workload minimization

First, we discuss how well Talon can minimize peak workload run on reserved resources (CM1). Here, we focus on how well each approach can minimize peak workload run on reserved resources and deadline violations incurred. Fig. 5.7 shows the effectiveness of each approach under CM1 relative to TRADITIONAL.

Comparison against TRADITIONAL. TRADITIONAL, being the normal approach used by most that schedules everything on reserved machines, is able to complete all jobs without violating any deadlines, but also uses the most reserved resources at its workload peak. Talon's peak reserved resource usage is 62% that of TRADITIONAL, while only violating 0.004% of job deadlines.

Comparison against GHDP. We find that GHDP is adept at minimizing peak reserved resource usage, attaining a peak of 50% relative to that of TRADITIONAL, compared to Talon at 62%. This is because GHDP runs without any task replication, and thus can run significant portions of its workload on transient resources. However, this comes with a significant caveat, as GHDP violates the deadlines of 0.59% of deadline-sensitive jobs. These deadline violations were mainly caused

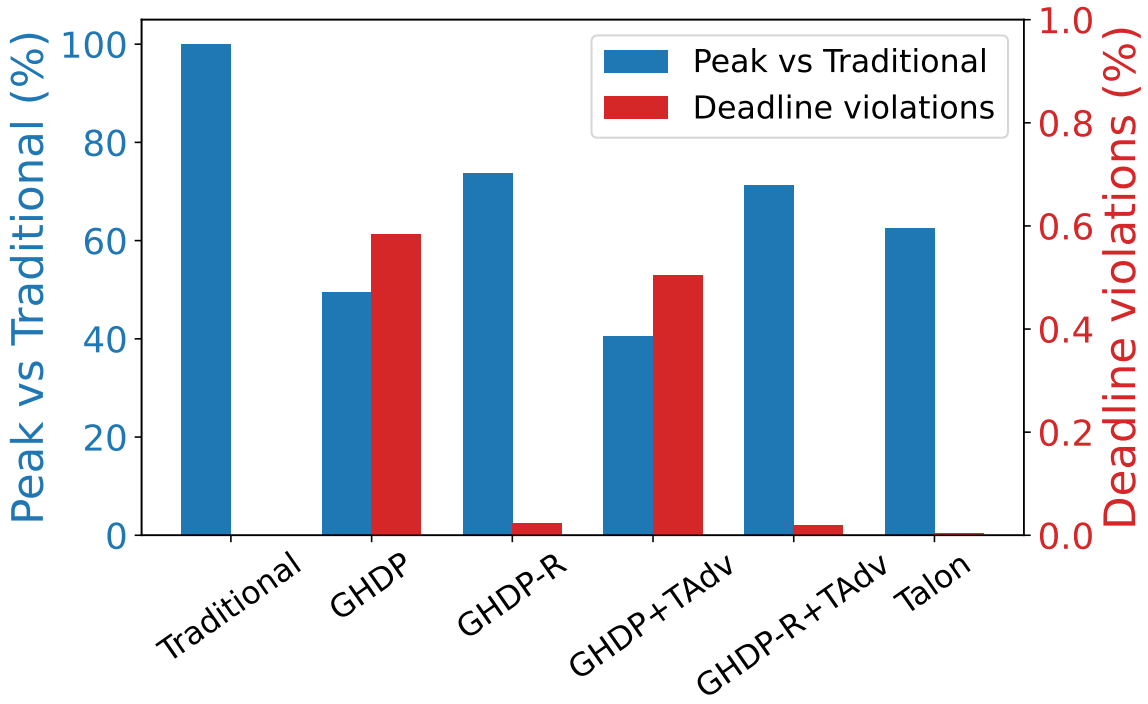


Figure 5.7: Comparison against state-of-the-art (CM1). This figure shows the performance of Talon against compared scheduling policies (§5.5.3). Blue bars represent reserved resource peak relative to TRADITIONAL (y-axis to the left) and red bars represent job deadline violation rate (y-axis to the right). Lower is better for both types of bars. TRADITIONAL's peak is 100% relative to itself and achieves 0 deadline violations. GHDP and GHDP-R+TAdv attain lower peaks vs Talon, but also incurs more job deadline violations. Task replication (GHDP-R and GHDP-R+TAdv) on transient resources significantly reduces deadline violations, but incurs higher reserved resource peak. Talon balances reserved resource peak minimization with SLO attainment.

by task preemptions and retries, and are worsened with inter-dependent jobs, as late upstream jobs can cause a chain of downstream jobs to violate their deadlines.

Comparison against GHDP-R. GHDP-R significantly reduces the number of GHDP job deadline violations by replicating tasks run on transient resources, yielding a job deadline violation rate of 0.02%. However, this comes with the caveat that GHDP's ability to minimize peak reserved workload is also reduced. GHDP-R attains a reserved resource peak of 73% relative to TRADITIONAL.

Comparison against GHDP-R+TAdv and GHDP-R+TAdv. We provide GHDP and GHDP-R with Talon's job advancement guidance to see how much benefit additional load-shifting flexibility brings. GHDP-R+TAdv (Talon advancement guided GHDP) reduces reserved resource usage peak compared to GHDP, attaining a peak of 41% relative to that of TRADITIONAL. While improved due to job advancement opportunities, GHDP-R+TAdv still violates the deadlines of 0.5% of deadline-sensitive jobs. Task preemptions and retries are still the main cause of deadline violations in GHDP-R+TAdv, as GHDP-R+TAdv runs without task replication.

On the other hand, we do not see as much improvement when providing GHDP-R with Talon's job advancement guidance in GHDP-R+TAdv. GHDP-R+TAdv attains a reserved resource usage

peak of 71% relative to TRADITIONAL, a 2% reduction from GHDP-R, and still yields a low deadline violation rate of 0.018%.

Examining why providing Talon's job advancement guidance does not yield much benefit to GHDP-R+TAdv in reserved peak reduction in its *provisioning planning phase*, we found that while compared to Talon, GHDP-R+TAdv's job admission and placement policies are much more aggressive in admitting advanced jobs, it often cannot fully take advantage of it. Indeed, GHDP-R+TAdv's aggressive placement and admission policies tries to take advantage of any immediately available transient resource, even when reserved workload is low. This in turn leads to high transient resource utilization by non-advanced jobs, which unintentionally reduces the amount of job resource-time that GHDP-R+TAdv can load-shift off of workload peaks. GHDP-R+TAdv only load-shifts 7% of all job resource-time by more than 15 minutes, compared to Talon at 35%, in the provision planning phase.

Both GHDP-R+TAdv and Talon use available transient resources to determine if jobs can be advanced in their provision planning phases. Talon, with its judicious use of transient resources such as by purposefully placing some ad-hoc jobs on reserved resources and by taking advantage of running advanced jobs with long deadline-slacks on transient resources without task replication, leaves resources on the table for admitting more advanced jobs.

CM2: Reserved contracts in public cloud

Here, we discuss how Talon reduces monetary cost running entirely on resources rented from a public cloud (e.g., Azure), using Reserved VM instances [2] as long-term reserved capacity (CM2). Our experiments use 3-year Reserved VM (costing 39% vs on-demand) and Harvest VM (costing 16% vs on-demand) instances. Costs were computed using average monthly costs across all VM types, as reported by Azure [4].

Fig. 5.8 shows how TRADITIONAL, GHDP, GHDP-R, and Talon perform in CM2. What we observe in CM2 is consistent with what we observe in CM1. We find that Talon is able to complete the workload with lower cost compared to GHDP-R as Talon's capacity planning was tighter, allowing effective usage of reserved resources: Talon costs 69% that of TRADITIONAL, out-performing GHDP-R, which costs 76% that of TRADITIONAL. While GHDP incurs the lowest cost, costing only 55% of TRADITIONAL, it incurs more deadline violations compared to strategies that utilize task replicas such as Talon and GHDP-R.

Note that in both CM1 and CM2 we assume right-sized reserved capacity planning from the provisioning phase, and that allocating insufficient reserved capacity can lead to more job deadline violations, as resource requests may be queued. We study the sensitivity of reserved capacity planning on Talon in the actual scheduling phase in §5.6.3.

CM3: Pay-as-you-go contracts in public cloud

Here, we discuss how Talon performs in monetary cost when exclusively using resources under pay-as-you-go contracts. In CM3, we use the same load-shifting approach as we do in Talon's provisioning phase and do not try to use as many reliable resources as we do in in actual scheduling, as these resources are now pay-as-you-go and are charged significantly more compared to reserved and transient resources.

