LIBRA: An Economical Hybrid Approach for Cloud Applications with Strict SLAs

Ali Raza, Boston University, araza@bu.edu
Zongshun Zhang, Boston University, zhangzs@bu.edu
Nabeel Akhtar, Akamai Technologies Inc., nakhtar@akamai.com
Vatche Isahagian, IBM Research, vatchei@ibm.com
Ibrahim Matta, Boston University, matta@bu.edu

***Abstract*—Function-as-a-Service (FaaS) has recently emerged to reduce the deployment cost of running cloud applications compared to Infrastructure-as-a-Service (IaaS). FaaS follows a serverless "pay-as-you-go" computing model; it comes at a higher cost per unit of execution time but typically application functions experience lower provisioning time (startup delay). IaaS requires the provisioning of Virtual Machines, which typically suffer from longer cold-start delays that cause higher queuing delays and higher request drop rates. We present LIBRA, a balanced (hybrid) approach that leverages both VM-based and serverless resources to efficiently manage cloud resources for the applications. LIBRA closely monitors the application demand and provisions appropriate VM and serverless resources such that the running cost is minimized and Service-Level Agreements are met. Unlike state of the art, LIBRA not only hides VM cold-start delays, and hence reduces response time, by leveraging serverless, but also directs a low-rate bursty portion of the demand to serverless where it would be less costly than spinning up new VMs. We evaluate LIBRA on real traces in a simulated environment as well as on the AWS commercial cloud. Our results show that LIBRA outperforms other resource-provisioning policies, including a recent hybrid approach – LIBRA achieves more than 85% reduction in SLA violations and up to 53% cost savings.**

***Index Terms*—EC2, Lambda, IaaS, FaaS.**

I. Introduction

In recent years, fueled by increased cloud adoption, business demands, and advances in technology (*e.g.* AI, edge computing, etc.), public cloud providers expanded their offerings to include a plethora of services. These services cater for a wide spectrum of customers[1] from those who want to control their own development stack to those who want to focus on designing and delivering new features without worrying about the underlying infrastructure. All these services come with varying configurations and have different performance and pricing models. Given all the service options available from the cloud providers, choosing the best suitable service to deploy applications is a challenging problem.

Infrastructure-as-a-Service (IaaS) is one of the most common cloud services available. It allows a tenant to lease VMs with particular configurations to deploy a cloud application. A customer can add or remove VM instances in response to fluctuating demand of the application. Depending on the VM configuration and cloud provider, it can take hundreds of seconds to set up an instance of a VM (cold-start). This can have an adverse effect on the performance of a cloud application, particularly in the presence of flash-crowd traffic demand, where the application demand increases quicker than the VM resource provisioning time. To deal with this problem, cloud providers offer auto-scaling service. This service monitors resource utilization (*e.g.*, CPU or memory), and scales out (increase) or in (decrease) VM instances based on user defined thresholds.

Function-as-a-Service (FaaS) [1] is a recent popular offering from most cloud providers. In this model, a tenant writes the code of the application (called serverless function) in one of the supported languages, and submits the code to a cloud provider along with dependencies and associated execution triggers. On an invocation, the code is executed in a sandbox environment, and the result is returned to the triggering event. In the past few years, FaaS has evolved to the point that most cloud applications [2]–[9] can be developed using FaaS, or given the application, it can be easily translated into the FaaS programming model [10]. A value-added of FaaS is that it does not require explicit scaling instructions from a tenant. The serverless platform is responsible for scaling the resources to address fluctuating demand.

There are two distinct features that set FaaS apart from IaaS: *1) Quick provisioning:* serverless functions are executed in sandbox environments which can be provisioned within a few milliseconds [11]. This feature makes FaaS an ideal service to serve flash-crowd traffic demands. *2) Pricing model:* While both FaaS and IaaS follow the spirit of "pay as you go" pricing model, FaaS promises real "pay as you go" with no waste of resources [1]. Under the IaaS model, customers are charged for the entire duration a VM is leased independent of the utilization (*i.e.* whether or not the application is running). Under the FaaS model, customers are only charged for the execution time of the application function. While the cost per unit time of execution in FaaS is comparatively higher than IaaS with the same resources, its cost model – charging the customer for the precise amount of time an application is running – makes it an ideal choice for low-duty demand cycles, where leasing a VM instance for a longer time with little to no utilization can be expensive.

