Optimal Resource Consumption with an Application to Cloud Computing via Data-Driven Prophet Inequalities

Muhammad J. Amjad
Operations Research Center
Massachusetts Institute of Technology
email: mamjad@mit.edu

Vivek F. Farias
Sloan School of Management
Massachusetts Institute of Technology
email: vivekf@mit.edu

Andrew A. Li
Operations Research Center
Massachusetts Institute of Technology
email: aali@mit.edu

Devavrat Shah
Electrical Engineering & Computer Science
Massachusetts Institute of Technology
email: devavrat@mit.edu

June 16, 2017

Abstract

The use of virtualized cloud infrastructure has transformed the nature and scope of enterprise computing. A salient challenge for cloud users today is to minimize the cost of completing computational tasks, and the recent introduction of market-driven pricing by major cloud providers such as Amazon and Google only intensifies the potential for cost savings through efficient, intelligent resource provisioning. We model this problem as a k-choice consumption problem and introduce a simple, new approach to modeling uncertainty – the Data-Driven Prophet Model – that treads the line between stochastic and adversarial modeling, and is amenable to the very common situation where stochastic modeling is challenging, despite the availability of copious historical data. We propose a simple, scalable algorithm that is shown to be order-optimal in this setting. We deploy this algorithm in open-source software that allows buyers to achieve a dramatic reduction in costs by taking advantage of spot markets. Experiments using our software tool on Amazon's cloud platform show cost reductions of up to an order of magnitude over the current practice approach.

Keywords: cloud computing, resource consumption, prophet inequality, adversarial models, optimal stopping, data-driven optimization

1. Introduction

The notion of the *cloud* is now part of the typical consumer's lexicon. The impact of the cloud, however, is perhaps currently felt strongest in the world of enterprise computing, where an estimated 80% of organizations use the cloud (RightScale (2016)), and an estimated $734 billion will be spent in 2016 (Gartner (2016)). Indeed, the use of virtualized cloud infrastructure has transformed the nature and scope of enterprise computing. The nature of this change is perhaps best explained with a few examples that will make the goal of this work more precise.

Batch Computation in the Enterprise: Consider the online commerce website Walmart.com. This website receives well in excess of 10 million unique visitors every month (Quantcast (2015)), many of whom are likely to be existing Walmart customers. Walmart collects data on the preferences of these customers over time with the intent of *personalizing* the customer experience. Personalization is seen as a critical area of competence for any e-commerce property. Personalization must also be *timely*, so that a customer's searches and browsing history on Walmart.com today, impact her experience and what is shown to her on the website tomorrow. Walmart.com is not unique in this regard – indeed, the need for such personalization is broad and felt across a swathe of other retailers, as well as outside of retail – for instance, the same problem applies to a service such as Netflix.

The typical approach to this problem is as follows: Customer preferences are predicted via the use of an approach such as collaborative filtering (Sarwar et al. (2001)) or via a 'Personalized PageRank' (Bahmani et al. (2010)). The key feature of this computation is that it is divisible, or parallelizable, across customers. As such, a retailer or service that benefits from personalization will, at regular intervals (say, every night), use incremental data collected on each customer to update its predictions of that customer's preferences using one of the aforementioned algorithms. Naturally, this computational task must be executed relatively quickly to have impact.

Batch computational tasks like this make up a significant portion of computing today, beyond just personalization. Other examples include simulations for scientific computing, post-processing of financial transactions for services like PayPal, and analysis of large data sets. Thanks to the divisibility of the computation, the task can be spread across a large fleet of machines and executed in parallel. This is particularly well-suited for commercial cloud infrastructures, where computing resources can be rented in real time and at low cost, so much so that even large retailers and service providers such as Netflix will rent the resources for such a computation: *they will rent cloud infrastructure as opposed to investing in their own data centers and dedicated compute infrastructure.*

The Switch to Cloud Infrastructure: The mechanics of provisioning computing resources in the cloud are incredibly simple which perhaps explains the rapid adoption of these services, even in environments where in-house infrastructure is available. The typical public cloud provider offers computing resources, in the form of virtual machines, for rent by the hour. Specifically, a user renting a specific type of computing resource would pay a fixed fee per hour of use. In a nutshell, these machines, which we will call *on-demand* machines, can be rented anytime at a fixed price that does not vary over time.

Another option for cloud buyers is the offering of dynamically priced resources that major cloud providers have experimented with over the last few years. The most prevalent example of this are *spot* machines offered by Amazon Web Services (AWS; the largest public cloud infrastructure

provider). In contrast to the traditional fixed price model for on-demand machines, the prices of spot machines are highly dynamic, presumably functions of supply and demand. The precise mechanism through which spot machines are priced and rented is distinct and will be discussed in detail in Section 6. For now, we can take spot machines to be available any time at a price that evolves randomly over time.

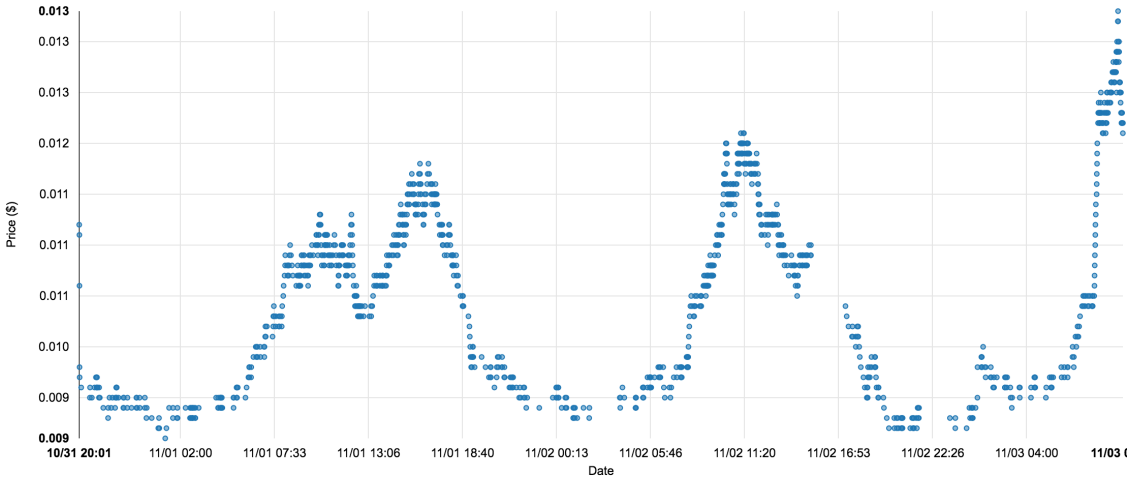


Figure 1: The variations in price for one type ('m3.medium') of spot machine on AWS, depicted over a three day period in 2016. The fixed on-demand price for the same type of machine was \$0.067, significantly higher than the spot price. Figure taken from `https://ec2price.com/`

Spot machines present a unique opportunity and challenge for users. The opportunity for significant cost savings exists because while spot prices are extremely volatile, they are on average lower, and often *much* lower, than the corresponding fixed, on-demand prices; see Figure 1. Not surprisingly, many large enterprise users already leverage spot machines to significantly reduce computation costs; for example, the ridesharing company Lyft, which uses public cloud infrastructure to run tests at massive scale before deploying updates, was able to save 75 percent on its computation cost by switching from on-demand instances to spot instances (Amazon (2016b)).

The challenge is that any computation generally comes with a deadline, and so a natural dynamic optimization problem arises wherein cloud users must acquire resources dynamically to minimize cost within a fixed time period. This problem involves many complicated features unique to the cloud setting, such as bidding and machine cancellation, but we will see that the core problem reduces to a classic consumption/allocation problem: *given a sequence of T prices that arrives sequentially, select exactly K of the prices so as to minimize the sum of the selected prices.* This problem will be the main theoretical study of our work, and in fact, the problem has many applications beyond cloud computing, such as automated mechanism design (Hajiaghayi et al. (2007)).

1.1. Our Contributions

The main contributions of our work can be summarized as follows:

Modeling Spot Prices: Our first contribution is to define a novel way of modeling uncertainty that is well-suited to spot machine prices. We show that the natural approach of modeling and fitting a stochastic process is not appropriate in many settings including our cloud problem. This motivates an adversarial approach, the most natural being a model typically studied in the area of Prophet Inequalities. The 'prophet' model is almost ideally suited to spot prices: it allows for arbitrary autocorrelation structures (a property not shared by other popular adversarial models). Moreover, our allocation problem has been well-studied under this model, and by now the story of so-called K-choice prophet inequalities is fairly complete: Hajiaghayi et al. (2007) first showed that the best achievable competitive ratio for the problem is $1 - O(1/\sqrt{K})$, and more recently, Alaei (2014) demonstrated that a simple policy achieves this theoretical upper bound.

Unfortunately, the prophet model assumes too much knowledge of the buyer: the buyer is assumed to know the marginal price distribution exactly. In reality, one almost certainly does not have this information, but instead has data in the form of past prices. Thus motivated, we propose a data-centric generalization of the prophet model, dubbed the *Data-Driven Prophet Model*, that still allows for the same freedom in autocorrelation structure, but only assumes access to historical data. The model naturally captures the relationship between past and future prices, and is parameterized by the amount of historical data so that it interpolates between having no knowledge whatsoever (zero data) and the original prophet model (infinite data).

Order-Optimal Threshold Policy: Our second contribution is to propose a policy for our allocation problem under the data-driven prophet model. The policy we propose is a simple threshold policy that amounts to selecting prices that fall below a threshold calculated from historical data. The simplicity of this policy yields immediate advantages for implementation: the policy is easily coded into lightweight software, is scalable to large numbers of machine types and amounts of historical price data, and requires almost no calibration. Moreover, as we discuss in Section 6, the precise bidding mechanism through which spot machines are rented precludes the implementation of more complicated policies.

We analyze our policy's performance and derive a strong performance guarantee. In particular, our guarantee reveals that, with 'enough' data, the performance of our policy is as strong as that of optimal policies for the regular prophet model which assume *full information*. We will quantify precisely how much is 'enough' data, but essentially it suffices to have as much data as the horizon on which the computation is being completed. This yields an immediate insight that we use in our software implementation. We also prove a corresponding upper bound that shows that our guarantee is order-optimal.

Software Implementation and Experiments: Our third and final contribution is the implementation of a software tool, called *OptScale*, that automatically completes deadline-constrained batch computations through the cloud. This load balancing and auto scaling tool connects to AWS and uses a combination of on-demand and spot machines to complete computational tasks by user-specified deadlines, all while using logic driven by our threshold policy to minimize cost. As part of this project, we will *open-source version 1.0 of our tool and make it publicly available to be readily usable with Amazon AWS.*

In both live and offline experiments over six months, we test our software tool on a Personalized PageRank application. These experiments highlight the massive savings that are possible through intelligent provisioning of machines: our tool consistently reduces cost over standard on-demand machines by an order of magnitude, and moreover, it incurs no more than 21% additional cost compared to a *clairvoyant* optimal policy (i.e. a policy that knows the future prices).

