Thesis proposal:
Realizing value in shared compute infrastructures

Andrew Chung
afchung@andrew.cmu.edu

1 Introduction

As 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 operators and users of such shared data infrastructures strive to optimize for *value.* Operators seek to satisfy the demands of their customers (i.e., help users maximize their value) to increase adoption and lower turnover, all the while without sacrificing cluster operation costs and overhead. 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 operators and users: Operators face difficult challenges in knowing how to assign compute resources to customers when heavily loaded. Users, on the other hand, have a wide variety of different types of compute resources available for rent, making it difficult for them to make *value-efficient* resource acquisition decisions for their applications, given application constraints. Indeed, maximizing value in shared data infrastructures necessarily requires effort from both operators and users.

In our work, we explore the problem of value attainment in shared data infrastructures from both the perspectives of operators and users. On the operator front, we explore using the notions of historic inter-job dependencies and expected job utility to inform cluster resource managers about upcoming jobs, their resource requirements, and the potential value they generate to users. Cluster resource managers can in turn use the information to effectively allocate cluster resources to jobs to achieve high user value attainment. 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, one for running online services and the other for general batch analytics jobs, with each demonstrating significant cost savings for users.

2 Thesis statement

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

My thesis proposal will first describe ongoing and planned work for operators of shared clusters to achieve value for cluster users. It will then provide an overview of prior work on optimizing for user value through frameworks supporting user applications in shared cluster environments (e.g., in AWS EC2).

Realizing value through dependency-aware resource management (§3). My dissertation will support my thesis by exploring opportunities for cluster operators and resource managers to realize user value through analyses of historical inter-job dependencies, effectively prioritizing jobs which are shown to be more historically valuable to users in the setting of large, multi-tenant corporate clusters (Microsoft Cosmos). This work consists of two parts:

(i) Ongoing work: ***Analysis of inter-job dependencies in Cosmos: Challenges & opportunities for resource management*** **(§3.2).** Inter-job dependencies pervade shared data analytics infrastructures, yet are largely ignored in current cluster resource management. I am performing a detailed analysis of inter-job dependencies in a 50K+ node analytics cluster within Microsoft's Cosmos infrastructure, based on job and data provenance logs, deriving consequential observations and untapped opportunities. We find that nearly 80% of all jobs depend on at least one other job, with 20% of jobs depending on jobs submitted by a different company organization sharing the data lake. Yet, even in an expertly-managed and business-critical setting, coordination appears challenging: 34% of jobs are submitted without checking if the job output they depend on is available, failing if not; perhaps worse, 13% of jobs depend on jobs that execute at a lower priority, creating non-trivial risk of priority inversion. We also find, however, that over 68% of jobs exhibit their dependencies in a recurring fashion, creating potential for using inter-job dependencies to improve resource management. Our results expose several opportunities to use historical inter-job dependencies to improve resource management.

(ii) Planned work: ***Value-aware, inter-job dependency driven scheduling*** **(§3.3).** Cluster job schedulers generally consider jobs independently of one another while ignoring their dependencies on the outputs of other jobs. This independent consideration of jobs may have worked well in siloed clusters, but in today's highly shared cluster environments, we believe significant benefits are forfeited by not considering inter-job dependencies. In my remaining work, I plan to build an inter-job dependency aware cluster scheduler and evaluate the benefits of dependency awareness using traces from a production cluster in Microsoft's Cosmos infrastructure. Our new scheduler will feature a new data representation for job value, more expressive than that of priority assignments and utility functions, which considers probabilistic views of a job's *value impact* on other jobs historically dependent on it. We will compare value-attained by our scheduler against state-of-the-art value-aware cluster schedulers.

Realizing value through user applications (§4.1—§4.3). To support my thesis statement, I will also describe two case studies of research software systems 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*** **(§4.2).** 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*** **(§4.3).** 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.

3 Realizing value through dependency-aware resource management

This section describes our ongoing work in realizing value through dependency-aware resource management from cluster operators. Work in this section is reliant primarily on workload from Microsoft's Cosmos clusters, with a focus on batch analytics jobs.

3.1 Background and motivation

This section motivates and provides background for our study of complex inter-job dependencies within modern shared data environments. More specifically, we describe characteristics Microsoft's Cosmos infrastructure and provide an overview of the different types of inter-job dependencies observed within Cosmos.

Shared data environments. Shared data environments such as *data lakes* and the public cloud have become core elements of modern data-driven enterprises, providing required data storage and analysis infrastructure. These environments 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 [17] and reduce overall computational resource-hours consumed. But, at the same time, such data re-use necessarily introduces more dependencies between jobs and the datasets they produce. Our ongoing work focuses on these *inter-job dependencies*[1] that arise from such data re-use.

Cosmos. Cosmos is a big data analytics platform deployed at Microsoft consisting of multiple clusters of 50k+ servers each. Even though numerous application types are executed in Cosmos's infrastructure, more than 80% of infrastructure capacity is used for batch data-analytics jobs (*Scope jobs*). We focus on Scope jobs and dependencies between them.

Motivation: Inter-job dependencies are prevalent and recurrent. In Cosmos, we find that almost 80% of submitted jobs depend on output generated by at least one other job. Indeed, interestingly, over half of the jobs are connected in a single dependency subgraph, and surprisingly, many dependencies are cross-organization, with 20% of jobs depending on jobs submitted by another organization. But, despite so much inter-job dependence, today's systems provide little support for addressing associated challenges.

Although it is difficult to know what future jobs will depend on the output generated by a current job, we find significant hope in recurrence. Previous workload studies have shown that most jobs in data analytics environments are *recurring* [23, 37], where a recurring job is one that is submitted many times over time, often to analyze fresh data. We find that inter-job dependency patterns are similarly recurrent, with jobs of the same template following similar input dependency patterns. This enables the use of *historically recurring dependencies* to (i) analyze and predict relationships between common, dependent recurring jobs, and (ii) pursue opportunities to exploit the dependency properties between jobs.

Our discussion makes the distinction between jobs and job templates, where a *job template* is a program to be executed (one or multiple times) in Cosmos, while a *job* is an actual execution of a job template: i.e., each submission of a job template results in a job. For brevity, in the rest of §3, job templates strictly refer to templates of recurring jobs.

3.2 Ongoing work: *Analysis of inter-job dependencies in Cosmos: Challenges & opportunities for resource management*

This section presents our study on inter-job dependencies within Cosmos. Analyses and observations made within this study are used to: (i) identify shortcomings in the current operation of shared data environments and shortcomings in existing job valuation schemes and (ii) explore how inter-job dependency awareness can inform cluster schedulers to achieve better scheduling (§3.3). We summarize our observations as follows:

Observation 1 (Recurring jobs & dependencies): *Most jobs and dependencies are recurring. Recurring jobs make up 68% of all jobs, while recurring dependencies make up 79% of all dependencies. These properties imply the predictability of both jobs and their dependencies.*

Challenges and opportunities. Recurrence of jobs dependencies lead users to form expectations with respect to both the latency of their jobs and the arrival time of the inputs their jobs depend on. These expectations, which present challenges to cluster operators [23, 15], both stem from and are reinforced by past

[1]We say that Job A depends on Job B if Job A takes as input an output file generated and stored into the shared distributed file system by Job B. i.e., our nomenclature and analysis focus on fundamental dataflow dependencies among batch analytics jobs, not distributed stream processing or artificial inter-relationships caused by resource contention.

experience. But, this recurrence also implies that the behavior of jobs and their dependencies are predictable: In our dataset, we observe that if a job x_1 of template X depends on a job of template Y, there is more than a 75% chance that another job x_2 of template X will also depend on a job of template Y. Predictable dependencies can allow resource managers to preemptively configure cluster state in anticipation of the arrival of future jobs based on jobs that have already arrived.

