Configuration and Placement of Serverless Applications using Statistical Learning

Ali Raza, *Student Member*, *IEEE*, Nabeel Akhtar, Vatche Isahagian, *Senior Member, IEEE*, Ibrahim Matta, *Senior Member, IEEE*, Lei Huang, *Student Member*, *IEEE*

***Abstract*—In the last decade, serverless computing emerged as a new compelling paradigm for the deployment of cloud 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 applications 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 resource configuration and placement for functions in a serverless application. 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 on available locations to select the "best" configuration parameters and placement for running a serverless application while satisfying customer objectives. We evaluate COSE on AWS Lambda with real-world applications consisting of multiple functions (both linear chains and service graphs), where we successfully found optimal/near-optimal configurations. We also evaluate COSE over a wide range of simulated distributed cloud environments that confirm the efficacy of our approach.**

***Index Terms*—Serverless, FaaS, AWS, IaaS**

I. Introduction

In the last decade, serverless computing[1] 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 more 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], Fn [2], Kubeless [3], OpenWhisk [4], *etc*. Commercial providers also allow a developer to deploy serverless applications in multiple geographical regions as well as closer to the end user (so called *edge*).

The serverless paradigm [5] at its core provides developers with a simplified programming model for creating cloud applications that abstract away most, *if not all*, operational concerns. Developers 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, and the backend is left to the cloud provider to manage. The paradigm also lowers the cost of deploying and running cloud code by charging for execution time – following a "pay as you go" pricing model [6], [7] – where a user is precisely charged for the time the application is running instead of allocation time. Any application deployed over a FaaS platform can be considered to be a serverless application. In this model, a developer writes the code of a cloud application in the form of stateless functions. The code, along with configuration parameters such as resources (memory/CPU), location (regions/edge) and triggers, is submitted to the platform, which in turn executes the code, in response to a triggering event, in a sandbox environment over an arbitrary infrastructure[2].

In serverless computing, the abstraction of backend infrastructure management and ease of deploying cloud applications come at the cost of little to no-control over the underlying resources and execution environment of the application. A developer has to rely on the configuration parameters of the individual function to obtain the desired performance, hence making it really critical to not only obtain desired performance but also optimize the cost of cloud usage.

Motivation: To highlight the effects of resource parameter configurations on the performance and cost of serverless functions, we deployed serverless functions written in Python on AWS Lambda, a popular FaaS platform. In AWS Lambda, the only resource configuration is the amount of memory allocated to the sandbox environment executing the function's code. We invoked each function with different memory configuration to observe the effect on cost and performance, *i.e.* run-time. Figure 1a shows how the run-time of these functions decreases with the increase of memory size allocated to the function. However, the marginal improvement in the run-time decreases as memory increases. Figure 1c shows the cost of corresponding runs, which is the product of price per unit time of computation with respect to allocated resources (Figure 1b) and run-time (Figure 1a). 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[3]. 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 1b), and the dependency between memory and CPU resource allocation – AWS Lambda allocates CPU power linearly in proportion to the amount of

Ali Raza, Ibrahim Matta and Lei Huang are with Boston University, USA. (e-mails: araza@bu.edu, matta@bu.edu, lei@bu.edu)
Nabeel Akhtar is with Akamai Technologies Inc. (email: nakhtar@akamai.com)
Vatche Isahagian is with IBM Research. (email: vatchei@ibm.com)

[1]Also referred to as Function-as-a-Service (FaaS) when offered as a service by a cloud provider.

[2]Some of the key features of major serverless platforms are shown in Table I.

[3]Our results are consistent with recent studies [8] on the cost of executing serverless functions.

	AWS Lambda	Google Cloud Function	IBM Cloud Function	Microsoft Azure Function
Memory (MB)	{128 ... 10240}	$128 \times i$ $i \in \{1,2,4,8,16,32\}$	{128 ... 2048}	Function can use up to 1536
Runtimes Supported	Node.js 14/12/10, Go 1.x, Ruby 2.7/2.5, Python 3.8/3.7/3.6/2.7, Java 11/8, .NET Core 3.1/2.1, and Custom Runtimes	Go 1.13, Python 3.9/3.8/3.7, Ruby 2.7/2.6, Java 11, Node.js 14/12/10, .NET Core 3.1, and PHP 7.4	Node.js 12, Python 3.7/3.6, Java 8, Swift 4.2, PHP 3.7, Ruby 2.5, Go 1.15, .NET Core 2.2, and Docker	.NET Core 3.1/2.1, .NET Framework 4.8, Node.js 14/12/10/8/6, Java 11/8, PowerShell 7/6, and Python 3.9/3.8/3.7/3.6
Billing	Execution time based on memory allocated	Execution time based on memory & CPU allocated	Execution time based on memory allocated	Execution time based on memory used
Configurable Resource	memory & location	memory & CPU-power	memory	n/a

Table I: Popular serverless platforms

memory configured [9]. We observed similar behavior when setting configurable parameters on Google Function.

The examples above highlight the effect of resources (configuration parameters) on the performance and cost of a serverless function. Configuring these resources is a challenging task because the execution behavior of a function can change over time depending on the cloud provider's policies and datacenter's conditions. Wang et al. [10] performed a detailed analysis of other factors affecting the performance of serverless functions and showed that underlying infrastructure and resource provisioning policies of a cloud provider can affect performance. For example, having diverse underlying infrastructure would cause instances of the same function to execute on different resources (hardware and VM types), hence resulting in variance in performance. Similarly, other factors such as instance packing strategy, cold starts, I/O and network conditions, and co-location with other functions would also cause inconsistent performance even when the function is allocated the same amount of resources. To elaborate on the effect of co-location on performance, we ran experiments on Apache OpenWhisk to show 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. A 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.

Similarly, the placement of functions can also affect a user's perceived latency of an application. Our experiments on AWS Lambda showed that different locations and regions have varying user-perceived latency depending on the user's location. Moreover, the cost can also vary based on the location – for example, AWS Lambda charges ~3x more for computation run at the edge locations [11], [6].

