COSE: Configuring Serverless Functions using Statistical Learning

Nabeel Akhtar
Boston University & Akamai
nabeel@bu.edu

Ali Raza
Boston University
araza@bu.edu

Vatche Ishakian
Bentley University
vishakian@bentley.edu

Ibrahim Matta
Boston University
matta@bu.edu

***Abstract*—Serverless computing has emerged as a new compelling paradigm for the deployment of applications and services. It represents an evolution of cloud computing with a simplified programming model, that aims to abstract away most operational concerns. Running serverless functions requires users to configure multiple parameters, such as memory, CPU, cloud provider, *etc*. While relatively simpler, configuring such parameters correctly while minimizing cost and meeting delay constraints is not trivial. In this paper, we present COSE, a framework that uses Bayesian Optimization to find the optimal configuration for serverless functions. COSE uses statistical learning techniques to intelligently collect samples and predict the cost and execution time of a serverless function across unseen configuration values. Our framework uses the predicted cost and execution time, to select the "best" configuration parameters for running a single or a chain of functions, while satisfying customer objectives. In addition, COSE has the ability to adapt to changes in the execution time of a serverless function. We evaluate COSE not only on a commercial cloud provider, where we successfully found optimal/near-optimal configurations in as few as five samples, but also over a wide range of simulated distributed cloud environments that confirm the efficacy of our approach.**

I. Introduction

Serverless computing has emerged as a new and compelling paradigm for the deployment of cloud applications and services. It promises new capabilities that make writing scalable microservices easier and cost effective. Most of the prominent cloud computing providers have released serverless computing platforms, and there are also several open-source efforts including OpenLambda [1] and OpenWhisk [2].

The serverless paradigm [3] at its core provides developers with a simplified programming model for creating cloud applications that abstracts away most, if not all, operational concerns. They no longer have to worry about provisioning and managing servers, and other infrastructure issues. Instead, they can focus on the business aspects of their applications. The paradigm also lowers the cost of deploying cloud code by charging for execution time – following a "pay as you go" pricing model [4], [5] – rather than for allocated resources.

In serverless application-development [6], a developer implements the business functionality as a stateless or composition of stateless functions using one or a combination of the programming languages supported by major cloud providers. Currently, Python and Nodejs are the most common scripting languages supported by major serverless platforms (*c.f.* Table I). The developer then submits the code to the cloud provider along with dependencies (*e.g.* libraries), and specifies configuration parameters such as memory size or CPU power. The cloud provider stores this code, and on invocation – which can be triggered through events or HTTP requests – executes this code either in containerized environments [2] or virtual machines over varying underlying physical infrastructures, with the specified configurations. Table I highlights serverless platforms from three major cloud providers [1]. The table shows programming languages supported, billing methodology, and memory size or CPU-power options that a user can select.

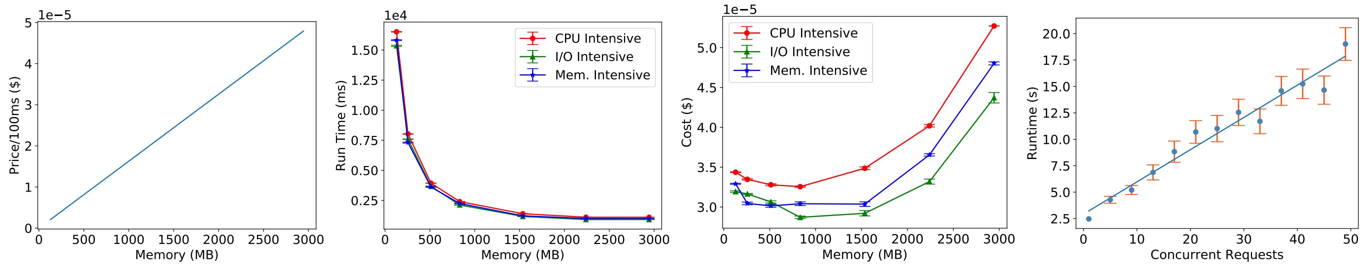
	AWS Lambda	**Google Function**	**IBM Cloud Function**
Memory (MB)	$64 \times i$ $i \in \{2,3, ... 47\}$	$128 \times i$ $i \in \{2,4,8,16,24\}$	{256 ... 2048}
Language	Python, Nodejs & others	Nodejs	Python, Nodejs & others
Billing	Execution time based on memory	Execution time based on memory & CPU-power	Execution time based on memory
Configurable	memory	memory & CPU-power	memory

Table I: Serverless platforms

Serverless computing has given a much-needed agility to developers, abstracted away the management and maintenance of physical resources, and provided them with a relatively small set of configuration parameters: memory and CPU. While relatively simpler, configuring the "best" values for these parameters while minimizing cost and meeting performance and delay constraints poses a new set of challenges. This is due to several factors that can significantly affect the running time of serverless functions.

To highlight the effects of parameter configuration on the performance and cost of serverless functions, we deployed serverless functions written in Python on AWS Lambda. In AWS Lambda, a customer is allowed to configure the amount of memory allocated to a serverless function. We ran this function for different memory sizes and studied the effect of varying memory sizes on the performance of the function, *i.e.* run-time. Figure 1b shows how the run-time of these serverless functions decreases with the increase of memory size allocated to the function. However, the marginal improvement in the run-time decreases as the memory increases. Figure 1c shows the cost of corresponding runs, which is the product of price

[1]Note that Microsoft Azure Functions does not provide users with the ability to configure functions and the cost is based on per-second resource consumption and execution time. In this work, we focus on configurable functions where users (or service providers) can configure serverless functions to meet service requirements.



(a) AWS price per 100ms (b) Run-time of serverless functions on Amazon Lambda (c) Cost for running functions on Amazon Lambda (d) Co-location effect on a serverless function on Apache OpenWhisk

Figure 1: Serverless function's performance with different memory sizes and co-location

(Figure 1a) and run-time (Figure 1b). As shown in Figure 1c, choosing too small a value or too large a value for memory can result in higher costs for running the function[2]. This behavior is because the pricing model as exposed by the cloud providers is tightly coupled with the amount of resources specified to execute the serverless function (*c.f.* Figure 1a), and the dependency between memory and CPU resource allocation – AWS Lambda allocates CPU power linearly in proportion to the amount of memory configured [8]. We observed similar behavior when setting configurable parameters on Google Function. Running similar experiments on Apache OpenWhisk showed that co-location has a significant impact on the run-time. Figure 1d shows the effect of co-locating serverless functions on OpenWhisk running on a single-CPU machine.