The remainder of this paper proceeds as follows. We conclude this section with a summary of related work. We then begin in Section 2 by formulating a more complete version of our cloud computing problem and reducing it to a simple consumption problem. In Section 3, we analyze potential approaches to modeling spot prices and ultimately propose our Data-Driven Prophet Model. Our threshold policy and theoretical guarantees are presented in Section 4; proofs of these guarantees are split between Section 5 and the Appendix. Details about our software implementation are given in Section 6, and experiments utilizing this software are summarized in Section 7. Section 8 concludes the paper.

1.2. Summary of Related Work

Our work nicely settles into two broad areas of research – one that addresses cloud problems similar to the one we have described, and the other that develops the various methodologies that we extend and apply.

Cloud Setting: Computation on cloud infrastructures has received significant attention in recent years. In practice, there are a variety of cloud resource providers, including Amazon Web Services, Microsoft Azure, IBM Cloud, Google Cloud Platform, VMWare vCloud, and Rackspace. Support for executing compute jobs is offered directly by cloud providers, like AWS Elastic Load Balancing (Amazon (2016a)) and Microsoft Azure Batch (Microsoft (2016)), and by independent solution providers like Batchly[1], Optispotter[2], and RightScale[3].

The introduction of market-priced spot resources has presented important avenues of investigation. Spot price traces have been studied, e.g. by Javadi et al. (2011) and Agmon Ben-Yehuda

[1] `http://www.batchly.net/`
[2] `http://www.optispotter.com/`
[3] `https://www.rightscale.com/`

et al. (2013), with attempts to model spot prices as stochastic processes. Similar to our batch computation problem, there have been a number of works Chaisiri et al. (2011), Li et al. (2014), Menache et al. (2014), Song et al. (2012), Tang et al. (2012), Zhao et al. (2012) that address the use of different machines and efficient scheduling to lower the cost of computation, along with important systems considerations such as startup times for virtual machines Mao and Humphrey (2012) and efficient execution of MapReduce jobs Isard et al. (2009), Palanisamy et al. (2013). Our work is distinguished by our data-driven adversarial approach that uniquely allows for simple policies that do not require calibration, and enjoy provable performance guarantees.

Finally, the problem of market design in the rapidly growing cloud space is becoming increasingly important. From the standpoint of cloud providers, spot markets have inspired important questions on how best to allocate cloud resources. Most of the work has been focused on design problems, including work by Zhang et al. (2011), Jain et al. (2012, 2014), Xu and Li (2012) and Zaman and Grosu (2013), with the goal of revenue or welfare maximization.

Methodology: Our data-driven prophet model is inspired by the area of prophet inequalities, which is concerned with algorithms and guarantees for a broad class of optimal-stopping problems. Prophet inequalities originated in the seminal works by Krengel et al. (1977), Krengel and Sucheston (1978), which spawned a rich area of research into numerous variations and extensions; see Hill and Kertz (1992) for a nice survey of the many classical results. Research on prophet inequalities has seen a resurgence recently due to its link with algorithmic mechanism design: Hajiaghayi et al. (2007) first noted a direct correspondence between prophet inequalities and online auctions. This has led to work most closely associated to our cloud problem involving so-called K-choice prophet inequalities, particularly Alaei (2014) and Kleinberg and Weinberg (2012).

The goal of this work is to consider these same problems in the setting where we have data as opposed to the full distributional information assumed in prior work. One work that is in this spirit is that of Azar et al. (2014), who study the same K-choice allocation problem in the setting when, rather than knowing full distributional information, we have a fixed number of samples from this distribution. One important distinction from our work is that our policy and all of our theoretical results apply to any amount of data, as opposed to the fixed amount considered there – the amount of data needed to achieve meaningful results is a central topic here.

Our buyer's problem is also loosely related to the area of job scheduling, and in fact this has been taken as an approach to reducing costs on cloud machines Azar et al. (2013), Mao and Humphrey (2011), Yao et al. (2014). However, in our setting, jobs are independent and small enough that they can be treated as a fluid workload, and therefore scheduling is not as relevant to cost savings as machine provisioning costs.

In the context of adversarial models, our allocation problem has been studied under another,

more restrictive adversarial model called the random permutation model. Under this model, a special case our problem is the classical secretary problem (Lindley (1961), Dynkin (1963)), and the more recently studied K-secretary problem (Babaioff et al. (2008), Kleinberg (2005)). The random permutation model has gained significant attention beyond the secretary problem recently, moving to online packing (Buchbinder and Naor (2009), Feldman et al. (2009), Molinaro and Ravi (2013)), online matching and the adwords problem (Devanur and Hayes (2009), Feldman et al. (2009), Goel and Mehta (2008)), and general online optimization (Agrawal et al. (2014)).

2. Problem Formulation

To begin, we briefly establish the nomenclature related to cloud computing that we will use for the remainder of this work. We will assume that a cloud provider offers virtual machines, or simply *machines*, for rent by the hour. These machines are of different *types* – varying along many dimensions, the most important of which include the number of cores (which can be thought of, loosely, as related to the extent of parallelism the machine can support), and the amount of main memory or RAM (which can be thought of, loosely, as constraining the scale of an individual computation). For example, Figure 2 shows the specifications of four different machine types offered by AWS. Any type of machine is offered both as *on-demand* machines and *spot* machines: on-demand machines maintain a fixed price, e.g. Figure 2 shows the on-demand prices, and separately the same machine types are offered with varying spot prices.

	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux/UNIX Usage
m3.medium	1	3	3.75	1 x 4 SSD	$0.067 per Hour
m3.large	2	6.5	7.5	1 x 32 SSD	$0.133 per Hour
m3.xlarge	4	13	15	2 x 40 SSD	$0.266 per Hour
m3.2xlarge	8	26	30	2 x 80 SSD	$0.532 per Hour

Figure 2: Specifications and on-demand prices of four types of machines on AWS. This figure is not exhaustive – at the time of writing, AWS offers at least 50 types of machines.

Let us now formalize our cloud computation problem as a dynamic optimization problem: we have K indivisible tasks that may be performed in parallel and must be performed over at most T periods (denoted $t = 1, \ldots, T$). To do this, we can rent machines of M distinct types (denoted $m = 1, \ldots, M$), where machine type m is capable of performing k_m tasks over a single period. The parameters k_m, depend on the characteristics of the machine types and the nature of the tasks. We are assuming that the K tasks are computationally identical, as is the case in most batch jobs,

and that the user is able to accurately measure or estimate k_m – this estimation is typically done by benchmarking each machine type with a small number of tasks.

The only constraint we face, besides the requirement of completing the tasks within T periods, is a rental limit that is imposed by most large cloud providers. Specifically, for each machine type m, no more than c machines can be concurrently rented over a given period. Note that all of the values so far, (K, T, M, k_m, c), are problem-specific constants that are known to us.

The spot price of machine type m in period t is denoted X_t^m – this is the key source of uncertainty. At the start of each period t, having observed the current price vector $X_t = (X_t^1, \ldots, X_t^M)$, we choose the number of machines of each type m, $x_t^m \in \{0, \ldots, c_m\}$, we will use over the subsequent period. Then a total of $\sum_m k_m x_t^m$ tasks are completed in period t at a cost of $\sum_m X_t^m x_t^m$. Our problem then, is to *allocate each of the K tasks to machines over time so as to minimize the expected cost incurred while guaranteeing completion of all tasks within T periods:*

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & \sum_{m,t} X_t^m x_t^m \\
\text{subject to} \quad & \sum_{m,t} k_m x_t^m \geq K, \\
& x_t^m \in \{0, \ldots, c\}.
\end{aligned} \tag{1}
$$

The problem above will be the main focus of this paper. Note that the problem, as it stands now, does not capture some of the specific challenges of the cloud computing market that we alluded to in the previous section, for example spot instance bidding and the possibility of machine cancellation or failure. We will address these issues in later sections and in our software tool, but for now note that in terms of cost-efficiency, these issues turn out to be much less important than identifying low prices as they arrive.

While the focus of this paper is on the problem of cost-efficient computing for cloud buyers, the problem we consider is also more broadly a resource allocation problem that has many applications. We have formulated problem (1) in terms of minimizing cost, but consider a closely-related formulation:

$$
\begin{aligned}
\underset{x}{\text{maximize}} \quad & \sum_{m,t} X_t^m x_t^m \\
\text{subject to} \quad & \sum_{m,t} k_m x_t^m \leq K, \\
& x_t^m \in \{0, \ldots, c\}.
\end{aligned} \tag{2}
$$

Problem (2) is a well-studied resource allocation problem, and our algorithm and guarantee will

apply to the same; note that (2) is not the dual of (1). For example, imagine we have K indivisible goods to allocate to M different streams of agents that arrive over time; letting $k_m = 1$ for all m (we will refer to this assumption later on as the *homogeneous* setting), X_m^t may represent the utility or revenue of allocating a good to an agent from stream m at time period t, and the problem is to maximize total utility or revenue. The single stream problem ($M = 1$) is a well-studied problem in online mechanism design (Hajiaghayi et al. (2007)), and our algorithm later on will correspond to a posted-price mechanism for the problem.

For the remainder of this work, unless otherwise stated, we will assume the machines are *homogeneous*, meaning $k_m = k$ for all m. This assumption is fairly harmless, since providers currently offer machines with identical capabilities as different products based on the data center or physical region they are housed in (e.g. machines on AWS are available from separate 'regions' and 'availability zones' within each region); all of these identical products are available to users and they have different prices. Furthermore, providers generally offer a suite of resources with exponentially different computing capabilities, i.e. a given machine type is twice as powerful as the next weakest type and half as powerful as the next strongest type. This means for a given set of tasks, there is often a single machine type that is ideally suited for the job.

With homogeneous machines, we will rescale k and c to be 1. This simply amounts to batching the tasks so that K instead represents the number of periods it takes the maximum amount of a single machine type to complete the entire workload. Then the problem above can be restated quite simply as follows: *given a sequence of price vectors $X_1, \ldots, X_T$, select exactly K of the component prices so as to minimize the sum of the selected prices.*

To conclude this section, we point out some challenges that have already emerged:

- High-dimensionality: This is the de rigueur challenge one might expect from such a problem. In particular, even if we assumed a very simple model for prices (an auto-regressive process, for instance), the problem at hand is an $O(M)$ dimensional dynamic program. M is large in practice: for example, on AWS, accounting for the various 'availability zones' and 'regions', there are tens of potential machine types that we might want to consider, so the dynamic optimization problem has dimension at least in the low tens leading to an impractically large state space.

