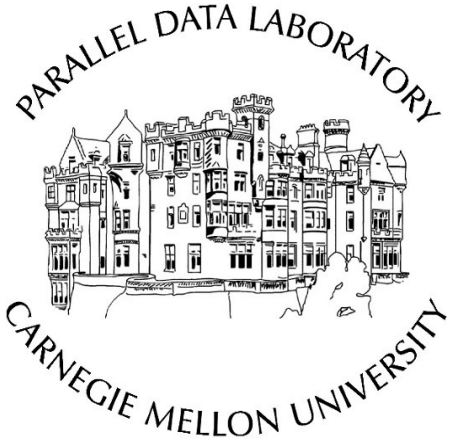


Tributary: spot-dancing for elastic services with latency SLOs

Aaron Harlap[§*], Andrew Chung[§*], Alexey Tumanov[†],
Gregory R. Ganger[§], Phillip B. Gibbons[§]

[§]*Carnegie Mellon University,* [†]*UC Berkeley*
[*]*Denotes equal contribution*

CMU-PDL-18-102
Feb 2018

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The Tributary elastic control system embraces the uncertain nature of transient cloud resources, such as AWS spot instances, to manage elastic services with latency SLOs more robustly and more cost-effectively. Such resources are available at lower cost, but with the proviso that they can be preempted en masse, making them risky to rely upon for business-critical services. Tributary creates models of preemption likelihood and exploits the partial independence among different resource offerings, selecting collections of resource allocations that will satisfy SLO requirements and adjusting them over time as client workloads change. Although Tributary's collections are often larger than required in the absence of preemptions, they are cheaper because of both lower spot costs and partial refunds for preempted resources. At the same time, the often-larger sets allow unexpected workload bursts to be handled without SLO violation. 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.

Acknowledgements: We thank Henggang Cui for all his feedback. We thank the members and companies of the PDL Consortium: Broadcom, Dell EMC, Google, HP Labs, Hitachi, IBM Researh, Intel, Microsoft Research, MongoDB, NetApp, Oracle, Salesforce, Samsung, Seagate Technology, Two Sigma, Toshiba, Veritas and Western Digital for their interest, insights, feedback, and support. This research is supported in part by Intel as part of the Intel Science and Technology Center for Visual Cloud Systems (ISTC-VCS), National Science Foundation under awards CNS-1042537, CCF-1725663, CCF-1533858, CNS-1042543 (PRObE [16]) and DARPA Grant FA87501220324.

Keywords: Big Data infrastructure, Big Learning systems

1 Introduction

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 [20, 34, 22]. 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 [23, 17, 14, 15, 31].

Elastic service scaling schemes generally assume that each of the machines used fail infrequently and independently of the others, 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 [3]). 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 [1]) at a discount of 70–80% [7], 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.

This paper describes 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, but 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 [19]. 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 AWS 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 for a short time. 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