Observation 2 (Priority mis-configurations): *Potential priority mis-configurations are frequent: jobs of 21% of job templates have the chance to be systematically priority-inverted — i.e., recurring jobs consuming their output are configured with a higher priority assignment. In addition, up to 33% of ad-hoc jobs are assigned higher priority than the average recurring job submitted within the same organizational queue*[2].

Challenges and opportunities. This observation confirms two commonly-held beliefs: that (i) properly configuring job priorities is highly challenging, evident from the fact that a non-trivial amount of job templates are assigned priorities that can lead to priority inversions, and that (ii) users often care most about the completion of their own jobs, often assigning high priority to their ad-hoc jobs that exceed that of the average recurring job, among many of which are production jobs [23]. Analysis of jobs and their dependencies can enable us to apply strategies to: (i) fix recurring dependencies that can lead to priority inversions, and (ii) identify and warn users who abuse organizational resources for their benefit.

Observation 3 (Uncoordinated jobs): *Many jobs are submitted without coordination with respect to the completion of their upstream jobs. Such jobs make up 38% of recurring jobs. Furthermore, 89% of these recurring jobs have input dependencies that are identified as hard dependencies, meaning that up to 34% of recurring jobs are susceptible to failure due to missing input data from a late upstream job.*

Challenges and opportunities. Uncoordinated, hard dependencies[3] between downstream and upstream jobs are fragile. Unlike polling dependencies, the submission of the downstream job in such a dependency is not coordinated with the completion of the upstream job by checking for its output; and unlike roll-up dependencies, these dependencies do not allow any missed inputs. Jobs with such dependencies on their outputs are more prone to violate the expectations set by their downstream consumers. A way to mitigate the situation is to convert dependent recurring jobs with upstream dependencies that are both hard and uncoordinated into polling jobs, which, although might add overhead to users by making their job submission logic more complicated, can provide significant benefits to the smoothness of the execution of these jobs.

Observation 4 (Cross-org jobs & dependencies): *Cross-org jobs and dependencies are very 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. Futhermore, 28% of jobs and 23% of recurring jobs are involved in a cross-org dependency relation.*

Challenges and opportunities. While the design of systems today largely neglects inter-job dependencies, the proliferation of cross-org dependencies, the potential mis-configuration of priority assignments, and the prevalence of uncoordinated dependent job templates reveal many opportunities in system design that can improve cluster operation efficiency and communication of contractual agreements across organizations.

Observation 5 (WCC shapes and sizes): *The structure of weakly-connected-components (WCCs) in Cosmos's job dependency graph G*[4] *is highly-diverse, with a mix of both complex and simple WCCs of various shapes and sizes. We observe that most WCCs are small, but large WCCs cover most jobs. As WCCs grow in size, root jobs become more impactful, as measured by its ratio over the number of leaf jobs.*

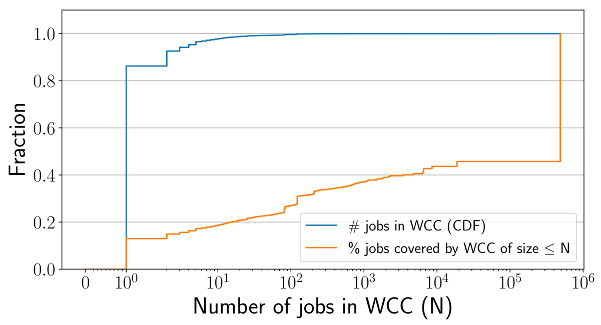
Fig. 1a shows that $> 95\%$ of WCCs are small and consist of less than three jobs, and that more than 50% of jobs belong to a single WCC. Within all WCCs, we analyze the structures of WCCs in G with sizes of at least 10 jobs, which cover more than 75% of all jobs.

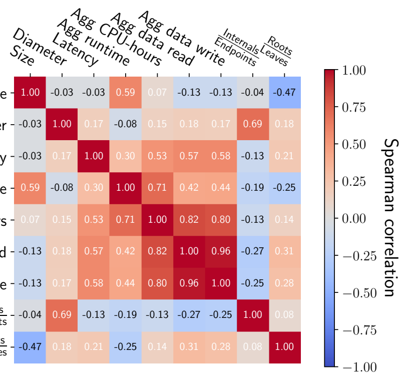
To identify *simple* WCCs, we induce undirected subgraphs from each WCC and identify ones that are trees, where a *tree* is an undirected graph in which any two vertices are connected by exactly one path. Trees

[2] *Hierarchical queues* are used to designate the resource share of an organization in a cluster at Microsoft, and priority assignments are only comparable within jobs of the same queue.

[3] Dependencies are *hard* if the downstream job *requires* the output(s) of the upstream job to be able to run successfully.

[4] The *job dependency graph (G)* is a directed acyclic graph (DAG) where each vertex in the graph is a job. A job x dependent on job y forms a directed edge (y, x), with y as the source and x as the target. A *weakly connected component (WCC)* in G is defined as a maximal subgraph $S \in G$ such that each pair of vertices in S is connected by at least one undirected path in S.





(a) **WCC sizes.** This figure shows that most WCCs are small (*# jobs in WCC*), and that most jobs are covered by large WCCs (*% jobs covered by WCC of size* $\leq$ *N*).

(b) **Spearman's correlation for WCC properties.** This figure shows the correlation between WCC properties for WCCs with size ≥ 10, discussed in Observation 5.

Figure 1: Weakly connected component (WCC) properties.

are structurally simple, but can be more fragile; i.e., for a path connecting any pair of jobs, the failure or late-arrival of any single job on the path can potentially lead to cascading failures. Our analysis shows that the complexity of WCCs are varied, and that not all large WCCs are complex: 55% of analyzed WCCs are trees, and there exist trees in WCCs as large as up to 800 vertices.

To learn more about the shapes and properties of WCCs, we analyze features of WCCs and try to correlate them using *Spearman's correlation* (Fig. 1b). Surprisingly, for some properties that we expected strong positive correlation with WCC size we find weak correlation (e.g., diameter, latency, aggregate CPU-hours) or even negative correlation (e.g., aggregate data read/write). But, We do observe that as the sizes of WCCs grow, $\frac{\text{Roots}}{\text{Leaves}}$ shrinks, i.e., larger graphs will appear narrower at the top and wider at the bottom. This shows that there are often more leaf than root jobs in analyzed WCCs, suggesting that some jobs' outputs are more important than others and have wide impact downstream. Indeed, 14% of all jobs are dependent upon the output of the top 0.1% of jobs, making failure or delay of these jobs much more disruptive.