- Need for simplicity: Any approach to solving this problem would eventually need to be embedded into software called an 'auto scaler'. This software must be lightweight, so that even a task such as solving an LP could prove onerous. Further, the ideal approach would be 'transparent' to users.

In addition to these, is the challenge of modeling prices. So far, we have treated the price vectors X_t as random, but have not specified how exactly the prices are modeled. This challenge will turn

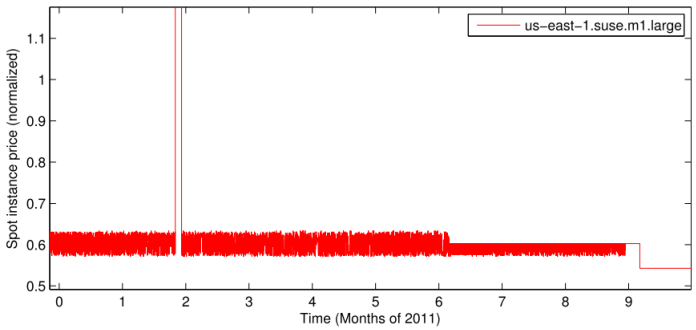


Figure 3: The futility of modeling prices as stochastic processes (reproduced from Ben-Yehuda et al. (2013))

out to be, from a practical perspective, substantially tricker than either of those above. This is the subject of the next section.

3. Modeling Prices with the Data-Driven Prophet Model

The key prerequisite to solving the cloud buyer's problem is a realistic approach to modeling prices. Perhaps the most natural approach would be to model prices via a stochastic process. Such a procedure would typically proceed in two steps: (1) we would specify a class of stochastic processes, and then (2) use historical data to select a model from within this class. This procedure already presents significant difficulty. First, it is unclear that any such calibration would be 'stable' – for instance, providers often make changes to the available capacity, and such changes are unannounced and have a non-trivial impact on prices. Moreover, while the second step is already a nontrivial task, in our context even the first step presents a formidable challenge in real world settings. As evidence of this, consider that in recent work by Ben-Yehuda et al. (2013), an attempt was made to model spot prices on AWS as an auto-regressive process, which was apparently an excellent fit to historical data. However, in a revision to this work with fresher data, it was found that in fact *the autocorrelation structure of the prices had changed dramatically* and that auto-regressive processes no longer provided a good fit (see Figure 3). This sort of evidence makes it very difficult to propose traditional stochastic process models as our uncertainty primitive for cloud prices.

3.1. Adversarial Models

The futility of attempting to model prices with a specific stochastic process motivates the second traditional approach: an adversarial analysis. The adversarial approach in general is to introduce an adversary that chooses prices with the goal of maximizing our total price paid (i.e. in complete opposition of our own objective). The adversary knows our algorithm prior to choosing the prices,

but is subject to some restriction in this choice. We then evaluate our algorithm by its performance against the adversary, and any performance guarantee that we show applies within the entire class of processes available to the adversary.

One popular adversarial model is the *Secretary model*. This originally appeared in the classical Secretary problem (Babaioff et al. (2008), Kleinberg (2005)), and is now well studied in the literature for online optimization (Agrawal et al. (2014), Devanur and Hayes (2009), Feldman et al. (2010)), sometimes under the name 'random permutation model'. Adapted to our setting, the precise price model is as follows: there are T vectors of prices (each of size M) chosen by the adversary without restriction, and unknown to the algorithm. Nature then takes a random permutation of these price vectors, such that each permutation is equally likely, and this permutation is given to the algorithm in that order.

Even though the adversary is free to choose any prices, online algorithms are able to perform well, essentially by leveraging the random permutation assumption to be able to learn about the values online. Unfortunately, the random permutation is quite a restrictive modeling assumption, particularly for modeling price behavior. In fact, one is hard pressed to find any reasonable stochastic price process that can be modeled by a random permutation of values, as e.g. most price processes exhibit strong correlations over time.

Another common adversarial model is the *Prophet model*, usually studied in the context of prophet inequalities. While there is some variation in the literature, the model we consider here is as follows. The adversary chooses a multivariate distribution over M variables, and this distribution is known to the algorithm. Nature draws T vectors of prices independently from this distribution, and the adversary chooses the order in which to give these values to the algorithm.

Here, there are also algorithms that perform well, as they are able to use the distributional information effectively. The Prophet model offers a different form of flexibility to the adversary than the Secretary model, as the adversary is allowed to choose the order of prices. In fact, a great variety of processes can be formulated as independent draws from a fixed distribution that are then reordered in some way. The only real restriction here is that the process exhibit some loose form of stationarity (and in fact, we should not expect to do much if stationarity is not assumed). However, the troubling assumption here is the assumption of perfect knowledge of the distribution, which in our setting is unrealistic as spot prices appear to exhibit major qualitative changes quite frequently.

Our ideal model would allow for adversarial ordering of prices, but not require full distributional information. Instead, while we can not expect to characterize the price process exactly, it is often the case that we have relevant historical data at our disposal. For example, if we would like to optimize over a 12-hour period starting this Monday afternoon, we may have prices from the same

12-hour period on Mondays of previous weeks. Prompted by this, we propose a class of adversarial models: the *Data-Driven Prophet models*.

Definition 1 (Data-Driven Prophet Model)**.** *Given a historical time period of length N, a future time period of length T, and a total of M machine types, the data-driven prophet model proceeds as follows:*

1. *The adversary selects $N + T$ price vectors, each of length M.*
2. *Nature randomly partitions the $N + T$ vectors into a set of N and a set of T vectors. This selection is done uniformly at random so that each partition is equally likely.*
3. *Prior to its use, the algorithm is given the set of N prices as historical data (in arbitrary order).*
4. *Over the course of its use, the algorithm is fed the remaining T prices, in an order chosen by the adversary.*

In this model, the N parameter controls, in a very natural way, the amount of history available. Specifically, the N past price vectors represent 'relevant' observations that we have, where 'relevance' means exchangeability with the T future prices, and any algorithm we design will use nothing more in terms of data than these N pieces of history. In practice, the user provides this relevant history (this is the only effort required of the user). One approach, which is the approach we have used successfully in experiments, is to simply take the immediate N previous periods, where N is on the order of T (we used $N = T$), though this is by no means the only approach.

The adversary in this model maintains the ability to select the order of the prices, at the cost of having the exact prices chosen from a larger subset. We may view this model as a generalization of the prophet model: the case when $N \gg T$ is approximately the prophet model, as the T future prices can be thought of as being drawn independently from the empirical distribution formed by the historical prices, which is known to the algorithm. On the other hand, when $N = 0$, the adversary has essentially unlimited freedom, and we should not expect an algorithm to achieve any meaningful performance guarantee. One of our key results will be to characterize how much information is needed, i.e. how large N needs to be, in order to achieve a meaningful guarantee.

4. The Fixed Threshold Policy

We now propose a simple, natural algorithm for provisioning machines: the *fixed threshold policy.*
The Policy: If the price of a machine is lower than a precalculated threshold, purchase it; otherwise wait. Terminate when K purchases have been made.

The Threshold: The threshold, which we denote γ, is calculated using historical prices. Specifically, the threshold is set to be the $(K(N+1)/T)^{\text{th}}$ largest price over all MN historical prices. More precisely, for each machine type m, we utilize historical prices from the past N periods, which we denote $X^m_{-(N-1)}, \ldots, X^m_0$, to calculate the threshold. Given that there are M different types of machines, we have a total of MN different historical prices. The threshold, γ, is set to be the $(K(N+1)/T)^{\text{th}}$ largest price over all MN historical prices. If $(K(N+1)/T)$ is not an integer, then either the $\lfloor (K(N+1)/T) \rfloor^{\text{th}}$ or $\lceil (K(N+1)/T) \rceil^{\text{th}}$ largest is used, where for any x, we denote the largest integer less than x by $\lfloor x \rfloor$ (this is slightly different than the usual floor function), and we denote the smallest integer greater than or equal to x as $\lceil x \rceil$. The choice of whether to round up or down is done randomly.

The complete policy is specified below. The only additions to the above description are steps to ensure that, if the final periods are reached and not enough computation has been completed, then machines are purchased even if their price exceeds the threshold.

Fixed Threshold Policy

---

1. Calculate fixed threshold γ:
   (a) Set $x = K(N+1)/T$.
   (b) Generate a bernoulli variable U such that $\mathbb{P}(U = 1) = (x - \lfloor x \rfloor)/(\lceil x \rceil - \lfloor x \rfloor)$.
   (c) Let γ be the $(\lfloor x \rfloor + U)^{\text{th}}$ largest value in $\{X^m_n\}^{1 \le m \le M}_{-(N-1) \le n \le 0}$.
2. Set $K_1 = K$.
3. For $t = 1, \ldots, T$:
   (a) Set $Y_t = \sum_m \mathbb{1}(X^m_t \le \gamma)$.
   (b) Accept the lowest $\max\{\min\{K_t, Y_t\}, K_t - (T-t)M\}$ prices in X_t.
   (c) Set $K_{t+1} = K_t - \max\{\min\{K_t, Y_t\}, K_t - (T-t)M\}$.

Before discussing theoretical guarantees, it is worth noting a number of practical advantages of the fixed threshold policy. In particular, the policy is automatically calibrated, scalable, and simple.

Automatically Calibrated: The threshold is taken to be a quantile of historical prices. This approach does not require any calibration such as fitting a stochastic processes, and in practice, the only work

required of the user is to specify historical data. Fortunately, simple approaches to this, such as using price data from the preceding week, work extremely well.

Scalable: The work required to calculate the threshold and execute the policy essentially amounts to sorting historical prices and comparing current prices to these values. Thus, the policy is tractable for a large number of machine types and historical data.

Simple: Given the calculated threshold, the policy is easy to implement and easily understood by the user. The threshold is nicely interpreted as the maximum willingness to pay for a machine. This is particularly important given the mechanics of renting spot machines. Specifically, while we have so far assumed that we can observe the price of a spot machine and rent it at that price, in reality these machines need to be bid for in a spot market. The threshold naturally corresponds to the amount we should bid for a machine.

4.1. The Performance Ratio

Before presenting our theoretical guarantees, we discuss exactly how performance is measured. Recall that the adversary selects $N + T$ price vectors (possibly randomly), then we observe a (random) subset of N of these vectors, and select K prices from the remaining T vectors (with a possibly randomized algorithm). After all randomness has been realized, i.e. along each sample path, we can identify the K lowest prices that ideally should have been chosen. We call the proportion of these K lowest prices that are chosen by our algorithm the *performance ratio*, denoted Z. Note that Z is measured pathwise and is in general random with respect to any randomization performed by the adversary, nature, or our algorithm. Our guarantee will be a lower bound on the expected performance ratio $\mathsf{E}[Z]$, with the expectation taken over all randomness.