The examples above highlights some factors that can affect the performance of running serverless functions. However, they are not the only factors. A recent study [9] showed that the underlying infrastructure and resource provisioning can vary significantly depending on multiple factors including function placement, cold starts, I/O and network conditions, type of VMs/containers, and co-location with other functions. The user is oblivious to all these other factors, and has only limited control over a few parameters affecting performance, *i.e.* memory and processing power.

Given the issues raised above and the limited control a user has over the underlying system parameters, finding the "best" configuration to run a function while minimizing cost and meeting performance and delay constraints poses a new set of challenges. The problem is even more challenging when a user is running a chain of interdependent functions – where a user can still meet the performance requirement of the chain by trading off the performance of some of the functions in the chain for lower cost – and when a user is presented with the option to pick between multiple locations, *i.e.* edge and core [10] [11] [12].

In this paper we present COSE, a framework that uses Bayesian Optimization to statistically learn the relationship between cost/run-time and unseen configurations of a serverless function. Using this learned relationship, henceforth referred to as *performance model* of the serverless function, our framework is able to pick the best configuration for a serverless function which not only minimizes the cost but also meets user-specified performance criteria such as response time/delay of running a serverless function or chain of these functions. Our framework is lightweight and has the ability to dynamically adapt to changes in the execution time of a serverless function. It can be incorporated into an offering by cloud providers; it could be implemented as a value-added proposition by service providers; or it could be directly leveraged by customers. We evaluate our framework not only on a commercial cloud provider, where we successfully found optimal/near-optimal configurations in as few as five samples, but also over a wide range of simulated distributed cloud environments that confirm the efficacy of our approach.

[2]Our results are consistent with recent studies [7] on the cost of executing serverless functions.

II. System Description

Figure 2 provides an overview of our COSE framework. It consists of two main components: a *Performance Modeler* component, which is responsible for learning the application's *performance model*, *i.e.* the relation between cost/run-time and configurations for the serverless function, and the *Config Finder* component whose goal is to find the "best" configuration that minimizes cost while satisfying the delay bound on the running time of the serverless function. As indicated earlier, COSE can be incorporated into an offering by cloud providers; it could be implemented as a value-added proposition by service providers; or it could be directly leveraged by customers. For the rest of this work, we will assume that our COSE framework has been adopted by a Service Provider, and through standard APIs, a client registers her serverless function with the COSE service.

Figure 2 highlights the interactions between our COSE framework and its environment. Application clients, *e.g.* mobile and IoT devices, issue requests to the cloud provider to invoke a serverless function. Once the function is invoked, a trace log, containing the cost and execution time, is generated and stored. Our framework acts as a monitoring service and utilizes the information from the trace to learn the performance model of the function. After the learning phase converges, COSE uses *Config Finder* to find the "best" configuration that minimizes cost while satisfying the delay bound on the running time of the serverless function or a chain of functions. To account for delays associated with different locations/services supported by a designated cloud provider (e.g. Amazon Lambda's "edge" vs. "core"), a client reports the response times of its serverless function invocations to

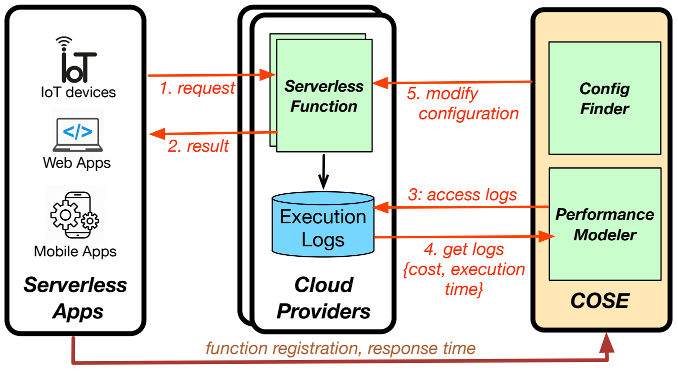


Figure 2: System overview

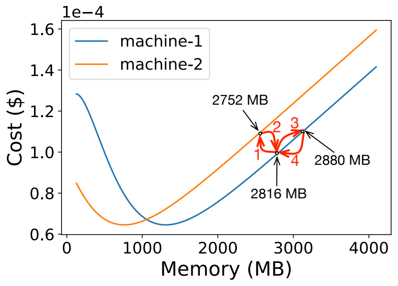


Figure 3: Example of AIAD getting stuck at a local minimum

COSE[3]. If a change to the previous configuration is needed, COSE connects to the designated cloud provider using APIs to modify the configuration. Next, we discuss the approaches and choices for the components of our COSE framework.

III. COSE: The *Performance Modeler* component

Our COSE framework has been entrusted to execute a single or a set of serverless functions on a designated cloud provider. It has the ability to configure the parameters of the serverless function, *e.g.* amount of memory, or the location of running the function by requesting it from the Cloud Provider (CP). The goal is to learn the application's performance model. There are several ways to achieve this:

1) *Exhaustive Search for the best cloud configuration:* This method runs the serverless function under all or a subset of possible configurations to find the configuration that minimizes the cost [13]. This methodology has very high overhead. Amazon Lambda alone has over 45 different memory types with a choice of location between "edge" and "core". To learn configurations across multiple CPs needs hundreds, if not thousands, of function executions. Moreover, the performance of the function can vary depending on the type of physical resources the function is deployed to execute on, and whether the function is co-located with other functions. This can lead to repeating the exhaustive search to find the best configuration.

2) *Algorithms for parameter descent:* As an alternative to doing exhaustive search, this method performs the search using parameter descent algorithms. The algorithms choose parameter values in the direction of decreasing cost. For example, if the memory value 512MB gives lower cost than 448MB,[4] the algorithm chooses a value greater than 512MB in anticipation of decreasing the cost further. Algorithms such as Additive Increase Additive Decrease (AIAD) and Gradient Descent, can be used. Such algorithms have tendency to get stuck in local minima, which leads to sub-optimal configuration for a serverless function.