Challenges. Our observations show that many jobs are inter-connected through data dependencies and imply that certain jobs need to be scheduled and placed more carefully than others, since their output is depended upon by many other jobs downstream. The parameters (e.g., submission time and priority) of these jobs should only be tuned upon careful consideration, since seemingly harmless changes made by job owners can potentially cause a ripple effect downstream.

3.3 Planned work: *Value-aware, inter-job dependency driven scheduling*

This section describes the background for and sets up requirements for our exploratory work in Talon, an inter-job dependency and value aware cluster scheduler.

3.3.1 Today's production clusters: priority based scheduling and automating job values

Companies can achieve lower costs-of-goods-sold through effective value-oriented cluster job scheduling and data pricing, ensuring that jobs producing data with more business value complete on time. To do this today, schedulers at 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 directly into a priority assignment on the job—the greater a job's value, the higher its priority. But, as noted in Observation 2, priorities are difficult to set correctly.

Dependency-driven job valuation. Although determining the true dollar-value of jobs is difficult, if not impossible, we find it promising to evaluate the importance of jobs based on their historical inter-job dependencies. Through our analysis of connected components within our job dependency graph (Observation 5), combined with the fact that many recurring jobs are uncoordinated with respect to the availability of their input (Observation 3), we find that the delay or failure of certain jobs are much more impactful than that of other jobs. Hitches in the execution of these jobs are likely to cause much more financial and operational damage to users and organizations within the company. With historical inter-job dependencies, we can objectively measure the impact of recurring jobs based on past telemetry of their downstream consumers, which we believe can make a good proxy for dollar-value of jobs, where our method proposed in Owl [12] uses a job's impact on downstream users (measured by features such as aggregate output downloads) and impact on downstream jobs (measured by features such as aggregate compute-hours) as a proxy for job value[5].

3.3.2 Dependency-driven utility functions

Even if priority assignments were perfectly configured to reflect job value, a fundamental limitation of the priority assignment construct limits its effectiveness in cluster scheduling. Specifically, there is only one dimension to priority assignments, so they cannot express both *job importance* and *urgency* at the same time. On the other hand, *utility functions* allow schedulers to estimate job value *realizable* if it were to complete a job by time T by expressing job value as a function of job completion time, adding the dimension of *time* to job value assignments.

A simple example shows the shortcoming of priority assignments: Assuming perfect knowledge of job value, suppose two jobs A and B, both requesting access to a singleton cluster resource (i.e., only one job gets to run), are waiting in the scheduler's job queue. A is more important, and therefore assigned a higher priority, but is not latency sensitive. B on the other hand, has a tight deadline to meet, but is less important. The perfect scheduler would assign the resource to B while delaying the execution of A, where in reality, most priority-based schedulers would give resource preference to job A.

Utility functions are commonly used in scheduling literature [39, 28] and have been shown to vastly improve a schedulers' decision making to achieve higher value attained. But, utility functions have not been deployed in any real production cluster environments, presumably due to the difficulty involved in constructing reliable utility functions that truly reflect a job's value over time.

Our work in Owl [12] describes a data-driven approach to automate utility function construction using dependency-based job value, removing the burden of manually creating reliable utility functions: Suppose each job j has inherent value $v_j(\tau)$, which is value rewarded for completing j by time τ relative to the submission time of the j (i.e., $v_j(t) = val_j$ if $t \leq \tau$, 0 otherwise). We define the *base case* of utility functions for leaf jobs j with no dependent jobs as $u_j(t) = v_j(\tau)$. We then define the *recursive case* of utility functions for non-leaf jobs j with directly dependent jobs D_j as $u_j(t) = (\sum_{l \in D_j} \chi_{(t \leq s_l^j)}(t) * u_l(0)) + v_j(\tau)$, where $\chi_{(t \leq s_l^j)}(t)$ is a step function that evaluates to 1 if j completes prior to the submission time of l relative to j's submission time (term s_l^j), 0 otherwise; $u_l(0)$ represents the maximum achievable utility by l directly downstream of j. Fig. 2a illustrates the construction of a utility function with a toy example, and Fig. 2b shows the real utility function for recurring jobs of a real job template in production.

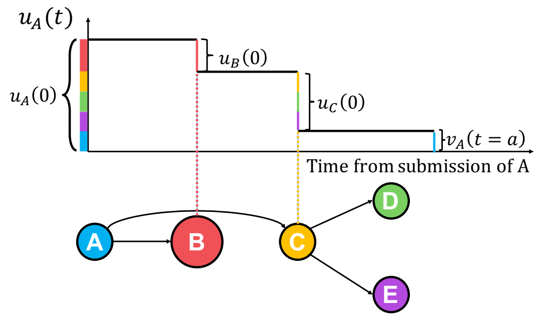
3.3.3 Talon: Value-aware, inter-job dependency driven scheduling

Shortcomings of utility functions. Utility functions, as described in §3.3.2, reflect an assumption that dependent jobs will fail upon submission if their inputs are missing. In reality, however, there are cases when the downstream job can tolerate missing (e.g., jobs that *roll-up* and compute aggregate statistics from outputs of previous recurring jobs) or late input (e.g., jobs that *poll* for their input).

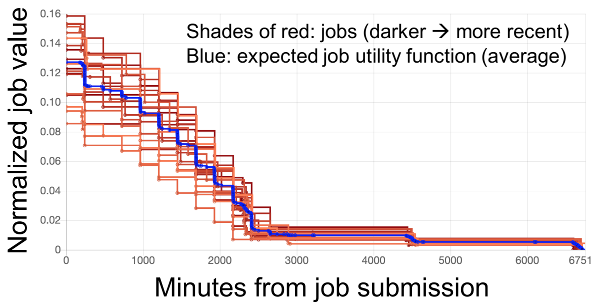
Our utility functions also cannot encode value attainable for completing an upstream job when a downstream job is dependent on multiple upstream jobs. For example, suppose two jobs A and B are submitted and queued at a cluster, both of whose output are required for the success of a future job C. In this case, both the utility functions of jobs A and B will show a drop in value corresponding to the expected submission of job C; but, the scheduler will not be able to ascertain that both A and B are required to complete in order for C's value to be attainable.

Talon and research problems to solve. Below, we list several key research problems that must be addressed to achieve Talon's goals:

[5]A short experiment adopting Owl's scheme for job valuation [12] to see how dependency-driven job values match up with pre-existing notions of job importance, using a list of six highly important recurring job templates hand-curated by the Cosmos resource management team at Microsoft, show that Owl's method of valuing jobs yields mostly consistent relative rankings of importance compared to existing priority-based importance rankings.



(a) Utility function toy example. This figure shows a toy example of a utility function for job A, directly dependent upon by jobs B and C. A starts out holding value $u_A(0) = u_B(0) + u_C(0) + val_A$. As directly dependent jobs are submitted and fail, A loses realizable value: e.g., $u_A(t = s_B^A) = u_C(0) + val_A$.



(b) A real utility function. This figure shows a utility function of a recurring job in production which has a directly dependent periodic job. The job drops in value every time an instance of the directly dependent job arrives, corresponding to a drop in realizable value if it fails to complete in time.

Figure 2: Utility functions.

