CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics

Omid Alipourfard, Yale University; Hongqiang Harry Liu and Jianshu Chen, Microsoft Research; Shivaram Venkataraman, University of California, Berkeley; Minlan Yu, Yale University; Ming Zhang, Alibaba Group

https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard

This paper is included in the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17).

March 27–29, 2017 • Boston, MA, USA

ISBN 978-1-931971-37-9

Open access to the Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.

CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics

Omid Alipourfard[1], Hongqiang Harry Liu[2], Jianshu Chen[2], Shivaram Venkataraman[3], Minlan Yu[1], Ming Zhang[4]

[1] *Yale University,* [2] *Microsoft Research,* [3] *University of California, Berkeley,* [4] *Alibaba Group*

Abstract – Picking the right cloud configuration for recurring big data analytics jobs running in clouds is hard, because there can be tens of possible VM instance types and even more cluster sizes to pick from. Choosing poorly can significantly degrade performance and increase the cost to run a job by 2-3x on average, and as much as 12x in the worst-case. However, it is challenging to automatically identify the best configuration for a broad spectrum of applications and cloud configurations with low search cost. *CherryPick* is a system that leverages Bayesian Optimization to build performance models for various applications, and the models are just accurate enough to distinguish the best or close-to-the-best configuration from the rest with only a few test runs. Our experiments on five analytic applications in AWS EC2 show that *CherryPick* has a 45-90% chance to find optimal configurations, otherwise near-optimal, saving up to 75% search cost compared to existing solutions.

1 Introduction

Big data analytics running on clouds are growing rapidly and have become critical for almost every industry. To support a wide variety of use cases, a number of evolving techniques are used for data processing, such as MapReduce, SQL-like languages, Deep Learning, and in-memory analytics. The execution environments of such big data analytic applications are structurally similar: a cluster of virtual machines (VMs). However, since different analytic jobs have diverse behaviors and resource requirements (CPU, memory, disk, network), their *cloud configurations* – the types of VM instances and the numbers of VMs – cannot simply be unified.

Choosing the right cloud configuration for an application is essential to service quality and commercial competitiveness. For instance, a bad cloud configuration can result in up to 12 times more cost for the same performance target. The saving from a proper cloud configuration is even more significant for *recurring* jobs [10, 17] in which similar workloads are executed repeatedly. Nonetheless, selecting the best cloud configuration, e.g., the cheapest or the fastest, is difficult due to the complexity of simultaneously achieving high accuracy, low overhead, and adaptivity for different applications and workloads.

Accuracy The running time and cost of an application have complex relations to the resources of the cloud instances, the input workload, internal workflows, and configuration of the application. It is difficult to use straightforward methods to model such relations. Moreover, cloud dynamics such as network congestions and stragglers introduce substantial noise [23, 39].

Overhead Brute-force search for the best cloud configuration is expensive. Developers for analytic applications often face a wide range of cloud configuration choices. For example, Amazon EC2 and Microsoft Azure offer over 40 VM instance types with a variety of CPU, memory, disk, and network options. Google provides 18 types and also allows customizing VMs' memory and the number of CPU cores [2]. Additionally, developers also need to choose the right cluster size.

Adaptivity Big data applications have diverse internal architectures and dependencies within their data processing pipelines. Manually learning to build the internal structures of individual applications' performance model is not scalable.

Existing solutions do not fully address all of the preceding challenges. For example, Ernest [37] trains a performance model for machine learning applications with a small number of samples but since its performance model is tightly bound to the particular structure of machine learning jobs, it does not work well for applications such as SQL queries (poor adaptivity). Further, Ernest can only select VM sizes within a given instance family, and performance models need to be retrained for each instance family.

In this paper, we present *CherryPick*—a system that unearths the optimal or near-optimal cloud configurations that minimize cloud usage cost, guarantee application performance and limit the search overhead for recurring big data analytic jobs. Each configuration is represented as the number of VMs, CPU count, CPU speed per core, RAM per core, disk count, disk speed, and network capacity of the VM.

The key idea of *CherryPick* is to build a performance model that is *just accurate enough* to allow us to distinguish near-optimal configurations from the rest. Tolerating the inaccuracy of the model enables us to achieve both low overhead and adaptivity: only a few samples

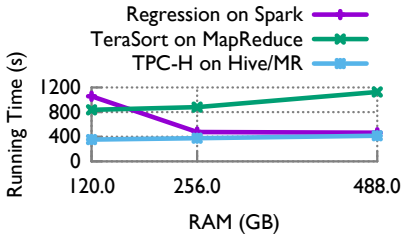


Figure 1: Regression and TeraSort with varying RAM size (64 cores)

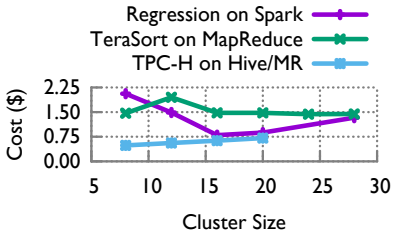


Figure 2: Regression and TeraSort cost with varying cluster size (M4)

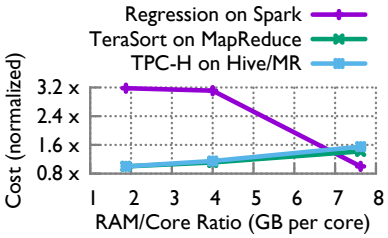


Figure 3: Regression and TeraSort cost with varying VM type (32 cores)

are needed and there is no need to embed application-specific insights into the modeling.

CherryPick leverages Bayesian Optimization (BO) [13, 28, 33], a method for optimizing black-box functions. Since it is non-parametric, it does not have any pre-defined format for the performance model. BO estimates a *confidence interval* (the range that the actual value should fall in with high probability) of the cost and running time under each candidate cloud configuration. The confidence interval is improved (narrowed) as more samples become available. *CherryPick* can judge which cloud configuration should be sampled next to best reduce the current uncertainty in modeling and get closer to go the optimal. *CherryPick* uses the confidence interval to decide when to stop the search. Section 3 provides more details on how BO works and why we chose BO out of other alternatives.

To integrate BO in *CherryPick* we needed to perform several customizations (Section 3.5): i) selecting features of cloud configurations to minimize the search steps; ii) handling noise in the sampled data caused by cloud internal dynamics; iii) selecting initial samples; and iv) defining the stopping criteria.

We evaluate *CherryPick* on five popular analytical jobs with 66 configurations on Amazon EC2. *CherryPick* has a high chance (45%-90%) to pick the optimal configuration and otherwise can find a near-optimal solution (within 5% at the median), while alternative solutions such as coordinate descent and random search can take up to 75% more running time and 45% more search cost. We also compare *CherryPick* with Ernest [37] and show how *CherryPick* can improve search time by 90% and search cost by 75% for SQL queries.

2 Background and Motivation

In this section, we show the benefits and challenges of choosing the best cloud configurations. We also present two strawman solutions to solve this problem.

2.1 Benefits

A good cloud configuration can reduce the cost of analytic jobs by a large amount. Table 1 shows the arithmetic mean and maximum running cost of configurations compared to the configuration with minimum running cost for four applications across 66 candidate configurations. The details of these applications and their cloud configurations are described in Section 5. For example, for the big data benchmark, TPC-DS, the average configuration costs 3.4 times compared to the configuration with minimum cost; if users happen to choose the worst configuration, they would spend 9.6 times more.

Application	Avg/min	Max/min
TPC-DS	3.4	9.6
TPC-H	2.9	12
Regression (SparkML)	2.6	5.2
TeraSort	1.6	3.0

Table 1: Comparing the maximum, average, and minimum cost of configurations for various applications.