Figure 3 illustrates a simple example where AIAD gets stuck at a local minimum. AIAD, as its name implies, uses a small fixed amount – 64MB in our example – to either increase or decrease requests for resources. Imagine a CP with two machines with different hardware configurations: machine-1 and machine-2. The performance of the serverless function will be different on machine-1 and machine-2 since these are shared resources and the performance depends on the utilization of each machine. Initially, AIAD requests a large memory configuration, *e.g.* 2816MB. Our serverless function is placed on machine-1. Using AIAD, it descends in the direction of decreasing cost. When the memory requested decreases from 2816 MB to 2752 MB (shown by arrow 1), the CP decides – potentially due to cost savings from colocating it with other functions – to place the serverless function on machine-2. Since the cost of execution is high at memory 2752 MB, AIAD changes direction and asks for a higher memory value of 2816 MB, which makes the CP place the function back on machine-1 (shown by arrow 2). In the next step, AIAD will further go in this same cost-reducing direction and ask for a higher memory value of 2880 MB (shown by arrow 3). However, the cost becomes higher at 2880 MB when compared with 2816 MB, so AIAD will change direction and in the next iteration, ask for 2816 MB (shown by arrow 4). The process will keep repeating and AIAD will be stuck at a local minimum. We implemented AIAD and Gradient Descent and tested them on commercial cloud providers – these results are not shown in this paper due to lack of space.

Another drawback of parameter descent algorithms is that they do not continually learn the underlying relation between cost/execution time and configuration, and so if the underlying conditions or requirements change, the whole process needs to be repeated.

3) *Statistical Learning for finding the best configuration:* This approach uses a statistical learning model to predict the performance of a serverless function under different configurations. It involves sampling different configuration values to successfully model the performance of the function and predict the configuration that will minimize cost. In this paper, we use **Bayesian Optimization** as the statistical learning approach to find the "best" configuration for a serverless function.

A. Our Approach: Leveraging Bayesian Optimization

The objective of Bayesian Optimization (BO) is to optimize over a black-box function $g(x)$. In our case, the function $g(x)$ that we want to learn is the relationship between performance/cost and all possible configurations x, not to be

[3]While this requires changes to the client, in practice, techniques to estimate the response time across geographically distributed clients can be incorporated without requiring any changes to the client.

[4]Recall that AWS memory options available from AWS Lambda increases or decreases in 64MB increments.

confused with the serverless function/code itself that we want to execute. Knowing this relationship $g(x)$, one can readily locate the configuration that minimizes cost, or that meets a certain performance/delay requirement.

BO constructs a probabilistic model for $g(x)$ in a predefined parameter space and exploits this model to make decisions about where to next sample/evaluate the function. It uses the information from all previous observations of $g(x)$ to find the next sample. The goal is to learn $g(x)$ in a few number of samples. Compared to deterministic searching/learning, BO *dynamically adapts* its search based on its current characterization and confidence interval of the prediction model. BO dynamically picks the next sample that gives more information and avoids unnecessary samples. BO stops searching when it has high confidence in the predicted model and the expected improvement for the predicted model is small for new samples.

BO at work: BO observes the objective function $g(x)$ at different sampled values. It models $g(x)$ as a stochastic process and computes a confidence interval for $g(x)$ based on the samples collected. Figure 4 shows a simple example where configuration x consists of a single dimension, *i.e.,* memory for a serverless function. The *actual* underlying function is given by the solid blue line. The confidence interval is an area around the predicted function where the actual function is passing through with 95% probability. In Figure 4a, there are only two samples collected and the confidence interval is higher in the region further away from the observed values. The black dashed line is the predicted objective function of $g(x)$. As BO collects more samples (in Figures 4b and 4c), the confidence interval gets smaller and the prediction is closer to the actual values. BO intelligently samples the next point to evaluate/observe $g(x)$, based on the so-called *acquisition function* – Expected Improvement (EI) in this case. EI is calculated for each possible configuration in the search space. A configuration with the highest EI value is selected as the next sample. As shown in the lower part of Figure 4a, the highest EI value is at 128 MB and this value is used as a configuration value for the next run of the serverless function as shown in Figure 4b.

A key part of BO is choosing an *acquisition function* to determine the next configuration point to evaluate. There are a number of possible acquisition functions and we chose EI as it is the most widely used - it has been shown to outperform others and it does not require parameter tuning [14]. Intuitively, EI samples at a point x where we are most likely to see an improvement in cost when compared to the best configuration value we have seen so far. Due to lack of space, we skip the mathematical details [15], but the EI expression has two terms: an exploitation term, and an exploration term. A parameter ω is used to define the amount of exploration, where higher values lead to more exploration (*i.e.* evaluating $g(x)$ at new configuration values x), and lower values lead to exploitation (*i.e.* re-evaluating $g(x)$ at already explored configuration values).

B. Adapting BO for Serverless Functions

To make BO run efficiently and accurately for serverless functions, we made changes to the classical BO. These changes are highlighted next.

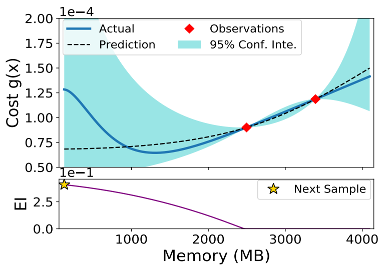
(i) *Initial Points:* The choice of the initial points can guide the search for the optimal solution in Bayesian Optimization. A random choice of initial points can lead to longer convergence time. Since a serverless function tends to have a convex relation between the cost of execution and the chosen configuration (cf. Figure 1c), we chose the initial points as uniformly distributed in the search space. For this work, we chose four initial points.

(ii) *Reduce the Search Space:* We reduce the search space by discretizing the possible configuration parameters. BO has a computational complexity of $O(C^4)$, where C is the number of data samples. We used the *memory values* and the *cloud providers* as the set of parameters to choose from. We follow the choices of memory values available to customers on Amazon Lambda as shown in Table I. Possible memory values that can be selected on Amazon Lambda are between 128MB and 3008MB, with offset of 64 MB, *i.e.,* $m \in \{128MB, 192MB, 256MB, ..., 3008MB\}$. We calculated EI only for these discrete values of possible input memory size, which decreased the running time significantly. For this work, we have only two cloud providers or two locations/services supported by a cloud provider, *Edge-Cloud* and *Core-Cloud*, which are readily discrete.