1. **Prediction model for recurring dependencies**: While Observation 1 hints that recurring dependencies can be predictable, we will need to build and evaluate a statistics-based model that demonstrates the predictability of recurring dependencies. Given a set of jobs that have already arrived, the model should predict both whether a dependent job will arrive and its time-to-dependency.
2. **Talon job value representation**: We should design a data representation for job value that overcomes the limit of utility functions, as discussed above. The data representation should explicitly consider: (i) jobs already in the queue, (ii) the types of dependencies of upcoming, directly-dependent jobs on queued jobs, (iii) the probability that directly-dependent jobs will arrive, using the prediction model described above, and (iv) a summarized view of properties of upcoming jobs (e.g., their aggregate downstream value and manually set deadlines downstream).
3. **Talon scheduling algorithm**: A new scheduling algorithm is needed to take advantage of the more expressive data representation described above.
4. **Evaluation of Talon with dependency-preserving workloads**: We will need to evaluate Talon against alternative, value-aware schedulers—e.g., ones using priority assignments and utility functions, with inter-job dependency-preserving workloads.

4 Realizing value through user applications

This section describes our contributions to realize value for users through cost-aware user application frameworks specialized to the shared cloud environments.

4.1 Background

This section describes background for our work in application frameworks that can help users reduce their cost of running applications in shared environments such as the public cloud.

4.1.1 Public vs private clouds

A *cloud* is a datacenter that provides software services supported by remote hardware [10]. When a cloud is offered to the public via the Internet, most often in a pay-as-you-go-manner, the cloud is a *public cloud*; if the cloud is operated by and only available internally within a business or corporation, the cloud is a *private cloud*. Compute and storage resources in clouds, both public and private, are highly shared by multiple users, each potentially of a different organization. Our studies (§4.2, §4.3) are mainly concerned with realizing user value when running their applications on virtual machines (VMs) rented from public clouds. However, we believe our strategies can be equally effective in the context of private clouds, where organizations within companies are often given fair shares of resources, but idle, excess capacity can acquired through bidding [36, 27] or bonus token mechanisms [24, 11].

4.1.2 VM instance offerings in clouds — AWS EC2 as a concrete example

Cloud service providers (CSPs) offer an effectively infinite (from most customers' viewpoints) set of VM instances[6] 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 [3, 7, 6] are *reliable* and *transient*. Instances leased under an *reliable* contract are non-preemptible. Instances leased under a *transient contract* are made available, on a best-effort basis, at decreased cost (in for-pay settings) and/or at lower priority (in private settings), and can be unilaterally revoked by the CSP at any time. These *transient instances* are often offered as a way to increase utilization in the CSPs' datacenters. This section describes transient instances in AWS EC2, both to provide a concrete example and because our software systems specialize to EC2 behavior.

EC2 offers "on-demand instances", which are VMs rented under reliable contracts billed at a flat per-second rate. EC2 also offers the same VM types as "spot instances", which are transient but are usually billed at prices significantly lower (70% - 80%) than the corresponding on-demand price. EC2 may preempt spot instances at any time, thus presenting users with a trade-off between reliability and cost savings.

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 [9]. (2) Price movements among spot markets are not always correlated, even for the same instance type in a given region [31]. (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 2nd, 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*. While many bidding strategies for EC2 spot instances have been studied, but the most popular strategy by far is to bid the on-demand price to minimize the odds of preemption [31].

4.2 *Tributary: Spot-dancing for elastic services with latency SLOs*

This section describes Tributary [19], a control system for elastic web services with latency SLOs that exploits the low cost of transient cloud VM instances (e.g., AWS spot instances), to robustly and cost-effectively scale-in and scale-out web services run in the public cloud.

4.2.1 Problem statement and challenges

Elastic service scaling schemes generally assume independent and infrequent failures, which is a safe assumption for high-priority allocations in private clouds and *non-preemptible* allocations in public clouds (e.g., on-demand instances in EC2). 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 resources (§4.1) at a discount, creating an opportunities for lower-cost 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 [20]. Akin to co-occurring failures, bulk preemptions can cause traditional scaling schemes to have sizable gaps in SLO attainment. Tributary is an elastic control system that considers properties of and exploits these transient, preemptible instances to reduce cost and increase robustness to unexpected workload bursts in web services run on the public cloud.

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

4.2.2 Tributary-specific background and related work

Scaling policies and resource acquisition schemes. Elastic web services dynamically acquire and release machine resources to adapt to time-varying client load. In this document, 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. Generally, an adaptive scaling policy seeks to use just the number of machines required to achieve its 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 [21, 26]. 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 [16, 33].

Related work. *AWS AutoScale* [2] 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 spot instances, the user can use a so-called "spot fleet" [4] consisting of multiple instance type and availability zone options. In this case, the user configures AutoScale to use one of two strategies. Our experiments focus on the *lowestPrice* strategy, which will always select cheapest current spot price of the specified options.

ExoSphere [32] is a virtual cluster framework for spot instances. Its instance acquisition scheme based on market portfolio theory, relies 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 considered spot market. ExoSphere acquires instances from each spot market bidding the on-demand price.

Proteus [20] 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.

4.2.3 System design

Tributary is an elastic control system that comprises both a pluggable scaling policy and a resource acquisition scheme tailored to cloud resource markets. In order to run web services robustly at a lower cost using transient instances, Tributary explicitly recognizes bulk preemption risk (§4.2.1), exploiting the fact that preemptions are often not highly correlated across different pools of resources in heterogeneous clouds. *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 machine learning (ML) models to predict the preemption probability of transient 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.

Resource allocations and service utility functions. AcquireMgr interacts with AWS to acquire resources. To do so, AcquireMgr builds sets of *allocation requests*, which specifies the instance type, availability zone, bid price, and number of instances to acquire. An *allocation* is defined as a set of instances of the same type acquired at the same time and price. AcquireMgr's *footprint* is a set of such allocations.

AcquireMgr abstracts away the resource type which is being optimized for. For workloads described in this paper, 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. 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 $v()$, 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.

Prediction model. When acquiring spot instances on AWS, there are four parameters that affect the preemption probability of an instance: its time-of-bid, type, availability zone, and bid price. We build an LSTM model, a popular model often used on temporal datasets, that predicts the probability of an instance being preempted within an hour given the above parameters.

Our model is trained offline 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 hour, as described in §4.1). 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.

AcquireMgr overview. 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. Here, we briefly describe how AcquireMgr computes these values.

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. We also recall that an *allocation* is specified by a single resource request to EC2 and uniquely identifiable by a tuple of request time, bid price, and resource pool. Note that an allocation can contain multiple instances, and that Tributary can request and manage multiple allocations of VMs from the same resource pool.

The expected cost for a given set of allocations can be calculated as the sum over the expected cost of individual allocations, where the expected cost of an allocation (a) is computed by considering the probability of preemption within the allocation's preemption window at a given bid delta (through the ML model). There are exactly two possibilities: an allocation will either be preempted with probability β_a (wherein the allocation will be free-of-cost) or it will reach the end of its preemption window in the remaining t_a minutes with probability $1 - \beta_a$, in which case we would voluntarily release the allocation. The expected cost for the allocation can then be written as: $(1 - \beta_a) * P_a * k_a * t_a + \beta_a * 0 * k_a * t_a$, where k_a is the number of instances in the allocation, and P_a is the market price for the instance type at the time of the allocation request.