Picking a good cloud configuration is even more important for recurring jobs where similar workloads are executed repeatedly, e.g. daily log parsing. Recent studies report that up to 40% of analytics jobs are recurring [10, 17]. Our approach only works for repeating jobs, where the cost of a configuration search can be amortized across many subsequent runs.

2.2 Challenges

There are several challenges for picking the best cloud configurations for big data analytics jobs.

Complex performance model: The running time is affected by the amount of resources in the cloud configuration in a *non-linear* way. For instance, as shown in Figure 1, a regression job on SparkML (with fixed number of CPU cores) sees a diminishing return of running time at 256GB RAM. This is because the job does not benefit from more RAM beyond what it needs. Therefore, the running time only sees marginal improvements.

In addition, performance under a cloud configuration is not deterministic. In cloud environments, which is shared among many tenants, stragglers can happen. We measured the running time of TeraSort-30GB on 22 different cloud configurations on AWS EC2 five times. We then computed the coefficient of variation (CV) of the five runs. Our results show that the median of the CV is about 10% and the 90 percentile is above 20%. This variation is not new [17].

Cost model: The cloud charges users based on the amount of time the VMs are up. Using configurations with a lot of resources could minimize the running time, but it may cost a lot more money. Thus, to minimize cost, we have to find the right balance between resource prices and the running time. Figure 2 shows the cost of running Regression on SparkML on different cluster sizes where each VM comes with 15 GBs of RAM and 4 cores in AWS EC2. We can see that the cost does not monotonically increase or decrease when we add more resources into the cluster. This is because adding resources may accelerate the computation but also raises the price per unit of running time.

The heterogeneity of applications: Figure 3 shows different shapes for TPC-DS and Regression on Spark and how they relate to instance types. For TeraSort, a low memory instance (8 core and 15 GBs of RAM) performs the best because CPU is a more critical resource. On the other hand for Regression, the same cluster has 2.4 times more running time than the best candidate due to the lack of RAM.

Moreover, the best choice often depends on the application configurations, e.g., the number of map and reduce tasks in YARN. Our work on identifying the best cloud configurations is complementary to other works on identifying the best application configurations (e.g., [19, 38]). *CherryPick* can work with any (even not optimal) application configurations.

2.3 Strawman solutions

The two strawman solutions for predicting a near optimal cloud configuration are modeling and searching.

Accurate modeling of application performance. One way is to model application performance and then pick the best configuration based on this model. However, this methodology has poor adaptivity. Building a model that works for a variety of applications and cloud configurations can be difficult because the knowledge of the internal structure of specific applications is needed to make the model effective. Moreover, building a model through human intervention for every new application can be tedious.

Static searching for the best cloud configuration. Another way is to exhaustively search for the best cloud configuration without relying on an accurate performance model. However, this methodology has high overhead. With 40 instance types at Amazon EC2 and tens of cluster sizes for an application, if not careful, one could end up needing tens if not hundreds of runs to identify the best instance. In addition, trying each cloud configuration multiple times to get around the dynamics in the cloud (due to resource multiplexing and stragglers) would exacerbate the problem even further.

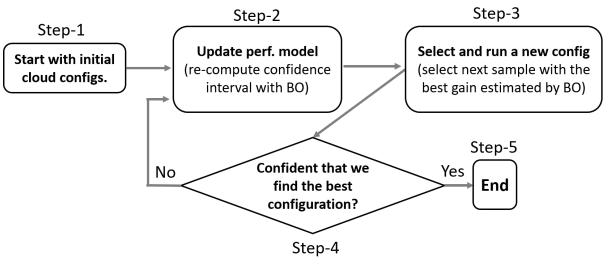


Figure 4: *CherryPick* workflow

To reduce the search time and cost, one could use *coordinate descent* and search one dimension at a time. Coordinate descent could start with searching for the optimal CPU/RAM ratio, then the CPU count per machine, then cluster size, and finally disk type. For each dimension, we could fix the other dimensions and search for the cheapest configuration possible. This could lead to suboptimal decisions if for example, because of bad application configuration a dimension is not fully explored or there are local minima in the problem space.

3 CherryPick Design

3.1 Overview

CherryPick follows a general principle in statistical learning theory [36]: "If you possess a restricted amount of information for solving some problem, try to solve the problem directly and never solve a more general problem as an intermediate step."

In our problem, the ultimate objective is to find the best configuration. We also have a very restricted amount of information, due to the limited runs of cloud configurations we can afford. Therefore, the model does not have enough information to be an accurate performance predictor, but this information is sufficient to find a good configuration within a few steps.

Rather than accurately predicting application performance, we just need a model that is accurate enough for us to separate the best configuration from the rest.

Compared to static searching solutions, we dynamically adapt our searching scheme based on the current understanding and confidence interval of the performance model. We can dynamically pick the next configuration that can best distinguish performance across configurations and eliminate unnecessary trials. The performance model can also help us understand when to stop searching earlier once we have a small enough confidence interval. Thus, we can reach the best configuration faster than static approaches.

Figure 4 shows the joint process of performance modeling and configuration searching. We start with a few initial cloud configurations (e.g., three), run them, and input the configuration details and job completion time into the performance model. We then *dynamically* pick the next cloud configuration to run based on the perfor-

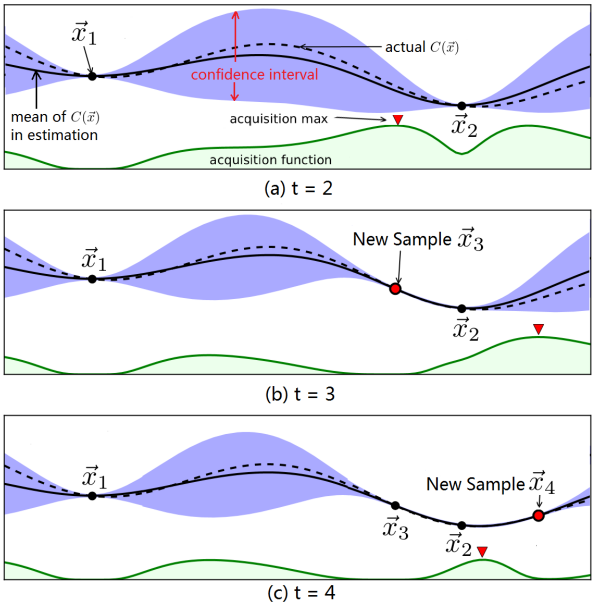


Figure 5: An example of BO's working process (derived from Figure 1 in [13]).

mance model and feed the result back to the performance model. We stop when we have enough confidence that we have found a good configuration.

3.2 Problem formulation

For a given application and workload, our goal is to find the optimal or a near-optimal cloud configuration that satisfies a performance requirement and minimizes the total execution cost. Formally, we use $T(\vec{x})$ to denote the running time function for an application and its input workloads. The running time depends on the cloud configuration vector $\vec{x}$, which includes instance family types, CPU, RAM, and other resource configurations.

Let $P(\vec{x})$ be the price per unit time for all VMs in cloud configuration $\vec{x}$. We formulate the problem as follows:

$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & C(\vec{x}) = P(\vec{x}) \times T(\vec{x}) \\ \text{subject to} \quad & T(\vec{x}) \leq \mathscr{T}_{max} \end{aligned} \tag{1}$$

where $C(\vec{x})$ is the total cost of cloud configuration $\vec{x}$ and $\mathscr{T}_{max}$ is the maximum tolerated running time[1]. Knowing $T(\vec{x})$ under all candidate cloud configurations would make it straightforward to solve Eqn (1), but it is expensive because all candidate configurations need to be tried. Instead, we use BO (with Gaussian Process Priors, see Section 6) to directly search for an approximate solution of Eqn (1) with significantly smaller cost.