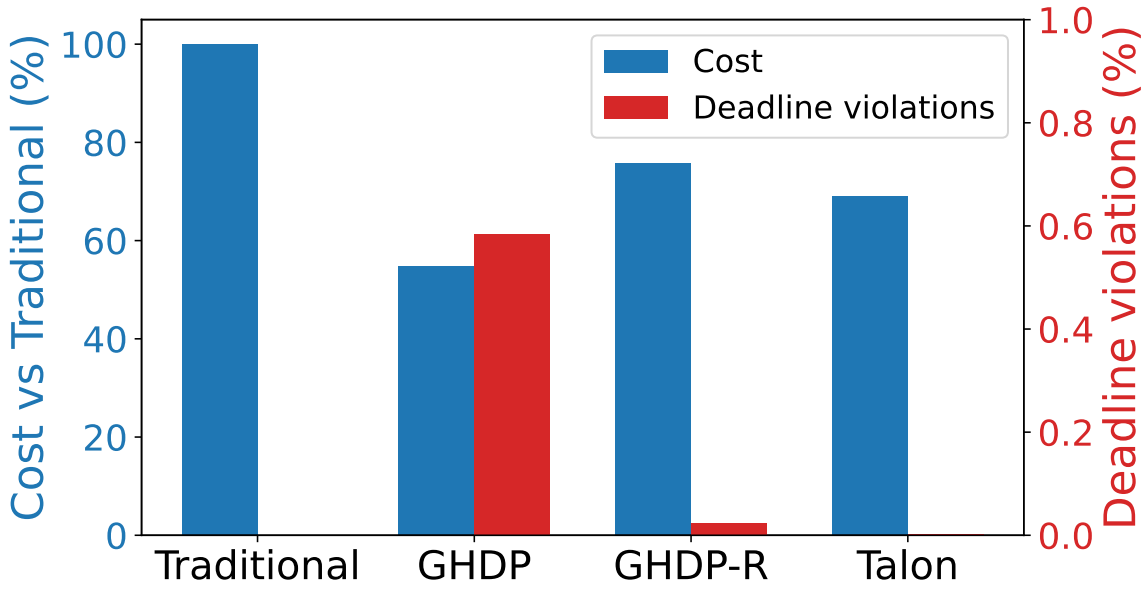


Figure 5.8: Costs of using reserved contracts in Azure (CM2). This figure shows the costs and job deadline violation rates of compared policies operating using Reserved VMs and Harvest VMs in Azure as reserved and transient resources, respectively. We find that cost results are consistent to that observed in CM1: GHDP incurs the least cost but the most deadline violations, GHDP-R reduces the deadline violations of GHDP at more cost, while Talon strikes a balance between both, costing 31% less than TRADITIONAL while maintaining a low number of deadline violations.

Fig. 5.9 shows each compared approach's performance in CM3, normalized to the cost of TRADITIONAL. While Talon in CM3 still reduces cost relative to TRADITIONAL, costing 67% that of TRADITIONAL, we found that Talon costs more compared to other CM3 approaches. GHDP only costs 27% and GHDP-R only costs 63% that of TRADITIONAL. This is because compared to GHDP and GHDP-R, Talon uses more reserved resource-time and less transient resource-time overall, as it tries to leave enough of a transient resource buffer to handle retry storms upon bulk resource preemptions. On the other hand, without a pre-determined amount of reliable resources that "soak-up" workload (e.g., in CM1 and CM2) GHDP and GHDP-R use transient resources much more aggressively, scheduling tasks on transient resources whenever they become available. We observe that in our execution traces of CM3, the GHDP and GHDP-R often exhaust all available transient resources, such that excessive queueing of unfulfilled retry requests happen during bulk preemptions. Indeed, this aggressive usage of transient resources cause GHDP and GHDP-R to be significantly impacted by bulk transient resource preemptions: GHDP and GHDP-R incur a 0.96% and 0.61% deadline violation rate, respectively. Talon robustly handles bulk preemptions, incurring only a 0.005% deadline violation rate, while only costing 4% of TRADITIONAL's cost more compared to GHDP-R.

Applying Talon's policy to green energy. We find that Talon does not perform as well as GHDP-R in terms of cost-savings in CM3, and we expect these results to extend to other cost models that similarly charge in a pay-as-you-go fashion for reliable resources. An example of such a cost model would be one where machines powered by brown energy cost more to operate

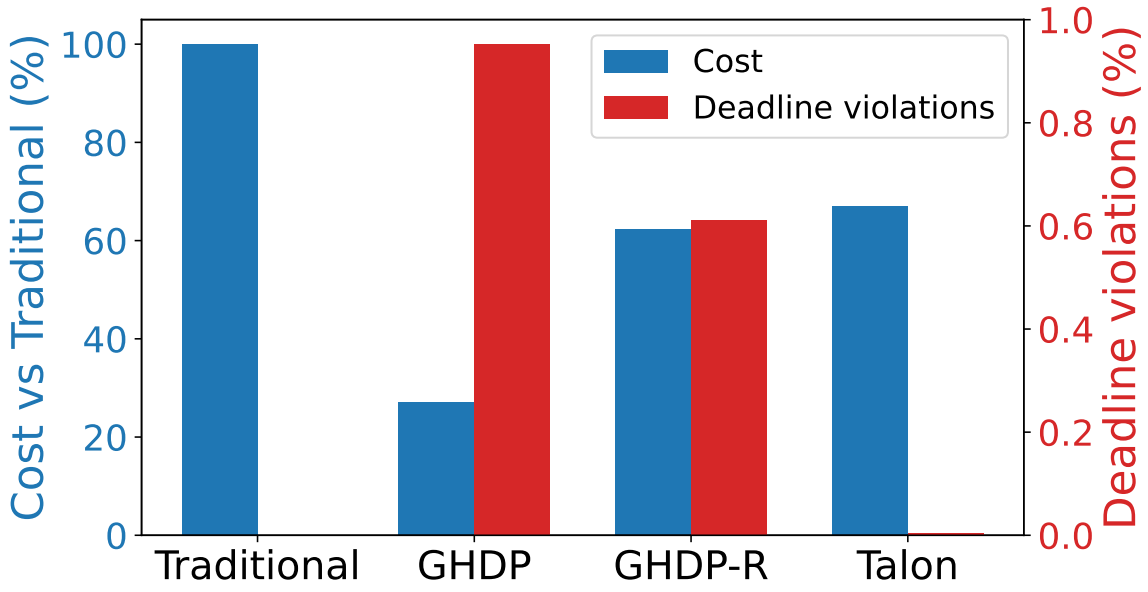


Figure 5.9: Costs of using pay-as-you-go contracts in Azure (CM3). This figure shows the costs and job deadline violation rates of compared policies using On-Demand reliable VMs and Harvest VMs in Azure as "reserved" and transient resources, respectively. While Talon under-performs GHDP-R other policies in cost under CM3, it robustly handles bulk transient resource preemptions, allowing it to achieve lower rates of job deadline violations.

compared to machines powered by intermittently available green (e.g., solar/wind) energy. Indeed, Talon's main cost reduction target is to reduce peak reserved resource usage: under pay-as-you-go cost models for reliable resources, when Talon opts to use more expensive reliable resources to schedule tasks to reduce deadline violations, peak workload run on reliable resources may not be raised, but extra costs will necessarily be incurred. Under these cost models, we expect approaches that fully utilize transient resources to incur less cost. When considering brown and green energy, policies like GHDP may even further benefit in terms of incurring less deadline violations, as transient resource "preemptions" in such a cost model can be controlled to some degree, such as via effective power distribution.

5.6.2 Attribution of benefits

This section attributes the benefits of features of Talon by following a series of design progressions, evaluating each compared approach's effectiveness in minimizing reserved resource commitment and job deadline violation rates (shown in Fig. 5.10).

Progression: REP, a simple replica approach. REP represents the *replica approach* to scheduling jobs, in which jobs are submitted on their original schedule. REP also also tries to run jobs on transient resources with replicas when transient resources are available: When jobs start, REP queries the resource manager for the availability of transient resources. If there is enough resources to run all of the job's tasks with replicas, the job requests transient resources; otherwise, the job requests reserved resources. REP achieves a reserved resource peak 74% that of TRADITIONAL, and incurs 0.02% of job deadline violations.