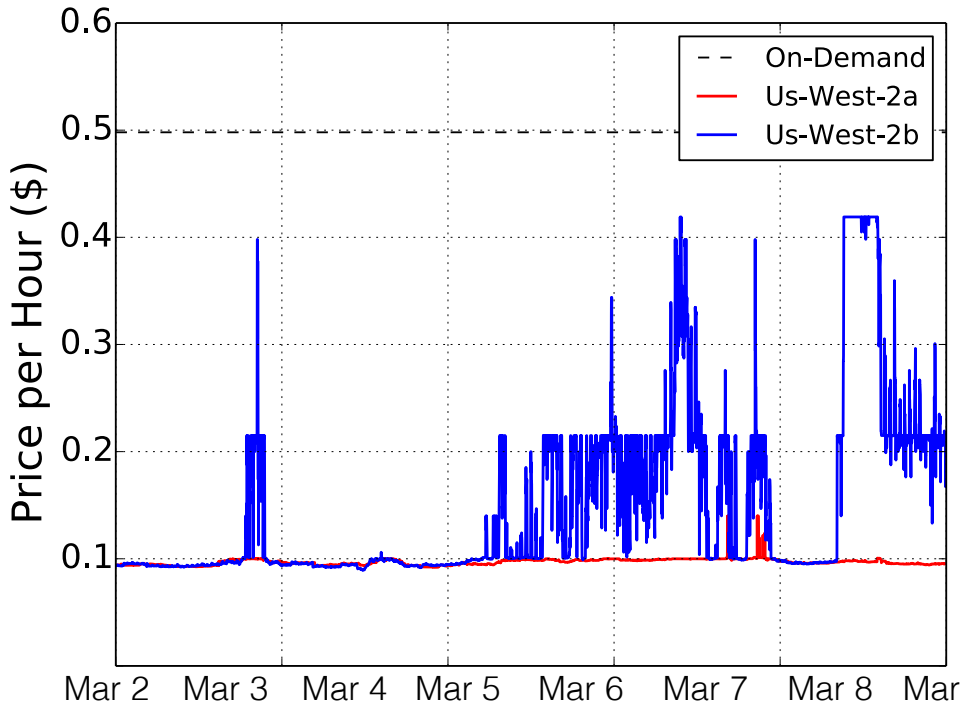


Figure 1: AWS spot prices over time. Spot prices are shown for the *c4.2xlarge* (8 VCPUs) instance type in two availability zones of the US-West-2 region for March 2-9, 2017. The unchanging on-demand price for *c4.2xlarge* instance is also shown.

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* [2]) 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 [29, 19], Tributary provides higher SLO attainment at significantly lower cost.

This paper 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 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, elastic service scaling policies, and resource acquisition strategies to put Tributary and its new approach to resource acquisition into context.

2.1 Preemptible resources in AWS EC2

In addition to non-preemptible, or *reliable* resources, most cloud infrastructures offer preemptible resources as a way to increase utilization in their datacenters. Preemptible resources are made available, on a best-effort basis, at decreased cost (in for-pay settings) and/or at lower priority (in private settings). This subsection describes preemptible resources in AWS EC2, both to provide a concrete example and because Tributary and most related work specialize to EC2 behavior.

EC2 offers "on-demand instances", which are reliable VMs billed at a flat per-second rate. EC2 also offers the same VM types as "spot instances", which are preemptible 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 (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 prior work has shown that AWS's spot markets are not truly free markets [11]. (2) Recent work [28] reports that price movements among spot markets are not always correlated, even for the same instance type in a given region (e.g., see Fig. 1). (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, the customer cannot modify it. (4) A customer owns a spot instance, and is billed the spot market price (not the bid price), 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. For spot instances, 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. (Prior to 10/2/2017, billing was per-hour, and the last partial-hour of a spot instance was refunded upon preemption.) 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 [11, 39, 36]. The most popular strategy by far is to bid the on-demand price to minimize the odds of preemption [28, 25], since AWS charges the market price rather than the bid price.

2.2 Scaling Policies for Elastic Services

Scaling policies have been extensively studied and can be broadly classified as reactive, predictive, or hybrids thereof [24]. Tributary does not innovate on the scaling policy front and has been designed to accommodate most existing scaling policies—our experiments evaluate it with three popular options. As such, we only briefly overview scaling policy options here.

Reactive scaling policies observe system metrics (e.g., CPU utilization and request rate) periodically and set the target number of resources required based on the resources required to handle the load in recent observation periods. Reactive scaling policies, as suggested by the name, do not try to predict future load. A reactive scaling policy may thus violate SLOs when client load increases faster than the policy determines that more resources are needed and additional resources are acquired and deployed.

Predictive scaling policies use past observations to infer future demand and set the target number of resources to meet the *predicted* demand. A predictive scaling policy will thus miss the target latency when its predicted request rate is lower than the actual request rate, and it will incur extra cost when its predicted request rate is higher than the actual request rate. Various techniques have been used to identify patterns and predict service loads, including signal processing [33, 17, 27], control theory [10], and ML [37, 13].

2.3 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 [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 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"[4] 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 [31] 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 [32] 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 [30] 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.

ExoSphere [29] 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 Section 5.5 that Tributary achieves lower cost, because it bids aggressively (resulting in more first-hour preemptions), and higher SLO attainment, because it explicitly predicts preemptions and selects resource sets based on sufficient tolerance of bulk preemptions.

Proteus [19] 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 Section 5.5 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.

3 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 the 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 is reaching the end of its preemption window, (3) the scaling policy specifies a change in resource requirement, and/or (4) when 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 paper, virtual CPUs (VCPUs) are the bottleneck resource; however, it is possible to optimize for memory, network bandwidth, or other resource types instead. Resource scaling characteristics are provided to AcquireMgr via a *utility function* $\upsilon()$ by the service linked to it. The *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 with respect to the number of resources given a load on the service. 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. Using a utility function allows Tributary to accommodate various scaling characteristics of services.

In addition to providing $\upsilon()$, the service also provides the SLO in terms of a percentage of requests required to meet the target latency. Upon receiving this information, AcquireMgr acquires the right amount of resources to meet SLO in expectation while optimizing for expected cost. In deciding which resources to acquire, AcquireMgr uses the prediction models described in Sec. 3.1 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 (Sec. 3.2). AcquireMgr greedily acquires allocations until the expected utility is greater than or equal to the SLO percentage requirement (Sec. 3.3).