Considering the performance and pricing models for IaaS and FaaS, in this paper we show that a combination of IaaS and FaaS is ideal to cater to the dynamic demand of a cloud application. Previous hybrid approaches [2], [12], [13] have leveraged the quick-provisioning-time feature of FaaS by directing a portion of the demand to FaaS *temporarily* while scaling out IaaS based resources (*i.e.* VMs), which leads to lower service-level agreement (SLA) violations. To the best

[1]We use the terms "customer" and "tenant" interchangeably.

of our knowledge, no work has investigated the simultaneous use of FaaS to reduce the overall cost by *consistently* directing the low-rate bursty portion of the demand to FaaS.

In this paper, we present LIBRA, a load balancing approach for cloud applications with strict SLA requirements[2]. Two contributions set LIBRA apart from previous hybrid approaches: 1) LIBRA continually monitors the demand for an application and procures resources in IaaS, FaaS, or both to cater to the demand while optimizing the cost and performance of an application. In contrast, previous hybrid approaches employ VMs as their primary resource and only leverage FaaS to hide VM startup delays; 2) We compare LIBRA to Spock [12] (a recently proposed hybrid approach) and other resource provisioning policies. Our evaluation of LIBRA on both Amazon Web Services (AWS) and a simulated cloud shows that LIBRA outperforms these other approaches in lowering the overall cost of a cloud application while reducing SLA violations in the presence of dynamic demand.

Our contributions are summarized as follows:

- In Section II, we present an economic model to analyze the cost of using IaaS versus FaaS to serve a given application demand. We derive a "cost-indifference point" (CIP) that determines an upper bound on the demand rate to direct to FaaS that guarantees a lower cost compared to using IaaS.
- We present the architecture of LIBRA (Section III), a hybrid load balancing approach that *simultaneously* uses both FaaS and IaaS, motivated by our analysis that a portion of the demand sent to FaaS at a rate lower than CIP can be served in a cost-effective way.
- We evaluate LIBRA in a simulated cloud environment (Section IV) and on AWS services (Section V) with real application demand traces to establish its efficacy. Our results show that LIBRA outperforms other resource provisioning policies, including Spock [12], in reducing both cost (48% compared to FaaS, 53% compared to VM over-provisioning, and up to 20% compared to VM auto-scaling and Spock) and SLA violations (more than 85% compared to auto-scaling and Spock).

Section VI summarizes related work, and Section VII concludes the paper.

II. Background & Motivation

Despite development and deployment challenges, IaaS and FaaS are popular choices for building and deploying cloud applications. Typically, developers target one of these services for an application deployment, but recent trends have shown that simultaneous use of these services can improve the application's performance [2], [12]. We contend that simultaneously leveraging both services (IaaS and FaaS) not only improves the performance but can also decrease an application's operational cost. In particular, IaaS is more economical for supporting the high-rate steady portion of the load of application requests, while FaaS is more economical for supporting the low-rate bursty portion of that load.

FaaS provisions serverless functions in as little as 10s of milliseconds [11]. This quick provisioning time of FaaS has been leveraged in the literature to hide VM cold-start delays and hence lowering SLA violations [2], [12], [13]. In contrast to these previous approaches, LIBRA consistently monitors the application load, and directs a portion of the application load to FaaS when it is a cheaper alternative to provisioned VMs.

In this section, we present an economic (cost) model of an application deployed using either FaaS or IaaS cloud service. We use this model to derive the "cost indifference point" (CIP) as a function of the request arrival rate, where the costs of using IaaS or FaaS for an application are equal. If the application load is below this CIP value, it is more economical to use FaaS. For higher loads, IaaS is more economical. This analysis motivates the design of our LIBRA approach.

A. FaaS: Serverless Pricing Model

Serverless platforms follow a "pay as you go" pricing model where the user is only charged for the execution time of the serverless function based on a particular configuration (*e.g.*, memory) [14].