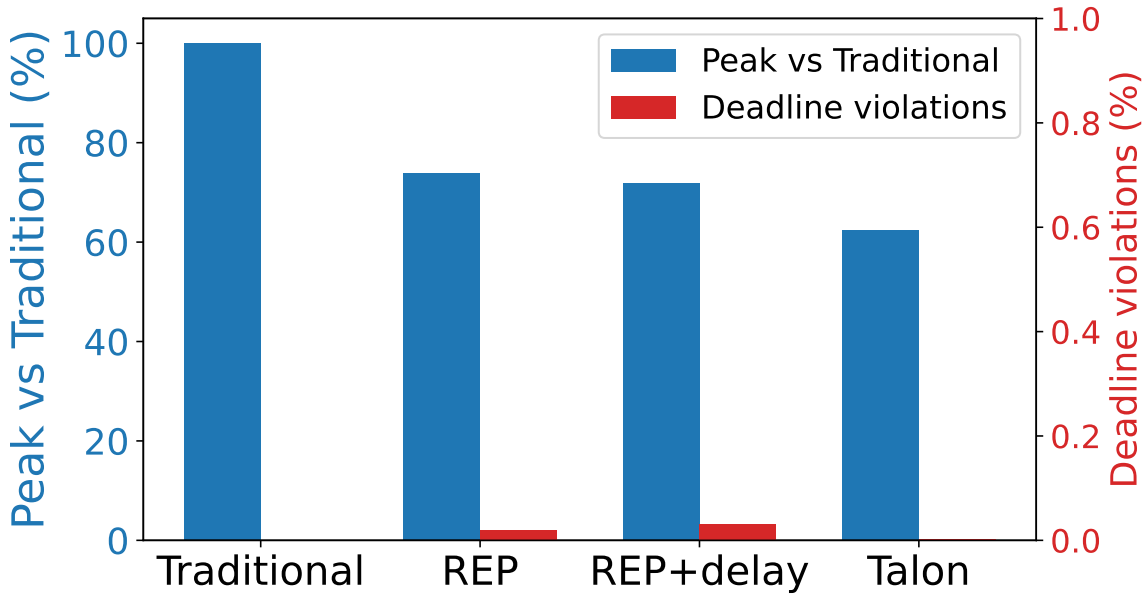


Figure 5.10: Progression results. This figure shows the progression of features we evaluate that lead us to Talon. REP, a load-shifting approach that runs jobs on transient resources with replicas, incurs a higher reserved resource usage peak. Enhancing REP with the ability to delay jobs (REP+delay) does not help much to reduce reserved resource capacity, as opportunities to delay jobs are limited (§5.3.4). Talon's exploitation of job advancement opportunities reduces reserved resource commitment to 62% that of TRADITIONAL.

Progression: REP+delay, REP with job delays. While REP performs reasonably well just by using transient resources to reduce the amount reserved resource committed, we wanted to see if the conventional method of load-shifting by delaying jobs based on deadline and job run time information can help further reduce reserved resource peaks. We thus introduce REP+delay, a load-shifting policy that, in addition to running jobs on transient resources with replicas, prioritizes and delays jobs based on their estimated slack when there are not enough transient resources to run the whole job (similar to GHDP-R). We provide REP+delay with perfect job run time and deadline information, but we find that delaying jobs did not help much, reducing reserved resources only by 2% while incurring slightly more job deadline violations. This confirms our earlier hypothesis that job delays do not provide much opportunity (§5.3.4). Indeed, GHDP-R, which uses a similar approach to REP+delay, performs similarly when only considering job delays.

Progression: Talon. In analyzing inter-job dependencies, we find that many future-arriving recurring jobs are predictable based on input availability, and that there exists significant gap in many jobs between when their inputs are available and when the jobs are submitted. We develop Talon to exploit this opportunity, advancing jobs to when transient resources are abundant to reduce reliance on reserved resources. Talon achieves a reserved resource usage peak at 62% that of TRADITIONAL, and achieves a low amount of job deadline violations. As an added bonus, since Talon is more careful in its transient resource usage as to not trigger retry storms during bulk preemptions, it also incurs significantly lower job deadline violation rates (0.005% job deadlines

violated) compared to other load-shifting policies in CM3.

5.6.3 Sensitivity analyses

This section describes sensitivity analyses of the various tunable parameters in Talon's policies.

Sensitivity of tunable knobs on CM1

We perform sensitivity analyses on the tunable knobs available in Talon's policies. Fig. 5.11 shows the effects of the tunable resource slack (t) and run time slack (p) parameters on reserved workload peak reduction. Our discussion in this section will focus on the provision planning phase of Talon, as deadline violation rates are less than 0.02% across all CM1 sensitivity experiments. For all values of t and p, Talon out-performs its variants (REP-TP and REP-TP+TAdv) and GHDP-R in reserved workload peak reduction, while incurring little deadline violations. Our primary results in §5.6.1 and §5.6.2 set $t = 2$ and $p = 1$.

Varying resource slack (t). The resource slack parameter (t) affects both admission and placement policies. The larger the resource slack, the more advanced jobs admitted, and the more likely jobs are run with replicas on transient resources.

On our evaluated Cosmos workload, keeping p constant, across evaluated values of p, using $t = 2$ and $t = 3$ minimizes reserved resource usage peaks. With $t = 1$, we find that Talon tends to be too conservative, preferring to run jobs on reserved resources and does not take enough advantage of load-shifting. Indeed, $t = 1$ often incurs the least deadline violations, keeping p constant. On the other hand, with $t = 4$, Talon can be too aggressive with advancing jobs and running them on transient resources, leading to jobs being advanced to periods where workload is generally busier.

Varying placement run time slack (p). The run time slack p affects whether an advanced job determined to run on transient resources should run with replicas. The larger p is, the more likely it is that the job should run with replicas.

In our evaluated workload, we find that for fixed t's, the larger p is, the higher the reserved resource usage peak. This is because for advanced jobs, using a larger p value introduces more transient resource competition, leading to more jobs that arrive later being run on reserved resources. Values of smaller p, however, also introduces more deadline violations, as more jobs are set to run without replicas.

We find that setting $p = 1$ strikes a good balance between taking risks to minimize reserved resource peak usage and maintaining low numbers of job deadline violations. Using $p = 1$ allows jobs advanced by more than its run time to run without task replicas, such that even if its tasks were preempted, it would have ample time to finish its execution. By contrast, if p is set to 0.5, jobs would have a higher chance of violating their deadlines should its tasks get preempted.

Varying admission run time slack (m). Keeping resource slack (t) and placement run time slack (p) constant, varying admission run time slack (m) yielded up to a 2% difference in reserved resource peak usage, for evaluated values of $m > p$ (m and $p \in \{0.5, 1, 2, 3\}$). m should be set to values higher than p, as setting it to values less than or equal to p will not allow Talon's placement policy to opportunistically schedule advanced jobs with long deadline slack without task replicas, thereby increasing reserved resource peak usage (up to a 6% difference).