3.1 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 [19] 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 average spot price [38], number of times the spot price has changed, magnitude of spot price fluctuations, day of week, time of day, among others. These features allow AcquireMgr to construct a more complex prediction model, leading to higher prediction accuracy (Sec. 5.6).

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 [35]. Examples of such workloads include language modeling, machine translation, and stock market prediction. Unlike traditional 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 (Sec. 5.6). For every current instance held by AcquireMgr and any instance that it is considering acquiring, AcquireMgr queries the prediction model to get the probability of the instance being preempted within the preemption window. Each query to the trained LSTM model consists of ten previous spot market time-stamps and prices and the current time-stamp and price. The prediction model returns the probability of preemption for each query.

3.2 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 1 for symbol reference.

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

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

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

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 1: Summary of parameters used by AcquireMgr

.

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:

$$C_A[a_{i,j}] = (1 - \beta_{i,j}) * P_{i,j} * k_{i,j} * t_{i,j} + \beta_{i,j} * 0 * k_{i,j} * t_{i,j}, \tag{2}$$

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:

$$V_A = \sum_{r=0}^{resc(A)} P(R = r) * \upsilon(r), \tag{3}$$

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. 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

$$P(a_{i,j}|a_{i,j-1}) = \frac{P(a_{i,j} \wedge a_{i,j-1})}{P(a_{i,j-1})} = \frac{\beta_{i,j}}{\beta_{i,j-1}}. \tag{4}$$

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

$$P(a_{i,j}|a_{i,j-1}) = \begin{cases} 0 & \text{if allocation } a_{i,j-1} \notin S, \\ \beta_{i,j}/\beta_{i,j-1} & \text{else.} \end{cases} \tag{5}$$

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:

$$\lambda_i = \gamma * \sum_{l=1}^{size(A)} \rho_{i,l} * \frac{resc(A_i) + resc(A_l)}{2 * resc(A)}. \tag{6}$$

where $\rho_{i,l}$ is the Pearson correlation between resource pools i and l, and $\gamma \in R \geq 0$ 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 $\gamma = 0.01$.

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 ($1 - P(a_{i,j}|a_{i,j-1})$).

$$\begin{aligned} P(S) = \prod_{i=1}^{size(A)} \prod_{j=1}^{size(A_i)} \Big(& d_{i,j} * P(a_{i,j}|a_{i,j-1}) \\ & + (1 - d_{i,j}) * (1 - P(a_{i,j}|a_{i,j-1})) \Big) \end{aligned} \tag{7}$$

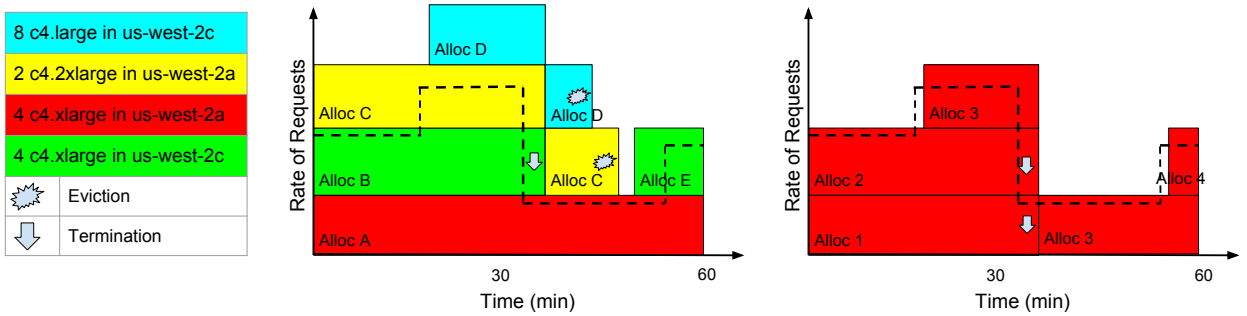
Finally, AcquireMgr formulates the probability of r resources remaining after preemption $P(R = r)$ (Eq. 3) as the sum of the probabilities of all sets S where the number of resources not preempted in S equals to r:

$$P(R = r) = \sum_{S \subseteq A, resc(S) = resc(A) - r} P(S) \tag{8}$$

which it uses to calculate the expected utility of a set of allocations A (Eq. 3).

3.3 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



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

Figure 2: 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.

acquire resources. At each decision point, AcquireMgr continues 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) and utility of the current set of allocations (Eq. 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 market 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.