Our previous work [15] studies the effect of configurable resources on various types of serverless functions in AWS Lambda. It showed that AWS Lambda's execution time follows exponential decay (*i.e.*, diminishing return, as shown in Figure 1a), and can be expressed as follows:

$$t_f^{FaaS}(m) \approx t_f^{FaaS}(m_{max}) + \left(t_f^{FaaS}(m_{min}) - t_f^{FaaS}(m_{max})\right) e^{-\lambda(m-m_{min})} \quad (1)$$

where $t_f^{FaaS}(m)$ is the execution time of a function f when allocated memory m MB, $t_f^{FaaS}(m_{min})$ is the running time of f at the smallest possible memory configuration (m_{min} = 128 MB for Amazon Lambda), $t_f^{FaaS}(m_{max})$ is the running time at the largest possible memory configuration (m_{max} = 3008 MB for Amazon Lambda), and λ is a decay constant.

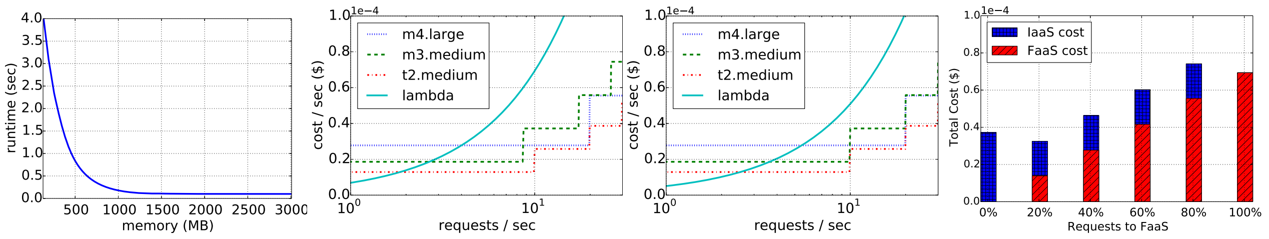
Consider an application, deployed using FaaS, that receives N requests per second, where each request causes the execution of a serverless function f. The usage cost per second can be calculated as follows:

$$cost_{FaaS} = \sum_{i=1}^{N} \left(t_f^{FaaS}(m) \times C_{FaaS}(p,m) + G_{FaaS}(p)\right) \quad (2)$$

where $C_{FaaS}(p,m)$ is the cost per unit time[3] of executing a serverless function as specified by the serverless platform p for a given configuration m, and $G_{FaaS}(p)$ is the total fixed cost charged by the cloud provider (such as API-gateway cost for AWS Lambda [14]). This cost model also holds for other cloud providers which follow similar pricing models for FaaS, such as IBM Functions, Google Cloud Functions, *etc.*

[2]Examples of such applications are machine learning inference models, IoT applications that collect/process data, etc.

[3]Serverless platforms currently charge for every 1ms or 100ms of execution time, depending on the platform [14], [16]–[18].



(a) AWS Lambda Execution Model (b) Cost at Request Memory=512MB (c) Cost at Request Memory=3008MB (d) Hybrid Case

Fig. 1: Cost comparison of Amazon Lambda and EC2 instances for varying average request arrival rate

B. *IaaS: VM Pricing Model*

In the IaaS model, a tenant leases a VM with a particular configuration, such as memory, CPU, and storage, to deploy an application. A tenant is charged the cost of the VM independent of its utilization. A VM with a particular configuration can only serve a certain number of requests in a given time period while meeting SLA requirements.

If a VM can host at most r_{max} requests per second without violating the SLA, and the IaaS based deployment receives N requests per second, the cost per second can be calculated as follows:

$$cost_{IaaS} = \lceil \frac{N}{r_{max}} \rceil \times C_{vm}(p) \tag{3}$$

where $C_{vm}(p)$ is the cost per second[4] of renting a particular VM (vm) from a certain cloud provider p.