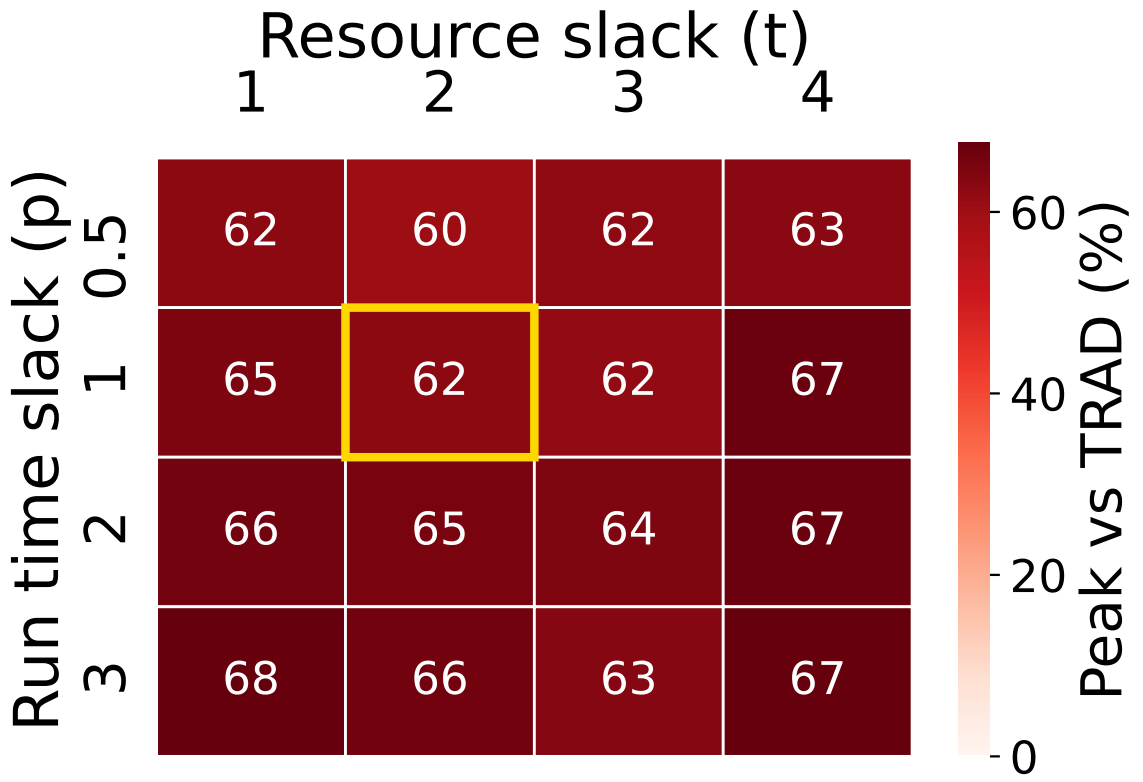


Figure 5.11: Sensitivity of results to Talon parameters. This figure explores how Talon's tunable parameters t and p affect reserved resource usage peak relative to that of TRADITIONAL. Lighter colors (lower value) are better. The gold-outlined box ($t = 2$ and $p = 1$) refer to our primary experiment settings. Keeping p constant, setting $t = 2$ and $t = 3$ yield the highest reliable resource peak reduction. Keeping t constant, a lower p yields lower peaks.

Sensitivity of reserved capacity on actual scheduling

Careful capacity planning is necessary to optimize for actual scheduling. If not enough reserved capacity is planned, load-shifting approaches can incur extra job deadline violations, as job resource requests will be queued due to insufficient resources. We study the sensitivity of capacity planning on job deadline violations with Talon. The right-sized reserved capacity for Talon is at 62% of TRADITIONAL (§5.6.1). If we allocate only 95% of the right-sized capacity, we find that Talon's job deadline violation rate increases from 0.004% to 0.01%. If we allocate 90% of the right-sized capacity, Talon's job deadline violation rate further increases to 0.03%. If we further allocate less capacity at 80%, Talon's job deadline violation rate increases to 0.17%.

Our results find Talon to be robust to incorrect capacity planning, largely due to its ability to (1) reduce workload peaks by advancing jobs, (2) advance jobs to times when resource contention is scarce, and (3) reliably run advanced jobs without replicas on transient resources, reducing contention of both reserved and transient resources for other jobs.

5.7 Related work

Cluster workload analysis. Much work has been done to study cluster workloads of various types [25, 37, 64, 106]. One particular area of study within cluster workload analyses is in job run time prediction [45, 67]. An idea often used in literature is to use historical similar/recurring job runs to predict future job run times [38, 104, 117, 123, 124]. Our work leverages history-based recurring job run time prediction techniques to provide Talon with job run time predictions.

Different from prior work, our work on Talon considers jobs which consider cluster jobs independent of one another. Our work uses the analyses from Wing (Chapter 4) to uncover inter-job dependencies. From there, Talon identifies and exploit recurring jobs that can be load-shifted due to the slack between when jobs' inputs are available and when jobs are actually submitted, and sheds light on the significant opportunity available in job load-shiftability.

Workflow managers [73, 96, 97] for batch analytics jobs have been widely studied and deployed throughout industry. However, workflow managers today depend on users to manually specify inter-job dependencies between jobs in a workflow, while Talon infers inter-job dependencies from historical job and data provenance logs. Furthermore, Talon allows users and cluster resource operators to save on cost by selectively running load-shiftable workloads off-peak and on cost-effective resources, while working in-tandem with the cluster resource manager to ensure that jobs do not violate their deadlines.

Cluster scheduling is an area of research that has enjoyed a long history of study, and work has been done to support general batch analytics, low-latency, interactive jobs, streaming jobs, and mixed workloads thereof. [30, 52, 53, 54, 55, 61, 78, 100, 102, 109, 124, 126] However, different from most prior work, which assume that jobs run independently of one another, Talon exploits the inter-job dependencies between submitted jobs for effective load-shifting.

Load-shifting and green-energy-aware scheduling. A particularly relevant area of cluster scheduling lies in green-energy-aware job scheduling. These schedulers load-shift jobs in order to match job energy consumption with the availability of green energy (e.g., solar or wind energy). However, we find that many of these schedulers require accurate user-provided information on job run times and/or job load-shiftability [23, 56, 57, 83, 84, 86]. We have adapted GreenHadoop [56] as a baseline for comparison against Talon (§5.5.3), and shown that Talon is more effective at both reducing peak reserved resources used and attaining lower numbers of job deadline violations.

Transient resources: Offerings and usage. Transient resources such as Spot VMs [3] and Harvest VMs [24, 129] are often offered at a discount compared to regular VMs by cloud service providers to increase data center utilization. Many prior work have been proposed to take advantage of these lower-cost offerings to reduce the overall cost of executing their respective workloads, including in areas of batch analytics [116, 132] (e.g., Chapter 3), machine learning model training [68], and long-running services (e.g., Chapter 2). We consider our work in Talon complementary to prior work in this area, as Talon mainly focuses on using load-shifting and transient resources to reduce peak reserved resource usage.

Systems using data provenance. There has also been prior work that collects [40, 95] and exploits provenance to improve batch data analytics computation [63, 76]. Our work is similarly exploits job data provenance, but focuses on using the information to load-shift jobs.

Inter-job dependency aware frameworks. Our work extends upon a series of prior work in

inter-job dependency analyses. Owl (§4.7) is a tool that helps users visualize inter-job dependency characteristics of their jobs, while Guider [95] and Wing (Chapter 4)) propose and realize the idea of scheduling to maximize job utility attainment based on inter-job dependency awareness, respectively. Talon uses historical inter-job dependencies to derive job load-shiftability, and exploits transient resources to reliably load-shift jobs to reduce cluster peak resource demand while minimizing job deadline violations.

5.8 Summary

Talon effectively lowers long-term reserved resource capacity needed to run workloads by exploiting job load-shifting via inter-job dependencies. Considering load-shifting in-tandem with intermittently available transient resources allows Talon to more reliably execute jobs with low impact on cluster resource usage and job deadline violations. Our experiments find that Talon can reduce reserved resource commitment by up to 38% compared to the traditional approach of reserving resources based on workload peak, while experiencing only minimal job deadline violations.

Chapter 6

Conclusion and future directions

6.1 Conclusion

This dissertation demonstrates that *value-realized in shared data environments can be improved both by cost- and heterogeneity-aware applications from users and by value- and dependency-aware resource management systems from cluster operators* by presenting four case study systems: two in user applications, and two in dependency-aware resource management systems.

In user applications, we presented Tributary, an elastic control system that allows users to realize more value in long-running elastic cloud services with latency SLO targets by enabling these services to reliably use low-cost, transient cloud resources. Tributary does so by creating models of transient resource preemption likelihood and exploiting low correlation between different transient resource pools, allocating resources from diverse pools to mitigate preemption risk and to satisfy application SLO requirements. Our experiments show that Tributary reduces cost for achieving a given SLO by 81-86% compared to scaling on non-preemptible resources, and by 47-62% compared to alternative approaches of the same scaling with transient resources.

Tackling another class of user applications in batch data analytics, we presented Stratus, a cost-aware virtual cluster scheduler that allows users to attain more value by reducing cost. Different from prior schedulers, Stratus focuses on the fact that cloud resources are continuously charged-for while allocated. It therefore tries to minimize cost by attempting to fully utilize resources while they are allocated, using job runtime predictions as guidance. Our simulation experiments on Google and TwoSigma cluster traces show that Stratus can reduce cost by 17-44% compared to state-of-the-art approaches in virtual cluster scheduling.