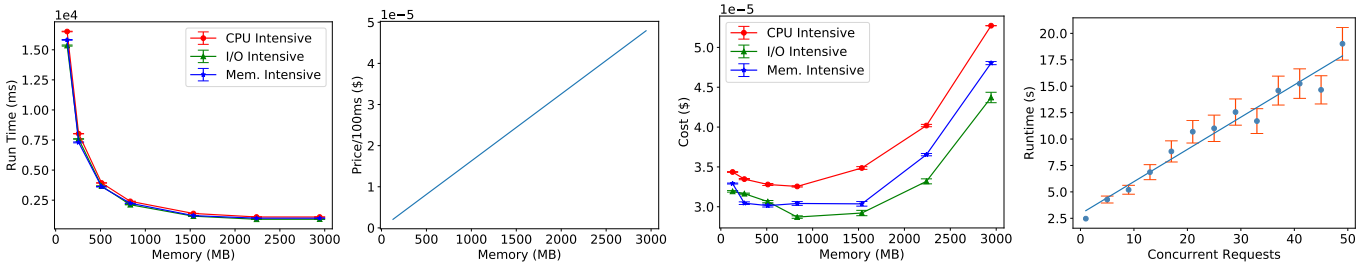
Ideal Configurator: Given the issues raised above and limited user control, finding the "best" configuration to run a function while minimizing cost and meeting performance and service-level objectives (SLOs) can be a daunting task. The problem is even more challenging when a user is running an application composed of interdependent functions (referred to as *service graphs*) – where a user can still meet the end-to-end performance requirement of the application by trading off the performance of some of the functions for lower cost – and when a user is presented with the option to pick between multiple locations, *i.e.* different regions, edge, and core [12] [13] [14]. Considering the above challenges, we believe an ideal resource configuration and placement strategy should address the following three aspects:

- ***Sampling Cost:*** Sampling the performance of an application for all possible configurations can be expensive as AWS Lambda alone offers tens of thousand of configuration options (memory values, edge/core locations, regions).
- ***Dynamic Adaptation:*** Serverless applications can be running in varying conditions based on the data-center resources and cloud provider's policies. An ideal strategy should adapt according to the current condition as one-time configurations may not always be optimal.
- ***Service Graphs & Placement:*** In the case of an application composed of multiple serverless functions, individual resource configurations may not lead to the globally optimal solution. Moreover, placement can affect the user's perceived latency and SLO.

Our Contributions: In this paper, we view the serverless platform as a closed-loop feedback control system and present COSE, a framework that uses Bayesian Optimization to statistically learn (with as few samples as five) 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 can pick the best configuration and placement for a serverless application which not only minimize the cost but also meet user-specified performance criteria (SLO) such as response time/delay of running a serverless application. Our framework is lightweight and has the ability to dynamically adapt to changes in the execution time of a serverless function.

Our previous work on COSE [15] was one of the earliest works that explored the configuration and placement of serverless applications to optimize cost and performance using a Machine Learning technique, namely Bayesian Optimization (BO). In this paper, we make the following additional contributions to [15]:

- We extend COSE to accommodate applications consisting of arbitrary acyclic service graphs.
- We compare COSE with some of the state-of-the-art solutions proposed and discuss COSE's advantages.
- We evaluate COSE on two real-world applications, a multi-function Personal Protective Equipment (PPE) detection application and an Image Processing application.



(a) Run-time of serverless functions on Amazon Lambda

(b) AWS price per 100ms

(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

- We enrich our review of related work by discussing most recent work in the domain of configuring serverless applications.

How to deploy COSE? It can be incorporated into an offering by cloud providers similar to *AWS Compute Optimizer [16]*; 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. In the next section, we describe the COSE framework and its components in detail.

II. System Description

Figure 2 provides an overview of our COSE framework. We view the system as a closed-loop feedback control system, where configurations are control parameters and performance/end-to-end latency are feedback signals. 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 runtime configurations for the serverless function, and the *Config Finder* component whose goal is to find the "best" configuration and placement that minimize cost while satisfying the SLO. 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. Moreover, the client application also reports the end-to-end delay (response time) to COSE. After the learning phase converges, COSE uses *Config Finder* to find the "best" configuration for an application that minimizes cost while satisfying the SLO. To account for delays associated with different locations/services supported by a designated cloud provider (*e.g.* AWS Lambda's "edge" vs. "core"), a client reports the response times of its serverless function invocations to COSE. If a change to the previous configuration is needed, COSE connects to the designated cloud provider using its APIs to modify the configuration. Next, we discuss the approaches and choices for the components of our COSE framework.

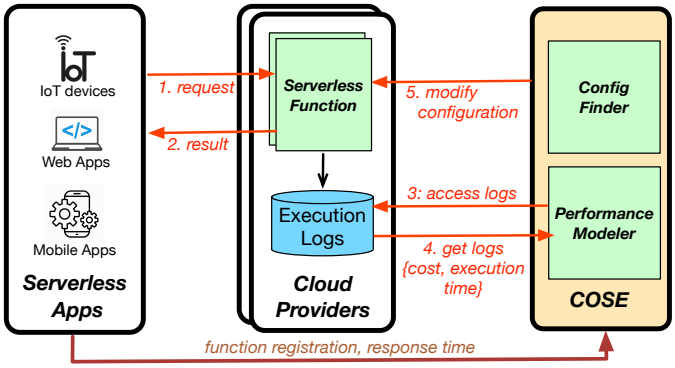
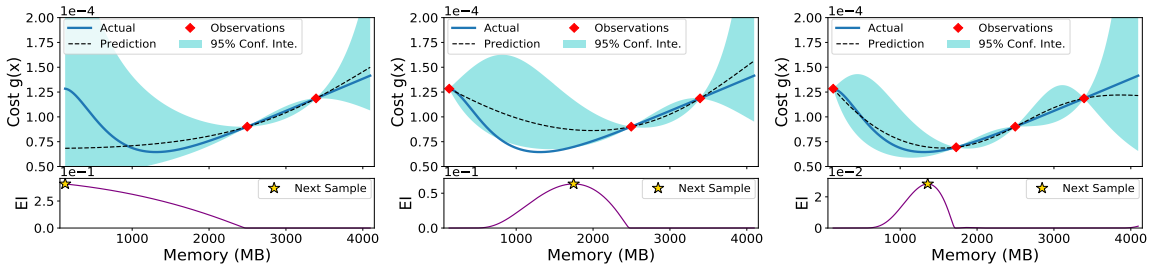


Figure 2: System overview

III. COSE: The *Performance Modeler* component

Our COSE framework has been entrusted to execute a single or a composition of serverless functions on a designated cloud provider. It has the ability to configure the parameters of individual functions in an application, *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 and cost for various configurations. We rely on statistical learning to learn this model since other approaches, such as exhaustive search and parameter decent algorithms, fail to capture an accurate estimate of the dynamic execution model or have high exploration cost as described in our previous work [15].