C. *Cost Analysis*

Using Equations (2) and (3), we compare the cost of deploying an application using FaaS or IaaS, respectively. We evaluate the cost for varying demand given by N, the rate of requests for the application, where each request causes the invocation of application code deployed using FaaS or IaaS.

The execution model of the application/function used is shown in Figure 1a and follows an exponential decay (*i.e.*, diminishing return) in the running time of the function with respect to the amount of resources (memory) allocated [15], [22]. It gives the execution time $t_f^{FaaS}(m)$ of the application for different memory m settings when deployed using FaaS. IaaS based deployment would follow a slightly different execution model as underlying resources can differ from FaaS.

Thus, the execution model of the IaaS based deployment with respect to FaaS can be described as:

$$t_f^{vm}(m) = \tau \times t_f^{FaaS}(m) \tag{4}$$

where $t_f^{vm}(m)$ is the execution time of the IaaS based deployment when allocated memory m to each request, and τ is a constant whose value is a real positive number and can vary based on the application and underlying resources. Without loss of generality, we show results where under both IaaS and FaaS, the application follows the same execution model (*i.e.*, $\tau = 1$), and memory is the bottleneck resource in the execution of the function as most FaaS platforms allow only memory as a configurable resource[5]. Note that setting τ to values different than 1 does not qualitatively affect the results of our analysis.

Using $t_f^{FaaS}(m)$ and m, we calculate $cost_{FaaS}$ using Equation (2), where the costs $C_{FaaS}(p, m)$ and $G_{FaaS}(p)$ are taken from AWS Lambda pricing [14].

For IaaS, the $cost_{IaaS}$ is calculated using Equation (3), where r_{max}, the maximum number of requests that a VM with memory M can handle in one second, can be derived using Little's Law [23]:

$$\frac{M}{m} = r_{max} \times t_f^{vm}(m)$$

$\frac{M}{m}$, the long-term average number of concurrent requests in the system, equals the arrival rate of these requests (r_{max}) times the (average) time that a request spends in the system ($t_f^{vm}(m)$). We thus have:

$$r_{max} = \frac{M}{m} \times \frac{1}{t_f^{vm}(m)} \tag{5}$$

We use the AWS Elastic Compute Cloud (EC2) pricing model for different types of EC2 instances (m4.large, m3.medium, and t2.medium). The cost $C_{vm}(p)$ and memory resources M of these instances are specified in EC2 pricing [20]. Figure 1b compares the cost of cloud usage when an application is deployed in AWS Lambda or in various instances of EC2 for varying request rate and memory m of 512MB. The x-axis is drawn on a logarithmic scale for better readability. We observe that the FaaS model is cost effective when the request rate is below 4 requests/second for the m4.large EC2 instance (the point where the m4.large and lambda cost curves intersect). This represents the cost-indifference point (CIP) beyond which the IaaS model is cheaper to be used. The CIP is obtained by equating Equations (2) and (3). Figures 1c shows a similar behavior when each request is using memory m of 3008MB.

Though the results shown here are obtained using AWS pricing, the cost model is applicable to other cloud services (*e.g.*, from IBM and Google) that follow a similar pricing model. To summarize the key takeaways from our analysis:

- The FaaS model is cheaper to use for low duty-cycle application, *i.e.* when the average request rate N is below the CIP. For higher values of N, IaaS is cheaper.

[4]IaaS resources can be rented on an hourly basis, while a user can also be charged for partial usage (per second) [19]–[21].

[5]Other resources, such as CPU, I/O, Network, etc., can be bottleneck in the execution of a function. These resources can be substituted here to get similar analysis.

- The value of CIP depends on the amount of resources used by each request and the type of VM instance. A tenant can find the appropriate resource configuration by profiling the application or using inference approaches, as proposed in [15], [24].

D. LIBRA's Motivation

Demand for an application can significantly vary across certain hours of the day and certain days of the week. Based on our analysis in previous sections, an ideal hybrid load balancing approach will have two main characteristics:

(a) It would continually monitor the demand for an application and when the demand is below CIP, it will only provision FaaS resources to cater to the demand as they are more cost-effective in such a scenario. This is a feature that previous hybrid approaches [2], [12] lack, as they only use FaaS either for transient demand, or during scaling out VM resources to avoid SLA violations.