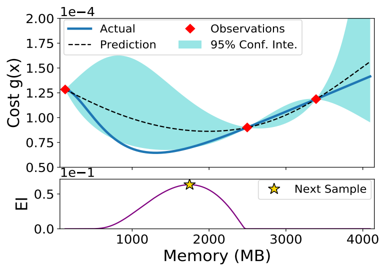
(iii) *Handling Noise:* A cloud environment is shared and there are uncertainties introduced because of the sharing of resources. Co-location of functions, cold-start, hardware failures, resource-overuse, *etc*., can impact the execution time of a serverless function running under the same configuration. We assume additive white Gaussian noise with hyperparameter α. Finding the best value for α is outside the scope of this work. Our experimental results show that $\alpha = 0.01$ captures most uncertainties in the serverless cloud platforms and we chose this value for our system.

(iv) *Accurately Predicting Changing Performance Model:* BO assumes that the target performance model is not changing while the samples are being collected. However, in practice, the target cost-configuration relation can change because of multiple reasons, *e.g.* migration of the serverless function to a different machine by the cloud provider, change in execution time based on the change in the input data to the function, *etc.* To predict the dynamics in the target cost-configuration relation, we keep a history of the configuration points sampled, and we discard the "old" sampled points as we collect new samples using a sliding window approach. This helps BO sample points again in the search space where it had sampled in the past, thus capturing the changes in the target cost-configuration relation.

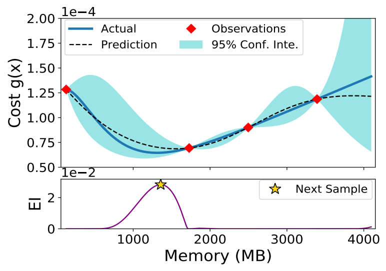
(v) *Convergence Criteria:* As mentioned earlier, we use Expected Improvement (EI) as the convergence criterion for BO. When the EI for the next collected sample is below 5%, we consider that BO has converged and we henceforth use our



(a) Prediction after two samples. Next sample with highest EI at 128 MB



(b) Prediction after three samples. Next sample with highest EI at 1728 MB



(c) Prediction after four samples. Next sample with highest EI at 1344 MB

Figure 4: Bayesian Optimization example

BO predicted cost/run-time vs. configuration model to find the least-cost configuration that satisfies the delay constraint on the serverless function.

Our COSE-based service provider runs BO to predict the performance model of the serverless function, *i.e.* the black-box cost/run-time vs. configuration relation. It will keep on sampling until the BO has converged on the performance model. Once converged, COSE can use the predicted model to find the "best" configuration that satisfies the delay constraint for a serverless function. In Section IV, we show how delay constraints for serverless functions are met.

IV. COSE: The *Config Finder* Component

Given the performance model of a serverless function, predicted by the *Performance Modeler*, it is easy for the second component of our COSE framework, *Config Finder*, to locate the configuration that minimizes the cost. However, real-world applications typically have delay constraints on the running time of a serverless function or chain of functions. In this section, we discuss how *Config Finder* picks the least-cost configurations that satisfy delay constraints.

The cost of running a serverless function f is given by:

$$g^f(x) = t^f(x) \times p^f(x)$$

where $t^f(x)$ is the execution time of the function using configuration x, and $p^f(x)$ is the price (cost per unit time) for running the function using configuration x. $p^f(x)$ is provided by the cloud provider and the predicted $g^f(x)$ is provided by the BO, so we can use these values to calculate the predicted execution time $t^f(x)$ for a serverless function under configuration x. The total time to run a function (end-to-end delay) is given by:

$$T^f(x) = t^f(x) + d(x)$$

where $d(x)$ is the delay other than the execution time of a function (such as network, queuing delay, *etc.*) Note that $d(x)$ is specific to a cloud provider or location and we estimate it by taking the difference $T^f(x) - t^f(x)$ of multiple samples collected by BO for the cloud providers or locations. This will help us predict the total time to run a serverless function, *i.e.* response time, on any cloud provider/location for a given configuration value.

With all the information above, it is easy for a service provider to find a configuration that satisfies the (end-to-end) delay constraint for a single serverless function. However, the problem gets complicated when we have a chain of functions (service chain) that need to execute one after another. All current serverless cloud providers support the chaining of functions. The service provider needs to select a configuration for each serverless function such that the cost to execute the service chain is minimized, while satisfying the delay constraint on the running time of the whole chain. The *Config Finder* module in COSE solves this problem. Using Integer Linear Programming (ILP), we formulate the problem as an optimization problem. *Config Finder* solves the ILP to find the best configuration for a chain of functions. Note that to solve for a single function, we consider the degenerate case of a chain of size one.

We assume that for each serverless function f in chain F, we choose the cloud provider $v \in V$ and the memory $m \in M$ such that the total cost for placing the chain is minimized and the delay constraint D_F on service chain F is satisfied.

Define $Y_x^f \in \{0,1\} = 1$ if function $f \in F$ is deployed using configuration $x \in C$, 0 otherwise.

The objective of the *Config Finder* is to minimize the total price paid for the chain of functions. This is given by:

$$\mathit{minimize}\left(\sum_{f\in F}\sum_{x\in C} g^f(x)Y_x^f\right) \tag{1}$$

subject to:
1) The delay requirement for the service chain is satisfied:

$$\sum_{f\in F}\sum_{x\in C} T^f(x)Y_x^f \leq D_F \tag{2}$$

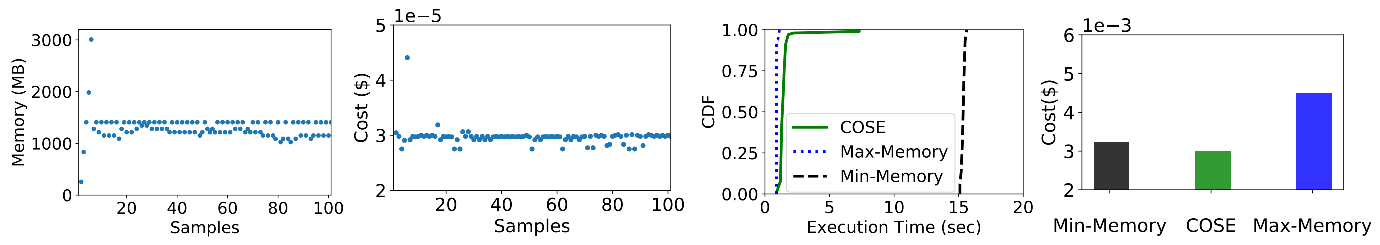
where $T^f(x)$ is the predicted end-to-end delay for running serverless function $f \in F$ using configuration $x \in C$.
2) A single configuration $x \in C$ is selected for each serverless function in the chain.