3.3 Solution with Bayesian Optimization

Bayesian Optimization (BO) [13, 28, 33] is a framework to solve optimization problem like Eqn. (1) where the objective function $C(\vec{x})$ is unknown beforehand but can be observed through experiments. By modeling $C(\vec{x})$ as a stochastic process, e.g. a Gaussian Process [26], BO can compute the *confidence interval* of $C(\vec{x})$ according to one or more samples taken from $C(\vec{x})$. A confidence interval is an area that the curve of $C(\vec{x})$ is most likely (e.g. with 95% probability) passing through. For example, in Figure 5(a), the dashed line is the actual function $C(\vec{x})$. With two samples at $\vec{x}_1$ and $\vec{x}_2$, BO computes a confidence interval that is marked with a blue shadowed area. The black solid line shows the expected value of $C(\vec{x})$ and the value of $C(\vec{x})$ at each input point $\vec{x}$ falls in the confidence interval with 95% probability. The confidence interval is updated (posterior distribution in Bayesian Theorem) after new samples are taken at $\vec{x}_3$ (Figure 5(b)) and $\vec{x}_4$ (Figure 5(c)), and the estimate of $C(\vec{x})$ improves as the area of the confidence interval decreases.

BO can smartly decide the next point to sample using a pre-defined *acquisition function* that also gets updated with the confidence interval. As shown in Figure 5, $\vec{x}_3$ ($\vec{x}_4$) is chosen because the acquisition function at $t = 2$ ($t = 3$) indicates that it has the most potential gain. There are many designs of acquisition functions in the literature, and we will discuss how we chose among them in Section 3.5.

BO is embedded into *CherryPick* as shown in Figure 4. At Step 2, *CherryPick* leverages BO to update the confidence interval of $C(\vec{x})$. After that, at Step 3, *CherryPick* relies on BO's acquisition function to choose the best configuration to run next. Also, at Step 4, *CherryPick* decides whether to stop the search according to the confidence interval of $C(\vec{x})$ provided by BO (details shown in Section 3.5).

Another useful property of BO is that it can accommodate observation noise in the computation of confidence interval of the objective function. Suppose in practice, given an input point $\vec{x}$, we have no direct access to $C(\vec{x})$ but can only observe $C(\vec{x})'$ that is:

$$C(\vec{x})' = C(\vec{x}) + \varepsilon \tag{2}$$

where ε is a Gaussian noise with zero mean, that is $\varepsilon \sim \mathscr{N}(0, \sigma_\varepsilon^2)$. Because $C(\vec{x})'$ is also Gaussian, BO is able to infer the confidence interval of $C(\vec{x})$ according to the samples of $C(\vec{x})'$ and ε [13]. Note that in our scenario, the observation noise on $C(\vec{x})$ is negligible because the measurement on running time and price model is accurate enough. However, the ability to handle the additive noise of BO is essential for us to handle the uncertainty in clouds (details in Section 3.6).

In summary, by integrating BO, *CherryPick* has the ability to learn the objective function quickly and only take samples in the areas that most likely contain the minimum point. For example, in Figure 5(c) both $\vec{x}_3$ and $\vec{x}_4$ are close to the minimum point of the actual $C(\vec{x})$,

[1] $C(\vec{x})$ assumes a fixed number of identical VMs.

leaving the interval between $\vec{x}_1$ and $\vec{x}_4$ unexplored without any impact on the final result.

3.4 Why do we use Bayesian Optimization?

BO is effective in finding optimal cloud configurations for Big Data analytics for three reasons.

First, BO does not limit the function to be of any predefined format, as it is non-parametric. This property makes *CherryPick* useful for a variety of applications and cloud configurations.

Second, BO typically needs a small number of samples to find a near-optimal solution because BO focuses its search on areas that have the largest expected improvements.

Third, BO can tolerate uncertainty. *CherryPick* faces two main sources of uncertainty: (i) because of the small number of samples, *CherryPick*'s performance models are imperfect and usually have substantial prediction errors; (ii) the cloud may not report a stable running time even for the same application due to resource multiplexing across applications, stragglers, etc. BO can quantitatively define the uncertainty region of the performance model. The confidence interval it computes can be used to guide the searching decisions even in face of model inaccuracy. In Section 3.6, we leverage this property of BO to handle the uncertainty from cloud dynamics.

One limitation of BO is that its computation complexity is $O(N^4)$, where N is the number of data samples. However, this is perfectly fine because our data set is small (our target is typically less than 10 to 20 samples out of hundreds of candidate cloud configurations).

Alternatives Alternative solutions often miss one of the above benefits: (1) linear regression and linear reinforcement learning are not generic to all applications because they do not work for non-linear models; (2) techniques that try to model a function (e.g., linear regression, support vector regression, boosting tree, etc.) do not consider minimizing the number of sample points. Deep neural networks [27], table-based modeling [11], and Covariance matrix adaptation evolution strategy (CMA-ES) [25] can potentially be used for black-box optimization but require a large number of samples. (3) It is difficult to adapt reinforcement learning [27, 35] to handle uncertainty and minimize the number of samples while BO models the uncertainty so as to accelerate the search.

3.5 Design options and decisions

To leverage Bayesian Optimization to find a good cloud configuration, we need to make several design decisions based on system constraint and requirements.

Prior function As most BO frameworks do, we choose to use Gaussian Process as the prior function. It means that we assume the final model function is a sample from Gaussian Process. We will discussion this choice in more details in Section 6.

We describe $C(\vec{x})$ with a mean function $\mu(\cdot)$ and covariance kernel function $k(\cdot,\cdot)$. For any pairs of input points $\vec{x}_1, \vec{x}_2$, we have:

$$\mu(\vec{x}_1) = \mathbb{E}[C(\vec{x}_1)]; \mu(\vec{x}_2) = \mathbb{E}[C(\vec{x}_2)]$$
$$k(\vec{x}_1,\vec{x}_2) = \mathbb{E}[(C(\vec{x}_1) - \mu(\vec{x}_1))(C(\vec{x}_2) - \mu(\vec{x}_2))]$$

Intuitively, we know that if two cloud configurations, $\vec{x}_1$ and $\vec{x}_2$ are similar to each other, $C(\vec{x}_1)$ and $C(\vec{x}_2)$ should have large covariance, and otherwise, they should have small covariance. To express this intuition, people have designed numerous formats of the covariance functions between inputs $\vec{x}_1$ and $\vec{x}_2$ which decrease when $||\vec{x}_1 - \vec{x}_2||$ grow. We choose *Matern5/2* [31] because it does not require strong smoothness and is preferred to model practical functions [33].

Acquisition function There are three main strategies to design an acquisition function [33]: (i) Probability of Improvement (PI) – picking the point which can maximize the probability of improving the current best; (ii) Expected Improvement (EI) – picking the point which can maximize the expected improvement over the current best; and (iii) Gaussian Process Upper Confidence Bound (GP-UCB) – picking the point whose certainty region has the smallest lower bound(when we minimize a function). In *CherryPick* we choose EI [13] as it has been shown to be better-behaved than PI, and unlike the method of GP-UCB, it does not require its own tuning parameter [33].