In this paper, we use ***Bayesian Optimization*** as the statistical learning approach to find the "best" configuration for a serverless function. It builds a probabilistic model of the underlying relation between cost and configurations, and intelligently samples the performance of an application under various resource configurations in order to reduce the sampling cost.



(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 3: Bayesian Optimization example

A. Our Approach: Leveraging Bayesian Optimization

The objective of Bayesian Optimization (BO) is to optimize over a black-box function, suppose $g(x)$. 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 confused with the serverless function/code itself that we want to execute. Each configuration x is the combination of resources (i.e. memory, CPU or both) and available locations. 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 decide where to sample next. The goal is to learn $g(x)$ in as few number of samples as possible. 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, *i.e.* performance/cost at different configurations. It then models $g(x)$ as a stochastic process and computes a confidence interval for $g(x)$ based on the samples collected. Figure 3 shows a simple example where configuration x consists of a single dimension, *i.e.,* memory for a serverless function. The *actual* underlying function (cost) 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 3a, 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 3b and 3c), the confidence interval gets smaller and the prediction is closer to the actual values. BO intelligently chooses samples to predict the underlying function $g(x)$, based on the *acquisition function* – Expected Improvement (EI) in this case. EI for a sample collected at configuration x indicates the expected improvement in the prediction of the underlying function $g(x)$. EI is calculated for each possible configuration in the unexplored search space. A configuration with the highest EI value is selected as the next sample. As shown in the lower part of Figure 3a, the highest EI value is at 128 MB for the next sample and this value is used as a configuration value for the next run of the serverless function as shown in Figure 3b. The mathematical details of *acquisition function* EI are available at [17].

B. Adapting BO for COSE

To make BO run efficiently and accurately for serverless functions, we had to make certain 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 possible configurations (data samples). In this work, we used the *memory values* and the *cloud providers* as the set of configuration parameters to choose from. We follow the choices of memory values available to customers on AWS Lambda as shown in Table I. A developer can assign a memory value between 128 MB and 10240 MB, with granularity of 1 MB. However, change in performance/cost for a function is negligible for small changes in memory values. To reduce the search space, we use a granularity of 64 MB at which we start noticing impact on performance.

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 among tenants and a provider may decide to host multiple serverless functions/applications on the same underlying resources. This resource sharing can lead to performance uncertainty. Other factors such as cold-start, hardware failures, resource-overuse, *etc*., can also impact the execution time of a serverless function running under the same configuration. We used a noise parameter to account for this performance uncertainty. 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) *Adapting to Temporal Changes:* 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 accurately capture these dynamic changes in the model, 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 potential changes in the target cost-configuration relation (cf. Section V-C).

(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 BO predicted cost/run-time vs. configuration model to find the least-cost configuration that satisfies the delay constraint on the serverless function. Next we show how delay constraints for serverless functions are met.

IV. COSE: The *Config Finder* Component

Given the performance/cost model of each serverless function in an application over available locations, predicted by the *Performance Modeler*, it is easy for the second component of our COSE framework, *Config Finder*, to find the configuration that minimizes the cost and meet the SLO. In this section, we discuss how *Config Finder* picks the configurations and placement of functions comprising the application. We assume that various functions of an application are being orchestrated by one entity/user [18].

The *Performance Modeler* essentially provides two predicted models to the *Config Finder*. First, it provides the cost model $g^f(x)$ for each function f – this is achieved by retrieving logs information, *i.e.* cost and execution time. Second, it also provides the performance model that estimates the end-to-end delay $T^f(x)$ – this model is built using the user perceived latency reported by the application.

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 service graph where functions in the application represent nodes in a directed acyclic graph (DAG) with the output of one function fed into the next function(s).

Config Finder solves this problem in two steps. We assume the application forms a directed acyclic graph G. For the whole application to execute within the SLO, every unique path should execute within the SLO. So, first, it identifies the set of all possible paths P from the start function in the graph (source) to the end function (sink). This is typically a one-time path computation as the service graph will only change if the developer alters the application. It then uses Integer Linear Programming (ILP) to find the best configuration for all functions such that the cost is minimized and SLO is satisfied. *Note that to solve for a single-function application, the service graph degenerates to a single node.*

We assume that for each serverless function f in graph G, we choose a configuration consisting of the cloud provider $v \in V$ and memory $m \in M$, such that the total cost is minimized and for each path p in P the delay constraint D is satisfied. Note the cloud provider can be an edge, core, or a region.

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

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

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

subject to:

1) Every path meets the SLO so as to guarantee that the whole application will execute within SLO:

$$\sum_{f \in p} \sum_{x \in C} T^f(x) Y_x^f \leq D \quad \forall p \in P \tag{2}$$

where $T^f(x)$ is the predicted end-to-end delay for running the serverless function $f \in p$ using configuration $x \in C$.

2) A single configuration $x \in C$ is selected for each serverless function in the application.

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

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 service graph. Note that one can argue that ILP takes long to solve. However, since a service graph typically consists of a few nodes [19], [18], [20], the total time to execute this ILP on a CPLEX solver [21] is only a few seconds.

Micro Evaluation of Config Finder

To evaluate the *Config Finder*'s ability of finding the right configurations while minimizing cost for service graph applications, we use the Image Processing workflow [22] shown in Figure 4. This application generates a thumbnail of an image uploaded to the S3 database. It first makes sure the image has a face (F1) and not a duplicate (F2) then in parallel, it indexes the face (F3) and creates the thumbnail (F4). The last step is to persist metadata. This application creates four serverless functions implementing the above functionality and was deployed using AWS Step Functions [23].

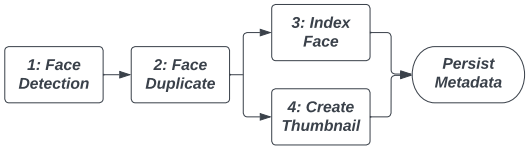


Figure 4: Image Processing Workflow

Evaluation Results: We ran the Image Processing workflow with various memory configurations of the four serverless functions to obtain each function's behaviour with respect to changes in memory allocated and then performed curve fitting to obtain function profiles as shown in Figure 5. It can be observed that the minimum achievable SLO is around 1346.91 ms and indeed when we ran our *Config Finder*, we obtained the following memory allocations $\{F1 : 701, F2 : 3000, F3 : 232, F4 : 3000\}$. We also ran our *Config Finder* with other SLOs and the results are shown in the figure – as we relax the SLO, the cost goes down as expected.