Z is a natural performance measure to use, as large Z immediately implies lower bounds on both the absolute and relative overpayment:

Proposition 1. *Suppose all prices lie in the interval $[a, b]$, and let S and S^* be the total of the selected prices and the lowest K prices, respectively. Then,*

$$S - S^* \leq (1 - Z)K(b - a), \quad \textit{and}$$

$$(S - S^*)/S^* \leq (1 - Z)(b - a)/a.$$

Proof. Let $p_{(i)}$ denote the i^{th} largest price, so $a \leq p_{(1)} \leq \cdots \leq p_{(MT)} \leq b$. Then

$$S^* = \sum_{i=1}^{K} p_{(i)} \geq (1 - Z)Ka + \sum_{i=(1-Z)K+1}^{K} p_{(i)}, \quad \text{and}$$

$$S \leq \sum_{i=(1-Z)K+1}^{K} p_{(i)} + \sum_{i=MT-ZK+1}^{MT} p_{(i)}$$
$$\leq (1-Z)Kb + \sum_{i=(1-Z)K+1}^{K} p_{(i)}.$$

Replacing S and S^* with the above bounds, the first result follows immediately, and the second result comes by noticing that the resulting quantity is maximized by setting $p_{(i)} = a$ for $i = (1-Z)K+1, \ldots, K$. ■

Note that in the worst case, the absolute error is $K(b-a)$ and the relative error is $(b-a)/a$. Therefore, Proposition 1 implies that both the relative and absolute errors are bounded away from their worst cases when Z is large. Furthermore, the assumption of bounded prices is not at all limiting for our application: since any spot machine can be replaced by an on-demand machine of the same type, we can treat prices as being bounded above by the fixed on-demand price.

4.2. Performance of the Fixed Threshold Policy

We are now ready to state the first of two main theoretical results: a performance ratio guarantee for the fixed threshold policy.

Theorem 1. *Under the data-driven prophet model, the fixed threshold policy achieves an expected performance ratio of*

$$\mathsf{E}[Z] \geq 1 - \sqrt{\frac{M}{K}} - \sqrt{\frac{MT}{K(N+2)}} - \frac{(M-\frac{1}{2})T}{K(N+1)}.$$

Parsing the expression in Theorem 1 yields a few important points:

1. **Extending the Prophet Model:** As discussed earlier, if $N \gg T$ the data-driven prophet model and the prophet model itself are very similar. The problem for a univariate distribution ($M = 1$ in our parlance) has previously been studied under the prophet model, and in particular, Alaei (2014) showed an algorithm that achieves an order-optimal competitive ratio of $1 - O(1/\sqrt{K})$. We should hope for the same 'gold-standard' $1 - O(1/\sqrt{K})$ guarantee in our problem when $N \gg T$ and $M = 1$, and Theorem 1 says that this is precisely the case. There are at present no results available in the multivariate $M > 1$ case for the prophet model – Theorem 1 gives a $1 - O(\sqrt{M/K})$ guarantee, which we will show in the next subsection is order-optimal as well.

2. **Amount of Data:** In general, Theorem 1 offers a guarantee of the order

$$1 - O\left(\sqrt{\frac{M}{K}}\right) - O\left(\sqrt{\frac{MT}{KN}}\right).$$

This result characterizes precisely the role played by N and highlights an extremely interesting dependence: observe that as long as $N = \Omega(T)$, we get the same order-optimal scaling of $1 - O(\sqrt{M/K})$. This immediately yields a valuable prescription when implementing the policy in practice – in general, when choosing the amount of historical data to use, it suffices to use as much data as the time horizon of the computation. In our own experiments, we used this idea directly, taking a week's worth of historical data for computation with a deadline of one day.

3. **Constants:** While a guarantee of the order $1 - O(\sqrt{M/K}) - O(\sqrt{MT/KN})$ is extremely significant, Theorem 1 is even stronger, as it provides an exact calculable bound.

The complete proof of Theorem 1 is delayed until Section 5 of the appendix; we provide some basic intuition here. In doing so, let us denote by Y the number of prices that fall below the threshold γ, among the T price vectors shown to the algorithm in adversarial order. Of course, we are hoping that Y is as close to K as possible. Specifically, if Y were less than K by some number Δ, then the algorithm would have to make up the shortfall using whatever prices the adversary made available after the point in time it was clear that a shortfall would occur. On the other hand if Y were larger than K by Δ, then the adversary could succeed in fooling the algorithm away from picking the first Δ smallest prices. In summary, this argument can allow us to see that

$$\mathsf{E}\left[Z|Y\right] \geq 1 - \frac{|Y - K|}{K}.$$

The question is now to characterize the quantity $|Y - K|/K$ in the data-driven prophet model. Here we observe that the only randomness is the random split of the $N + T$ price vectors originally chosen by the prophet. As it turns out, we can show after some work that this randomness suffices to show that Y can be modeled via a *Polya Urn* process. This unlocks the analysis of the quantity $\mathsf{E}|Y - K|$, ultimately yielding the result of the theorem.

4.3. Upper Bound

Our second theoretical result speaks to the optimality of the fixed threshold policy. In the previous subsection, Theorem 1 gave a guarantee for the expected performance ratio of $1 - O(\sqrt{M/K}) - O(\sqrt{MT/KN})$. The goal of this subsection is to demonstrate that this rate is in fact the best possible rate for any online algorithm. The expression has four parameters (K, M, N, T), and so care needs to be taken in describing optimality of results in terms of the scaling of all four parameters.

Let $\alpha = N/T$, i.e. the ratio of the amount of historical data to the amount of future data. The quantity α is akin to an information ratio, and note that in terms of scaling, the guarantee

in Theorem 1 only depends on N and T through α. In particular, fixing the parameters K and M, and taking all problem instances where $N \geq \alpha T$, Theorem 1 implies an expected performance ratio of

$$1 - O\left(\frac{1 + 1/\sqrt{\alpha}}{\sqrt{K/M}}\right) \tag{3}$$

One necessary condition then for Theorem 1 to say anything meaningful is that α must be at least constant, i.e. if $\alpha \to 0$, then no guarantee is made. A natural question is whether this is a necessary condition for any online algorithm to achieve a meaningful result. Moreover, even though Theorem 1 makes a guarantee for constant α, another question is whether the form of the dependence on α is optimal. Theorem 2 answers both of these.

Theorem 2. *Let* $\text{ONLINE}(K, M, N, T)$ *be the largest achievable expected performance ratio for any online algorithm over problems with parameters* $K, M, N,$ *and* T*.*

a. *Let* $\{N_1, N_2, \ldots\}$ *be any non-decreasing sequence of nonnegative integers such that* $\lim_{T\to\infty} N_T/T = 0$*. Then for any* K *and* M*, there exists* T *such that*

$$\text{ONLINE}(K, M, N_T, T) \leq 1 - c,$$

where $c > 0$ *is a universal constant.*

b. *Fix* $\alpha > 0$*. For any* K *and* M *such that* K/M *is sufficiently large, there exists* T *and* $N \geq \alpha T$ *such that*

$$\text{ONLINE}(K, M, N, T) \leq 1 - c\frac{1 + 1/\sqrt{\alpha}}{\sqrt{K/M}},$$

where $c > 0$ *is a universal constant.*

Theorem 2 answers both of our questions in the affirmative. First, part a shows that if $\alpha \to 0$, i.e. if N is scaling slower than T, then we can not make a meaningful guarantee for any algorithm in terms of the other parameters K and M, and so $N = \Omega(T)$ is a necessary condition. Second, part b shows that the dependence on α in (3) is order-optimal. The proof of Theorem 2 can be found in Section A of the Appendix, and relies on carefully constructing specific values for the adversary to choose.

5. Proof of Theorem 1

Recall that the adversary chooses $N + T$ price vectors, each of length M. Nature then randomly partitions these $N + T$ vectors into a set of N historical price vectors, and a set of T future price

vectors. Finally, the adversary selects the order in which the prices are presented to the algorithm. Without loss of generality, we will assume all prices are distinct.[4] The goal is to choose the K lowest prices from among the MT future prices. Z is the proportion of these K lowest prices that are correctly chosen.

Consider any fixed threshold algorithm that sets a fixed threshold γ based on the first N price vectors, and let Y be the number of the remaining MT prices that are less than γ. Then we can bound $\mathsf{E}[Z]$ conditional on Y:

$$\mathsf{E}[Z|Y] \geq 1 - \frac{|Y-K|}{K}. \tag{4}$$

We can see this by treating the cases $Y < K$ and $Y \geq K$ separately. If $Y < K$, then the Y lowest prices are less than γ, and the algorithm will select all of them, no matter what order the prices are presented in, so $\mathsf{E}[Z|Y] \geq Y/K = 1 - (K-Y)/K$. If $Y \geq K$, then the best the adversary can do is to first present the highest K prices from among the Y lowest prices. Of these K highest prices, exactly $(K-(Y-K))^+$ belong to the original K lowest prices, so $\mathsf{E}[Z|Y] \geq 1 - (Y-K)/K$.

Thus, to lower bound $\mathsf{E}[Z]$, it suffices to upper bound $\mathsf{E}[|Y-K|]$. The following two lemma works toward this. Lemma 1 exactly characterizes the distribution of Y, which we then use to bound the deviation of Y from its mean.

Lemma 1. *Fix a deterministic value $w \in \{1, \ldots, N\}$. Suppose a subset of size N is taken from $N+T$ distinct prices uniformly at random, and let γ be the w^{th} smallest price in this subset. Let Y be the number of the remaining T prices that are less than γ. Then Y is a beta-binomial random variable. In particular, $Y|P \sim Binomial(T, P)$ where $P \sim Beta(w, N+1-w)$.*

Proof of Lemma 1. We define the random variables $X_1, \ldots, X_{N+T}$, which take on a random permutation of the $N+T$ prices such that each permutation is equally likely. Then $\{X_1, \ldots, X_N\}$ is a random subset of the $N+T$ prices. For $t = 1, \ldots, T$, let Y_t be the indicator variable for the event that X_{N+t} is less than γ, i.e. $Y_t = \mathbb{1}(X_{N+t} < \gamma)$. Note that $Y = \sum_{t=1}^{T} Y_t$.

We can treat $\{Y_t\}$ as a stochastic process indexed by t, and evaluate the probability of $Y_t = 1$, i.e. $X_{N+t} < \gamma$, as follows: initially, there are $N+t-1$ sticks (representing prices) with $N+t$ 'gaps' in between them including end-points. Since the prices were randomly permuted, a new stick (i.e. price) can fall into any of these gaps (we are using the 'distinct prices' assumption here only) with equal probability. Moreover, $w + \sum_{s=1}^{t-1} Y_s$ of these gaps lie before the stick identified by γ. Summarizing this succinctly in equation, we obtain

$$\mathsf{E}[Y_t | Y_s, 1 \leq s \leq t-1] = \frac{w + \sum_{s=1}^{t-1} Y_s}{N+t}.$$

[4]We only require that the prices can be strictly ordered. If multiple equal prices are presented, we assume the algorithm will choose a random permutation of them to be the strict ordering.