Jones et al. [22] derive an easy-to-compute closed form for the EI acquisition function. Let X_t be the collection of all cloud configurations whose function values have been observed by round t, and $m = \min_{\vec{x}}\{C(\vec{x})|\vec{x} \in X_t\}$ as the minimum function value observed so far. For each input $\vec{x}$ which is not observed yet, we can evaluate its expected improvement if it is picked as the next point to observe with the following equation:

$$EI(\vec{x}) = \begin{cases} (m - \mu(\vec{x}))\Phi(Z) + \sigma(\vec{x})\phi(Z), & \text{if } \sigma(\vec{x}) > 0 \\ 0, & \text{if } \sigma(\vec{x}) = 0 \end{cases} \quad (3)$$

where $\sigma(\vec{x}) = \sqrt{k(\vec{x},\vec{x})}$, $Z = \frac{m-\mu(\vec{x})}{\sigma(\vec{x})}$, and Φ and ϕ are standard normal cumulative distribution function and the standard normal probability density function respectively.

The acquisition function shown in Eqn (3) is designed to minimize $C(\vec{x})$ without further constraints. Nonetheless, from Eqn 1 we know that we still have a performance constraint $T(\vec{x}) \leq \mathcal{T}_{max}$ to consider. It means that when we choose the next cloud configuration to evaluate, we should have a bias towards one that is likely to satisfy the performance constraint. To achieve this goal, we first build the model of running time function $T(\vec{x})$ from $\frac{C(\vec{x})}{P(\vec{x})}$.

Then, as suggested in [18], we modify the EI acquisition function as:

$$EI(\vec{x})' = P[T(\vec{x}) \leq \mathscr{T}_{max}] \times EI(\vec{x}) \quad (4)$$

Stopping condition We define the stopping condition in *CherryPick* as follows: when the expected improvement in Eqn.(4) is less than a threshold (e.g. 10%) and at least N (e.g. $N = 6$) cloud configurations have been observed. This ensures that *CherryPick* does not stop the search too soon and it prevents *CherryPick* from struggling to make small improvements.

Starting points Our choice of starting points should give BO an estimate about the shape of the cost model. For that, we sample a few points (e.g., three) from the sample space using a quasi-random sequence [34]. Quasi-random numbers cover the sample space more uniformly and help the prior function avoid making wrong assumptions about the sample space.

Encoding cloud configurations We encode the following features into $\vec{x}$ to represent a cloud configuration: the number of VMs, the number of cores, CPU speed per core, average RAM per core, disk count, disk speed and network capacity of a VM.

To reduce the search space of the Bayesian Optimization, we normalize and discretized most of the features. For instance, for disk speed, we only define fast and slow to distinguish SSD and magnetic disks. Similarly, for CPU, we use fast and slow to distinguish high-end and common CPUs. Such discretization significantly reduces the space of several features without losing the key information brought by the features and it also helps to reduce the number of invalid cloud configurations. For example, we can discretize the space so that the CPUs greater (smaller) than 2.2GHz are fast (slow) and the disks with bandwidth greater (smaller) than 600MB/s are fast (slow). Then, if we suggest a (fast, fast) combination for (CPU, Disk), we could choose a 2.5Ghz and 700MBs instance (or any other one satisfying the boundary requirements). Or in place of a (slow, slow) configuration we could pick an instance with 2Ghz of speed and 400MB/s of IO bandwidth. If no such configurations exist, we can either remove that point from the candidate space that BO searches or return a large value, so that BO avoids searching in that space.

3.6 Handling uncertainties in clouds

So far we assumed that the relation between cloud configurations and cost (or running time) is deterministic. However, in practice, this assumption can be broken due to uncertainties within any shared environment. The resources of clouds are shared by multiple users so that different users' workload could possibly have interference with each other. Moreover, failures and resource overloading, although potentially rare, can impact the completion time of a job. Therefore, even if we run the same workload on the same cloud with the same configuration for multiple times, the running time and cost we get may not be the same.

Due to such uncertainties in clouds, the running time we can observe from an actual run on configuration $\vec{x}$ is $\tilde{T}(\vec{x})$ and the cost is $\tilde{C}(\vec{x})$. If we let $T(\vec{x}) = \mathbb{E}[\tilde{T}(\vec{x})]$ and $C(\vec{x}) = \mathbb{E}[\tilde{C}(\vec{x})]$, we have:

$$\tilde{T}(\vec{x}) = T(\vec{x})(1+\varepsilon_c) \quad (5)$$

$$\tilde{C}(\vec{x}) = C(\vec{x})(1+\varepsilon_c) \quad (6)$$

where ε_c is a multiplicative noise introduced by the uncertainties in clouds. We model ε_c as normally distributed: $\varepsilon_c \sim \mathcal{N}(0, \sigma^2_{\varepsilon_c})$.

Therefore, Eqn (1) becomes minimizing the expected cost with the expected performance satisfying the constraint.

BO cannot infer the confidence interval of $C(\vec{x})$ from the observation of $\tilde{C}(\vec{x})$ because the latter is not normally distributed given that BO assumes $C(\vec{x})$ is Gaussian and so is $(1+\varepsilon_c)$. One straightforward way to solve this problem is to take multiple samples at the same configuration $\vec{x}$, so that $C(\vec{x})$ can be obtained from the average of the multiple $\tilde{C}(\vec{x})$. Evidently, this method will result in a big overhead in search cost.

Our key idea to solve this problem (so that we only take one sample at each input) is to transform Eqn. (1) to the following equivalent format:

$$\begin{aligned} \underset{\vec{x}}{\text{minimize}} \quad & \log C(\vec{x}) = \log P(\vec{x}) + \log T(\vec{x}) \\ \text{subject to} \quad & \log T(\vec{x}) \leq \log \mathscr{T}_{max} \end{aligned} \quad (7)$$

We use BO to minimize $\log C(\vec{x})$ instead of $C(\vec{x})$ since:

$$\log \tilde{C}(\vec{x}) = \log C(\vec{x}) + \log(1+\varepsilon_c) \quad (8)$$

Assuming that ε_c is less than one (e.g. $\varepsilon_c < 1$), $\log(1+\varepsilon_c)$ can be estimated by ε_c, so that $\log(1+\varepsilon_c)$ can be viewed as an observation noise with a normal distribution, and $\log \tilde{C}(\vec{x})$ can be treated as the observed value of $\log C(\vec{x})$ with observation noise. Eqn.(8) can be solved similar to Eqn.(2).

In the implementation of *CherryPick*, we use Eqn. (7) instead of Eqn. (1) as the problem formulation.

4 Implementation

In this section, we discuss the implementation details of *CherryPick* as shown in Figure 6. It has four modules.

1. Search Controller: Search Controller orchestrates the entire cloud configuration selection process. To use *CherryPick*, users supply a representative workload (see Section 6) of the application, the objective (e.g. minimizing cost or running time), and the constraints (e.g. cost budget, maximum running time, preferred instance

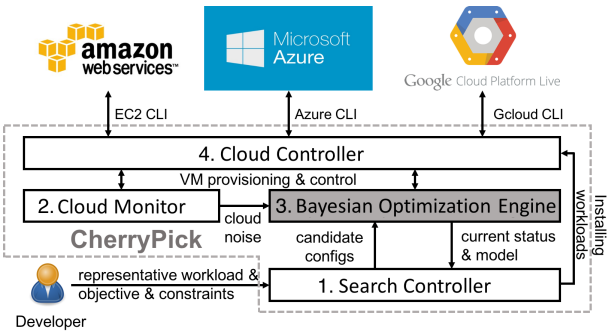


Figure 6: Architecture of *CherryPick*'s implementation.

types, maximum/minimum cluster size, etc.). Based on these inputs, the search controller obtains a list of candidate cloud configurations and passes it to the Bayesian Optimization Engine. At the same time, Search Controller installs the representative workload to clouds via Cloud Controller. This process includes creating VMs in each cloud, installing the workload (applications and input data), and capturing a customized VM image which contains the workload. Search Controller also monitors the current status and model on the Bayesian Optimization engine and decides whether to finish the search according to the stopping condition discussed in Section 3.5.