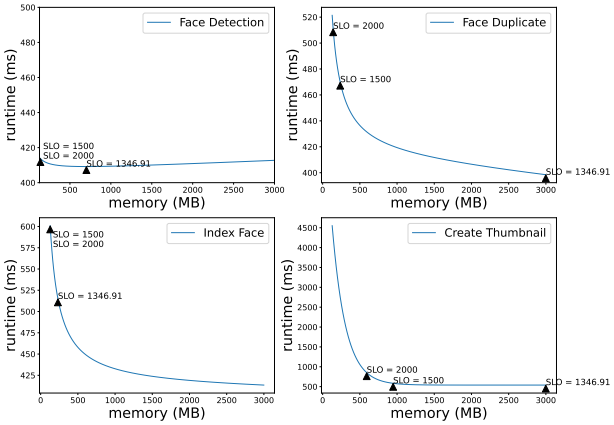


Figure 5: Function performance w.r.t. memory allocated

V. Evaluation in a Distributed Cloud Environment

While the results of running our framework on AWS Lambda (shown later in Section VI) highlight the utility of our COSE framework, evaluating additional aspects (like convergence time, accuracy, robustness, *etc.*) 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.

To establish the efficacy of our COSE framework, in addition to the experiments on AWS Lambda, we also model a distributed cloud provider and evaluate the framework across a range of scenarios 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 follow the execution/pricing model of AWS Lambda. Other aspects of serverless platforms such as cold start, instance lifetime, and the effect of co-location on performance are obtained from our experiments or previous studies such as [10]. We also simulate the edge and core cloud similar to the AWS Lambda offering, where the edge is more expensive but closer to the end user. Details of our modeled cloud provider can be found in [15], [24].

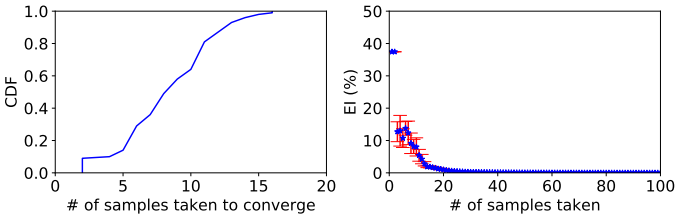


Figure 6: BO's Convergence and EI

B. Performance Metrics

We 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. In addition to other unique scenarios, we look at the following two performance metrics: **(1) *Convergence:*** COSE assumes that the *Performance Modeler* has learnt the underlying cost/performance model if the EI falls below 5%. As explained earlier, an ideal approach to configure serverless applications would learn the model quickly and have a small sampling cost as serverless platforms charge users on per execution basis. Moreover, more time spent on sampling means the application is running with suboptimal configuration which can affect both cost and SLO. **(2) *Prediction Accuracy:*** Using the learnt model, we also look at how well our approach predicted the optimal configurations.

We also tested COSE's ability to adapt in the face of changes in the execution model and to configure applications consisting of multiple serverless functions where each function follows a different performance/cost model.

C. Simulation Results

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[4]. Also, COSE showed significant cost savings without compromising on performance[5].

Single Function with Static Execution Model

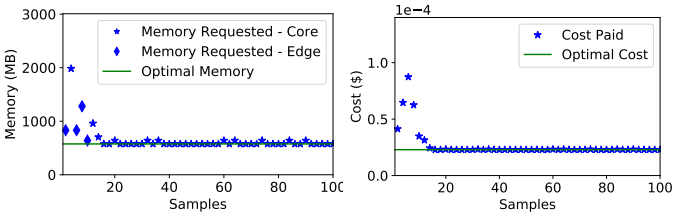
Convergence: In this experiment, we look at how long it takes for our BO-based *Performance Modeler* to converge and find the underlying cost/performance relation. Since we are using a *cloud provider model*, we know the underlying performance function. Figure 6a shows the CDF (taken over 100 runs) of the number of configuration samples taken for BO to converge. We observed that BO can converge, 95% of the time, in 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.

Prediction Accuracy: After BO converges, COSE starts picking the "best" possible configuration for serverless functions

[4]We note that for a commercial cloud provider with one parameter, COSE was able to find a near-optimal configuration in 5 samples [15]

[5]COSE code and simulation parameters are available at [24].

using *Config Finder*. We use a function whose execution model has 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 and locations. After the BO in COSE converges, *Config Finder* starts picking optimal/near-optimal configurations for the function, *i.e.* the corresponding price paid for 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

Single Function with Dynamic Execution Model

Dynamicity: The performance of a serverless function can be affected by factors like co-location, hardware it is placed on, resource provisioning policy of provider, *etc*. Most of these factors are beyond the control of the developer, hence the change in execution model is inevitable. 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. Recall that COSE keeps discarding older samples in order to adapt to new runtime conditions. In order to test COSE's ability to adapt to a change in the underlying execution model, we created 500 requests for a serverless function. For the first 250 requests, the function follows a certain execution model and has certain optimal configurations. For the next 250 requests, we change the execution model hence the optimal configurations. It will take time proportional to the history for COSE to successfully unlearn the previous execution model, learn the new model, and start predicting the new optimal configurations.

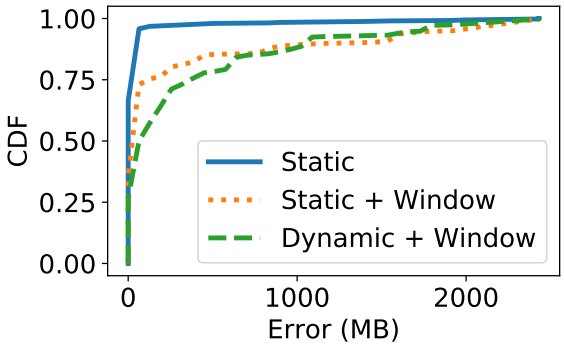


Figure 9: COSE for serverless function with dynamic execution model

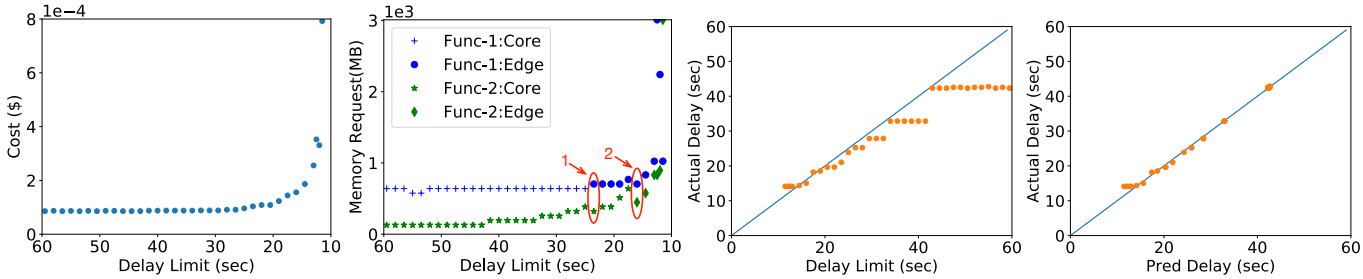
In Figure 9, we show the performance of COSE in terms of sample error in memory for three scenarios. The sample error E^s is defined as the difference between the sampled memory and the optimal memory: $E^s = |m^s - m^o|$, where m^s is the memory being sampled by BO and m^o is the optimal memory. In 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 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.