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. 3). While the primary goal of these additional resources is to account for preemptions, they often have the added benefit of being able to handle unexpected increases in load. Experiments with Tributary show that these resource buffers both increase the fraction of requests meeting latency targets and decrease cost (Sec. 5.3).

3.4 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.

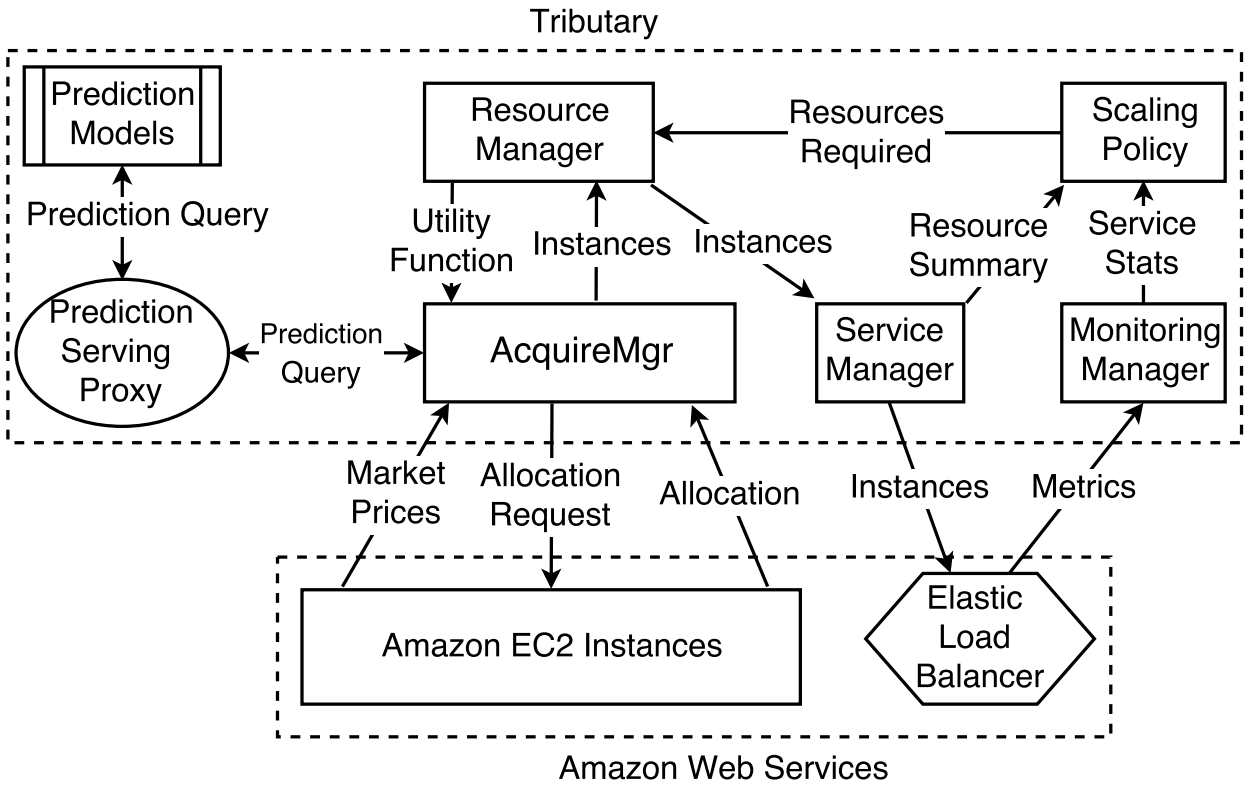


Figure 3: Tributary architecture.

3.5 Example

Fig. 2 shows how Tributary and AutoScale handle a sample workload. The example illustrates how the extra resources Tributary acquires to handle preemption events are able to handle an unexpected increase in the rate of requests and how it may be able to get resources for free by intelligently selecting allocations to terminate, thus lowering its cost.

4 Tributary Implementation

Figure 3 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* (Sec. 3.1) are trained offline using TensorFlow [9] and deployed using Tensorflow Serving [8]. 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 monitors AWS for instance preemption notifications. AcquireMgr considers modifying the resource footprint at every *decision point*, and it follows the procedure described in Sec. 3.3 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. Upon verifying that 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. Upon reaching a decision to terminate resources or noticing a preemption from AWS,[1] AcquireMgr sends a

[1]AWS does not send preemption notifications, instead AcquireMgr continuously polls AWS checking for preemption notifications.

message containing the instance IDs of the instances being removed.

Scaling Policy. The *Scaling Policy* component determines dynamic sizing of the resource target, as defined in Sec. 2.2. 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 to the *Resource Manager*.