$$\sum_{x\in C} Y_x^f = 1 \quad \forall f \in F \tag{3}$$

The solution to this problem yields a least-cost feasible solution, *i.e.* the resulting Y_x^f, that gives the configuration x of each serverless function f in the chain. Note that one can argue that ILP takes long to solve. However, since a chain typically consists of a few functions, the total time to execute this ILP on a CPLEX solver [16] is only a few milliseconds.

V. Experimental Results: Running COSE on Amazon Lambda

We test the proposed COSE system on AWS Lambda, a very popular serverless cloud provider. We start by describing the class of functions that we tested on AWS Lambda.



(a) Parameters (Memory) (b) Cost (c) Runtime of Serverless Function (d) Cost for running 100 iterations

Figure 5: COSE performance on Amazon Lambda for *I/O Intensive* serverless function

1) Representative Functions: we test our COSE framework across four different representative functions for serverless computing. These functions represent the different types of computation (combination of *I/O-*, *CPU-*, *network-* and *memory*-intensive tasks) that a serverless application typically performs: (i) *CPU-intensive:* This is a function that calculates the trigonometric function *atan* of multiple numbers, hence making it more CPU-heavy function; (ii) *Memory-intensive:* This function applies a filter on a large image. This requires extensive use of memory; (iii) *I/O-intensive:* This function performs multiple I/O related operations on a file, *i.e.,* opening, reading and closing a file; (iv) *Network-intensive:* This function downloads a large file from a server.

These functions were implemented in Python3.6/3.7 and deployed on AWS Lambda. Each function was deployed as a separate AWS Lambda function. Figure 1b shows the run-time for the *CPU-intensive, memory-intensive* and *I/O-intensive* functions under different memory configurations. We do not show the results for the *network-intensive* function since change in memory had little/no impact on the running time of the function. This is because the network resources allocated to a function do not change as we change the memory requested. Figure 1c shows the price-memory relation for CPU-, memory- and I/O-intensive functions.

Even though CPU- and I/O-intensive functions do not use more than a certain amount of memory, their performance is affected by the memory requested for the function. The reason for that is, AWS Lambda assigns CPU share to each function in proportion to the memory configured for the function. Hence more memory will assign more CPU cycles to a function.

2) Evaluation Results: We ran the *CPU-intensive*, *I/O-intensive* and *memory-intensive* functions shown in Figure 1 on Amazon Lambda using COSE. Since the behavior of these functions is very similar to each other, we show results for only the *I/O-intensive* function here. To get an estimate of the optimal memory value, *i.e.* the memory value that minimizes the price, we ran the serverless functions multiple times across different memory values. As seen in Figure 1c, the *I/O-intensive* function has the lowest price in the memory range 900MB-1400MB.

We use COSE to find the optimal configuration for this function (with the goal of minimizing the cost, with no delay requirement on the execution time of the function). For the first few requests, as shown in Figure 5a, COSE explores different memory values and once it learns the underlying cost-memory relation, it starts suggesting optimal/near-optimal memory values in the range 900-1400MB. The corresponding cost for individual requests (function invocations) is given in Figure 5b. Again, COSE finds the optimal/near-optimal cost for the function ($\$2.9 \times 10^{-5}$ as seen in Figure 1c for the *I/O-intensive* function).

To compare COSE with static configurations, we invoked the *I/O-intensive* function 100 times with the maximum and minimum memory values, possible on AWS Lambda, to get the best/worst running times for the function, and also the corresponding cost. Figure 5c shows the running time of the serverless function when invoked with configurations picked by COSE, *maximum-memory*=3008MB, and *minimum-memory*=128MB. The *minimum-memory* configuration takes, on average, around 15 seconds to complete a request, while the *maximum-memory* configuration takes, on average, 1 second to complete the request. COSE performance is very close to *maximum-memory*. However, the cost incurred when using COSE is even less than the cost for *minimum-memory*, as shown in Figure 5d, due to lower execution time under COSE.

VI. Evaluation in a Distributed Cloud Environment

While the results of running our framework on AWS Lambda highlight the utility of our COSE framework, evaluating additional aspects of our framework presents a new set of challenges because we have little or no knowledge about the underlying infrastructure, the decisions made by the cloud provider regarding the allocation of resources, which functions are co-located, if the function had cold-start or warm-start, and queuing, propagation and other delays in the system. While it was possible in the previous section to compare the performance of an *I/O-intensive* serverless function in a simple scenario by exhaustively searching the memory space, and finding where the optimal memory value for this function lies, this approach may not be practical for scenarios where we have multiple functions (and possibly chains of functions). To establish the efficacy of our COSE framework, we model a distributed cloud provider and evaluate the framework across a set of multiple functions using extensive simulations. Since in the simulated cloud environment we know the target function that COSE is trying to optimize, we can compare the performance of COSE against the "ground truth".

A. Modeling Cloud Provider

We model our cloud provider by adopting the following aspects of commercial cloud providers.

(i) *Co-location*: We modeled the effect of co-location of functions by deploying the open-source serverless platform, Apache OpenWhisk [2], on Chameleon Cloud [17]. We deployed multiple functions on the same machine. The effect of co-location is given in Figure 1d. We modeled this in the cloud provider.

(ii) *Life-time and Cold-start*: If a function is not executed for a certain period of time (i.e. function *life-time*), the function is evicted by the cloud provider, and the subsequent request for running the function will experience extra delay (i.e. function *cold-start*). We use Amazon Lambda function's *life-time* of 26 minutes and *cold-start* delay of 0.25 seconds, as shown by previous studies [9], [18].