Multi-Functions Application

Serverless applications are often composed of multiple functions (forming a service graph) [25], where the output of one function serves as input to the next, and the whole application can be subjected to an SLO. These multi-function applications can be either orchestrated from a workflow manager such as Hyperflow [18], or cloud providers provide special services for this purpose such as AWS Step Functions [25]. In this experiment, we show how COSE can successfully find the *best* resource and placement configuration for such applications.

In this setup, each request is a service chain consisting of two or more functions. Each function has a different execution model, hence different optimal configuration. Initially, we let the BO in our *Performance Modeler* 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 application such that the SLO for the whole application (*i.e.* end-to-end latency) is satisfied. Although we tested COSE for different chain sizes and service graphs, for simplicity, we show results for an application consisting of two functions.

In Figure 8a, 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 end-to-end delay (response time). Both of these choices will raise the cost and that is why we see an increase in the cost as the SLO becomes tighter. In Figure 8b, we show the corresponding configurations selected for varying SLO. Initially, because of looser SLO, 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



(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 8: Delay bounded chaining of serverless functions (application) deployed across edge and core clouds

smaller, COSE has to increase the memory available to either function or change the location 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 8c, 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 8d shows that the actual delay experienced by the chain is very close to the delay predicted by COSE.

VI. Running COSE on AWS Lambda

In the previous section, we have shown by extensive simulations that COSE is able to learn the underlying execution model, and to configure single functions and service chains in a cost-effective way (less sampling). In this section, we run COSE to configure applications running on a real cloud provider. We run the COSE system on AWS Lambda, a serverless offering from AWS. AWS Lambda allows users to allocate up to 10GB memory to individual serverless functions[6] and also allows the edge deployment of certain functions to reduce the access latency [12]. We employed COSE to perform resource configuration and placement of single and multi-function serverless applications. We next describe the applications and COSE performance in detail.

Single Function Application on AWS

We test our COSE framework across four different representative functions for serverless computing. These functions can be considered equivalent to an application or task that can be implemented as one serverless function, such as inference models, various DevOps, *etc.* These functions represent the different types of computation (combination of *I/O-*, *CPU-*, *network-* and *memory*-intensive tasks) that a serverless application typically performs. Briefly, we describe these functions as follows: (i) *CPU-intensive:* This is a function that calculates the trigonometric function *atan* of multiple numbers, hence making it a 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.

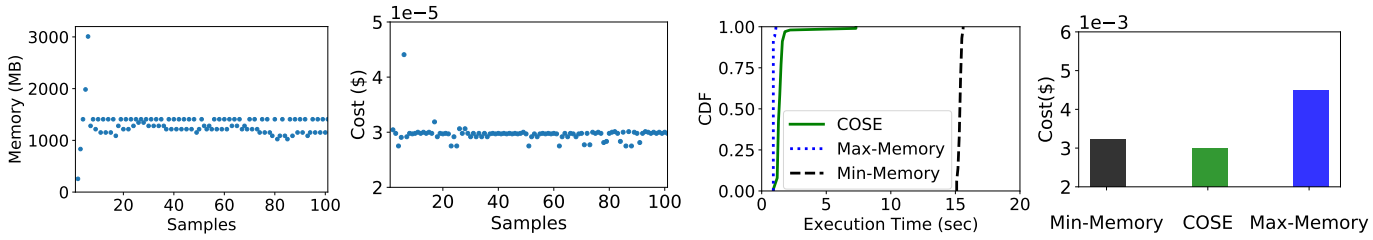
These functions were implemented in Python 3.6/3.7 and deployed on AWS Lambda. Each function was deployed as a separate AWS Lambda function. Figure 1a shows the run-time for the *CPU-intensive, memory-intensive* and *I/O-intensive* functions under different memory configurations. We do not show the results for the *network-intensive* function since changes 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. This is because 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.

Evaluation Results: We ran the *CPU-intensive*, *I/O-intensive* and *memory-intensive* functions shown in Figure 1 on AWS Lambda and employed COSE to configure resources. Since the behavior of these functions is very similar to each other, we show results for only the *I/O-intensive* function in Figure 10. To get an estimate of the optimal memory value, *i.e.* the memory value that minimizes the price, we ran the serverless function 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 10a, COSE explores different memory values to build the performance model and

[6]In AWS Lambda, CPU is allocated in proportion to memory.



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

Figure 10: COSE performance on Amazon Lambda for *I/O-Intensive* serverless function

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 10b. Again, COSE finds the optimal/near-optimal cost for the function ($\$2.9\times10^{-5}$ 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 10c shows the running time of the serverless function when invoked with configurations picked by COSE, *maximum-memory*=3008MB[7], 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*. Furthermore, the cost incurred when using COSE is even less than the cost for *minimum-memory*, as shown in Figure 10d, because of lower execution time due to near-optimal memory configuration under COSE.

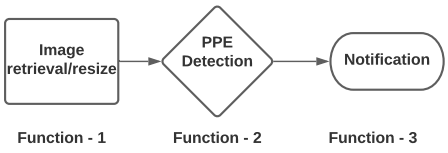


Figure 11: PPE Detection Application

Multi-Functions Application on AWS

We next evaluate COSE on a multi-function Personal Protective Equipment (PPE) detection application, which upon receiving an image from an end device, such as a surveillance camera, performs PPE detection, and notifies authorities if anyone violates PPE policy. The application's workflow is shown in Figure 11. It consists of three main functions: a) image preprocessing, b) detecting PPE, and c) notification. All three functions were implemented using Python 3.7 and utilize various AWS services such as S3 for storage, *Amazon Rekognition* to perform PPE detection, and *Simple Notification Service* (SNS) for notifications. For edge deployment we used AWS Lambda@Edge. AWS Lambda@Edge has some key

[7]Now, AWS Lambda allows maximum memory up to 10GB.