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 each *monitoring period* and forwards them to the scaling policy. When Tributary is used with AWS EC2, the MonMgr uses AWS CloudWatch to fetch and provide metrics. Currently, the most fine-grained monitoring period supported by AWS is one minute.

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.[2] The ResMgr also receives instance allocations and termination notices from AcquireMgr and forwards them to the SvcMgr.

Service Manager (SvcMgr). The *Service Manager* manages the web service. The SvcMgr is integrated with EC2 and manages the ELB and its load balancing set. On receiving instance messages from the ResMgr, it makes API calls to ELB to manage its load balancing set accordingly. SvcMgr relays instance events as resource events to the Scaling Policy.

5 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 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) Using standard bidding policies for spot instances and configuring scaling policies to match Tributary's number of SLO latency misses with 47%–62% lower cost with Tributary; (4) Tributary outperforms state-of-the-art resource managers in running elastic services, as shown in Sec. 5.5; (5) Tributary's preemption prediction models improve accuracy significantly, resulting in 37% lower cost than previous prediction approaches.

5.1 Experimental Setup

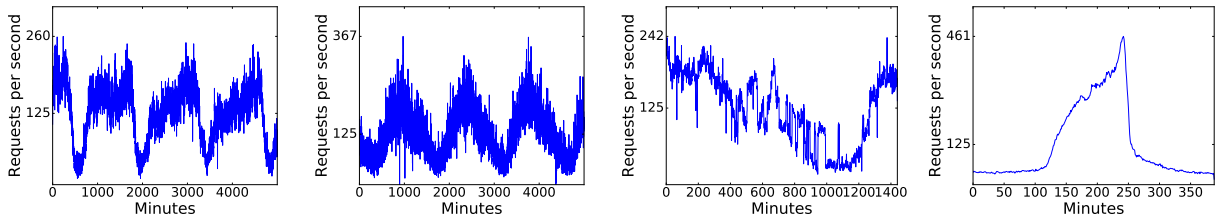
Simulated Platform. We report simulation results corresponding to use of three AWS EC2 spot instance types: *c4.large*, *c4.xlarge*, and *c4.2xlarge*. These instance types have 2, 4 and 8 VCPUs and 3.75, 7.5, and 15 GiB of memory per instance, respectively. The results correspond to the *us-west-2* region, which consists of three availability zones. Using the three instance types in each availability zone, our experiments involve nine resource pools.

Simulated Workload. The simulated workload uses a real-world trace for request arrival times, with each request consisting of the derivation of the PBKDF2 [21] 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.

Simulation 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 [26]. Network latency is configured based on a distribution of observed round-trip network latencies of HTTP requests to EC2 ELBs with various cluster sizes. The framework models a load balancer based on documentation of the EC2 Classic ELB [5] and implements a least-outstanding requests routing policy.

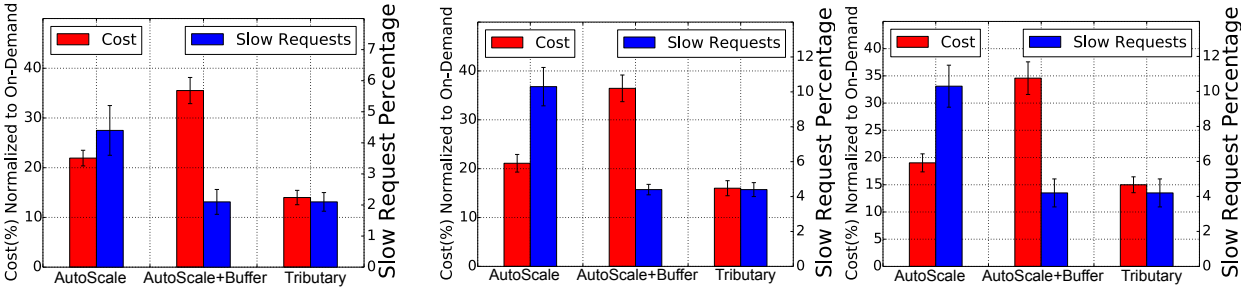
SLO and Scaling. The target service latency is set to one second, and we verified on EC2 that a VCPU

[2]The process of constructing the utility function is described in Sec. 5.2.



(a) Berkeley Periodic[12] (b) ClarkNet Periodic[12] (c) WITS Large Variation[18] (d) WCup98 Slow Spike[12]

Figure 4: Traces used in system evaluation.



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

Figure 5: Cost savings (red) and percentage of "slow" requests (blue) for the *ClarkNet* trace.