In addition to computing expected cost for a set of allocations, AcquireMgr also computes the set's *expected utility*, which is the expected percentage of requests that will meet the latency target, given the allocations within the set and the client demand, taking into account the probability of allocation preemptions.

The expected utility V_A of a set of allocations A is calculated as: $V_A = \sum_{r=0}^{resc(A)} P(R = r) * v(r)$, 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, v is the utility function provided by the service, and $resc(A)$ is the function that reports the *total amount of resources* in a set of allocations A (e.g., in number of VCPUs).

Our equation 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, as AcquireMgr continuously reevaluates its constituent instances.

In deriving $P(R = r)$, AcquireMgr computes the probability that r resources remain among all allocations of A, taking conditional preemption probabilities of allocations from the same resource pool into account. For example, if allocations x and y both come from the same resource pool, their probabilities of being preempted will be not be independent, and AcquireMgr takes this into consideration.

Tributary also introduces a regularization term for each resource pool to increase bidding in markets with low correlation, where a weighted, configurable penalty is applied based on recent market price correlation between resource pools. This encourages Tributary to create a diversified footprint, reducing the probability that significant portions of instances are preempted simultaneously.

Scaling out. 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. At each decision point, AcquireMgr's objective is to acquire resources until the expected utility θ 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 and utility of the current set of allocations. AcquireMgr then calculates the missing number of resources required to meet the target SLO and builds a set of possible allocations that consists of allocations from different resource pools at different bid prices. For each possible new allocation, AcquireMgr records the new expected utility divided by the new expected cost, choosing the allocation that maximizes this value. AcquireMgr continues to add possible allocations until it achieves the target SLO in expectation.

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. While the primary goal of these additional resources is to account for preemptions, they have the added benefit to handle unexpected increases in load. Our experiments (§4.2.4) show that these resource buffers both increase the fraction of requests meeting latency targets and decrease cost.

Scaling in. Aside from preemptions, Tributary also tries to scale in voluntarily. When an allocation reaches the end of its preemption window, it is terminated and replaced with a new allocation if required, since spot instances that run for more than an hour cannot benefit from preemption refunds. When resource requirements decrease, Tributary also considers terminating allocations for those still within their preemption windows—starting with allocations least likely to be preempted (i.e., allocations least likely to be refunded for preemption). During this process, Tributary chooses the allocation with the least time remaining in the hour, computes the expected utility 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 is greater than the target SLO.

Example. Fig. 3 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.

4.2.4 Evaluation

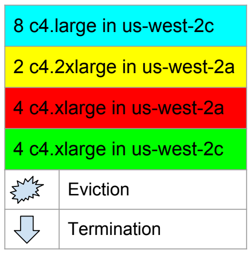
This section evaluates Tributary's effectiveness, with more details available in our paper [19].

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. Our simulated workload uses four real-world traces for request arrival times: *ClarkNet*, *Berkeley*, *WorldCup98* [14], and *WITS* [40], each with different characteristics. All traces are scaled to have an average of 125 requests per second in order to generate sufficient load for the experiments. Our discussion focuses on the *ClarkNet* trace, which is periodic and diurnal, while evaluation of our system on other traces is described in further detail in our paper. Each request in our trace consists of a CPU-bottlenecked computation that can be processed in ∼100ms on a single VCPU. Our experiments' resource requirements are therefore characterized by requests-per-second-per-VCPU, with our target service latency per request set to one second. From here on, we define a *"slow" request* as a request that does not meet the latency target. Each VM instance maintains a queue of requests, and we simulate the queueing effects using a discrete event simulator. The queue size per instance is 10x the number of VCPUs in the instance.

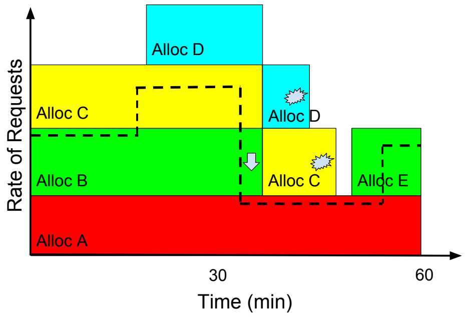
Spot market traces. To achieve fair comparisons across a wide range of data points, we perform cost analysis using historical spot market traces between January 23, 2017 and March 23, 2017 in *us-west-2*.

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 since our workload characteristic is embarrassingly

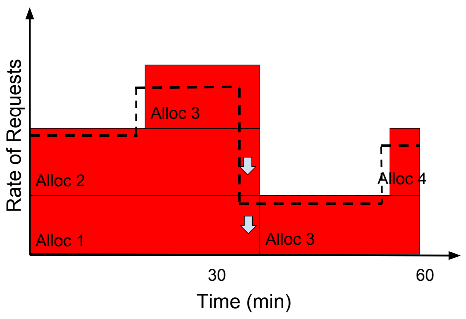


8 c4.large in us-west-2c
2 c4.2xlarge in us-west-2a
4 c4.xlarge in us-west-2a
4 c4.xlarge in us-west-2c
Eviction
Termination

(a) Legend

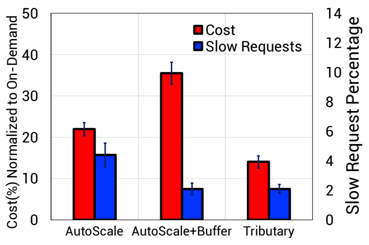


(b) Tributary

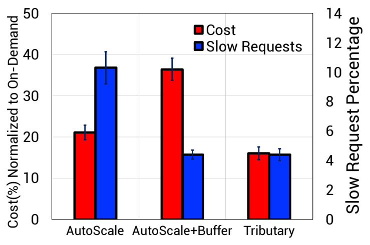


(c) AutoScale

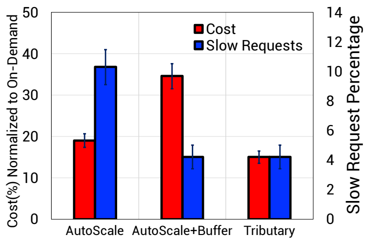
Figure 3: Figures (b) and (c) show how Tributary and AutoScale handle a sample workload respectively. Figure (a) is the legend for (b) and (c), color-coding each allocation. The black dotted lines in (b) and (c) signify the request rates over time. At **minute 15**, the request rate unexpectedly spikes 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.