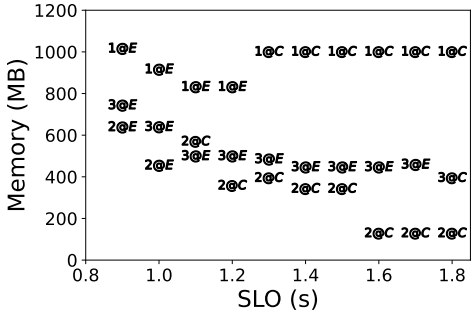


Figure 12: Memory and location w.r.t. SLO

characteristics: i) the cost of running a function on the edge can be 3x the cost of running the function on the core[26], ii) propagation delays are much smaller in the case of Lambda@Edge, and iii) resources at the edge can be limited. In our case, the core region was AWS region *us-east-1*, and the end client was placed in a university campus in *Minneapolis, Minnesota*. Lambda@Edge would execute deployment closer to the user. Note that the current offering of Lambda@Edge supports only certain types of computation (in response to events generated by the Amazon CloudFront content delivery network (CDN)) and we believe cloud providers should enhance their service by providing a more general-purpose computation utility. For our experiment, we executed our functions on the edge using the *origin-request* trigger as this is the only viable trigger type that can configure large memory and intercept the request at the edge.

Evaluation Results: Given the *Performance Modeler* has learnt the cost and performance relations of all the functions, in Figure 12 we show the memory and location configuration for various SLOs. SLO is measured as the time from first request is sent and the final response of function-3 is received,. Each marker is in the form $function\_id@deployed\_at$ where $function\_id$ can be 1, 2 or 3 corresponding to Figure 11 and $deployed\_at$ can be edge (E) or core (C). Similar to the simulation results, it can be seen that initially, when the SLO is strict, COSE places all the functions on the edge because the edge is closer to the end user (despite being more expensive) but as the SLO is relaxed, COSE first reduces the memory allocated to reduce the cost, but as SLO is further relaxed, COSE places the functions on the core given the core is cheaper.

Cost & SLO Tradeoff: In COSE, SLO dictates the overall cost, stricter SLO can be met with higher cost. In Figure 13, we show the total cost and breakdown of individual function

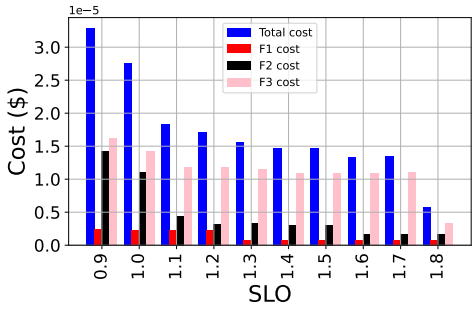


Figure 13: Cost ($) breakdown

System	Config.	Placement	Dynamic	Cost
COSE	✓	✓	✓	*Low*
Costless [19]	✓	✓	×	*Low*
Sizeless [20]	✓	×	×	*High*
SLAM [30]	✓	×	✓	*High*
StepConf [31]	✓	×	✓	*Low*

Table II: COSE and other configuration techniques

cost. Initially, when the SLO is 0.9s, all the functions are placed at the edge, as it has less propagation delays, but 3x the cost. When the SLO is relaxed, the cost decreases rapidly because now *Config Finder* can either ask for less resources on the edge or move functions to the core cloud as it is more cost-effective despite higher propagation delays.

VII. Other Configuration Techniques

Finding optimal running configurations for cloud applications is a well-studied topic and various algorithms and techniques have been proposed to address this problem. These include Reinforcement Learning, Genetic Algorithms, Machine Learning, *etc*. [27], [28]. Most of these techniques can also be extended to configure serverless applications [20], [19], [29]. We compare COSE and other configuration approaches in Table II). We will next discuss two such techniques Sizeless [20] and Costless [19] in detail.

A. Sizeless

Sizeless [20] is a recent ML-based approach to configure a serverless application. It trains a multi-target regression model with the profiling data of thousands of synthetic functions. Then using this model, Sizeless can predict the execution model of a serverless application on unseen memory configurations. The code and replication packages of Sizeless have been released, including the measurement data of different applications/functions running on AWS Lambda [32]. To compare COSE against Sizeless, we took the measurement data of these functions – Facial Recognition (*FaceSearch*), Airline Booking (*NotifyBooking*), *etc.* – and performed curve fitting on each function to obtain the actual execution model. Then we fed this execution model to COSE to see how long it takes for COSE to learn the model. In Figure 14, we report the performance of COSE on one of these functions (*FaceSearch*) – COSE performs similarly on other functions. In Figure 14a, we show the measurement data from the Sizeless replication package and the curve fit to that data. Figures 14b, 14c and 14d show COSE's execution model prediction after 3, 7 and 10 samples. We observe that after 10 samples, COSE was able to accurately predict the execution model. In our experiments, COSE took anywhere between 6 to 10 samples to predict the execution model with high accuracy. While we do not claim that COSE will always take less samples than Sizeless to converge to the execution model of a given function, we believe COSE offers certain logistical advantages as discussed next.

Logistical Advantages of COSE: COSE has certain logistical advantages over Sizeless: 1) COSE has the ability to configure applications consisting of multiple serverless functions (*service graphs*) and meeting the SLO, while Sizeless targets individual function configuration, which may not always lead to the *most* optimal configurations for such applications; 2) Currently, the released Sizeless code only supports applications written in JavaScript, while COSE can support any function written in any language as long as the application reports the user perceived latency; 3) COSE does not have any extra cost overhead other than sampling, while Sizeless needs to run thousands of synthetic functions on a serverless platform to generate the training dataset (for offline training); and 4) COSE can adapt to changes in the execution model resulting from underlying infrastructure and temporal changes, while Sizeless performs one-time configuration.