2. Cloud Monitor: Cloud Monitor runs benchmarking workloads of Big Data defined by *CherryPick* on different clouds. It repeats running numerous categories of benchmark workloads on each cloud to measure the upper-bound (or high percentile) of the cloud noise [2]. The result is offered to Bayesian Optimization engine as the ε_c in Eqn. (8). This monitoring is lightweight; we only need to run this system every few hours with a handful of instances.

3. Bayesian Optimization Engine: Bayesian Optimization Engine is built on top of Spearmint [6] which is an implementation of BO in Python. Besides the standard BO, it also has realized our acquisition function in Eqn (3) and the performance constraint in Eqn (4). However, Spearmint's implementation of Eqn (4) is not efficient for our scenario because it assumes $C(\vec{x})$ and $T(\vec{x})$ are independent and trains the models of them separately. We modified this part so that $T(\vec{x})$ is directly derived from $\frac{C(\vec{x})}{P(\vec{x})}$ after we get the model of $C(\vec{x})$. Our implementation of this module focuses on the interfaces and communications between this module and others. For taking a sample of a selected cloud configuration, the BO engine submits a cluster creation request and a start workload request via the Cloud Controller.

4. Cloud Controller: Cloud Controller is an adaptation layer which handles the heterogeneity to control the clouds. Each cloud has its own APIs and semantics to do the operations such as create/delete VMs, create/delete virtual networks, capturing images from VMs, and list the available instance types. Cloud Controller defines a uniform API for the other modules in *CherryPick* to perform these operations. In addition, the API also includes sending commands directly to VMs in clouds via SSH, which facilitates the control of the running workload in the clouds.

The entire *CherryPick* system is written in Python with about 5,000 lines of code, excluding the legacy part of Spearmint.

[2] Over-estimating ε_c means more search cost.

5 Evaluation

We evaluate *CherryPick* with 5 types of big data analytics applications on 66 cloud configurations. Our evaluations show that *CherryPick* can pick the optimal configuration with a high chance (45-90%) or find a near-optimal configuration (within 5% of the optimal at the median) with low search cost and time, while alternative solutions such as coordinate descent and random search can reach up to 75% more running time and up to 45% more search time than *CherryPick*. We also compare *CherryPick* with Ernest [37] and show how *CherryPick* can reduce the search time by 90% and search cost by 75% for SQL queries. We discuss insights on why *CherryPick* works well and show how *CherryPick* adapt to changing workloads and various performance constraints.

5.1 Experiment setup

Applications: We chose benchmark applications on Spark [44] and Hadoop [41] to exercise different CPU/Disk/RAM/Network resources: *(1) TPC-DS* [7] is a recent benchmark for big data systems that models a decision support workload. We run TPC-DS benchmark on Spark SQL with a scale factor of 20. *(2) TPC-H* [8] is another SQL benchmark that contains a number of ad-hoc decision support queries that process large amounts of data. We run TPC-H on Hadoop with a scale factor of 100. Note that our trace runs 20 queries concurrently. While it may be possible to model each query's performance, it is hard to model the interactions of these queries together. *(3) TeraSort* [29] is a common benchmarking application for big data analytics frameworks [1, 30], and requires a balance between high IO bandwidth and CPU speed. We run TeraSort on Hadoop with 300 GB of data, which is large enough to exercise disks and CPUs together. *(4) The SparkReg* [4] benchmark consists of machine learning workloads implemented on top of Spark. We ran the regression workload in SparkML with 250k examples, 10k features, and 5 iterations. This workload heavily depends on memory space for caching data and has minimal use for disk IO. *(5) SparkKm* is another SparkML benchmark [5]. It is

Instance Size	Number of instances					
large	16	24	32	40	48	56
xlarge	8	12	16	20	24	28
2xlarge	4	6	8	10	12	14
Number of Cores	32	48	64	80	96	112

Table 2: Configurations for one instance family.

a clustering algorithm that partitions a space into k clusters with each observation assigned to the cluster with the closest mean. We use 250k observations with 10k features. Similar to SparkReg, this workload is dependent on memory space and has less stringent requirements for CPU and disk IO.

Cloud configurations: We choose four families in Amazon EC2: M4 (general purpose), C4 (compute optimized), R3 (memory optimized), I2 (disk optimized) instances. Within each family, we used large, xlarge, and 2xlarge instance sizes each with 2, 4, and 8 cores per machine respectively. For each instance size, we change the total number of cores from 32 to 112. Table 2 shows the 18 configurations for each of the four families. We run a total of 66 configurations, rather than 72 (18×4), because the smallest size for I2 starts from xlarge. We do not choose more configurations due to time and expense constraints. However, we make sure the configurations we choose are reasonable for our objective (i.e., minimizing cost). For example, we did not choose 4xlarge instances because 4xlarge is more expensive than 2xlarge but shows diminishing returns in terms of running time.

Objectives: We define the objective as minimizing the cost of executing the application under running time constraints. By default, we set a loose constraint for running time so *CherryPick* searches through a wider set of configurations. We evaluate tighter constraints in Section 5.4. Note that minimizing running time with no cost constraint always leads to larger clusters, and therefore, is rather simple. On the other hand, minimizing the cost depends on the right balance between cluster size and cluster utilization.

***CherryPick* settings:** By default, we use EI= 10%, $N = 6$, and 3 initial samples. In our experiments, we found that EI= 10% gives a good trade-off between search cost and accuracy. We also tested other EI values in one experiment.

Alternative solutions: We compare *CherryPick* with the following strategies: *(1) Exhaustive search*, which finds the best configuration by running all the configurations; *(2) Coordinate descent*, which searches one coordinate – in order of CPU/RAM ratio (which specifies the instance family type), CPU count, cluster size, disk type – at a time (Section 2.3) from a randomly chosen starting point. The ordering of dimensions can also impact the result. It is unclear whether a combination of dimensions and ordering exists that works best across all applications. Similar approaches have been used for tuning configurations for Map Reduce jobs and web servers [24, 42]. *(3) Random search with a budget*, which randomly picks a number of configurations given a search budget. Random search is used by previous configuration tuning works [20, 43]. *(4) Ernest [37]*, which builds a performance model with common communication patterns. We run Ernest once per-instance type and use the model to predict the optimal number of instances.

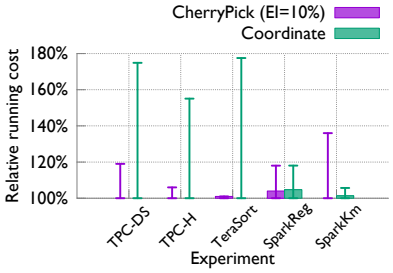
Metrics: We compare *CherryPick* with alternative solutions using two metrics: (i) the running cost of the configuration: the expense to run a job with the selected configuration; (ii) the search cost: the expense to run all the sampled configurations. All the reported numbers are normalized by the exhaustive search cost and running cost across the clusters in Table 2.

We run *CherryPick* and random search 20 times with different seeds for starting points. For the coordinate descent, we start from all the 66 possible starting configurations. We then show the 10th, median, and 90th percentile of the search cost and running cost of *CherryPick* normalized by the optimal configuration reported by exhaustive search.