(iii) *Edge-cloud and core-cloud*: To compare across different cloud providers or different locations provided by one cloud provider, we model two types of clouds, *edge-cloud* and *core-cloud*. We assume that *edge-cloud* is closer to the user and thus has a smaller round trip delay. However, *edge-cloud* is more expensive when compared with the *core-cloud*.

(iv) *Dynamic serverless function*: To test the adaptive performance of COSE, we run COSE for a dynamic function whose execution time changes over time.

(v) *Modeling price and execution time of a serverless function*: We develop an analytical model for the cost and execution time based on the experimental results of running these functions on Amazon Lambda (details next).

B. Modeling cost and execution time

Our analytical model for cost and execution time of serverless functions is based on AWS Lambda's pricing and execution model.

Cloud provider's pricing model: We use Amazon Lambda's pricing model. Amazon uses a linear pricing model, as shown in Figure 1a. We use this pricing model for the *core-cloud*. Since the price for *edge-cloud* is higher than the *core-cloud*, we set the edge resource price to be 1.5 times the price of resources at the *core-cloud*. This linear pricing model is captured by the following equation for serverless function f:

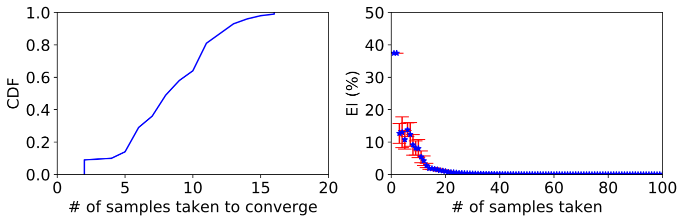
$$p^f(v,m) = K(v) \times m \tag{4}$$

where m is the memory used by function f, $v \in V$ is the cloud provider, and $K(v)$ is a constant and its value depends on the cloud provider's pricing.

Execution time model: The execution time for representative functions is shown in Figure 1b for Amazon Lambda. The execution time for these functions follows an exponential decay. Thus, we model the execution time for function f as:

$$t^f(v,m) = t^f(v,m_{min}) + t^f(v,m_{max}) \times e^{-\lambda(m-m_{min})} + h(v) \tag{5}$$

where $t^f(v,m_{min})$ is the running time for function f at the lowest possible memory ($m_{min} = 128MB$ for Amazon Lambda), $t^f(v,m_{max})$ is the running time at the highest possible memory ($m_{max} = 3008MB$ for Amazon Lambda), v is the cloud provider, and λ is the decay constant. By changing $t^f(v,m_{min})$, $t^f(v,m_{max})$ and λ, we can fit the execution model for any serverless function. The constant $h(v)$ captures the delay due to cold-start and co-location, and its value depends on the current state of the cloud provider.



(a) Convergence (b) Expected Improvement

Figure 6: BO's Convergence and EI

Cost for running a serverless function: The total cost $g^f(v,m)$ for running function f on cloud provider $v \in V$ is given by the product of the price per second ($p^f(v,m)$) and the total execution time ($t^f(v,m)$).

$$g^f(v,m) = p^f(v,m) \times t^f(v,m)$$

C. Simulation Results

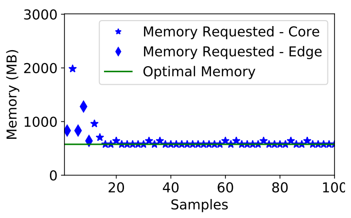
Using the above models allows us to simulate a cloud provider that has two parameters to be optimized, *i.e.* selection of location (edge vs. core) and memory value for deploying a serverless function. Since we have more control over the execution model and resource management, we evaluated the convergence and accuracy-related aspects of COSE. In addition, we evaluated some unique scenarios, for example, dynamically changing the underlying execution model to evaluate how well COSE adapts to changes. Our evaluation showed that COSE can learn the optimal or near-optimal configurations for a serverless function with as few as 13-15 samples and can adapt to changes well[5]. Also, COSE showed significant cost savings without compromising on performance[6].

Convergence: COSE uses Bayesian Optimization (BO) to predict the price function. A small convergence time for BO means that COSE can quickly find the "best" configuration that minimizes the price paid while satisfying the delay constraint. In this experiment, we looked at how long it takes for BO to converge and find the underlying cost-configuration relation. Since we are using a *cloud provider model*, we know the underlying performance function. As explained in Section III-A, we use expected improvement (EI) as the convergence criterion. Figure 6a shows the CDF (taken over 100 runs) of the number of configuration samples taken for BO to converge. We observed that BO can converge, 95% of the time, with as few as 15 samples. In Figure 6b we show how EI decreases as the number of configuration samples increases. The first few samples have the highest EI value. However, as COSE takes more samples, the EI value rapidly decreases. With each new sample, BO improves its understanding of the underlying performance function and subsequent configuration samples contribute little to improving the prediction.

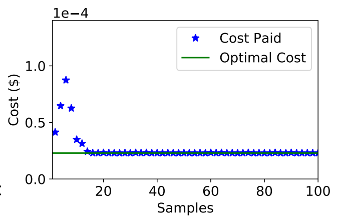
[5]We note that for a commercial cloud provider with one parameter, COSE was able to find a near-optimal configuration in 5 samples.

[6]Simulation parameters are available at [19].

Configuration Selection: After BO converges, COSE starts picking the "best" possible configuration for serverless functions using *Config Finder*. We used a function whose execution model had an optimal configuration of {*memory = 576MB, location = core-cloud*}. In Figures 7a and 7b, we show the configurations that COSE picked for each request and their corresponding cost. For the first few requests (up to 15 requests), COSE is exploring different configurations. After the BO in COSE converges, *Config Finder* starts picking optimal/near-optimal configurations for the function, *i.e.* the corresponding price paid of each serverless request is lowest.