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. For less external interference on request latency caused by queueing delays, we set the ELB surge queue size to be zero. The effectiveness of Tributary is not overly sensitive to the target service latency setting.

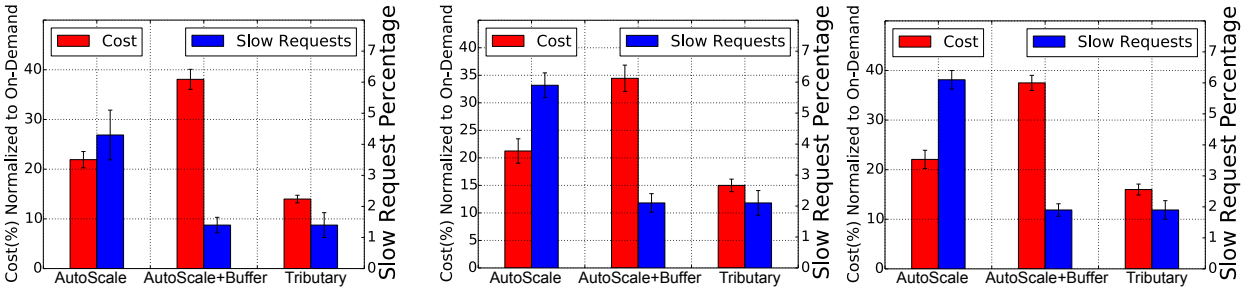
Traces. We use four real-world request arrival traces with differing characteristics, shown in Fig. 4. *Berkeley* is from the Berkeley Home IP proxy service and *ClarkNet* is from the ClarkNet ISP's HTTP servers [12]. Both exhibit a periodic, diurnal pattern. We use the first 2000 minutes of these two traces, which covers an entire period. *WITS* is from the Waikato Internet Traffic Storage (WITS) [18]. *WorldCup98* is the arrival trace of the workload on the 1998 FIFA World Cup HTTP Servers [12] on day 75 of the World Cup. The traces are scaled to have an average of 125 requests per second in order to generate sufficient load for the experiments.

5.2 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 Scaling 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. [15], to prevent premature scale-in in case the demand fluctuates widely in a short period of time. The *Predictive-MWA Scaling 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 *Predictive-LR Scaling 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 the cost of operation.



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

Figure 6: Cost savings (red) and percentage of "slow" requests (blue) for the *Berkeley* trace.

5.3 Improvements with Tributary

Here, we evaluate Tributary's ability to reduce cost and latency target misses against AutoScale with different settings.

AWS Autoscale. AWS AutoScale (Sec. 2.3) 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 [2] and integrate it with Tributary's SvcMgr to work with the more powerful scaling policies therein. From 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 scaling policy lowers resource requirements, 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 (Sec. 5.1) 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. [3]

Cost Savings and Service Latency Improvements. Fig. 5 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 resources. 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.

Fig. 6 shows cost savings and reduction in percentage of "slow" requests for the *Berkeley* trace. The

[3]The prediction models used were trained on data from 06/06/16 – 01/22/17.

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* 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, the results of which are shown in Tables 2 and 3, respectively.

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

Scaling Policy	Cost Saving	"Slow" request Reduction
Reactive	27%	55%
Predictive-LR	25%	57%
Predictive-MWA	22%	63%

Table 3: Cost and "slow" request improvements for Tributary compared to AutoScale for the *WorldCup* trace

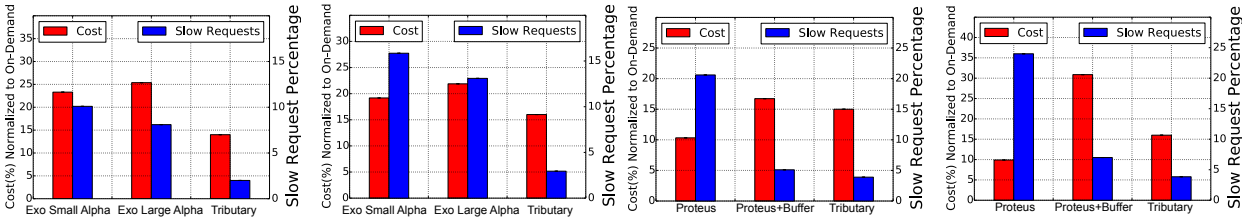
Attribution of Benefits. Tributary's superior performance arises from several factors. Much of the reduction in cost compared to AutoScale is due to Tributary's ability to get free instance hours. *Free instance hours* occur when an allocation does useful work but is preempted by AWS before the end of 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. 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 (Sec. 3.3). 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.