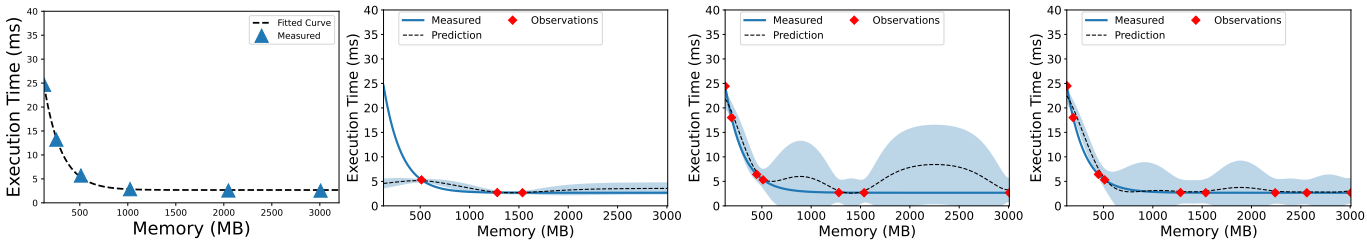
B. Costless

Costless [19] is another technique, which given a serverless application composed of multiple functions, decomposes it across edge and core clouds and perform memory configuration to minimize cost while meeting performance requirements. Costless does not infer/build the execution model of the functions but relies on profiling the application under various memory configurations and then feeds this data to a constrained shortest path optimization solver, similar to the *Config Finder* functionality of COSE. It fundamentally differs from COSE which attempts to build the cost/performance model of the application first and then perform configuration and placement. We believe approaches like Costless can benefit from COSE's *Performance Modeler* functionality to build a more accurate and robust execution model and then feed it to its optimization solver.

VIII. Related Work

Resource orchestration for cloud applications is a well-studied topic. Previous approaches [33], [34], [35], [36], [35] help a user allocate resources to applications in cluster settings to meet the SLOs and optimize resource usage.

From a developer's perspective, CherryPick [27] helps find a suitable VM configuration for cloud applications. Similarly, CloudCmp [37] recommends a suitable cloud provider for running a user application based on benchmarking results. Both CherryPick and CloudCmp are offline tools that are helpful to users *before* they deploy their applications. Similar to COSE, Ernest [38] and ARIA [39] build the performance



(a) *Face search* - Execution Model (b) *COSE's* prediction after 3 samples (c) *COSE's* prediction after 7 samples (d) *COSE's* prediction after 10 samples

Figure 14: COSE predicting the execution model

profile of various applications to suggest resources. Commercial cloud providers such as Google and AWS [40], [16] have developed systems that based on the collected performance metrics suggest suitable resource configurations. Most of these approaches target traditional cloud services such as IaaS. Unlike prior work that focuses on a particular application or on configuring resources beforehand, *i.e.* before deployment, the COSE framework can be used for any application running as a serverless function and can adapt configurations on the fly.

Another aspect of optimizing the performance of cloud applications is adding more instances of the application in case of a surge in demand. Most cloud providers offer autoscaling services for this purpose which, based on collected performance metrics, scale up or down the application. Researchers have also suggested various (reactive and predictive) autoscaling techniques [41], [42], [43], [44], [45], [46]. These techniques use methods such as performance modeling of the application, machine learning for forecasting demand, *etc*. In a serverless computing model, the scaling of an application is taken care of by the cloud provider.

While serverless computing has been around only for a few years, it has gained immense interest from the developer and research communities, hence there has been a large body of work as well as tools/techniques addressing challenges from deploying to managing cloud applications in this computing model [47], [48]. Detailed studies on different commercial serverless platforms aim to characterize and understand the architecture, resource provisioning, and performance variability offered by the cloud provider [10], [49], [50]. Similarly, benchmarking tools [51], [52] for serverless platforms have also been proposed to help the developer find a suitable provider. These studies support various insights made in our paper, and show that there can be high variability in performance because of co-location, cloud provider's resource packing policies, and underlying hardware, thereby establishing the need for more proactive approaches to configure resources to meet SLOs.

In regard to resource configuration for serverless applications, COSE is one of the first solutions in this space. Recently, several new systems have been proposed to tackle this issue including AWS's own Compute Optimizer [16] which, given a function's performance logs, can suggest optimal configurations for AWS Lambda – note, however, that since this is a proprietary solution, non-AWS users may not be able to employ it. Other works include Sizeless [20] and Costless [19], which we discuss in detail and compare to COSE in Section VII. SLAM [30] utilizes distributed tracing and performance modeling to find configurations for multi-function serverless applications. StepConf [31] is a more dynamic approach to configure serverless applications but does not perform function placement. Other approaches to configure resources for serverless functions include local simulations [53] or tracing the underlying infrastructure through logging [54]. These approaches are limited in nature as they only target function configuration. On the other hand, COSE targets both function configuration and placement dynamically and does not require changes to the application.

IX. Conclusion and Future Work

COSE is a statistical learning-based system that leverages Bayesian Optimization to learn the cost and execution time model of the serverless application. Given this model, it efficiently performs the resource configuration and placement for functions composing the application. COSE requires no application-side changes, has minimal sampling cost, and has the ability to *adapt* to changes in the execution time of serverless functions. We evaluated COSE in both simulated distributed cloud environments and real serverless applications running on AWS Lambda. Our results show that within a few samples, COSE provides optimal/near-optimal configurations and placement for serverless applications. We also discuss other state-of-the-art solutions and show how COSE has less logistical and cost overhead.

Future work includes the deployment of COSE as a service over larger-scale multi-cloud providers and adapting the solution for applications with wildly varying input workload.

Acknowledgement

This work has been supported by National Science Foundation Award CNS-1908677.

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] "Fn Project," https://fnproject.io/, 2021.
[3] "Kubeless Project," https://kubeless.io/, 2021.
[4] "Apache OpenWhisk," https://openwhisk.apache.org/, 2019.
[5] 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.
[6] "Amazon Lambda Pricing Model," https://aws.amazon.com/lambda/pricing/, 2019.
[7] "Google Function Pricing Model," https://cloud.google.com/functions/pricing, 2019.

[8] 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.

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

[10] 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.

[11] "Lambda@Edge Pricing," https://aws.amazon.com/lambda/pricing/, 2022.

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

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

[14] 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.

[15] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "Cose: Configuring serverless functions using statistical learning," in *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2020, pp. 129–138.

[16] "AWS Computer Optimizer," https://aws.amazon.com/compute-optimizer/, 2018.

[17] 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.

[18] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela, "Serverless execution of scientific workflows: Experiments with HyperFlow, AWS Lambda and Google Cloud Functions," *Future Generation Computer Systems*, 11 2017.

[19] T. Elgamal, "Costless: Optimizing cost of serverless computing through function fusion and placement," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 300–312.

[20] S. Eismann, L. Bui, J. Grohmann, C. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. New York, NY, USA: ACM, 2021, p. 248–259.

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

[22] "Serverless Image Processing," https://www.image-processing.serverlessworkshops.io/, 2022.