(a) config={*memory, cloud provider*} selected for each request



(b) Cost paid for each request

Figure 7: Configuration with COSE

Dynamicity: The performance of a serverless function can be affected by factors like co-location, hardware, resource provisioning policy, *etc*. In case any of these factors changes, the configurations that were optimal before the change may no longer be optimal. We designed COSE so it is resilient in the face of such changes and is able to find new optimal configurations. We tested COSE's ability to adapt to a change in the underlying execution model. We created 500 requests for a serverless function. The first 250 requests follow a certain execution model and have certain optimal configurations. For the next 250 requests, we change the execution model hence the optimal configurations. We observed that depending on the history, COSE can successfully unlearn the previous execution model, learn the new model, and start predicting the new optimal configurations.

In Figure 8, we show the performance of COSE in terms of sample error in memory for three scenarios. The sample error E^s is defined as the difference between the sampled memory and the optimal memory: $E^s = |m^s - m^o|$, where m^s is the memory being sampled by BO and m^o is the optimal memory. In the *static* case, the underlying function's execution model does not change, and COSE remembers the full history, hence it takes a few samples for COSE to converge and the error is small. In the *static+window* case, the underlying function's

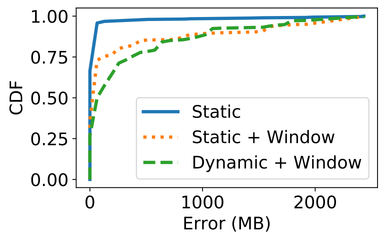


Figure 8: COSE for serverless function with changing execution model

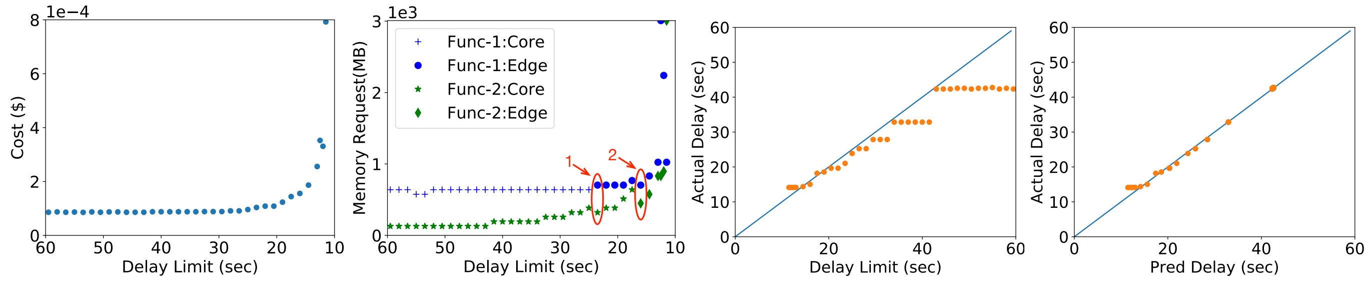
execution model does not change, and COSE remembers a limited history (last 40 samples). As BO loses historical data, the acquisition function periodically samples configuration points for exploration, and hence, there is high variability in memory selection leading to the higher error value. In the third (dynamic+window) case, COSE not only remembers a limited history but the serverless function's execution model also changes. We see the highest error in this scenario since COSE is constantly collecting samples to adapt to the changing execution model, and it takes time proportional to the history, to unlearn the previous execution model and learn the new one. That is a trade-off that COSE can make: more history would yield less sampling, but it would be slow to adapt to changes. A shorter history would result in more sampling, but COSE would adapt to changes more quickly. In all three scenarios, COSE converges to optimal/near-optimal values.

Delay Bounded Chaining: Serverless applications can have a chain of functions that triggers one another, where the output of one function serves as input to the next. This is called *service chaining*. Service chains usually have a delay constraint on the total execution time to meet quality-of-service requirements.

As each individual action is a serverless function, picking the right configurations is critical to the performance of the application and the cost of cloud usage. Here we show how COSE successfully finds the optimal configurations for serverless functions comprising a delay-bounded chain.

In this experiment, each request is a service chain consisting of two or more functions. Each function has a different execution model, hence different optimal configuration. As explained in Section IV, COSE uses BO's prediction of price and execution time to formulate the problem as an Integer Linear Program (ILP). COSE solves this ILP to find a suitable configuration for each function in the chain. Initially, we let the BO collect configuration samples and wait for it to converge. Once BO has converged, we observe the "best" configuration selected by COSE for each function in the chain such that the total (end-to-end) delay of the chain satisfies the delay bound. Although we tested COSE for different chain sizes, for simplicity, we show results for service chains of size two.

In Figure 9a, we look at how the delay bound affects the cost of cloud-usage. For loose delay requirement, COSE finds the "best" location and memory for both functions, hence lower cost. As the delay requirement becomes more stringent, COSE has to make a decision of either increasing the memory available to a function or placing it on the *edge-cloud* to reduce the delays. Both of these choices will raise the cost and that is why we see an increase in the cost as the delay bound becomes tighter. In Figure 9b, we show the corresponding configurations selected for varying delay bound. Initially, because of higher delay bound, COSE runs both functions on the *core-cloud* to lower the usage cost and selects the memory that lowers the cost. However, as the delay bound becomes smaller, COSE has to increase the memory available to either function or change the location



(a) Cost as a function of delay bound on the chain (b) Configurations selected by COSE to satisfy delay bounds (c) Delay bound on the chain vs actual delay (d) Predicted delay vs actual delay

Figure 9: Delay bounded chaining of serverless functions

where they are deployed. As the delay bound reduces to 42 seconds, COSE starts increasing the memory available to the second function. At around 24 seconds of delay bound, COSE cannot keep both functions on the *core-cloud* to meet the delay requirement. As shown by arrow 1, COSE moves the first function to the *edge-cloud* and decreases the memory needed for the second function on the *core-cloud*. As the delay bound becomes even smaller, COSE moves both functions to the *edge-cloud* as shown by arrow 2 at around 15 seconds. Since the *edge-cloud* has lower delays, COSE selects smaller memory values, compared to previous values, to minimize the cost while fulfilling the delay requirement.

In Figure 9c, we look at the actual delay experienced by the chain. COSE meets the delay requirement of the chain under most delay bounds. When the delay bound is higher than 42 seconds, we do not see an increase in the actual delay experienced by the chain because at the optimal configurations, the chain's total delay is 42 seconds. As explained in Section IV, COSE uses its estimation of delays in selecting the configurations of the serverless functions in the chain. It is critical that the predicted delay is close to the actual delay. Figure 9d shows that the actual delay experienced by the chain is very close to the delay predicted by COSE.