Given this equation, $\{Y_t\}$ is precisely the classical Pólya urn process: start with w red balls and $N+1-w$ blue balls in an urn, and at each time draw a ball at random and add one more ball of the same color to the urn. It is well known that $Y|P \sim \text{Binomial}(T, P)$ where $P \sim \text{Beta}(w, N+1-w)$. ■

For our particular fixed threshold policy, we define W as a random variable taking either $\lfloor K(N+1)/T \rfloor$ or $\lceil K(N+1)/T \rceil$, with probabilities chosen such that $\mathsf{E}[W] = K(N+1)/T$. For $m = 1, \ldots, M$, let W^m and Y^m be the number of machine m prices that are less than or equal to γ among the first N and last T prices, respectively. In addition, let V_m be the number of machine m prices among the last T prices that lie in the 'gap' containing γ formed by the first N prices. Then Y lies in the interval $[\sum_m Y^m, \sum_m Y^m + \sum_m V^m]$.

By Lemma 1, we know $Y^m|P^m \sim Binomial(T, P^m)$ where $P^m|W^m \sim Beta(W^m, N+1-W^m)$. Let $P = \sum_m P^m$. Then it can be checked that

$$\text{E}[P|W^m, 1 \le m \le M] = \sum_m \frac{W^m}{N+1} = \frac{W}{N+1}, \quad \text{and} \tag{5}$$

$$\begin{aligned} \text{Var}(P|W^m, 1 \le m \le M) &\le \left(\sum_m \sqrt{\text{Var}(P^m|W^m)} \right)^2 \\ &\le \left(\sum_m \sqrt{\frac{W^m}{(N+1)(N+2)}} \right)^2 \\ &\le \frac{MW}{(N+1)(N+2)}. \end{aligned} \tag{6}$$

Note that the first inequality of (6) follows from the subadditivity of standard deviation, and the last step follows from Hölder's inequality.

We proceed by showing concentration of $\sum_m Y^m$ to its conditional mean TP. Applying Jensen's inequality, we can bound the mean absolute deviation of Y_m for each m:

$$\begin{aligned}
\mathsf{E}\left[\left|\sum_m Y^m - TP\right|\right] &\leq \sum_m \mathsf{E}\left[|Y^m - TP^m|\right] \\
&\leq \sum_m \mathsf{E}\left[(Y^m - TP^m)^2\right]^{1/2} \\
&= \sum_m \mathsf{E}\left[\mathrm{Var}\left(Y^m|P^m\right)\right]^{1/2} \\
&= \sum_m \mathsf{E}\left[TP^m(1-P^m)\right]^{1/2} \\
&\leq T^{1/2}\sum_m \mathsf{E}[P^m]^{1/2} \leq (MK)^{1/2}.
\end{aligned}$$

Note that we apply Jensen's inequality in the second step and condition on P^m in the third step. The final step comes from Hölder's inequality and taking expectations on both sides of (5). Similarly, we apply Jensen's inequality to show concentration of P to its mean:

$$\begin{aligned}
\mathsf{E}\left[|TP-K|\right] &\leq \mathsf{E}\left[(TP-K)^2\right]^{1/2} \\
&= T\,\mathrm{Var}(P)^{1/2} \\
&\leq T\left(\frac{MK}{T(N+2)} + \frac{1}{4(N+1)^2}\right)^{1/2} \\
&\leq \left(\frac{MKT}{N+2}\right)^{1/2} + \frac{T}{2(N+1)}.
\end{aligned}$$

In the third step, we apply the law of total variance using (5) and (6), along with the fact that W lies in an interval of length 1, so $\mathrm{Var}(W) \leq 1/4$.

By an argument very similar to that from the proof of Lemma 1, we can show that $\mathsf{E}[V^m] = T/(N+1)$ for the $M-1$ machine types whose price histories do not include γ. Combining this with the previous two bounds, we conclude

$$\begin{aligned}
\mathsf{E}\left[|Y-K|\right] &\leq \mathsf{E}\left[|\Sigma_m Y^m - TP|\right] + \mathsf{E}\left[|TP-K|\right] + \mathsf{E}\left[|\Sigma_m V^m|\right] \\
&\leq (MK)^{1/2} + \left(\frac{MKT}{N+2}\right)^{1/2} + \frac{(M-\frac{1}{2})T}{N+1}.
\end{aligned}$$

We are now ready to conclude:

$$\mathsf{E}[Z] \geq 1 - \frac{\mathsf{E}[|Y-K|]}{K} \geq 1 - \sqrt{\frac{M}{K}} - \sqrt{\frac{MT}{K(N+2)}} - \frac{(M-\frac{1}{2})T}{K(N+1)}.$$

■

6. Software Implementation

We have built a software tool, called OptScale, that automatically completes deadline-constrained batch computations in the cloud in a cost-effective way. While the main logic is driven by the fixed threshold policy, there are specific mechanics to the cloud market that need to be addressed. The goal of this section is to describe the idiosyncrasies of the cloud market in detail, how the fixed threshold policy is adapted to these idiosyncrasies, and the implementation of the tool itself.

6.1. The Cloud Computing Landscape

As we have described previously, most cloud providers offer on-demand virtual machines for rent by the hour. These can be rented any time at a fixed, pre-announced price, so for example an 'm3.xlarge' machine (which implicitly specifies performance attributes) on AWS, would cost approximately 27 cents per hour of use, rounded *up* to the nearest hour. This figure (27 cents) *does not vary over time*. Moreover, machine failures are now exceedingly rare (CloudHarmony (2015)), and the time taken to provision a machine is under a minute from the time a request for the machine is placed.

A major challenge faced by cloud providers is the substantial amount of heterogeneity in the demand for computing resources. In particular, based on the nature of their tasks, users may have varied abilities to *substitute* across computing resources. Users are also likely to have a fair amount of heterogeneity with respect to their inter-temporal preferences; specifically there is likely to be wide variation in deadlines for completion across users. Finally, the mix of users is highly unlikely to be stationary over time given the varied objectives of their computing needs. As such, an obvious critique of the 'fixed price' model prevalent, is that it is likely to do a poor job of ensuring efficient resource allocation in the face of volatile, heterogeneous demand.

In response to this critique, cloud providers have recently begun experimenting with dynamic pricing. The most prevalent example of this is the *'spot' market* launched by AWS. In contrast to the traditional fixed price model, the prices on the spot market are highly dynamic and are functions of demand for a specific type of resource as seen by Amazon in a specific data center or 'region'. The mechanism via which machines are rented in this market is distinct and requires careful thought on the part of the user: (1) the user places a standing bid for a machine (or multiple machines of a given type); (2) should the spot price for that machine type fall below the user's bid, the user is allocated machine(s) of the type; and finally (3) should the spot price at any subsequent time (prior to the user terminating use of the machine) rise above the user's bid, the user instantly loses access to that machine.

6.2. Adapting the Fixed Threshold Policy

Using the fixed threshold policy in practice requires some adaptations regarding the bidding process, the possibility of losing machines if the spot price rises too high, and heterogeneous machine types.

Spot Machine Bidding. In our analysis, we assumed spot machines can be purchased at the current price. However, in a reality, we are required to bid on them, and since requests take some time (on the order of a few minutes on AWS), there is uncertainty in what price we will actually pay. This raises the question of what bid should be used. Fortunately, the fixed threshold can be interpreted as the maximum willingness to pay, and so we bid the threshold γ when we request a machine.

Machine Failure. As discussed previously, spot machines can be shut down at any time if the spot price rises above the bid. This presents a major challenge for computational tasks that require continuity, such as running a web server, but for batch computation it is not as much of a concern as we only lose progress on a single job within the batch of jobs when a machine is lost.

Moreover, from a cost perspective, machine termination may even be beneficial, as users are not charged for machines that are terminated. For example, if a machine is provisioned but terminated after 30 minutes, the user pays nothing even though some computation was completed. The only risk associated with cancellation is that we will not complete all of the jobs by the deadline, so in the final period (or final periods, if necessary), we use on-demand resources exclusively.

Heterogeneous Machines. Different machine types may differ in size and therefore the rate at which they complete jobs. The fixed threshold policy was described for machines with the same performance, but this can easily be generalized to machines of different sizes by equating each machine type to a multiple of a normalized compute resource.

We will describe how to calculate the thresholds for machines with different performance levels (there is a threshold γ_m for each machine type m). Let k_m be the scaling factor proportional to the number of jobs that machine type m can complete in a single period. Assume without loss of generality that $k_1 = 1$ and that this is the lowest value among all machine types. We also assume for convenience that k_m is integer-valued for all m. Note that instance type 1 can be thought of as the 'normalized' resource.

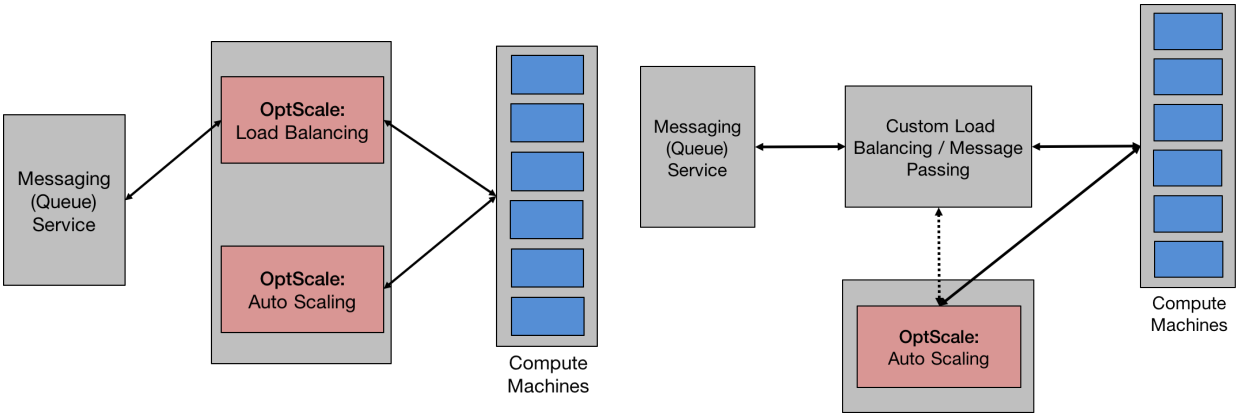
From the original pool of MN historical prices $\{X_n^m\}_{-(N-1)\le n\le 0}^{1\le m\le M}$, create a new pool of $\sum_m k_m N$ prices where for each historical price X_n^m from the original pool, the normalized price X_n^m/k_m is included k_m times in the new pool. Then γ_1, the threshold for instance type 1, is set to be the $\lceil (KN/T) \rceil^{\text{th}}$ largest over all $\sum_m k_m N$ prices in the new pool. For all other m, the threshold γ_m is $k_m \gamma_1$.