In dependency-aware resource management systems, we first presented Wing. In Wing, we found that inter-job dependencies widely pervade today's shared data clusters, yet are often not considered in resource management. The Wing dependency profiler analyzes job and data provenance logs to expose inter-job dependencies, characterizes them, and provides guidance to cluster schedulers to effectively prioritize the most impactful recurring jobs. In simulations driven by logs from a Cosmos cluster, we find that a traditional YARN scheduler that uses Wing-provided valuations in place of user-specified priorities extracts more value from a heavily-constrained cluster.

We further delved into opportunities to reduce cluster operating costs by presenting Talon.

Talon exploits job load-shifting with inter-job dependencies in-tandem with intermittently available transient resources to minimize long-term cluster resource commitment in shared cluster capacity planning. Our experiments find that Talon can reduce reserved resource commitment by up to 38% compared to the traditional approach of reserving resources based on workload peak, while experiencing only minimal job deadline violations.

6.2 Future directions

This section describes several directions of future research that extends upon work presented in this dissertation.

6.2.1 Cost-efficient resource acquisition mixed job types

This dissertation presented Tributary, which optimizes for cost and reliability of long-lived, latency sensitive, and elastic cloud services. It also introduced Stratus, which specializes in lowering the cost of shorter-running batch analytics jobs. An interesting future research direction is to design a scheduler/resource acquisition scheme that is capable of lowering cost across multiple different application types.

6.2.2 Dynamic dependency-aware value scheduling

The value scheduling policy proposed by Wing is a simple one that translates recurring aggregate downstream value impact in to a static priority assignment. While we found that this is effective and sufficient in the use-case of Cosmos due to the Zipfian nature of our job value distribution, it may not work as well for other shared computing environments or for other metrics of job value/utility (§6.2.3). In these cases, we may need a value scheduler that is aware of dynamic dependency changes, updating the value of jobs as their downstream dependencies come-and-go. A more sophisticated and effective version of such a scheduler could plan ahead and reserve resources for high-value downstream jobs that have high probabilities of arriving in the future.

6.2.3 Better metrics for job value/utility

The search for a good proxy-metric for job value has been a long-standing topic of research in scheduling. While Wing in this dissertation proposes to use job output download as a proxy-metric for the value jobs, which may work in certain contexts (e.g., output dataset popularity ranking), it is still an imperfect proxy-metric. For example, output downloads by certain users may outweigh the output downloads of other users (e.g., the output download by the CEO of Microsoft will likely be more financially impactful than the output download of a researcher running ad-hoc experiments). It is also difficult to place or measure value on the impact of jobs that are monitored but whose outputs are never downloaded (e.g., cluster canary jobs), ad-hoc jobs that may lead to important financial decisions, and the output of jobs that is only downloaded once but cached and reused elsewhere. This dissertation therefore leaves better metrics for job value as an important subject for future research.

6.2.4 Better predictions of time-to-output usage

While Wing can predict the arrival of a downstream recurring job fairly reliably (§4.2), in Talon, we found that it is difficult to predict when the output of a recurring job will first be consumed, either by a user or by downstream jobs (§5.3.4). Indeed, even predicting the minimum historical time to output usage, our predictor still over-predicts for 2% of recurring jobs, which is why in Talon we opted to not actively delay recurring jobs. Historical time-to-output-downloads may therefore not be sufficient here for accurate predictions, and additional job submission context and potentially input from users may be required. An accurate prediction of time-to-output usage can significantly increase the load-shiftability of jobs, while decreasing the number of output violations, for decisions made by a load-shifting scheduler.

6.2.5 Other cost-aware load-shifting applications

In Talon, we focused on reducing cost by reducing reserved resource commitment, and examined three cost models in total. Cost models CM1 and CM2 (§5.5.4) benefit from using Talon to reduce the peak reserved resource workload, but using Talon in CM3, which measures the cost of running a week-long workload from a Cosmos cluster entirely using pay-as-you-go resources, costs more compared to other load-shifting approaches. A direction for future research is to apply load-shifting principles discussed in Talon (i.e., with inter-job dependency information) to reduce cost in other cost-aware load-shifting applications. These applications include load-shifting to reduce costs by exploiting fluctuating prices in pay-as-you-go resources (e.g., Spot VMs) and load-shifting to better take advantage of environmentally friendly but intermittently available green energy.

Bibliography

[1] AWS Autoscale. https://aws.amazon.com/autoscaling/.

[2] Azure Reserved Virtual Machine Instances (2022). https://azure.microsoft.com/en-us/pricing/reserved-vm-instances/,.

[3] Azure Spot Virtual Machines (2022). https://azure.microsoft.com/en-us/services/virtual-machines/spot/,.

[4] Azure Linux Virtual Machine Pricing (2022). https://azure.microsoft.com/en-us/pricing/details/virtual-machines/,.

[5] Spot Bid Advisor. https://aws.amazon.com/ec2/spot/bid-advisor/.

[6] Add discrete event-based simulation to yarn scheduler simulator. https://issues.apache.org/jira/browse/YARN-1187,.

[7] Hadoop: Capacity Scheduler. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html,.

[8] Azure Virtual Machines. https://azure.microsoft.com/en-us/services/virtual-machines/, 2018.

[9] CRIU. http://criu.org/, 2018.

[10] AWS EC2. http://aws.amazon.com/ec2/, 2018.

[11] AWS EC2 Spot Instances. https://aws.amazon.com/ec2/spot/, 2018.

[12] Amazon ECS Container Instances. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_instances.html, 2018.

[13] Powering your Amazon ECS Clusters with Spot Fleet. https://aws.amazon.com/blogs/compute/powering-your-amazon-ecs-clusters-with-spot-fleet/, 2018.

[14] Amazon ECS Task Placement Strategies. https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-placement-strategies.html, 2018.

[15] AWS Fargate. https://aws.amazon.com/AWS/Fargate, 2018.

[16] Google Compute Engine. https://cloud.google.com/compute/, 2018.

[17] AWS EC2 Spot Fleet. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html, 2018.

[18] Tensorflow serving. https://tensorflow.github.io/serving, 2018.

[19] Apache Airflow. https://airflow.apache.org/, 2019.

[20] Luigi. https://github.com/spotify/luigi, 2019.

[21] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 16)*, OSDI'16, pages 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026899.

[22] Orna Agmon Ben-Yehuda, Muli Ben-Yehuda, Assaf Schuster, and Dan Tsafrir. Deconstructing amazon ec2 spot instance pricing. *ACM Transactions on Economics and Computation*, 1(3), sep 2013. ISSN 2167-8375. doi: 10.1145/2509413.2509416. URL https://doi.org/10.1145/2509413.2509416.

[23] Baris Aksanli, Jagannathan Venkatesh, Liuyi Zhang, and Tajana Rosing. Utilizing green energy prediction to schedule mixed batch and service jobs in data centers. In *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, HotPower '11. ACM, 2011. ISBN 9781450309813. doi: 10.1145/2039252.2039257. URL https://doi.org/10.1145/2039252.2039257.

[24] Pradeep Ambati, Iñigo 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*. OSDI '20. USENIX Association, USA, 2020. ISBN 978-1-939133-19-9.

[25] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *Proceedings of the 2018 USENIX Annual Technical Conference*, USENIX ATC '18. USENIX Association, 2018. URL https://www.usenix.org/conference/atc18/presentation/amvrosiadis.

[26] Anton Beloglazov and Rajkumar Buyya. Adaptive threshold-based approach for energy-efficient consolidation of virtual machines in cloud data centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science*, MGC '10. ACM, 2010. ISBN 978-1-4503-0453-5. doi: 10.1145/1890799.1890803. URL http://doi.acm.org/10.1145/1890799.1890803.

[27] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. *Concurrency and Computation : Practice and Experience*, 24(13), September 2012. ISSN 1532-0626. doi: 10.1002/cpe.1867. URL http://dx.doi.org/10.1002/cpe.1867.

[28] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. Datahub: Collaborative data science & dataset version management at scale. In *Proceedings of the 7th Biennial Conference on*

Innovative Data Systems Research, CIDR '15, January 2015.

[29] Luiz Fernando Bittencourt and Edmundo Roberto Mauro Madeira. HCOC: a cost optimization algorithm for workflow scheduling in hybrid clouds. *Journal of Internet Services and Applications*, 2(3), 2011. doi: 10.1007/s13174-011-0032-0. URL https://doi.org/10.1007/s13174-011-0032-0.

[30] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14. USENIX Association, 2014. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685071.