VII. Related Work

As serverless computing is gaining popularity, there has been a significant body of research that measures different aspects of the serverless paradigms [20], [21]. Detailed studies on different commercial serverless platforms aim to characterize and understand the architecture and resource management by the cloud provider [9], [22], [23]. With a better insight into the cloud provider's serverless platform, users can tailor their applications to efficiently use the cloud provider. In COSE, our focus is on modeling the application behavior at different configurations, regardless of the underlying architecture and resource management scheme used by cloud providers.

Commercial cloud providers have developed systems that suggest suitable configuration parameters to the user for running her tasks. Google provides a machine type recommendation system [24] that helps to maximize the resource utilization of user VM instances. AWS provides auto-scaling service [25] to the users for EC2 instances. Cloud provider's cluster managing systems, such as Google's Borg [26], Mesos [27], Paragon [28] and Quasar [29], allow the user to specify the need for the application and the system finds the best configuration. Currently, cloud providers do not provide a resource configuration facility for serverless computing. Moreover, to port any of these techniques to serverless computing, the user (or service provider) needs the complete knowledge of the underlying cloud infrastructure. Since this information is not available to the user/SP, these techniques cannot be applied by the user/SP for serverless computing.

Systems have been developed for users to infer cost/performance across different cloud configurations. CherryPick [30] uses Bayesian Optimization to predict a suitable VM configuration for an application to run by a cloud provider. CloudCmp [31] recommends a suitable cloud provider for running a user application. Both CherryPick and CloudCmp are offline tools that are helpful to users *before* they deploy their applications. Ernest [32] builds the performance model of machine learning applications. WebPerf [33] estimates the latency model of a web application. ARIA [34] builds the job profile and performance model for MapReduce and Hadoop applications. Unlike prior work that focuses on a particular application or on configuring resources beforehand, *i.e.* before execution/deployment, the COSE framework can be used for any application running as a serverless function and can adapt configurations on the fly.

VIII. Conclusion and Future Work

We presented COSE, a statistical learning based configuration finder for serverless functions. COSE uses Bayesian Optimization to learn the cost and execution time model for serverless functions across unseen configuration values. It supports function chaining, and has the ability to *adapt* to changes in the execution time of serverless functions. Our results on commercial cloud and simulated distributed cloud environments show that COSE provides optimal/near-optimal configurations for serverless functions in a few samples.

Future work includes the deployment of COSE as a service over larger scale multi-cloud providers. This will enable studying a wide range of workloads, application requirements, and cloud resource provisioning and pricing policies. We intend to extend our COSE simulator to accommodate more complex scenarios, such as service graphs. Note that although we did not test COSE for functions with wildly varying input workload, we believe COSE can be used for such scenarios if the input workload can be classified (*e.g.*, based on size) and COSE is trained for each class separately.

References

[1] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. Denver, CO: USENIX Association, 2016.

[2] "Apache OpenWhisk," https://openwhisk.apache.org/, 2019.

[3] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "Serverless programming (function as a service)," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 2658–2659.

[4] "Amazon Lambda Pricing Model," https://aws.amazon.com/lambda/pricing/, 2019.

[5] "Google Function Pricing Model," https://cloud.google.com/functions/pricing, 2019.

[6] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service (faas) in industry and research," *arXiv preprint arXiv:1708.08028*, 2017.

[7] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *Cloud Engineering (IC2E), 2018 IEEE International Conference on*. IEEE, 2018, pp. 257–262.

[8] "AWS Lambda Function Configuration," https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html, 2019.

[9] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 133–146.

[10] "AWS Lambda at Edge," https://aws.amazon.com/lambda/edge/, 2019.

[11] "OpenWhisk at Edge," https://github.com/kpavel/openwhisk-light, 2019.

[12] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: Extending serverless computing to the edge of the network," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: ACM, 2017, pp. 28:1–28:1.

[13] "AWS Lambda Power Tuning," https://github.com/alexcasalboni/aws-lambda-power-tuning, 2019.

[14] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'12. USA: Curran Associates Inc., 2012, pp. 2951–2959.

[15] D. R. Jones, M. Schonlau, and W. J. Welch, "Efficient global optimization of expensive black-box functions," *J. of Global Optimization*, vol. 13, no. 4, pp. 455–492, Dec. 1998.

[16] "IBM ILOG CPLEX Optimizer," https://www.ibm.com/analytics/cplex-optimizer, 2019.

[17] "Chameleon Cloud," https://www.chameleoncloud.org, 2019.

[18] "Cold Starts in AWS Lambda," https://mikhail.io/serverless/coldstarts/aws/, 2019.

[19] "COSE Simulation Parameters," https://github.com/akhtarnabeel/COSE-Serverless-Configuration, 2020.

[20] "AWS Lambda CPU allocation ," https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac, 2018.

[21] "Chaos of AWS Lambda," https://blog.symphonia.io/the-occasional-chaos-of-aws-lambda-runtime-performance-880773620a7e , 2018.

[22] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 405–410.

[23] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, April 2018, pp. 159–169.

[24] "Google Cloud Recommendations," https://cloud.google.com/compute/docs/instances/apply-sizing-recommendations-for-instances, 2018.

[25] "Amazon Auto Scaling," https://aws.amazon.com/ec2/autoscaling/, 2018.

[26] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys '15. New York, NY, USA: ACM, 2015, pp. 18:1–18:17.

[27] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[28] C. Delimitrou and C. Kozyrakis, "Qos-aware scheduling in heterogeneous datacenters with paragon," *ACM Trans. Comput. Syst.*, vol. 31, no. 4, pp. 12:1–12:34, Dec. 2013.

[29] ——, "Quasar: Resource-efficient and QoS-aware Cluster Management," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 127–144.

[30] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 469–482.

[31] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 1–14.

[32] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 363–378.

[33] Y. Jiang, L. R. Sivalingam, S. Nath, and R. Govindan, "WebPerf: Evaluating What-If Scenarios for Cloud-hosted Web Applications," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 258–271.

[34] 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*, ser. ICAC '11. New York, NY, USA: ACM, 2011, pp. 235–244.