6.3. OptScale Architecture

We have implemented a software tool called OptScale. OptScale is architected (see Figure 4) to be an efficient load balancer and cost-efficient auto scaler for deadline-constrained *batch-computation* in the cloud (public or private) that is comprised of independent, *embarrassingly parallel* atomic tasks. *Embarrassingly parallel* applications are those that readily split up in to parallel atomic tasks without requiring significant changes to the application structure, such as MapReduce jobs and the Personalized PageRank described in the introduction.

OptScale is designed as a replacement for existing load balancers and auto scaling schemes, and readily plugs in to existing infrastructure. It queues up requests for computation (atomic tasks) and intelligently scales the infrastructure (compute resources) to complete the tasks in a cost-efficient manner while ensuring that the tasks are completed within the user-specified deadline. In the near term future, *it will be made available as an open-source software which will be compatible (out-of-the-box) for use on Amazon AWS.*

Since OptScale is designed for the cloud, it is able to rapidly add/remove machines when they become economically feasible/infeasible. The number of machines that can be managed by OptScale is only upper-bounded by the limitations imposed by the cloud-provider, e.g. limits for the total number of machines allowed per user in a given data-center. OptScale assumes no universal limits and allows users to configure their own depending on the specifics of their application and circumstances.



(a) OptScale architecture layout for applications without custom load balancing needs. OptScale can connect to existing infrastructure by replacing load balancing and auto scaling schemes, while connecting to Messaging Queues managing demand, on the one hand, and compute nodes, on the other.

(b) OptScale architecture layout for applications with their own custom load balancing needs or an existing implementation of the Message Passing Interface. In this case, OptScale only provides auto scaling functionality, adding/removing machines in a cost-efficient manner, while leaving load balancing to the user.

Figure 4: Architecture layouts for OptScale, depending on the load balancing needs of the application.

6.3.1. Compatibility with Existing Infrastructure

OptScale can be run on any machine that is able to accept Socket connections. As shown in Figure 4a, it sits in series after the infrastructure managing incoming demand (i.e. compute tasks) and mediates each demand object's way to the worker nodes (i.e. compute machines). Tasks can be tracked and queued using any popular management scheme, e.g. messaging queue services like RabbitMQ[5]. RabbitMQ, for example, allows 'messages' to be passed between various components of a system. These 'messages' can be mapped one-to-one to atomic tasks that need to be performed by worker nodes. OptScale can easily be tailored to interface with any scheme that manages computation tasks in a manner similar to these messaging services.

Any task management scheme must expose the following interface to OptScale:

a. Ability to subscribe to the scheme, such as the `basic_consume()` function exposed by RabbitMQ.

b. A callback mechanism that can deliver 'messages' and any required parameters to OptScale.

c. An ability to acknowledge the completion of a task. This is required to ensure that if the OptScale service goes out of commission due to any reason, the messages are not assumed done. The `basic_ack()` function exposed by RabbitMQ is an example of such a mechanism.

In addition, OptScale needs the following configuration details from the user:

1. The deadline for completing the computation.

2. Up-to-date pricing information for all types of machines. By default, OptScale has a mechanism to retrieve this information from Amazon's EC2.

3. A copy of the image of the machine that needs to be launched on every compute node. Most commercial cloud providers have simple instructions that allow the creation and subsequent use of such images.

4. Set of machine types appropriate for the tasks. OptScale then uses any/all of these options. Assumed to be included are individual machine properties (memory, CPU etc), profiling information and any limits on the number of such machines that can be provisioned.

Regarding the last item, it is assumed that users are able to profile each candidate machine type for their application. This helps OptScale provision the right number of resources and also helps with efficient load balancing. Note that the tasks are can have randomly distributed completion

[5] `https://www.rabbitmq.com/`

times, but system profiling is expected to provide the mean and variance of these properties. For convenience, OptScale provides a module to help estimate these job-specific statistics.

Given the information identified, OptScale effectively becomes the 'middle-man' interfacing between the existing task management schemes and compute nodes in the cloud. This is exactly the role played by existing load balancers and auto scalers, but additionally, OptScale is designed to optimize for low-cost resource provisioning.

In the remainder of this section, we describe OptScale's key features.

Load Balancing: OptScale's load balancer maintains a state of tasks (queued, started, completed) and a state of each compute node. Just as in state-of-the-art load balancers, OptScale schedules tasks on machines as soon as there is available capacity, aspiring to achieve as close to 100% resource utilization as possible; Figure 5 illustrates the utilization of acquired resources during a live experiment with OptScale. Figure 4a shows a visual depiction of this system. Our load balancer schedules each atomic task on its own CPU utilizing Python's *multiprocessing* library.

Note that if the user's application requires tasks to communicate with each other, or report intermediate results, our load balancer can easily be adapted to integrate directly with an implementation of the Message Passing Interface[6] that the application might already be using. In such a scenario, OptScale effectively requires the user's custom load balancer to schedule and farm tasks out to compute nodes, as shown in Figure 4b.

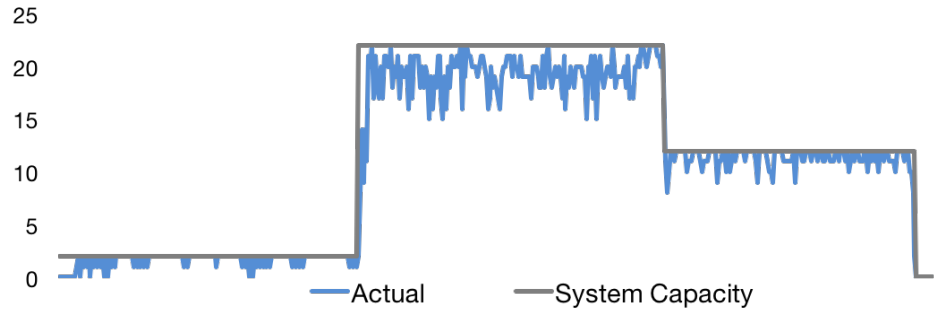


Figure 5: OptScale's load balancing as demonstrated in a live experiment. System capacity and actual usage, as measured by number of simultaneous tasks ongoing, are depicted over time.

Auto Scaling: OptScale's auto scaler provides dynamic decision-making about how many of each type of machine to add or remove. Launch and termination decisions are communicated to the health monitoring and load balancing services to appropriately monitor the nodes and schedule tasks as appropriate. The essential innovation of our work, cost-efficiency through the fixed threshold policy, is encoded within this auto scaler. In addition to cost-awareness, OptScale enforces a hard requirement of getting all tasks completed within the user-specified deadline, using on-demand

[6] `https://computing.llnl.gov/tutorials/mpi/`

machines exclusively as the deadline approaches.

Queuing of Tasks: In order to tightly control and track individual tasks being performed on each machine, OptScale maintains its own task-queue. This is a classic queue with a FIFO structure providing atomic tasks to the load balancing scheme as required. Tasks that fail to get scheduled on compute nodes are queued back.

Health-Monitoring: In addition to maintaining a dynamic list of available CPU units on machines, OptScale also needs to perform health checks on all available machines on a regular frequency. This is to ensure that machines in unhealthy states are taken out of commission and no tasks are scheduled on them. This health monitoring is a key feature of OptScale and executes on its own thread at an adjustable frequency which defaults to 10 minutes.

7. Experiments

To evaluate the performance of OptScale, we have conducted extensive experiments on Amazon's EC2 cloud for a Personalized PageRank application. Our results show that OptScale achieves cost close to the theoretical clairvoyant (optimal) policy that has knowledge of the future. OptScale also performs better than two other currently prevalent policies.

7.1. Experimental Setup

As a representative of large-scale batch computing, we chose Personalized PageRank computations. As mentioned before, performing such tasks are common place for consumer systems such as Netflix or Amazon. To personalize content to an individual's preferences based on her/his activity, such tasks are performed at a regular interval. For example, in a high velocity environment where available content is changing daily, such computation needs to be performed for each user in parallel creating a large batch of compute jobs. This is an ideal scenario for OptScale where each task is independent of others and few intermediate aggregations need to be performed.

For our experiments, we used 'm3.medium', 'm3.large', and 'm3.xlarge' machines described in Figure 2. All experiments were done through Amazon's data centers in Virginia, US (the 'us-east-1' region), within four different *availability zones* labeled 'a', 'c', 'd', and 'e'. Note that spot prices for the same machine type can vary greatly across availability zones, so we treat them as separate resources. In total, this gives twelve different spot machine types and three on-demand types. We provisioned at most four machines of each type at any time to stay within the limits imposed by Amazon. Businesses can request larger limits and the results of our experiments will scale. During experiments, each machine is booted with a prepared Image containing the Personalized PageRank implementation that can be triggered via an HTTP request. This Image is based on an

Intel $x86\_64$ architecture on an Ubuntu OS (version 12.04 or later). Each instance uses Apache to arbitrate incoming HTTP requests and direct them to the Personalized PageRank implementation.

We performed experiments with batches ranging from 1,000 to 12,000 tasks, and deadlines ranging from 6 to 24 hours. Each Personalized PageRank problem is on a matrix of size 16 million. Our machine profiling estimated that the completion time for a single task per CPU core is on average 36 seconds. Therefore, m3.medium machines can complete approximately 100 problems/hour, while m3.medium and m3.large machines can solve about 200 and 400 problems, respectively because these machines have 1, 2 and 4 CPU cores, respectively.

For each experiment, we used historical pricing data from the 7 days immediately preceding the experiment to calculate the thresholds for OptScale's fixed threshold policy. We compared OptScale against two commonly used natural benchmark policies and a clairvoyant 'optimal' policy that has full knowledge of future prices:

1. **Baseline**: This policy *only* uses on-demand machines. At the moment, most computation is still performed on on-demand resources exclusively, so it is worth measuring the potential savings from using spot machines intelligently.

2. **Myopic**: This is a natural *greedy* policy that utilizes both spot and on-demand machines. In particular, it always purchases instances of the cheapest machine type available at each time and does so myopically. This is a natural cost-aware policy utilized in practice. The policy will purchase on-demand machines, if needed, near the end of the deadline to perform all remaining computations.

3. **Clairvoyant**: This is the offline optimal calculated ex post, knowing the realized prices. This is used as a lower bound on the performance that can be achieved by any policy.

In all of our experiments, the policies (except for clairvoyant) are run simultaneously so that they are given the same price traces.

7.2. Results

We conducted two sets of experiments: the first set consists of two *live* experiments done directly on Amazon's cloud platform, and the second is a much larger set of *offline* experiments with thousands of historical price traces. For our live experiments, we used a deadline of 6 hours, and workloads of 4,600 and 7,200 Personalized PageRank computations.