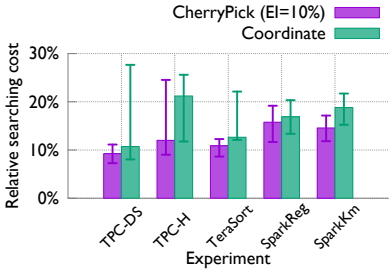
5.2 Effectiveness of *CherryPick*

***CherryPick* finds the optimal configuration in a high chance (45-90%) or a near-optimal configuration with low search cost and time:** Figure 7a shows the median, 10th percentile, and 90th percentile of running time for the configuration picked by *CherryPick* for each of the five workloads. *CherryPick* finds the exact optimal configuration with 45-90% chance, and finds a configuration within 5% of the optimal configuration at the median. However, using exhaustive search requires 6-9 times more search cost and 5-9.5 times more search time compared with *CherryPick*. On AWS, which charges on an hourly basis, after running TeraSort 100 times, exhaustive search costs $581 with $49 for the remainder of the runs. While *CherryPick* uses $73 for searching and $122 for the rest of the runs saving a total of $435.

In terms of accuracy, we find that that *CherryPick* has good accuracy across applications. On median, *CherryPick* finds an optimal configuration within 5% of the optimal configuration. For TPC-DS, *CherryPick* finds a configuration within 20% of the optimal in the 90th percentile; For TPC-H, the 90th percentile is 7% worse than optimal configuration; Finally, for TeraSort, SparkReg, and SparkKm *CherryPick*'s 90th percentile configuration is 0%, 18%, 38% worse than the optimal respectively. It is possible to change the EI of *CherryPick* to find even better configurations.



(a) Running cost



(b) Search cost

Figure 7: Comparing *CherryPick* with coordinate descent. The bars show 10th and 90th percentile.

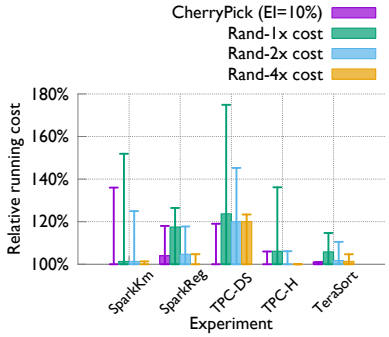


Figure 8: Running cost of configurations by *CherryPick* and random search. The bars show 10th and 90th percentile.

***CherryPick* is more stable in picking near-optimal configurations and has less search cost than coordinate descent.** Across applications, the median configuration suggested by coordinate descent is within 7% of the optimal configuration. On the other hand, the tail of the configuration suggested by coordinate descent can be far from optimal. For TPC-DS, TPC-H, and TeraSort, the tail configuration is 76%, 56%, and 78% worse than optimal, while using comparable or more search cost. This is because coordinate descent can be misled by the result of the run. For example, for TPC-DS, C4 family type has the best performance. In our experiment, if coordinate descent starts its search from a configuration with a large number of machines, the C4 family fails to finish the job successfully due to the scheduler failing. Therefore, the C4 family is never considered in the later iterations of coordinate descent runs. This leads coordinate descent to a suboptimal point that can be much worse than the optimal configuration.

In contrast, *CherryPick* has stronger ability to navigate around these problems because even when a run fails to finish on a candidate configuration, it uses Gaussian process to model the global behavior of the function from the sampled configurations.

***CherryPick* reaches better configurations with more stability compared with random search with similar budget:** Figure 8 compares the running cost of configurations suggested by *CherryPick* and random search with equal/2x/4x search cost. With the same search cost, random search performs up to 25% worse compared to *CherryPick* on the median and 45% on the tail. With 4x cost, random search can find similar configurations to *CherryPick* on the median. Although *CherryPick* may end up with different configurations with different starting points, it consistently has a much higher stability of the running cost compared to random search. *CherryPick* has a comparable stability to random search with 4x budget, since random search with a 4x budget almost visits all the configurations at least once.

***CherryPick* reaches configurations with similar running cost compared with Ernest [37], but with lower search cost and time:** It is hard to extend Ernest to work with a variety of applications because it requires using a small representative dataset to build the model. For example, TPC-DS contains 99 queries on 24 tables, where each query touches a different set of tables. This makes it difficult to determine which set of tables should be sampled to build a representative small-scale experiment. To overcome this we use the TPC-DS data generator and generate a dataset with scale factor 2 (10% of target data size) and use that for training. We then use Ernest to predict the best configuration for the target data size. Finally, we note that since Ernest builds a separate model for each instance type we repeat the above process 11 times, once for each instance type.

Figure 9 shows that Ernest picks the best configuration for TPC-DS, the same as *CherryPick*, but takes 11 times the search time and 3.8 times the search cost. Although Ernest identifies the best configuration, its predicted running time is up to 5 times of the actual running time. This is because, unlike iterative ML workloads, the TPC-DS performance model has a complex scaling behavior with input scale and this is not captured by the linear model used in Ernest. Thus, once we set a tighter performance constraint, Ernest suggests a configuration that is 2 times more expensive than *CherryPick* with 2.8 times more search cost.

***CherryPick* can tune EI to trade-off between search cost and accuracy:** The error of the tail configuration for SparkKm as shown in Figure 7a can be as high as 38%. To get around this problem, the users can use lower values of EI to find better configurations. Figure 10 shows the running cost and search cost for different values of EI. At $EI < 6\%$, *CherryPick* has much

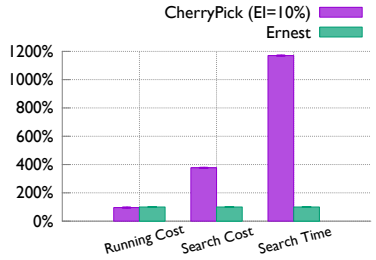


Figure 9: Comparing Ernest to *CherryPick* (TPC-DS).

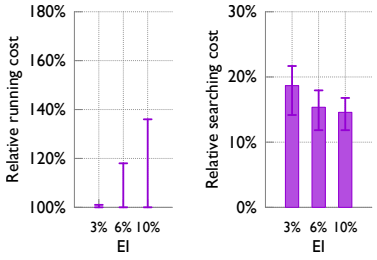


Figure 10: Search cost and running cost of SparkKm with different EI values.

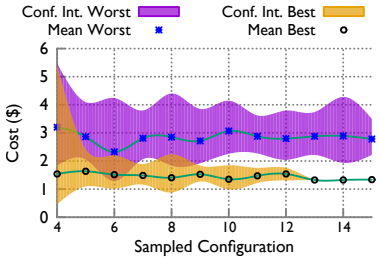
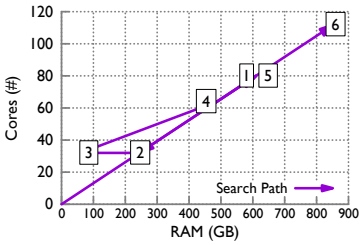
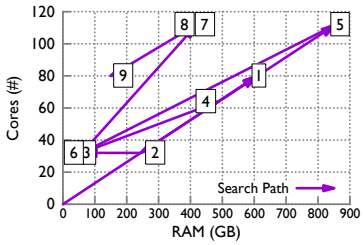


Figure 11: Bayesian opt. process for the best/worst configuration (TeraSort).



(a) SparkReg



(b) TPC-DS

	SparkReg		TPC-DS	
Step	**VM Type**	**# VMs**	**VM Type**	**# VMs**
1	r3.2xlarge	10	r3.2xlarge	10
2	r3.2xlarge	4	r3.2xlarge	4
3	c4.xlarge	8	c4.xlarge	8
4	r3.large	32	r3.large	32
5	i2.2xlarge	10	r3.2xlarge	14
6	r3.2xlarge	14	c4.2xlarge	4
7			m4.xlarge	28
8			m4.2xlarge	14
9			c4.2xlarge	10

(c) Search path for TPC-DS and SparkReg

Figure 12: Search path for TPC-DS and SparkReg

better accuracy, finding configurations that at 90th percentile are within 18% of the optimal configuration. If we set $EI < 3\%$, *CherryPick* suggests configurations that are within 1% of the optimal configuration at 90th percentile resulting in a 26% increase in search cost.