(a) Reactive



(b) Predictive-LR

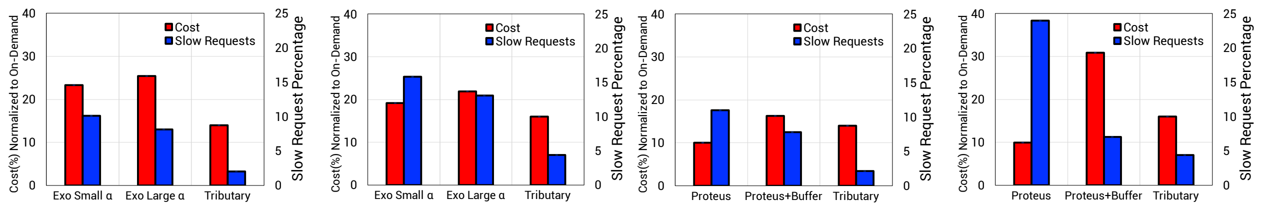


(c) Predictive-MWA

Figure 4: Cost savings (red) and percentage of "slow" requests (blue) for the *ClarkNet* trace.

parallel. If a workload exhibits different scaling characteristics, a different utility function can be employed.
Evaluation vs AutoScale. We evaluate Tributary's ability to reduce cost and latency target misses against AutoScale. AWS AutoScale, as offered, only supports the simplest scaling policies. To provide better comparison, we implement AWS AutoScale's resource acquisition policy according to its documentation [2].

Fig. 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 show that Tributary reduces cost and "slow" requests for all scaling policies. Cost savings are $\sim 85\%$ 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 scaling policies. Furthermore *AutoScale+Buffer*, shows the cost of provisioning AutoScale with a large enough resource 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.
Evaluation vs ExoSphere. We implemented ExoSphere's allocation strategy (§4.2.2) for comparison against Tributary. Fig. 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



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

Figure 5: Comparing to ExoSphere and Proteus. Predictive-MWA results not shown but similar.

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. 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.

Evaluation vs Proteus. We implemented Proteus's allocation strategy (§4.2.2), modified to acquire only spot resources. Fig. 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-VCPU cost. When told by the scaling policy to acquire additional buffer resources, (similar to AutoScale+Buffer), Proteus is still unable to match Tributary's number of "slow" requests no matter how large the buffer. This is due to the lack of diversity in the instances that Proteus acquires.

4.3 *Stratus: Cost-aware container scheduling in the public cloud*

This section describes Stratus [13], a cost-aware, specialized batch analytics job scheduler for orchestrating job execution on collections of VM instances rented from the public cloud. Stratus aggressively packs job tasks onto VMs, guided by job runtime estimates, to lower the dollar cost of executing batch job workloads.

4.3.1 Problem statement and challenges

Many organizations rely on public clouds to offload workload bursts from traditional on-premise clusters (so-called "cloud bursting") or even to replace these clusters entirely. Although traditional cluster schedulers could be used to manage a mostly static allocation of public cloud VM instances, such an arrangement would fail to exploit the public cloud's elastic on-demand properties and thus be unnecessarily expensive.

A common approach is to allocate an instance for each submitted task and then release that instance when the task completes. But, 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 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.

4.3.2 Stratus-specific background and related work

Autoscaling of virtual clusters. There are two parts to autoscaling a VC: determining the capacity to scale to and picking the right set of instances to meet said capacity. CSPs offer VC management frameworks (e.g., *Amazon EC2 Spot Fleet* [4]) for choosing and acquiring instances to scale based on a user-specified strategy, up to a target capacity.

Assigning containerized tasks to instances. Container services enable containerized user tasks to be run on a public cloud. In the server-based model [1], users provide a pool of instances while the container service schedules and packs tasks on to available VMs according to a configured placement policy. In the

container-based model [5], the container service automatically manages container placement, execution, and underlying infrastructure. But, this approach is currently significantly more expensive than other alternatives.
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. An example we compare to, *HotSpot* [34], 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.
Packing VC schedulers. Compared to the common approach of assigning a single-task-per-instance in existing VC scheduling literature, schedulers that *pack* tasks onto instances may reduce overall cost, as they reduce the risk of lower utilization due to imperfect fit. One reasonable approach packs containerized tasks on an elastic VC using CSP-offered services. Specifically, one can use server-based container services (e.g., ECS) to place containerized tasks on to instances, while maintaining a pool of running instances with an instance management frameworks (e.g., SpotFleet).

SuperCloud is a system that enables application migration across different clouds [35], and it includes a subsystem (SuperCloud-Spot) used for acquiring and packing spot instances [22]. SuperCloud-Spot appears to be designed primarily for a fixed set of long-running jobs (e.g., services); but, it represents an important step toward effective VC scheduling, and we include it in our evaluations.
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 that must be kept on. This goal draws a parallel to the goal of VC schedulers, whose primary objective is to minimize the cluster's bill. But, these schedulers generally do not address the opportunities created by instance heterogeneity or price variation aspects of VC scheduling. The closest scheme to Stratus is a scheduler proposed by Knauth et al [25], which packs VMs onto physical machines based on pre-determined runtimes.

4.3.3 System design

Stratus is a VC scheduler designed to achieve cost-effective job execution on public IaaS clouds, combining a new elasticity-aware packing algorithm with a cost-aware cluster scaler that exploits cloud 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. Fig. 6 presents the architecture and key components of Stratus, and walks the reader through the lifetime of a job processed by Stratus.
Packer. We describe the on-line packing component of Stratus here, which places newly arriving tasks on to already-running instances. The Scaler, 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.

The primary objective of Stratus is to minimize the cloud bill of the VC, which is driven mostly by the amount of resource-time 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.

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 and hardware requirements). **(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).

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 $[2^{i-1}, 2^{i})$. 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.
Packer algorithm. At the beginning of a scheduling event, the packer organizes tasks and instances into

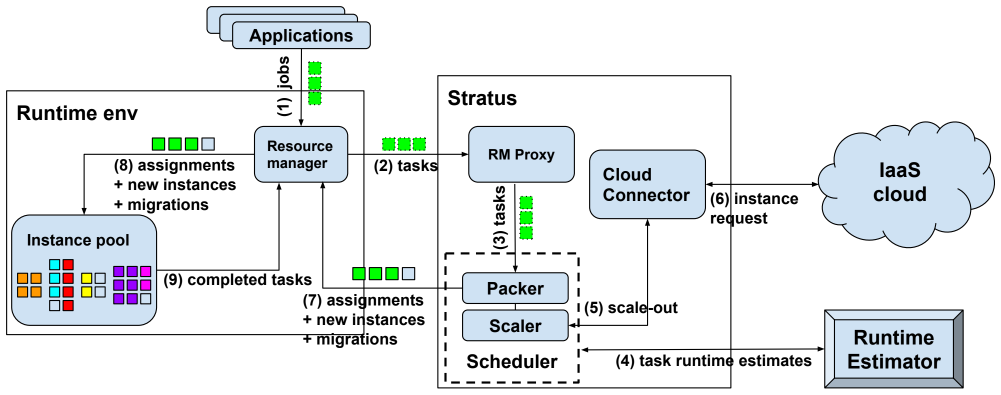
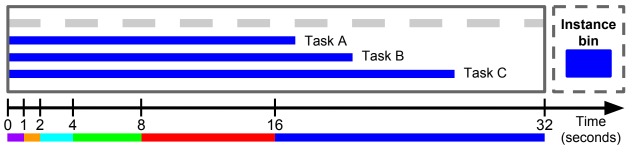


Figure 6: This figure shows the architecture of Stratus and walks the reader through the lifetime of a job. **(1)** Job requests are submitted by users and received by the Resource Manager (RM). **(2)** The RM spins off task requests from the job and dispatches them to the *RM Proxy*, which is responsible for receiving task state events (e.g., new task request) from the RM and routing them to the scheduler. **(3)** The *scheduler* consists of the *packer* and the *scaler*. 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. 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*. **(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 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 a pluggable module that acquires and terminates instances from the cloud. **(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, completion events are propagated to the RM. A job is completed when the RM receives all task completion events of the job's tasks.