Table 1 summarizes the results of the live experiments. For all four policies, we report the total cost along with the completion time. In terms of cost, both experiments have qualitatively similar results: we see that the clairvoyant policy is cheaper than the baseline policy by almost a

	Experiment 1		Experiment 2	
Policy	Cost	Time	Cost	Time
Baseline	\$3.36	57	\$5.26	91
Myopic	\$1.37	176	\$3.83	349
OptScale	\$0.39	177	\$0.61	285
Clairvoyant	\$0.38	∼240	\$0.58	∼360

Table 1: Summary of results for live experiments. The total cost and completion time (in minutes) of each of the four policies is reported. The experiments consisted of 4,600 and 7,200 Personalized PageRank computations, respectively, and both had deadlines of 6 hours.

factor of ten, suggesting the potential for massive savings by using spot machines intelligently. The myopic policy makes some progress to this effect with costs in both experiments that are much improved over the baseline policy, but still a gap remains above the clairvoyant policy. OptScale improves over the myopic policy and closes this gap almost completely – OptScale's cost is at most 5% greater than that of the clairvoyant policy.

The promising results of the live experiments are further backed by extensive offline experiments. For these experiment, price traces for Amazon's machines were collected at a one minute resolution over a six month period. Simulated experiments nearly identical to the online experiments were performed, the only difference being that prices were read from these historical traces. The results are summarized in Table 2.

		Average Computation Cost		
Deadline	Workload	Clairvoyant	OptScale	Myopic
12 hrs	1,000	\$100	\$109	\$131
	3,000	\$100	\$115	\$136
	6,000	\$100	\$121	\$143
24 hrs	2,000	\$100	\$105	\$136
	6,000	\$100	\$108	\$145
	12,000	\$100	\$116	\$155

Table 2: Summary of results for offline experiments. Average costs of OptScale and the clairvoyant and myopic policies are reported. For ease of comparison, we have normalized the cost of the clairvoyant policy to be \$100.

As shown in Table 2, deadlines of 12 and 24 hours were used, and for each of these, three different workloads were tested, giving six types of experiments. The experiments were done beginning every hour in the collected time horizon, giving over 4000 experiments for each of the six types of experiments. In every experiment, the costs of all four policies were measured.

Across all experiments, OptScale was no more than 21% more expensive than the clairvoyant

policy. This performance complements our theoretical result for the fixed threshold policy. At the same time, the myopic policy was as much as 55% more expensive than the clairvoyant policy. This mirrors the results of the live experiments. Also, very similar to the live experiments, the cost of the baseline policy was much higher than any of the other polices, almost eight times more expensive than the clairvoyant policy. Taken together, these experiments paint a promising picture for OptScale to significantly reduce costs of batch computations.

8. Conclusion

The goal of this paper was to solve a valuable problem customers in the cloud market: deadline-constrained batch computation at minimal cost. To do so, we modeled the problem as a classic K-choice consumption/allocation problem. In deciding how to model the randomly evolving prices of cloud resources, we saw the inefficacy of stochastic modeling, and the limitations of existing adversarial models. We proposed a data-driven generalization of the classic prophet model, called the Data-Driven Prophet Model, which properly captured the availability of historical prices. We then proposed a fixed threshold policy and showed that, under this model, the policy is order-optimal. We implemented OptScale, a cost-efficient load balancer and auto scaler that completes batch computation automatically on Amazon AWS, with logic driven by the fixed threshold policy. Experiments demonstrated that OptScale indeed reduces costs over existing, popular approaches, and achieves cost close to the offline optimal cost.

To conclude, our work suggests a number of exciting directions for future work:

1. Real-time applications: another class of computational tasks are real-time applications, such as content delivery and web servers, that need to handle random demand in real-time. As in this work, the main goal is to minimize cost, but rather than having fixed deadlines, the usual performance metric is a service level. A major contribution would be a similar tool that achieves a user-specified quality of service, through automatic provisioning of cloud resources.

2. Data-driven prophet model for general packing: we solved the K-choice allocation problem under the data-driven prophet model, but this adversarial model is more broadly applicable in a variety of problems, including general packing problems. It remains to be seen what sorts of policies arise in these broader settings.

A. Proof of Theorem 2

We will prove parts (a) and (b) separately.

Part (a): We being by proving part (a) for the case $M = 1$. Fix any K and $T \geq K$, and assume the adversary sets $N_T + T - K$ of the values to be equal to one. The remaining K values are either all set equal to two, or all equal to zero, each with probability $1/2$. Finally, after nature forms the partition, the adversary presents the T values in the following order: all of the ones, followed by all of the twos or zeros.

Let E be the event that nature's partition leaves all of the zeros or twos in the future prices, and so the past prices are only ones. If E occurs, then the best achievable performance ratio is $1/2$, and so

$$\mathsf{E}[Z] \leq \mathsf{P}(E^c) + \frac{1}{2}\mathsf{P}(E) = 1 - \frac{1}{2}\mathsf{P}(E).$$

Thus, it suffices to show that for any K, there exists some T such that we can lower bound $\mathsf{P}(E)$ by a universal constant. In fact, $\mathsf{P}(E)$ can be calculated exactly:

$$\mathsf{P}(E) = \prod_{n=1}^{N_T} \frac{T-K+n}{T+n} \geq \left(\frac{T-K}{T}\right)^{N_T} = \left(1-\frac{K}{T}\right)^{N_T} = \left[\left(1-\frac{K}{T}\right)^{T}\right]^{N_T/T}$$

Since $\lim_{T\to\infty}(1-K/T)^T = e^{-K}$, and by assumption $\lim_{T\to\infty} N_T/T = 0$, there exists some T^* such that $(1-K/T^*)^{T^*} \geq e^{-2K}$ and $N_{T^*}/T^* \leq 1/K$, and thus $\mathsf{P}(E) \geq e^{-2}$.

The proof for $M > 1$ is a simple generalization of the same arguments. Define $K' = K/M$, which we will take to be an integer. Assume the adversary sets the values within each vector to be equal. $N_T + T - K'$ of the vectors are set equal to ones, and the remaining K' vectors are set all equal to twos, or all equal to zeros, each with probability $1/2$. The rest of the proof follows identically to the argument above, with K' replacing K.

Part (b): We will now show part (b), again first considering the case $M = 1$. Fix K, and for any $T \geq K$, set $N = \alpha T$, which for the sake of simplicity we will assume is an integer. To select values, the adversary first draws a random variable A from a beta distribution:

$$A \sim \text{Beta}\left(\frac{\alpha K}{2}, \frac{\alpha K}{2}\right).$$

Having drawn A, the adversary draws $(\alpha+1)K$ of the values i.i.d. to be two with probability A and zero otherwise (in contrast to part (a), here there may be a combination of zeros and twos). The adversary then sets the remaining values equal to one. Finally, after nature forms the partition, the adversary orders the T values as all of the ones, followed by all of the twos, and finally all of

the zeros.

Let W equal the number of zeros and twos in the future values, and let Y_2 be the number of twos observed in the past values. Note that W and Y_2 are observed in the past values and fully encode the information provided by the past values. We will show that for certain realizations of W and Y_2, the expected performance ratio is bounded above, and that these realizations occur with constant probability.

First, W only depends on nature's partition: with respect to nature's randomization, W follows a hypergeometric distribution and it can be verified that:

$$\mathsf{E}[W] = K \quad \text{and} \quad \mathrm{Var}(W) = K\frac{N(1-\frac{K}{T})}{N+T-1} \leq \alpha K.$$

Throughout, we let $f_i(\cdot)$ denote functions that we will not explicitly define, but that satisfy $|f_i(x)| = O(x)$. Let E_1 denote the event that $|W - K| \leq f_1(\sqrt{\alpha K})$. By a Chebyshev bound, f_1 can be selected so that E_1 occurs with constant probability for K sufficiently large. For example, taking $f_1(\sqrt{\alpha K}) = 2\sqrt{\alpha K}$, we have $\mathsf{P}(E_1) \geq 3/4$ for all K.

Now conditional on W and A, and with respect to the adversary's random draw of zeros and twos, Y_2 follows a beta-binomial distribution: $(Y_2|W, A) \sim \mathrm{Bin}((\alpha+1)K - W, A)$. Thus it can be verified, through known closed-form expressions for moments of the beta-binomial distribution, that for any constant w such that $|w - K| \leq f_1(\sqrt{\alpha K})$,

$$\mathsf{E}[Y_2|W = w] = \frac{1}{2}\alpha K + \frac{K-w}{2} = \frac{1}{2}\alpha K + f_2\left(\sqrt{\alpha K}\right) \quad \text{and}$$

$$\begin{aligned}
\mathrm{Var}(Y_2|W = w) &= \frac{1}{4}\frac{(\alpha K + K - w)(2\alpha K + K - w)}{\alpha K + 1} \\
&\leq \frac{1}{4}\left(2\alpha K + 3(K-w) + \frac{(K-w)^2}{\alpha K}\right) \\
&\leq \frac{1}{2}\alpha K + f_3\left(\sqrt{\alpha K}\right).
\end{aligned}$$

Let E_2 denote the event that $|Y_2 - \alpha K/2| \leq f_4(\sqrt{\alpha K})$. Then again by a Chebyshev bound, f_4 can be selected so that $\mathsf{P}(E_2|E_1)$ is bounded below by a constant for sufficiently large K. Putting this together, we have for sufficiently large K,

$$\mathsf{P}(E_1 \cap E_2) = \mathsf{P}(E_1)\mathsf{P}(E_2|E_1) \geq c, \tag{7}$$

where c denotes a running, positive universal constant.