[31] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2), August 2008. ISSN 2150-8097. doi: 10.14778/1454159.1454166. URL http://dx.doi.org/10.14778/1454159.1454166.

[32] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study. In *Proceedings of the 25th International Symposium on Software Reliability Engineering*, ISSRE '14. IEEE Computer Society, Nov 2014. doi: 10.1109/ISSRE.2014.34.

[33] Brent N. Chun and David E. Culler. User-Centric Performance Analysis of Market-Based Cluster Batch Schedulers. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02. IEEE Computer Society, May 2002. ISBN 0769515827.

[34] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware Container Scheduling in the Public Cloud. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018. ISBN 978-1-4503-6011-1. doi: 10.1145/3267809.3267819. URL http://doi.acm.org/10.1145/3267809.3267819.

[35] Andrew Chung, Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Panagiotis Garefalakis, and Gregory R. Ganger. Peering Through the Dark: An Owl's View of Inter-job Dependencies and Jobs' Impact in Shared Clusters. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19. ACM, 2019. ISBN 978-1-4503-5643-5. doi: 10.1145/3299869.3320239. URL http://doi.acm.org/10.1145/3299869.3320239.

[36] Andrew Chung, Subru Krishnan, Konstantinos Karanasos, Carlo Curino, and Gregory R. Ganger. Unearthing inter-job dependencies for better cluster scheduling. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20. USENIX Association, November 2020. URL https://www.usenix.org/conference/osdi20/presentation/chung.

[37] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17. ACM, 2017. ISBN 978-1-4503-5085-3.

doi: 10.1145/3132747.3132772. URL http://doi.acm.org/10.1145/3132747.3132772.

[38] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You'Re Late Don'T Blame Us! In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '14. ACM, 2014. ISBN 978-1-4503-3252-1. doi: 10.1145/2670979.2670981. URL http://doi.acm.org/10.1145/2670979.2670981.

[39] Carlo Curino, Subru Krishnan, Konstantinos Karanasos, Sriram Rao, Giovanni M. Fumarola, Botong Huang, Kishore Chaliparambil, Arun Suresh, Young Chen, Solom Heddaya, Roni Burd, Sarvesh Sakalanaga, Chris Douglas, Bill Ramsey, and Raghu Ramakrishnan. Hydra: a federated resource manager for data-center scale analytics. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '19. USENIX Association, February 2019. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/curino.

[40] Sergio Manuel Serra da Cruz, Patricia M. Barros, Paulo Mascarello Bisch, Maria Luiza Machado Campos, and Marta Mattoso. Provenance Services for Distributed Workflows. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '08. IEEE Computer Society, 2008.

[41] Peter Danzig, Jeff Mogul, Vern Paxson, and Mike Schwartz. The internet traffic archive. *URL: http://ita.ee.lbl.gov/*, 2000.

[42] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, OSDI '04. USENIX Association, 2004.

[43] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. Hawk: Hybrid Datacenter Scheduling. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15. USENIX Association, 2015. ISBN 978-1-931971-225. URL http://dl.acm.org/citation.cfm?id=2813767.2813804.

[44] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. Kairos: Preemptive Data Center Scheduling Without Runtime Estimates. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018. ISBN 978-1-4503-6011-1. doi: 10.1145/3267809.3267838. URL http://doi.acm.org/10.1145/3267809.3267838.

[45] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14. ACM, 2014. ISBN 9781450323055. doi: 10.1145/2541940.2541941. URL https://doi.org/10.1145/2541940.2541941.

[46] Ruben Van den Bossche, Kurt Vanmechelen, and Jan Broeckhove. Cost-optimal scheduling in hybrid iaas clouds for deadline constrained workloads. In *IEEE International Conference on Cloud Computing*, CLOUD '10. IEEE Computer Society, 2010. doi: 10.1109/CLOUD.

2010.58. URL https://doi.org/10.1109/CLOUD.2010.58.

[47] Nosayba El-Sayed, Hongyu Zhu, and Bianca Schroeder. Learning from Failure Across Multiple Clusters: A Trace-Driven Approach to Understanding, Predicting, and Mitigating Job Terminations. In *Proceedings of the IEEE 37th International Conference on Distributed Computing Systems*, ICDCS '17. IEEE Computer Society, June 2017. doi: 10.1109/ICDCS.2017.317.

[48] Dror G Feitelson. *Workload modeling for computer systems performance evaluation*. Cambridge University Press, 2015.

[49] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12. ACM, 2012. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168847. URL http://doi.acm.org/10.1145/2168836.2168847.

[50] Guilherme Galante and Luis Carlos E. de Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4862-3. doi: 10.1109/UCC.2012.30. URL http://dx.doi.org/10.1109/UCC.2012.30.

[51] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Compututer Systems*, 30(4), November 2012. ISSN 0734-2071. doi: 10.1145/2382553.2382556. URL http://doi.acm.org/10.1145/2382553.2382556.

[52] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. Medea: Scheduling of Long Running Applications in Shared Production Clusters. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys '18. ACM, 2018. ISBN 978-1-4503-5584-1. doi: 10.1145/3190508.3190549. URL http://doi.acm.org/10.1145/3190508.3190549.

[53] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. Neptune: Scheduling Suspendable Tasks for Unified Stream/Batch Applications. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019. ISBN 9781450369732. doi: 10.1145/3357223.3362724. URL https://doi.org/10.1145/3357223.3362724.

[54] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11. USENIX Association, 2011. URL http://dl.acm.org/citation.cfm?id=1972457.1972490.

[55] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026886.

[56] Iñigo Goiri, Kien Le, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. GreenHadoop: Leveraging Green Energy in Data-processing Frameworks. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, 2012. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168843. URL http://doi.acm.org/10.1145/2168836.2168843.

[57] Iñigo Goiri, Md. E. Haque, Kien Le, Ryan Beauchea, Thu D. Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. Matching renewable energy supply and demand in green datacenters. *Ad Hoc Networks*, 25:520–534, 2015. doi: 10.1016/j.adhoc.2014.11.012. URL https://doi.org/10.1016/j.adhoc.2014.11.012.

[58] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management*, pages 9–16. IEEE, 2010. doi: 10.1109/CNSM.2010.5691343. URL https://doi.org/10.1109/CNSM.2010.5691343.

[59] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource packing for cluster schedulers. In *Proceedings of the 2014 ACM Special Interest Group on Data Communication*, SIGCOMM '14. ACM, 2014. ISBN 9781450328364. doi: 10.1145/2619239.2626334. URL https://doi.org/10.1145/2619239.2626334.

[60] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-resource Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026884.

[61] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. Graphene: Packing and Dependency-aware Scheduling for Data-parallel Clusters. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, 2016. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026885.

[62] WAND Network Research Group et al. Wits: Waikato internet traffic storage. *URL: http://wand.net.nz/wits/index.php*.

[63] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI '10, 2010.

[64] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces. In *Proceedings of the 2019 International Symposium on Quality of Service*, IWQoS '19. ACM, 2019. ISBN 978-1-4503-6778-3. doi: 10.1145/3326285.3329074. URL http://doi.acm.org/10.1145/3326285.3329074.

[65] Tian Guo, Upendra Sharma, Prashant Shenoy, Timothy Wood, and Sambit Sahu. Cost-aware cloud bursting for enterprise applications. *ACM Transactions on Internet Technology*,

13(3), may 2014. ISSN 1533-5399. doi: 10.1145/2602571. URL https://doi.org/10.1145/2602571.

[66] Alon Halevy, Flip Korn, Natalya F. Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. Goods: Organizing Google's Datasets. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16. ACM, 2016. ISBN 978-1-4503-3531-7. doi: 10.1145/2882903.2903730. URL http://doi.acm.org/10.1145/2882903.2903730.

[67] Mor Harchol-Balter and Allen B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '96, New York, NY, USA, 1996. ACM. ISBN 0897917936. doi: 10.1145/233013.233019. URL https://doi.org/10.1145/233013.233019.

[68] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ml elasticity through tiered reliability in dynamic resource markets. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17. ACM, 2017. ISBN 9781450349383. doi: 10.1145/3064176.3064182. URL https://doi.org/10.1145/3064176.3064182.

[69] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. Tributary: Spot-dancing for elastic services with latency slos. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18. USENIX Association, 2018. ISBN 9781931971447.

[70] John Hoxmeier and Chris Dicesare. System response time and user satisfaction: An experimental study of browser-based applications. *Proceedings of the Association of Information Systems Americas Conference*, 01 2000.