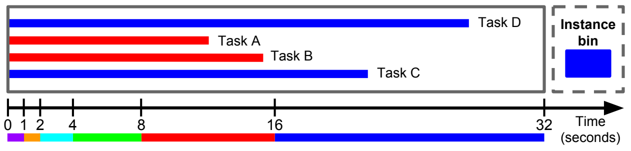
their appropriate bins. Tasks are then considered for placement in descending order by runtime—longest task first. For each task, the Packer attempts to assign it to an available instance in two phases: the *up-packing* phase and the *down-packing* phase.

We first describe the *up-packing phase*. In placing a task, the packer first looks at instances from the same runtime 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. 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. Fig. 7 shows a toy example of runtime binning in the up-packing phase on a single instance over time.

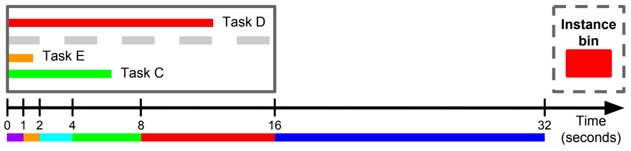
After all greater bins have been examined for VMs to schedule the task on, the Packer examines progressively *lesser* bins for a suitable VM in the *down-packing* phase, described here. If there are multiple candidate VMs from a lesser bin Stratus, like when up-packing, finds the VM with the most available 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. Promoting an instance also increases the chance of better utilizing the instance in later scheduling cycles, since tasks are always



(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 as tasks A and B finish. Task C moves down to [4, 8), and 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.

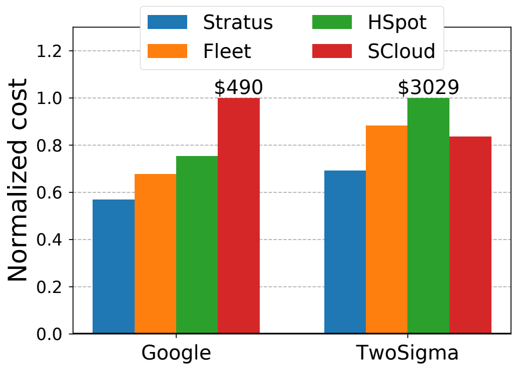
Figure 7: Toy example showing how runtime binning works with the scheduling of tasks on to an instance over time (subfig. a–c). This simple example assumes all tasks are uniformly sized, and that the instance can hold four tasks in total. The solid gray box outlines the instance. Runtime bins are color-coded (*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.

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.
Scaler. When Stratus cannot accommodate all tasks in a scheduling event, it scales out immediately and acquires new instances for 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 a new instance.

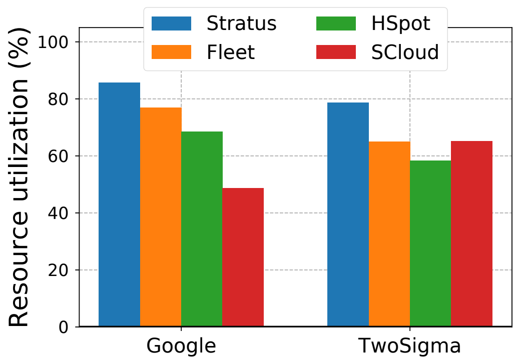
During scale-out, Stratus considers task packing options together with instance type options, seeking to achieve the most cost-efficient combination. In each iteration, it considers unscheduled tasks in each bin in descending order of runtime bins. The scaler constructs several candidate groups of tasks to be placed on the new instance. Each candidate group is assigned a *cost-efficiency score*[7] 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 (task packing and instance type selection) in tandem is crucial to achieving high cost-efficiency. Our approach balances the complexity of the large search space of combinations of task-potential instance assignments while exploring varied points in that space.

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 and continues until all tasks are scheduled.
Scaling in. Stratus terminates instances when: (1) when an instance does not have any tasks assigned to it or (2) when it continuously experiences low utilization, in which case its tasks are migrated off of it.
Runtime estimates. Runtime Estimator is the component that provides runtime estimates from a queryable task runtime estimate system for tasks submitted to Stratus. Stratus uses a modified copy of JVuPredict [38] to predict average task runtime rather than job runtime.
Handling runtime misestimates. While Stratus's use of exponentially-sized runtime bins already tolerates task runtime misestimates to a degree, we introduce two specialized heuristics to deal with larger misestimates: (1) Stratus readjusts task runtime underestimates by assuming that the task has already run for half of its runtime [18]. (2) Stratus migrates tasks away from instances that continuously experience low resource

[7]Computed as $\frac{\text{normalized used constraining resource}}{\text{instance price}}$, where the numerator is the amount of resource of the most heavily utilized resource type by percentage (specified as type r) of a hypothetical <task group, instance> assignment, divided by the amount of resource of type r available on the smallest instance that we can acquire. This is used to facilitate comparisons across <task group, instance> pairs with different constraining resource types.



(a) Average daily cost for each VC scheduler on the Google and TwoSigma workloads, normalized to the most costly option for the given trace.



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

Figure 8: The figures compare Stratus against other VC schedulers in cost and average resource utilization.

utilization[8] due to task runtime mis-alignments. For each instance chosen for migration, either all or none of its tasks are migrated, where migrated tasks are re-packed using Packer's heuristics.

4.3.4 Evaluation

This section evaluates Stratus's effectiveness, with more details available in our paper [13].

Simulator. We run simulation experiments using an event-based workload simulator to evaluate Stratus against other VC scheduling approaches. The simulator simulates instance allocation and task placement decisions made by schedulers, and considers instance spin-up delays consistent with observations on AWS. Container migration times are computed based on the container's memory footprint and a transfer rate of 160MBps for container memory [34]. To simulate the effect of spot market price movements, we use price traces provided by Amazon [8] spanning a three month period.

Instance types and regions available. We limit our experiments to use instances of the *m4* family in EC2 to avoid unknown performance comparisons among compute resources. We assume that valid instance requests are always fulfilled, and limit instance allocations to the *us-west-2* region.

VM acquisition/termination. For all evaluated schedulers, (1) instances are bid for at or above the on-demand price, and (2) are voluntarily released when no more tasks are running on it.

Workload traces. Our experiments use two traces from production clusters: the *Google trace* [30, 29] and the *TwoSigma Trace*. We filter out jobs that start before the trace start time and jobs that end after the trace end time. More details with regards to workload traces can be found in our paper [13].

Assumptions. We make the following assumptions about tasks in our simulation workloads: (1) tasks can be migrated without losing progress potentially using checkpoint-restore solutions, (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.