Finally, let Y be the number of twos in the future prices. Conditional on W and Y_2, and

with respect to the adversary's randomization, Y follows a beta-binomial distribution: $(Y|W,B) \sim \text{Bin}(W,B)$ where $(B|Y_2,W) \sim \text{Beta}(\alpha K/2+Y_2, \alpha K/2+(\alpha+1)K-W-Y_2)$. Thus for any constants w and y_2 such that $|w-K| \le f_1(\sqrt{\alpha K})$ and $|y_2 - \alpha K/2| \le f_4(\sqrt{\alpha K})$, we have

$$\mathsf{E}[B|Y_2 = y_2, W = w] = \frac{\alpha K/2 + y_2}{2\alpha K + K - w} = \frac{1}{2} + f_5\left(1/\sqrt{\alpha K}\right) \quad \text{and}$$

$$\begin{aligned}\text{Var}(B|Y_2 = y_2, W = w) &= \frac{(\alpha K/2 + y_2)(3\alpha K/2 + K - w - y_2)}{(2\alpha K + K - w)^2(2\alpha K + K - w + 1)} \\ &= \frac{1}{8\alpha K} + f_6\left(1/(\alpha K)^{3/2}\right).\end{aligned}$$

Let E_3 denote the event that $B \ge 1/2+1/\sqrt{\alpha K}$ and E_4 denote the event that $B \le 1/2-1/\sqrt{\alpha K}$. Then the above implies

$$\mathsf{P}(E_3|Y_2 = y_2, W = w) \ge c \quad \text{and} \quad \mathsf{P}(E_4|Y_2 = y_2, W = w) \ge c \tag{8}$$

for sufficiently large K. One way to see this is by the well known fact that the beta distribution converges in distribution to a normal random variable as both parameters increase together. Now for any constants w and b such that $|w-K| \le f_1(\sqrt{\alpha K})$ and $b \ge 1/2 + 1/\sqrt{\alpha K}$, we have

$$\mathsf{E}[Y|W = w, B = b] \ge \frac{w}{2} + \frac{w}{\sqrt{\alpha K}} \ge \frac{w}{2} + \sqrt{\frac{K}{\alpha}} + f_7(1) \quad \text{and}$$

$$\text{Var}(Y|W = w, B = b) \ge w\left(\frac{1}{4} - \frac{1}{\alpha K}\right) \ge \frac{K}{4} + f_8(\sqrt{\alpha K}).$$

It follows from (8) and the expressions above that, for constants w and y_2 such that $|w-K| \le f_1(\sqrt{\alpha K})$ and $|y_2 - \alpha K/2| \le f_4(\sqrt{\alpha K})$, the quantity

$$\mathsf{P}\left(Y \ge \frac{w}{2} + \sqrt{\frac{K}{\alpha}} + \sqrt{K} \middle| Y_2 = y_2, W = w\right)$$

is at least constant for sufficiently large K. An equivalent argument with event E_4 shows that the same holds for the quantity

$$\mathsf{P}\left(Y \le \frac{w}{2} - \sqrt{\frac{K}{\alpha}} - \sqrt{K} \middle| Y_2 = y_2, W = w\right),$$

and therefore for any online algorithm,

$$\mathsf{E}[Z|Y_2 = y_2, W = w] \leq 1 - c/\sqrt{K} - c/\sqrt{\alpha K}. \tag{9}$$

We are now ready to conclude. Combining (7) and (9), we have for sufficiently large K,

$$\begin{aligned}
\mathsf{E}[Z] &\leq (1 - P(E_1 \cap E_2)) + P(E_1 \cap E_2)\mathsf{E}[Z|E_1 \cap E_2] \\
&\leq 1 - P(E_1 \cap E_2)\left(\frac{c}{\sqrt{K}} - \frac{c}{\sqrt{\alpha K}}\right) \\
&\leq 1 - \frac{c}{\sqrt{K}} - \frac{c}{\sqrt{\alpha K}}
\end{aligned}$$

The proof for $M > 1$ is again a simple generalization of the same arguments. Define $K' = K/M$, which we will take to be an integer. The adversary sets the values within each vector to be equal, using the same procedure used above to randomly select values. The rest of the proof follows identically with K' replacing K. ■

References

Agmon Ben-Yehuda, Orna, Muli Ben-Yehuda, Assaf Schuster, Dan Tsafrir. 2013. Deconstructing amazon ec2 spot instance pricing. *ACM Transactions on Economics and Computation* **1**(3) 16.

Agrawal, Shipra, Zizhuo Wang, Yinyu Ye. 2014. A dynamic near-optimal algorithm for online linear programming. *Operations Research* **62**(4) 876–890.

Alaei, Saeed. 2014. Bayesian combinatorial auctions: Expanding single buyer mechanisms to many buyers. *SIAM Journal on Computing* **43**(2) 930–972.

Amazon. 2016a. Elastic load balancing. `https://aws.amazon.com/elasticloadbalancing/`.

Amazon. 2016b. Lyft case study-spot. `https://aws.amazon.com/solutions/case-studies/lyft-spot/`.

Azar, Pablo D, Robert Kleinberg, S Matthew Weinberg. 2014. Prophet inequalities with limited information. *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1358–1377.

Azar, Yossi, Naama Ben-Aroya, Nikhil R Devanur, Navendu Jain. 2013. Cloud scheduling with setup cost. *Proceedings of the twenty-fifth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 298–304.

Babaioff, Moshe, Nicole Immorlica, David Kempe, Robert Kleinberg. 2008. Online auctions and generalized secretary problems. *ACM SIGecom Exchanges* **7**(2) 7.

Bahmani, Bahman, Abdur Chowdhury, Ashish Goel. 2010. Fast incremental and personalized pagerank. *Proceedings of the VLDB Endowment* **4**(3) 173–184.

Ben-Yehuda, Orna Agmon, Muli Ben-Yehuda, Assaf Schuster, Dan Tsafrir. 2013. Deconstructing amazon ec2 spot instance pricing. *ACM Transactions on Economics and Computation* **1**(3) 16.

Buchbinder, Niv, Joseph Naor. 2009. Online primal-dual algorithms for covering and packing. *Mathematics of Operations Research* **34**(2) 270–286.

Chaisiri, Sivadon, Rakpong Kaewpuang, Bu-Sung Lee, Dusit Niyato. 2011. Cost minimization for provisioning virtual servers in amazon elastic compute cloud. *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on*. IEEE, 85–95.

CloudHarmony. 2015. Cloudsquare service status. `https://cloudharmony.com/status`. Accessed: 2015-02-08.

Devanur, Nikhil R, Thomas P Hayes. 2009. The adwords problem: online keyword matching with budgeted bidders under random permutations. *Proceedings of the 10th ACM conference on Electronic commerce*. ACM, 71–78.

Dynkin, Eugene B. 1963. The optimum choice of the instant for stopping a markov process. *Soviet Math. Dokl*, vol. 4.

Feldman, Jon, Monika Henzinger, Nitish Korula, Vahab S Mirrokni, Cliff Stein. 2010. Online stochastic packing applied to display ad allocation. *Algorithms–ESA 2010*. Springer, 182–194.

Feldman, Jon, Aranyak Mehta, Vahab Mirrokni, S Muthukrishnan. 2009. Online stochastic matching: Beating 1-1/e. *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*. IEEE, 117–126.

Gartner. 2016. Gartner says by 2020 "cloud shift" will affect more than $1 trillion in it spending. `http://www.gartner.com/newsroom/id/3384720/`.

Goel, Gagan, Aranyak Mehta. 2008. Online budgeted matching in random input models with applications to adwords. *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 982–991.

Hajiaghayi, Mohammad Taghi, Robert Kleinberg, Tuomas Sandholm. 2007. Automated online mechanism design and prophet inequalities. *AAAI*, vol. 7. 58–65.

Hill, Theodore P, Robert P Kertz. 1992. A survey of prophet inequalities in optimal stopping theory. *Contemporary Mathematics* **125**(1) 191.

Isard, Michael, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 261–276.

Jain, Navendu, Ishai Menache, Joseph Naor, Jonathan Yaniv. 2012. Near-optimal scheduling mechanisms for deadline-sensitive jobs in large computing clusters. *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 255–266.

Jain, Navendu, Ishai Menache, Joseph Seffi Naor, Jonathan Yaniv. 2014. A truthful mechanism for value-based scheduling in cloud computing. *Theory of Computing Systems* **54**(3) 388–406.

Javadi, Bahman, Ruppa K Thulasiram, Rajkumar Buyya. 2011. Statistical modeling of spot instance prices in public cloud environments. *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*. IEEE, 219–228.

Kleinberg, Robert. 2005. A multiple-choice secretary algorithm with applications to online auctions. *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 630–631.

Kleinberg, Robert, Seth Matthew Weinberg. 2012. Matroid prophet inequalities. *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. ACM, 123–136.

Krengel, Ulrich, Louis Sucheston. 1978. On semiamarts, amarts, and processes with finite value. *Advances in Prob* **4** 197–266.

Krengel, Ulrich, Louis Sucheston, et al. 1977. Semiamarts and finite values. *Bull. Amer. Math. Soc* **83**(4).

Li, Song, Yangfan Zhou, Lei Jiao, Xinya Yan, Xin Wang, Michael R Lyu. 2014. Delay-aware cost optimization for dynamic resource provisioning in hybrid clouds. *Web Services (ICWS), 2014 IEEE International Conference on*. IEEE, 169–176.

Lindley, Denis V. 1961. Dynamic programming and decision theory. *Applied Statistics* 39–51.

Mao, Ming, Marty Humphrey. 2011. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 49.

Mao, Ming, Marty Humphrey. 2012. A performance study on the vm startup time in the cloud. *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 423–430.

Menache, Ishai, Ohad Shamir, Navendu Jain. 2014. On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. *11th International Conference on Autonomic Computing (ICAC 14)*. USENIX Association.

Microsoft. 2016. Azure batch. http://azure.microsoft.com/en-us/documentation/services/batch/.

Molinaro, Marco, R Ravi. 2013. The geometry of online packing linear programs. *Mathematics of Operations Research* **39**(1) 46–59.

Palanisamy, Balaji, Aameek Singh, B Langston. 2013. Cura: A cost-optimized model for mapreduce in a cloud. *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 1275–1286.

Quantcast. 2015. Walmart.com traffic. `https://www.quantcast.com/walmart.com`. Accessed: 2015-02-08.

RightScale. 2016. State of the cloud report. `https://www.rightscale.com/lp/state-of-the-cloud/`.

Sarwar, Badrul, George Karypis, Joseph Konstan, John Riedl. 2001. Item-based collaborative filtering recommendation algorithms. *Proceedings of the 10th international conference on World Wide Web*. ACM, 285–295.

Song, Yang, Murtaza Zafer, Kang-Won Lee. 2012. Optimal bidding in spot instance market. *INFOCOM, 2012 Proceedings IEEE*. IEEE, 190–198.

Tang, ShaoJie, Jing Yuan, Xiang-Yang Li. 2012. Towards optimal bidding strategy for amazon ec2 cloud spot instance. *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 91–98.

Xu, Hong, Baochun Li. 2012. Maximizing revenue with dynamic cloud pricing: The infinite horizon case. *Communications (ICC), 2012 IEEE International Conference on*. IEEE, 2929–2933.

Yao, Min, Peng Zhang, Yin Li, Jie Hu, Chuang Lin, Xiang Yang Li. 2014. Cutting your cloud computing cost for deadline-constrained batch jobs. *Web Services (ICWS), 2014 IEEE International Conference on*. IEEE, 337–344.

Zaman, Sharrukh, Daniel Grosu. 2013. Combinatorial auction-based allocation of virtual machine instances in clouds. *Journal of Parallel and Distributed Computing* **73**(4) 495–508.

Zhang, Qi, Quanyan Zhu, Raouf Boutaba. 2011. Dynamic resource allocation for spot markets in cloud computing environments. *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*. IEEE, 178–185.

Zhao, Han, Miao Pan, Xinxin Liu, Xiaolin Li, Yuguang Fang. 2012. Optimal resource rental planning for elastic applications in cloud market. *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 808–819.