5.4 Risk Mitigation

A key feature of Tributary is that it encourages instance diversification, *i.e.*, acquiring instances from mostly independent resource pools, as described in Sec. 3.2. The default AutoScale policy is the lowest-price policy, which does not take diversification into account when acquiring instances; instead, it acquires the instance that is cheapest at the time of acquisition. An example of this is illustrated in Fig. 2. 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, and preemptions of allocations within a single instance spot market are highly correlated. 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. 2c was preempted prior to minute 35).



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

Figure 7: Comparing to ExoSphere and Proteus. Predictive-MWA results not shown but similar.

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.[4] Since Amazon charges users the market price and not the bid price, it is reasonable to predict that at some point in the future, Amazon will once again begin preempting instances bidding the on-demand price with regularity. Indeed, we recently observed this behavior in the *us-east* availability zones. Unlike Tributary's approach, AutoScale's resource acquisition approach is high-risk when using spot machines for a service with latency SLOs.

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 [4]. Acquiring instances from different spot markets reduces risk of losing instances to preemptions, 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.

5.5 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 Comparison. We implemented ExoSphere's allocation strategy, described in Sec. 2.3, 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 did 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. 7 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.[5] 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 as explained in Sec. 5.3.

Proteus Comparison. We implemented Proteus's allocation strategy, described in Sec. 2.3, modified to acquire only spot resources (reducing cost with no significant change in SLO attainment). Fig. 7 compares Tributary and Proteus for the *ClarkNet* trace, for two different scaling policies. While Proteus achieves lower

[4]From 01/23/17 to 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.

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

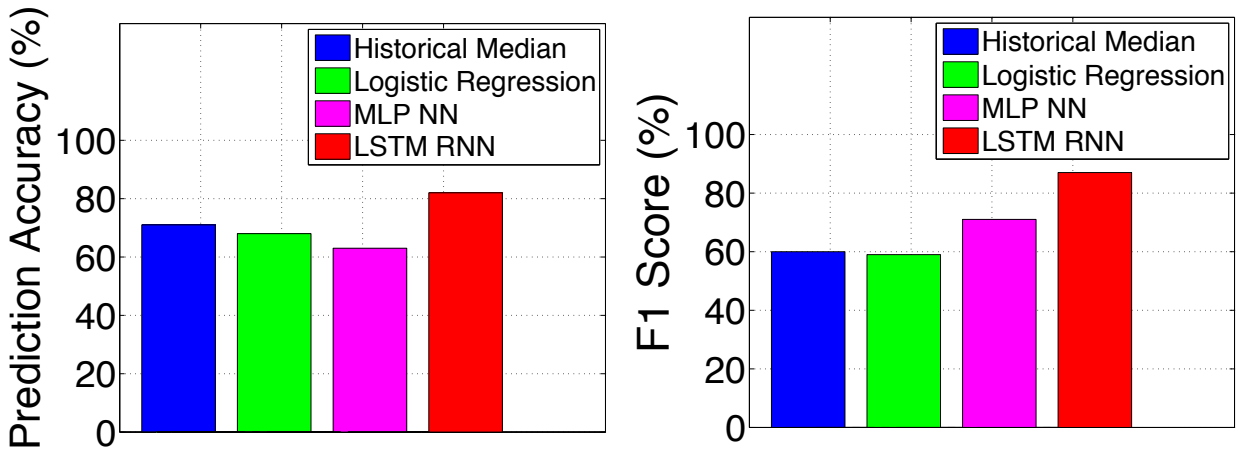


Figure 8: Prediction model accuracies and F_1 scores (accounts for data skew) for predicting preemption of AWS spot instances. The LSTM recurrent neural network outperforms prior techniques (blue bar) by 11% on the accuracy metric and 27% on the F_1 score metric.

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 (Sec. 5.3), Proteus is unable to match Tributary's number 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.

5.6 Prediction Model Evaluations

This section evaluates the accuracy of the preemption prediction models used by Tributary, which are described in Sec. 3.1. The recent Proteus system [19] 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. 8 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 the *us-west-2* region.

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 when calculating accuracy, are calculated using recall and precision: $F_1 = 2 * (recall * precision)/(recall + precision)$. *Recall* is the number of instances that were correctly identified (true positives) as preempted divided by the number instances that were actually preempted. *Precision* is the number of instances that were correctly identified as preempted divided by the number of instances that the model predicted to be preempted. F_1 is a good accuracy measurement because the data set is skewed toward preemptions at lower *bid deltas* and non-preemptions at higher *bid deltas*. The F_1 scores are able to account for this when evaluating the prediction models.

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 Sec. 3.1. 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.