This can be a knob where users of *CherryPick* can trade-off optimality for search cost. For example, if users of *CherryPick* predict that the recurring job will be popular, setting a low EI value can force *CherryPick* to look for better configurations more carefully. This may result in larger savings over the lifetime of the job.

5.3 Why *CherryPick* works?

We now show *CherryPick* behavior matches the key insights discussed in Section 3.1. For this subsection, we set the stopping condition EI<1% to make it easier to show how *CherryPick* navigates the space.

Previous performance prediction solutions require many training samples to improve prediction accuracy. *CherryPick* spends the budget to improve the prediction accuracy of those configurations that are closer to the best. Figure 11 shows the means and confidence intervals of the running cost for the best and worst configurations, and how the numbers change during the process of Bayesian optimization. Initially, both configurations have large confidence intervals. As the search progresses, the confidence interval for the best configuration narrows. In contrast, the estimated cost for the worst configuration has a larger confidence interval and remains large. This is because *CherryPick* focuses on improving the estimation for configurations that are closer to the optimal.

Although *CherryPick* takes a black-box approach, it automatically learns the relation between cloud resources and the running time. Figure 13 shows *CherryPick*'s final estimation of the running time versus cluster size. The real curve follows Amdahl's law: (1) adding more VMs reduces the running time; (2) at some point, adding more machines has diminishing returns due to the sequential portion of the application. The real running time falls within the confidence interval of *CherryPick*. Moreover, *CherryPick* has smaller confidence intervals for the more promising region where the best configurations (those with more VMs) are located. It does not bother to improve the estimation for configurations with fewer VMs.

Even though *CherryPick* has minimal information about the application, it adapts the search towards the features that are more important to the application. Figure 12 shows example search paths for TPC-DS and SparkReg from the same three starting configurations. For SparkReg, *CherryPick* quickly identifies that clusters with larger RAM (R3 instances) have better performance and redirects the search towards such instances. In contrast, for TPC-DS, the last few steps suggest that *CherryPick* has identified that CPU is more important, and therefore the exploration is directed towards VMs with better CPUs (C4 instances). Figure 12 shows that *CherryPick* directly searches more configurations with larger #cores for TPC-DS than for SparkReg.

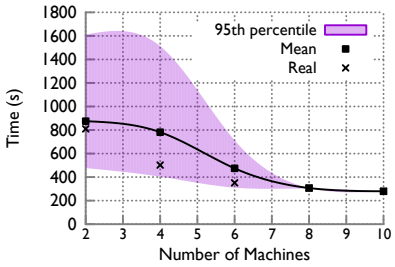


Figure 13: *CherryPick* learns diminishing returns of larger clusters (TPC-H, c4.2xlarge VMs).

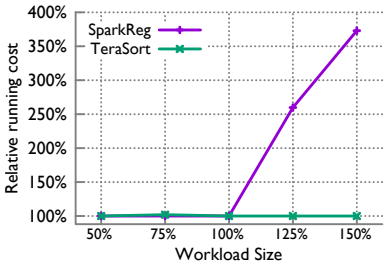


Figure 14: Sensitivity to workload size

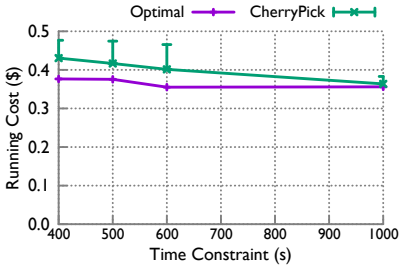


Figure 15: *CherryPick* works with time constraints (TPC-H).

5.4 Handling workload changes

CherryPick depends on representative workloads. Thus, one concern is *CherryPick*'s sensitivity to the variation of input workloads. In Figure 14, we keep the best configuration for the original workload (100% input size) C_{100} and test the running cost of the $C_{100\%}$ on workloads with 50% to 150% of the original input size. For TeraSort, we can continue to use $C_{100\%}$ to achieve the optimal cost with different input sizes. For SparkReg, $C_{100\%}$ remains effective for smaller workloads. However, when the workload is increased by 25%, $C_{100\%}$ can get to 260% the running cost of the new best configuration ($C_{125\%}$). This is because $C_{100\%}$ does not have enough RAM for SparkReg, which leads to more disk accesses.

Since input workloads usually vary in practice, *CherryPick* needs a good selection of representative workloads For example, for SparkReg, we should choose a relatively larger workload as the representative workload (e.g., choosing 125% gives you more stability than choosing 100%). We will discuss more on how to select representative workloads in Section 6.

When the difference between *CherryPick*'s estimation of the running cost and the actual running cost is above a threshold, the user can rerun *CherryPick*. For example, in Figure 14, suppose the user trains *CherryPick* with a 100% workload for SparkReg. With a new workload at size 125%, when he sees the running cost becomes 2x higher than expected, he can rerun *CherryPick* to build a new model for the 125% workload.

5.5 Handling performance constraints

We also evaluate *CherryPick* with tighter performance constraints on the running time (400 seconds to 1000 seconds) for TPC-H, as shown in Figure 15. *CherryPick* consistently identifies near-optimal configuration (2-14% difference with the optimal) with similar search cost to the version without constraints.

6 Discussion

Representative workloads: *CherryPick* relies on representative workloads to learn and suggest a good cloud configuration for similar workloads. Two workloads are similar if they operate on data with similar structures and sizes, and the computations on the data are similar. For example, for recurring jobs like parsing daily logs or summarizing daily user data with the same SQL queries, we can select one day's workload to represent the following week or month, if in this period the user data and the queries are not changing dramatically. Many previous works were built on top of the similarity in recurring jobs [10, 17]. Picking a representative workload for non-recurring jobs hard, and for now, *CherryPick* relies on human intuitions. An automatic way to select representative workload is an interesting avenue for future work.

The workload for recurring jobs can also change with time over a longer term. *CherryPick* detects the need to recompute the cloud configuration when it finds large gaps between estimated performance and real performance under the current configuration.

Larger search space: With the customizable virtual machines [2] and containers, the number of configurations that users can run their applications on becomes even larger. In theory, the large candidate number should not impact on the complexity of *CherryPick* because the computation time is only related with the number of samples rather than the number of candidates (BO works even in continuous input space). However, in practice, it might impact the speed of computing the maximum point of the acquisition function in BO because we cannot simply enumerate all of the candidates then. More efficient methods, e.g. Monte Carlo simulations as used in [6], are needed to find the maximum point of the acquisition function in an input-agnostic way. Moreover, the computations of acquisition functions can be parallelized. Hence, customized VM only has small impacts on the feasibility and scalability of *CherryPick*.

Choice of prior model: By choosing Gaussian Process as a prior, we assume that the final function is a sample from Gaussian Process. Since Gaussian Process is non-parametric, it is flexible enough to approach the actual function given enough data samples. The closer the ac-

tual function is to a Gaussian Process, the fewer the data samples and searching we need. We admit that a better prior might be found given some domain knowledge of specific applications, but it also means losing the automatic adaptivity to a set of broader applications.

Although any *conjugate distribution* can be used as a prior in BO [32], we chose Gaussian Process because it is widely accepted as a good surrogate model for BO [33]. In addition, when the problem scale becomes large, Gaussian Process is the only choice which is computationally tractable as known so far.