Alternative schedulers evaluated. Our experiments compare Stratus against a few reasonable, alternative VC scheduling solutions pieced together from related systems described in §4.3.2. Components of each VC scheduler implemented as closely as possible to its source documentation. The schedulers are described below:

(1) *HSpot* is a task-per-instance VC scheduler that implements HotSpot's migration and scaling policies, enhanced with perfect runtime knowledge. **(2)** *Fleet* is a VC scheduler that combines the most cost-efficient VM acquisition policy in AWS Spot Fleet (*lowestPrice* [4]) with the most cost-efficient packing strategy in ECS (*binpack* [1]). **(3)** *SCloud* is a VC scheduler that uses SuperCloud-Spot's greedy packing algorithm to schedule tasks on instances, enhanced with HotSpot's migration scheme and perfect task runtime knowledge.

Results: Cost savings. Fig. 8a shows the average costs of scheduling the Google and TwoSigma workloads

[8] 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 [3, 6, 7].

for each VC scheduler, normalized to the most expensive case for each trace. 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, they do not carry over well to workloads where task runtimes greatly vary. SCloud's scaling algorithm often bids for large VMs to reduce fragmentation and improve cost-per-resource at the time of packing. But, 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 at different times. Stratus reduces the cloud bill of Fleet by 17% (Google) and 22% (TwoSigma). Fleet's on-line packing is not as effective as Stratus's packing due to its use of Spot Fleet's *lowestPrice* scaling method. Fleet always acquires the cheapest (often 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 pending tasks. 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.
Results: Resource utilization. Much of Stratus's cost reduction comes from increased utilization of rented resources. Fig. 8b shows the utilization of the constraining resource (VCore in the case of the Google trace and memory for TwoSigma) for evaluated VC schedulers. Stratus attains the highest resource utilization, achieving 86% and 79% utilization 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. Stratus's selection of instance types during scale-out in light of different possible packing configurations, rather than only considering packing after selection, greatly increases utilization.

5 Proposed thesis timeline

Time	**Plan**
Nov. 2019	Submission of inter-job dependency analysis paper to EuroSys 2020
Dec. 2019 – Feb. 2020	Design and initial implementation of Talon
	Thesis proposal preparation
Feb. 2020	Thesis proposal
Feb. – Apr. 2020	Experiment design and refinement of Talon
May – Dec. 2020	Finish experiments on Talon and paper submissions (targeting OSDI 2020)
Jan. – May 2021	Dissertation writing, defense, and job search

References

[1] Amazon Elastic Container Service (2019). https://aws.amazon.com/ecs/.

[2] AWS Autoscale (2019). https://aws.amazon.com/autoscaling/.

[3] AWS EC2 (2019). http://aws.amazon.com/ec2/.

[4] AWS EC2 Spot Fleet (2019). https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-fleet.html.

[5] AWS Fargate (2019). https://aws.amazon.com/AWS/Fargate.

[6] Azure Virtual Machines (2019). https://azure.microsoft.com/en-us/services/virtual-machines/.

[7] Google Compute Engine (2019). https://cloud.google.com/compute/.

[8] Spot Instance Pricing History (2019). https://aws.amazon.com/ec2/spot/.

[9] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing amazon ec2 spot instance pricing. *ACM Trans. Econ. Comput.*, 1(3):16:1–16:20, Sept. 2013.

[10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[11] E. Boutin, J. Ekanayake, W. Lin, B. Shi, J. Zhou, Z. Qian, M. Wu, and L. 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, pages 285–300, Berkeley, CA, USA, 2014. USENIX Association.

[12] A. Chung, C. Curino, S. Krishnan, K. Karanasos, P. Garefalakis, and G. 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, pages 1889–1892, New York, NY, USA, 2019. ACM.

[13] A. Chung, J. W. Park, and G. R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 121–134, New York, NY, USA, 2018. ACM.

[14] P. Danzig, J. Mogul, V. Paxson, and M. Schwartz. The internet traffic archive. http://ita.ee.lbl.gov/, 2000.

[15] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, New York, NY, USA, 2012. ACM.

[16] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.*, 30(4), Nov. 2012.

[17] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 795–806, New York, NY, USA, 2016. ACM.

[18] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 13–24. ACM, 1996.

[19] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. 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, pages 1–13, Berkeley, CA, USA, 2018. USENIX Association.

[20] A. Harlap, A. Tumanov, A. Chung, G. R. Ganger, and P. 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, pages 589–604, New York, NY, USA, 2017. ACM.

[21] J. A. Hoxmeier and C. DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, page 347, 2000.

[22] Q. Jia, Z. Shen, W. Song, R. van Renesse, and H. Weatherspoon. Smart spot instances for the supercloud. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, page 5. ACM, 2016.

[23] S. A. Jyothi, C. Curino, I. Menache, S. Matthur Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Venkatraman Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *2016 International Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[24] K. Karanasos, S. Rao, C. Curino, C. Douglas, K. Chaliparambil, G. M. Fumarola, S. Heddaya, R. Ramakrishnan, and S. Sakalanaga. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 485–497, Berkeley, CA, USA, 2015. USENIX Association.

[25] T. Knauth and C. Fetzer. Energy-aware scheduling for infrastructure clouds. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 58–65. IEEE, 2012.

[26] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *Computer*, 40(9), 2007.

[27] Netflix Technology Blog. Creating Your Own EC2 Spot Market. https://medium.com/netflix-techblog/creating-your-own-ec2-spot-market-6dd001875f5, Apr 2017.

[28] J. W. Park, A. Tumanov, A. Jiang, M. A. Kozuch, and G. R. Ganger. 3sigma: Distribution-based cluster scheduling for runtime uncertainty. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 2:1–2:17, New York, NY, USA, 2018. ACM.

[29] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 7. ACM, 2012.

[30] C. Reiss, J. Wilkes, and J. L. Hellerstein. Google cluster-usage traces: format + schema. *Google Inc., White Paper*, pages 1–14, 2011.

[31] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 6:1–6:15, New York, NY, USA, 2016. ACM.

[32] P. Sharma, D. Irwin, and P. Shenoy. Portfolio-driven resource management for transient cloud servers. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1):5:1–5:23, June 2017.

[33] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh. A cost-aware elasticity provisioning system for the cloud. In *2011 31st International Conference on Distributed Computing Systems*, pages 559–570, June 2011.

[34] S. Shastri and D. Irwin. Hotspot: automated server hopping in cloud spot markets. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 493–505. ACM, 2017.

[35] Z. Shen, Q. Jia, G.-E. Sela, W. Song, H. Weatherspoon, and R. Van Renesse. Supercloud: A library cloud for exploiting cloud diversity. *ACM Transactions on Computer Systems (TOCS)*, 35(2):6, 2017.

[36] M. Stokely, J. Winget, E. Keyes, C. Grimes, and B. Yolken. Using a market economy to provision compute resources across planet-wide clusters. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, May 2009.

[37] H. Tian, Y. Zheng, and W. Wang. Characterizing and synthesizing task dependencies of data-parallel jobs in alibaba cloud. SoCC '19 (To appear), 2019.

[38] A. Tumanov, A. Jiang, J. W. Park, M. A. Kozuch, and G. R. Ganger. Jamaisvu: Robust scheduling with auto-estimated job runtimes. Technical report, Technical Report CMU-PDL-16-104. Carnegie Mellon University, 2016.

[39] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrisched: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 35:1–35:16, New York, NY, USA, 2016. ACM.

[40] WAND Network Research Group. Wits: Waikato internet traffic storage. http://wand.net.nz/wits/index.php.