[23] "AWS Step Functions," https://aws.amazon.com/step-functions/, 2022.

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

[25] "AWS Step Functions," https://aws.amazon.com/step-functions/, 2022.

[26] "AWS Lambda Pricing," https://aws.amazon.com/lambda/pricing/, 2022.

[27] 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.

[28] Z. Cao, V. Tarasov, S. Tiwari, and E. Zadok, "Towards better understanding of black-box auto-tuning: A comparative analysis for storage systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 893–907.

[29] L. Schuler, S. Jamil, and N. Kühl, "Ai-based resource allocation: Reinforcement learning for adaptive auto-scaling in serverless environments," in *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2021, pp. 804–811.

[30] G. Safaryan, A. Jindal, M. Chadha, and M. Gerndt, "Slam: Slo-aware memory optimization for serverless applications," in *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, 2022, pp. 30–39.

[31] Z. Wen, Y. Wang, and F. Liu, "Stepconf: Slo-aware dynamic resource configuration for serverless function workflows," in *IEEE INFOCOM 2022 - IEEE Conference on Computer Communications*, 2022, pp. 1868–1877.

[32] "Sizeless Replication Package," https://codeocean.com/capsule/6066333/tree/v2, 2021.

[33] 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.

[34] 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.

[35] 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.

[36] C. Delimitrou and C. Kozyrakis, "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.

[37] 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.

[38] 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.

[39] 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.

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

[41] W. Fang, Z. Lu, J. Wu, and Z. Cao, "RPPS: a novel resource prediction and provisioning scheme in cloud data center," in *2012 IEEE SCC, Honolulu, HI*.

[42] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, 2011, pp. 500–507.

[43] L. Aniello, S. Bonomi, F. Lombardi, A. Zelli, and R. Baldoni, "An architecture for automatic scaling of replicated services," in *Network Systems (NETYS)*, ser. 8593. Springer, August 2014.

[44] A. Y. Nikravesh, S. A. Ajila, and C. Lung, "Towards an autonomic auto-scaling prediction system for cloud resource provisioning," in *2015 IEEE/ACM SEAMS, Firenze, Italy*.

[45] A. H. Mahmud, Y. He, and S. Ren, "BATS: budget-constrained autoscaling for cloud performance optimization," in *2015 IEEE MASCOTS, Atlanta, GA*.

[46] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: exploiting cloud services for cost-effective, slo-aware machine learning inference serving," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1049–1062.

[47] A. Raza, I. Matta, N. Akhtar, V. Kalavri, and V. Isahagian, "SoK: Function-as-a-Service: From An Application Developer's Perspective," *Journal of Systems Research - JSys*, vol. 1, no. 1, 2021/09/19,01:00.

[48] J. Scheuner and P. Leitner, "Function-as-a-service performance evaluation: A multivocal literature review," *Journal of Systems and Software*, vol. 170, p. 110708, 06 2020.

[49] 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.

[50] 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.

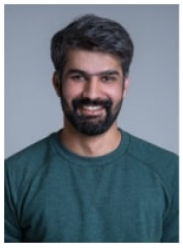
[51] P. Maissen, P. Felber, P. Kropf, and V. Schiavoni, "FaaSdom: a benchmark suite for serverless computing," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 73–84.

[52] T. Yu, Q. Liu, D. Du, Y. Xia, B. Zang, Z. Lu, P. Yang, C. Qin, and H. Chen, "Characterizing serverless platforms with Serverlessbench," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 30–44.

[53] J. Manner, M. Endreß, S. Bohm, and G. Wirtz, "Optimizing cloud function configuration via local simulations," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 168–178.

[54] R. Cordingly, W. Shu, and W. J. Lloyd, "Predicting performance and cost of serverless computing functions with saaf," in *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on*

Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), 2020, pp. 640–649.



Ali Raza (araza@bu.edu) is a Ph.D. candidate in the Computer Science department of Boston University (BU). Currently, his work deals with the orchestration of cloud resources for applications with strict performance requirements. Before joining BU, he was a researcher at NYU Abu Dhabi, where he worked on improving web connectivity in developing regions through advanced web-caching techniques. He is the recipient of the best paper award (research track) from IEEE IC2E'21 and best project award from IEEE INFOCOM-CNERT 2018.

Nabeel Akhtar (nakhtar@akamai.com) received his Ph.D. in Computer Science from Boston University in 2019. He is part of the Network Control group at Akamai Technologies Inc. His research involves Network Optimization and Modeling, Architecture Design, and Resource Optimization. He has served as a TPC member for multiple conferences. He has served as a reviewer for over 30 conferences and journals. He received the "Neal Shepherd Memorial Best Propagation Paper Award 2020" from IEEE Vehicular Technology Society and best paper award from IEEE IC2E'21. He has also received the Best Project Awards at ICMV 2010 and IEEE INFOCOM-CNERT 2018.

Vatche Isahagian (vatchei@ibm.com) is a Senior Research Scientist and manager at IBM Research in Cambridge, Massachusetts. He earned his Ph.D. in Computer Science from Boston University in 2013. His research spans a broad set of disciplines across Distributed Systems, Artificial Intelligence, and Business Processes. His current focus is on utilizing AI techniques such as natural language processing and multi-agent systems to enable AI-enhanced business automations. Vatche's work has resulted in multiple patent filings, peer-reviewed publications in conferences and journals, as well as two best paper awards. Additionally, he has organized several workshops and served as a member of program committees, as well as co-chair and publicity chair for various conferences. He is a senior member of both the IEEE and the ACM.

Ibrahim Matta (matta@bu.edu) (M'93-SM'06) received his Ph.D. in computer science from the University of Maryland at College Park in 1995. He is a professor and chair of computer science at Boston University. His research involves network protocols, architectures, and performance evaluation. He has published over 100 peer-reviewed articles. He received the National Science Foundation CAREER award in 1997 for his research on QoS routing. His work on wireless sensor networks and cloud resource management received best paper awards. He has served as the chair or co-chair of many technical committees, including IEEE 2011 CCW and 2005 ICNP. He is a senior member of ACM and IEEE.

Lei Huang (lei@bu.edu) is a Ph.D. student in the Computer Science department of Boston University (BU). His research interests are broadly in systems and networking, with a focus on trading off cost and performance in data center environments. Before joining BU, he obtained his Master's and Bachelor's degrees from the University of Minnesota, where he focused on achieving elasticity in edge environments for latency-sensitive applications.