7 Related Work

Current practices in selecting cloud configurations Today, developers have to select cloud configurations based on their own expertise and tuning. Cloud providers only make high-level suggestions such as recommending I2 instances in EC2 for IO intensive applications, e.g., Hadoop MapReduce. However, these suggestions are not always accurate for all workloads. For example, for our TPC-H and TeraSort applications on Hadoop MapReduce, I2 is not always the best instance family to choose.

Google provides recommendation services [3] based on the monitoring of average resource usage. It is useful for saving cost but is not clear how to adjust the resource allocation (e.g. scaling down VMs vs. reducing the cluster size) to guarantee the application performance.

Selecting cloud configurations for specific applications The closest work to us is Ernest [37], which we have already compared in Section 1. We also have discussed previous works and strawman solutions in Section 2 that mostly focus on predicting application performance [19, 21, 37]. Bodik *et al*. [12] proposed a framework that learns performance models of web applications with lightweight data collection from a production environment. It is not clear how to use such data collection technique for modeling big data analytics jobs, but it is an interesting direction we want to explore in the future.

Previous works [11, 40] leverage table based models to predict performance of applications on storage devices. The key idea is to build tables based on input parameters and use interpolation between tables for prediction. However, building such tables requires a large amount of data. While such data is available to data center operators, it is out of reach for normal users. *CherryPick* works with a restricted amount of data to get around this problem.

Tuning application configurations: There are several recent projects that have looked at tuning application configurations within fixed cloud environments. Some of them [19, 38, 45] propose to monitor resource usage in Hadoop framework and adjust Hadoop configurations to improve the application performance. Others search for the best configurations using random search [19] or local search [24, 42]. Compared to Hadoop configuration, cloud configurations have a smaller search space but a higher cost of trying out a configuration (both the expense and the time to start a new cluster). Thus we find Bayesian optimization a better fit for our problem. *CherryPick* is complementary to these works and can work with any application configurations.

Online scheduler of applications: Paragon [15] and Quasar [16] are online schedulers that leverage historical performance data from scheduled applications to quickly classify any new incoming application, assign the application proper resources in a datacenter, and reduce interferences among different applications. They also rely on online adjustments of resource allocations to correct mistakes in the modeling phase. The methodology cannot be directly used in *CherryPick*'s scenarios because usually, users do not have historical data, and online adjustment (e.g., changing VM types and cluster sizes) is slow and disruptive to big data analytics. Containers allow online adjustment of system resources, so it might be worth revisiting these approaches.

Parameter tuning with BO: Bayesian Optimization is also used in searching optimal Deep Neural Network configurations for specific Deep Learning workloads [9, 33] and tuning system parameters [14]. *CherryPick* is a parallel work which searches cloud configurations for big data analytics.

8 Conclusion

We present *CherryPick*, a service that selects near-optimal cloud configurations with high accuracy and low overhead. *CherryPick* adaptively and automatically builds performance models for specific applications and cloud configurations that are *just accurate enough* to distinguish the optimal or a near-optimal configuration from the rest. Our experiments on Amazon EC2 with 5 widely used benchmark workloads show that *CherryPick* selects optimal or near-optimal configurations with much lower search cost than existing solutions.

9 Acknowledgments

We like to thank our shepherd, John Wilkes, for his extensive comments. John's thoughtful interaction has substantially improved the presentation of this work. Further, thanks to Jiaqi Gao, Yuliang Li, Mingyang Zhang, Luis Pedrosa, Behnaz Arzani, Wyatt Lloyd, Victor Bahl, Srikanth Kandula, and the NSDI reviewers for their comments on earlier drafts of this paper. This research is partially supported by CNS-1618138, CNS-1453662, CNS-1423505, and Facebook.

References

[1] Apache Spark the fastest open source engine for sorting a petabyte. https://databricks.com/blog/2014/10/10/spark-petabyte-sort.html.

[2] Custom Machine Types - Google Cloud Platform. https://cloud.google.com/custom-machine-types/.

[3] Google VM rightsizing service. https://cloud.google.com/compute/docs/instances/viewing-sizing-recommendations-for-\instances.

[4] Performance tests for Spark. https://github.com/databricks/spark-perf.

[5] Performance tests for spark. https://github.com/databricks/spark-perf.

[6] Spearmint. https://github.com/HIPS/Spearmint.

[7] TPC Benchmark DS. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.3.0.pdf.

[8] TPC Benchmark H. http://www.tpc.org/tpc_documents_current_versions/pdf/tpch2.17.1.pdf.

[9] Whetlab. http://www.whetlab.com/.

[10] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data-parallel Computing. NSDI, 2012.

[11] E. Anderson. HPL-SSP-2001-4: Simple table-based modeling of storage devices, 2001.

[12] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. I. Jordan, and D. A. Patterson. Automatic exploration of datacenter performance regimes. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*. ACM, 2009.

[13] E. Brochu, V. M. Cora, and N. De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[14] V. Dalibard. A framework to build bespoke autotuners with structured Bayesian optimisation. Technical report, University of Cambridge, Computer Laboratory, 2017.

[15] C. Delimitrou and C. Kozyrakis. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. *ACM Trans. Comput. Syst.*, 2013.

[16] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and QoS-aware Cluster Management. ASPLOS. ACM, 2014.

[17] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, 2012.

[18] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, and J. Cunningham. Bayesian Optimization with Inequality Constraints. In *International Conference on Machine Learning*, 2014.

[19] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011.

[20] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Conference on Innovative Data Systems Research*, 2011.

[21] Y. Jiang, L. Ravindranath, S. Nath, and R. Govindan. WebPerf: Evaluating what-if scenarios for cloud-hosted web applications. In *SIGCOMM*, 2016.

[22] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 1998.

[23] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: comparing public cloud providers. In *SIGCOMM*, 2010.

[24] M. Li, L. Zeng, S. Meng, J. Tan, L. Zhang, A. R. Butt, and N. Fuller. MrOnline: MapReduce online performance tuning. In *International Symposium on High-performance Parallel and Distributed Computing*, 2014.

[25] I. Loshchilov and F. Hutter. CMA-ES for Hyperparameter Optimization of Deep Neural Networks. *arXiv preprint arXiv:1604.07269*, 2016.

[26] D. J. MacKay. Introduction to Gaussian processes. *NATO ASI Series F Computer and Systems Sciences*, 1998.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 2015.

[28] J. Mockus. *Bayesian approach to global optimization: theory and applications*. Springer Science & Business Media, 2012.

[29] O. O'Malley. Terabyte sort on apache hadoop. *Yahoo, available online at:* `http://sortbenchmark.org/Yahoo-Hadoop.pdf`, 2008.

[30] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A balanced large-scale sorting system. In *NSDI*, 2011.

[31] C. E. Rasmussen. Gaussian processes for machine learning. MIT Press, 2006.

[32] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. Taking the human out of the loop: a review of Bayesian optimization. *Proceedings of the IEEE*, 2016.

[33] J. Snoek, H. Larochelle, and R. P. Adams. Practical Bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, 2012.

[34] I. M. Sobol. On quasi-monte carlo integrations. *Mathematics and Computers in Simulation*, 1998.

[35] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press Cambridge, 1998.

[36] V. N. Vapnik. *Statistical Learning Theory*. John Wiley & Sons, 1998.

[37] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.

[38] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ACM, 2011.

[39] G. Wang and T. Ng. The impact of virtualization on network performance of Amazon EC2 data center. In *INFOCOM*, 2010.

[40] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with CART models. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*. IEEE, 2004.

[41] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.

[42] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang. A smart hill-climbing algorithm for application server configuration. In *International Conference on World Wide Web*, 2004.

[43] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *SIGMETRICS Perform. Eval. Rev.*, 2003.

[44] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, 2010.

[45] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives. In *International Conference on Autonomic Computing*, 2012.