6 Conclusion

Tributary exploits AWS spot instances to meet a latency SLO for an elastic service. 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 47–62% relative to traditional high-risk use of spot instances.

Although Tributary exploits AWS properties, such as billing with partial-hour refunds for preemptions, we believe its approach would also work for other clouds offering preemptible resources, if they expose a little more information. The most important requirement is some signal that can be used to predict preemption probabilities, which AWS provides via the visible spot market prices. Currently, *Google Cloud Engine* [6] does not expose such a signal for its preemptible instances. For private clouds, exposing preemption logs could provide the historical view, but even better predictions would likely be possible if scheduler state were exposed.

References

[1] Amazon EC2 Spot Instances. https://aws.amazon.com/ec2/spot.

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

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

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

[5] Classic Load Balancer. https://aws.amazon.com/elasticloadbalancing/classicloadbalancer/.

[6] Google Compute Engine. https://cloud.google.com/compute/.

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

[8] Tensorflow serving. https://tensorflow.github.io/serving.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI 16)*.

[10] T. Abdelzaher, Y. Lu, R. Zhang, and D. Henriksson. Practical application of control theory to web services. In *Proceedings of the 2004 American Control Conference*, volume 3, pages 1992–1997 vol.3, June 2004.

[11] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir. Deconstructing Amazon EC2 spot instance pricing. *ACM Transactions on Economics and Computation*, 1(3):16, 2013.

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

[13] X. Dutreilh, S. Kirgizov, O. Melekhova, J. Malenfant, N. Rivierre, and I. Truck. Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow. In *ICAS 2011, The Seventh International Conference on Autonomic and Autonomous Systems*, pages 67–74, 2011.

[14] G. Galante and L. C. E. d. Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, UCC '12, pages 263–270, Washington, DC, USA, 2012. IEEE Computer Society.

[15] 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):14:1–14:26, Nov. 2012.

[16] G. Gibson, G. Grider, A. Jacobson, and W. Lloyd. Probe: A thousand-node experimental cluster for computer systems research. *USENIX*, 38(3), 2013.

[17] Z. Gong, X. Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *6th IEEE/IFIP International Conference on Network and Service Management (CNSM 2010)*, Niagara Falls, Canada, 2010.

[18] W. N. R. Group et al. Wits: Waikato internet traffic storage. *URL: http://wand.net.nz/wits/index.php*.

[19] 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.

[20] 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.

[21] B. Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. 2000.

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

[23] H. Liu and S. Wee. *Web Server Farm in the Cloud: Performance Evaluation and Dynamic Architecture*, pages 369–380. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[24] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing*, 12(4):559–592, 2014.

[25] A. Marathe, R. Harris, D. Lowenthal, B. R. De Supinski, B. Rountree, and M. 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*, pages 279–290. ACM, 2014.

[26] K. Muller and T. Vignaux. Simpy: Simulating systems in python. *ONLamp. com Python Devcenter*, 2003.

[27] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *ICAC*, pages 69–82, 2013.

[28] P. Sharma, T. Guo, X. He, D. Irwin, and P. Shenoy. Flint: Batch-interactive data-intensive processing on transient servers. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys 16)*, page 6. ACM, 2016.

[29] P. Sharma, D. Irwin, and P. Shenoy. Portfolio-driven resource management for transient cloud servers. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 1(1):5, 2017.

[30] P. Sharma, S. Lee, T. Guo, D. Irwin, and P. Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, page 16. ACM, 2015.

[31] 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.

[32] S. Shastri and D. Irwin. Hotspot: Automated server hopping in cloud spot markets. 2017.

[33] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.

[34] S. Souders. Velocity and the bottom line. In *Velocity (Web Performance and Operations Conference)*, 2009.

[35] M. Sundermeyer, R. Schlüter, and H. Ney. LSTM neural networks for language modeling. In *Interspeech*, pages 194–197, 2012.

[36] S. Tang, J. Yuan, and X.-Y. Li. Towards optimal bidding strategy for Amazon EC2 cloud spot instance. In *Proceedings of the 5th IEEE International Conference on Cloud Computing(CLOUD 12)*, pages 91–98. IEEE, 2012.

[37] M. Wajahat, A. Gandhi, A. Karve, and A. Kochut. Using machine learning for black-box autoscaling. In *Green and Sustainable Computing Conference (IGSC0¡ 2016 Seventh International*, pages 1–8. IEEE, 2016.

[38] C. Wang, Q. Liang, and B. Urgaonkar. An empirical analysis of Amazon EC2 spot instance features affecting cost-effective resource procurement. In *ICPE*, 2017.

[39] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang. How to bid the cloud. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 71–84. ACM, 2015.