[71] David E. Irwin, Laura E. Grit, and Jeffrey S. Chase. Balancing risk and reward in a market-based task service. In *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, HPDC '04. IEEE Computer Society, June 2004. doi: 10.1109/HPDC.2004.1323519.

[72] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09. ACM, 2009. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629601. URL http://doi.acm.org/10.1145/1629575.1629601.

[73] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic Optimization for MapReduce Programs. *Proceedings of the VLDB Endowment*, 4(6):385–396, March 2011. ISSN 2150-8097. doi: 10.14778/1978665.1978670. URL https://doi.org/10.14778/1978665.1978670.

[74] Qin Jia, Zhiming Shen, Weijia Song, Robbert van Renesse, and Hakim Weatherspoon. Smart spot instances for the supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures and Platforms*, CrossCloud '16. ACM, 2016. ISBN 9781450342940. doi: 10.1145/2904111.2904114. URL https://doi.org/10.1145/2904111.2904114.

[75] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. Selecting Subexpressions to Materialize at Datacenter Scale. *Proceedings of the VLDB Endowment*, 11 (7):800–812, March 2018. ISSN 2150-8097. doi: 10.14778/3192965.3192971. URL https://doi.org/10.14778/3192965.3192971.

[76] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. Computation Reuse in Analytics Job Service at Microsoft. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18. ACM, 2018. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3190656. URL http://doi.acm.org/10.1145/3183713.3190656.

[77] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. Peregrine: Workload Optimization for Cloud Query Engines. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019. ISBN 9781450369732. doi: 10.1145/3357223.3362726. URL https://doi.org/10.1145/3357223.3362726.

[78] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Iñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '16. USENIX Association, November 2016. ISBN 978-1-931971-33-1. URL https://www.usenix.org/conference/osdi16/technical-sessions/presentation/jyothi.

[79] Burt Kaliski. Rfc2898: Pkcs #5: Password-based cryptography specification version 2.0, 2000.

[80] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, Raghu Ramakrishnan, and Sarvesh Sakalanaga. Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters. In *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC '15. USENIX Association, 2015. ISBN 978-1-931971-225. URL http://dl.acm.org/citation.cfm?id=2813767.2813803.

[81] Thomas Knauth and Christof Fetzer. Energy-aware scheduling for infrastructure clouds. In *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, CloudCom '12. IEEE Computer Society, 2012. doi: 10.1109/CloudCom.2012.6427569. URL https://doi.org/10.1109/CloudCom.2012.6427569.

[82] Ron Kohavi and Roger Longbotham. Online experiments: Lessons learned. *Computer*, 40(9), 2007. doi: 10.1109/MC.2007.328. URL https://doi.org/10.1109/MC.2007.328.

[83] Andrew Krioukov, Christoph Goebel, Sara Alspaugh, Yanpei Chen, David Culler, and Y Katz. Integrating renewable energy using data analytics systems: Challenges and opportunities. In *In Bulletin of the IEEE Computer Society Technical Committee*, 2011.

[84] Andrew Krioukov, Sara Alspaugh, Prashanth Mohan, Stephen Dawson-Haggerty, David E. Culler, and Randy H. Katz. Design and evaluation of an energy agile computing cluster.

Technical Report UCB/EECS-2012-13, EECS Department, University of California, Berkeley, Jan 2012. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-13.html.

[85] Shonali Krishnaswamy and Seng Wai Loke. Estimating computation times of data-intensive applications. *IEEE Distributed Systems Online*, 5(4), 2004. doi: 10.1109/MDSO.2004.1301253. URL https://doi.org/10.1109/MDSO.2004.1301253.

[86] Kien Le, Ricardo Bianchini, Jingru Zhang, Yogesh Jaluria, Jiandong Meng, and Thu D. Nguyen. Reducing electricity cost through virtual machine placement in high performance computing clouds. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 22:1–22:12. ACM, 2011. ISBN 978-1-4503-0771-0. doi: 10.1145/2063384.2063413. URL http://doi.acm.org/10.1145/2063384.2063413.

[87] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. EnaCloud: An Energy-Saving Application Live Placement Approach for Cloud Computing Environments. In *Proceedings of the 2009 IEEE International Conference on Cloud Computing*, CLOUD '09. IEEE Computer Society, Sep. 2009. doi: 10.1109/CLOUD.2009.72.

[88] Huan Liu and Sewook Wee. Web server farm in the cloud: Performance evaluation and dynamic architecture. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 9783642106644. doi: 10.1007/978-3-642-10665-1_34. URL https://doi.org/10.1007/978-3-642-10665-1_34.

[89] Qixiao Liu and Zhibin Yu. The Elasticity and Plasticity in Semi-Containerized Co-locating Cloud Workload: A View from Alibaba Trace. In *Proceedings of the 9th ACM Symposium on Cloud Computing*, SoCC '18. ACM, 2018. ISBN 978-1-4503-6011-1. doi: 10.1145/3267809.3267830. URL http://doi.acm.org/10.1145/3267809.3267830.

[90] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the 2019 ACM Special Interest Group on Data Communication*, SIGCOMM '19. ACM, 2019. ISBN 9781450359566. doi: 10.1145/3341302.3342080. URL https://doi.org/10.1145/3341302.3342080.

[91] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In Scott A. Lathrop, Jim Costa, and William Kramer, editors, *Conference on High Performance Computing Networking, Storage and Analysis*, SC '11. ACM, 2011. doi: 10.1145/2063384.2063449. URL https://doi.org/10.1145/2063384.2063449.

[92] Ming Mao and Marty Humphrey. A performance study on the VM startup time in the cloud. In Rong Chang, editor, *2012 IEEE Fifth International Conference on Cloud Computing, Honolulu, HI, USA, June 24-29, 2012*, pages 423–430. IEEE Computer Society, 2012. doi: 10.1109/CLOUD.2012.103. URL https://doi.org/10.1109/CLOUD.2012.103.

[93] Ming Mao, Jie Li, and Marty Humphrey. Cloud auto-scaling with deadline and budget

constraints. In *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing*, GRID '10. IEEE Computer Society, 2010. doi: 10.1109/GRID.2010.5697966. URL https://doi.org/10.1109/GRID.2010.5697966.

[94] Aniruddha Marathe, Rachel Harris, David Lowenthal, Bronis R. de Supinski, Barry Rountree, and Martin Schulz. Exploiting redundancy for cost-effective, time-constrained execution of hpc applications on amazon ec2. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14. ACM, 2014. ISBN 9781450327497. doi: 10.1145/2600212.2600226. URL https://doi.org/10.1145/2600212.2600226.

[95] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Phil Cudre-Mauroux. Dependency-Driven Analytics: a Compass for Uncharted Data Oceans. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*, CIDR '17, January 2017.

[96] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: A Progress Indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10. ACM, 2010. ISBN 9781450300322. doi: 10.1145/1807167.1807223. URL https://doi.org/10.1145/1807167.1807223.

[97] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *Proceedings of the IEEE 26th International Conference on Data Engineering*, ICDE '10. IEEE Computer Society, March 2010. doi: 10.1109/ICDE.2010.5447919.

[98] Klaus Muller and Tony Vignaux. Simpy: Simulating systems in python. *ONLamp. com Python Devcenter*, 2003.

[99] Shuangcheng Niu, Jidong Zhai, Xiaosong Ma, Xiongchao Tang, Wenguang Chen, and Weimin Zheng. Building semi-elastic virtual clusters for cost-effective HPC cloud resource provisioning. *IEEE Transactions on Parallel Distributed Systems*, 27(7), 2016. doi: 10.1109/TPDS.2015.2476459. URL https://doi.org/10.1109/TPDS.2015.2476459.

[100] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13. ACM, 2013. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522716. URL http://doi.acm.org/10.1145/2517349.2522716.

[101] Rina Panigrahy, Kunal Talwar, Lincoln Uyeda, and Udi Wieder. Heuristics for vector bin packing. January 2011. URL https://www.microsoft.com/en-us/research/publication/heuristics-for-vector-bin-packing/.

[102] Jun Woo Park, Alexey Tumanov, Angela Jiang, Michael A. Kozuch, and Gregory R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the 13th European Conference on Computer Systems*, EuroSys '18. ACM, 2018. ISBN 978-1-4503-5584-1. doi: 10.1145/3190508.3190515. URL http://doi.acm.org/10.1145/3190508.3190515.

[103] Florentina I. Popovici and John Wilkes. Profitable services in an uncertain world. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05. IEEE Computer

Society, 2005.

[104] Kaushik Rajan, Dharmesh Kakadia, Carlo Curino, and Subru Krishnan. Perforator: Eloquent performance models for resource optimization. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 415–427, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4525-5. doi: 10.1145/2987550.2987566. URL http://doi.acm.org/10.1145/2987550.2987566.

[105] Charles Reiss, John Wilkes, and Joseph L Hellerstein. Google cluster-usage traces: format+ schema. *Google Inc., White Paper*, pages 1–14, 2011.

[106] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12. ACM, 2012. ISBN 978-1-4503-1761-0. doi: 10.1145/2391229.2391236. URL http://doi.acm.org/10.1145/2391229.2391236.

[107] Andrea Rosà, Lydia Y. Chen, and Walter Binder. Predicting and mitigating jobs failures in big data clusters. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '15. IEEE Computer Society, 2015. ISBN 9781479980062. doi: 10.1109/CCGrid.2015.139. URL https://doi.org/10.1109/CCGrid.2015.139.

[108] Malte Schwarzkopf and Peter Bailis. Research for Practice: Cluster Scheduling for Datacenters. *Communications of the ACM*, 61(5):50–53, April 2018. ISSN 0001-0782. doi: 10.1145/3154011. URL https://doi.org/10.1145/3154011.

[109] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: Flexible, Scalable Schedulers for Large Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13. ACM, 2013. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465386. URL http://doi.acm.org/10.1145/2465351.2465386.

[110] Rathijit Sen, Alekh Jindal, Hiren Patel, and Shi Qiao. Autotoken: Predicting peak parallelism for big data analytics at microsoft. *Proceedings of the VLDB Endowment*, 13 (12):3326–3339, August 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415554. URL https://doi.org/10.14778/3415478.3415554.

[111] Liqun Shao, Yiwen Zhu, Siqi Liu, Abhiram Eswaran, Kristin Lieber, Janhavi Mahajan, Minsoo Thigpen, Sudhir Darbha, Subru Krishnan, Soundar Srinivasan, and et al. Griffon: Reasoning about Job Anomalies with Unlabeled Data in Cloud-Based Platforms. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019. ISBN 9781450369732. doi: 10.1145/3357223.3362716. URL https://doi.org/10.1145/3357223.3362716.

[112] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15. ACM, 2015. ISBN 9781450332385. doi: 10.1145/2741948.2741953. URL https://doi.org/10.1145/2741948.2741953.

[113] Prateek Sharma, Tian Guo, Xin He, David E. Irwin, and Prashant J. Shenoy. Flint: batch-interactive data-intensive processing on transient servers. In Cristian Cadar, Peter R. Pietzuch, Kimberly Keeton, and Rodrigo Rodrigues, editors, *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016*, pages 6:1–6:15. ACM, 2016. doi: 10.1145/2901318.2901319. URL https://doi.org/10.1145/2901318.2901319.

[114] Prateek Sharma, David E. Irwin, and Prashant J. Shenoy. Portfolio-driven resource management for transient cloud servers. *Proceedings of of the ACM on Measurement and Analysis of Computing Systems*, 1(1), 2017. doi: 10.1145/3084442. URL https://doi.org/10.1145/3084442.

[115] Upendra Sharma, Prashant J. Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *2011 International Conference on Distributed Computing Systems, ICDCS 2011, Minneapolis, Minnesota, USA, June 20-24, 2011*. IEEE Computer Society, 2011. doi: 10.1109/ICDCS.2011.59. URL https://doi.org/10.1109/ICDCS.2011.59.

[116] Supreeth Shastri and David Irwin. Hotspot: Automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17. ACM, 2017. ISBN 9781450350280. doi: 10.1145/3127479.3132017. URL https://doi.org/10.1145/3127479.3132017.

[117] Warren Smith, Ian T. Foster, and Valerie E. Taylor. Predicting application run times using historical information. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, IPPS/SPDP '98. Springer-Verlag, 1998. ISBN 3540648259.

[118] Ozan Sonmez, Nezih Yigitbasi, Alexandru Iosup, and Dick Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, HPDC '09. ACM, 2009. ISBN 9781605585871. doi: 10.1145/1551609.1551632. URL https://doi.org/10.1145/1551609.1551632.

[119] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *INTERSPEECH 2012, 13th Annual Conference of the International Speech Communication Association, Portland, Oregon, USA, September 9-13, 2012*. ISCA, 2012. URL http://www.isca-speech.org/archive/interspeech_2012/i12_0194.html.

[120] ShaoJie Tang, Jing Yuan, and Xiang-Yang Li. Towards optimal bidding strategy for amazon ec2 cloud spot instance. In *2012 IEEE Fifth International Conference on Cloud Computing*, 2012. doi: 10.1109/CLOUD.2012.134.

[121] Huangshi Tian, Yunchuan Zheng, and Wei Wang. Characterizing and Synthesizing Task Dependencies of Data-Parallel Jobs in Alibaba Cloud. In *Proceedings of the 10th ACM Symposium on Cloud Computing*, SoCC '19. ACM, 2019. ISBN 9781450369732. doi: 10.1145/3357223.3362710. URL https://doi.org/10.1145/3357223.3362710.

[122] Dan Tsafrir, Yoav Etsion, and Dror G Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed*

Systems, 18(6), 2007.

[123] Alexey Tumanov, Angela Jiang, Jun Woo Park, Michael A Kozuch, and Gregory R Ganger. JamaisVu: Robust scheduling with auto-estimated job runtimes. Technical report, Technical Report CMU-PDL-16-104. Carnegie Mellon University, 2016.

[124] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters. In *Proceedings of the 11th European Conference on Computer Systems*, EuroSys '16. ACM, 2016. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901355. URL http://doi.acm.org/10.1145/2901318.2901355.

[125] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SoCC '13. ACM, 2013. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523633. URL http://doi.acm.org/10.1145/2523616.2523633.

[126] 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 10th ACM European Conference on Computer Systems*, EuroSys '15. ACM, 2015. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741964. URL http://doi.acm.org/10.1145/2741948.2741964.

[127] Paul Voigt and Axel von dem Bussche. *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer Publishing Company, Incorporated, 1st edition, 2017. ISBN 3319579584, 9783319579580.

[128] Wei Wang, Baochun Li, and Ben Liang. To reserve or not to reserve: Optimal online multi-instance acquisition in iaas clouds. In Jeffrey O. Kephart, Calton Pu, and Xiaoyun Zhu, editors, *Proceedings of the 10th International Conference on Autonomic Computing*, ICAC '15. USENIX Association, 2013. URL https://www.usenix.org/conference/icac13/technical-sessions/presentation/wang_wei.

[129] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: Harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 1–16. ACM, 2021. ISBN 9781450383349. doi: 10.1145/3447786.3456225. URL https://doi.org/10.1145/3447786.3456225.

[130] Zichen Xu, Christopher Stewart, Nan Deng, and Xiaorui Wang. Blending on-demand and spot instances to lower costs for in-memory storage. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications*, INFOCOM '16. IEEE Press, 2016. doi: 10.1109/INFOCOM.2016.7524348. URL https://doi.org/10.1109/INFOCOM.2016.7524348.

[131] Neeraja J. Yadwadkar, Bharath Hariharan, Joseph E. Gonzalez, Burton Smith, and Randy H.

Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17. ACM, 2017. ISBN 978-1-4503-5028-0. doi: 10.1145/3127479.3131614. URL http://doi.acm.org/10.1145/3127479.3131614.

[132] Youngseok Yang, Geon-Woo Kim, Won Wook Song, Yunseong Lee, Andrew Chung, Zhengping Qian, Brian Cho, and Byung-Gon Chun. Pado: A data processing engine for harnessing transient resources in datacenters. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17. ACM, 2017. ISBN 9781450349383. doi: 10.1145/3064176.3064181. URL https://doi.org/10.1145/3064176.3064181.

[133] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, 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 Symposium on Networked Systems Design and Implementation*, NSDI '12, San Jose, CA, 2012. USENIX Association. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[134] Liang Zheng, Carlee Joe-Wong, Chee Wei Tan, Mung Chiang, and Xinyu Wang. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15. ACM, 2015. ISBN 9781450335423. doi: 10.1145/2785956.2787473. URL https://doi.org/10.1145/2785956.2787473.