(b) It should employ FaaS consistently for a low-rate bursty portion of the demand that the system can not serve using IaaS resources within the SLA. This would be beneficial in two ways: first, it will reduce the SLA violations, as sudden spikes in demand would be handled by FaaS, which has negligible cold-start delays and can natively scale-out. Second, consistently employing FaaS for a certain portion of the demand can lead to significant cost savings.

To demonstrate the cost saving of such an approach, we leverage the cost analysis in Section II-C. Consider the scenario shown in Figure 1b, where an application is running on an EC2 instance of the type *m3.medium* and it has a steady demand of 10 requests per second where each request requires 512MB of memory. In Figure 1d, we compare the cost of serving all the demand or a certain portion of it using FaaS while serving the rest through IaaS. We observe that a hybrid approach, where around 20% requests are served by FaaS and the remaining by IaaS is the most cost-effective as compared to IaaS or FaaS only scenarios. This is because 20% of the demand is below the CIP for this particular case and is cheaper to be served through FaaS than spinning up a new VM which would be underutilized.

III. LIBRA Architecture

In this paper, we present LIBRA, a balanced approach that leverages both IaaS and FaaS. It closely monitors the demand from an application and provisions appropriate VM capacity for the IaaS deployment to handle a portion of the requests while directing the rest to be handled by the FaaS based implementation of the application. Motivated by our analysis in Section II, the design of LIBRA derives from the following goals:

- An efficient load balancing approach would utilize FaaS for bursty demand to avoid SLA violations, leveraging FaaS's quick provisioning time.
- As explained in Section II-C, a steady-rate of traffic below a certain limit (CIP) can be cheaper to serve through serverless functions (FaaS). An efficient load balancing approach would leverage FaaS for a steady (low) rate of traffic *consistently* to reduce the overall cost of cloud usage.

The above two goals present load balancing between FaaS and IaaS as a marginal analysis problem [25], where a user can direct a small/bursty portion of the demand to FaaS (additional activity) to be served cost effectively instead of provisioning new VMs. This would not only reduce cost, but also lead to lower SLA violations as VMs take significantly longer to start (cold starts). Figure 2 gives an overview of our proposed approach. The load balancing across the IaaS and FaaS based implementations of the application is performed through a *Load Balancer*, which also updates the traffic statistics and share them with a *Traffic Monitor* using the control plane. Based on the traffic demand, a *Scaling Manager* provisions VM resources for IaaS and updates the Load Balancer to enforce appropriate forwarding rules. Henceforth, we refer to all three components of LIBRA as LIBRA Gateway (LG). LIBRA can be deployed by a cloud/service provider as a value-added service or can be directly leveraged by the customers. In what follows, we explain each component of LIBRA in detail.

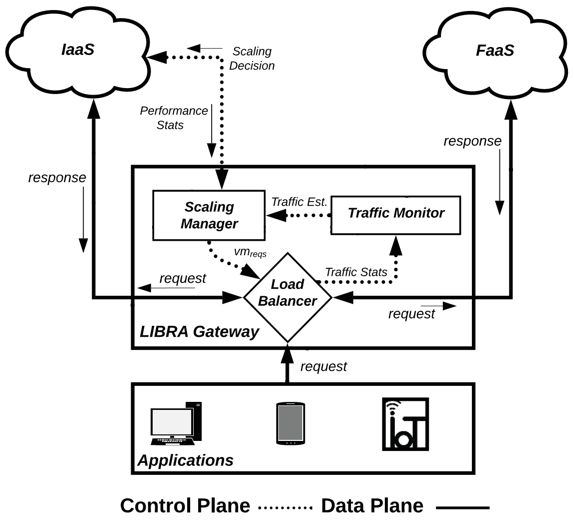


Fig. 2: LIBRA architecture

A. Traffic Monitor

The Traffic Monitor (TM) continually receives traffic updates from the *Load Balancer*. Using the historical data of these updates, the TM estimates the future load for the application. The forehand knowledge of traffic is critical particularly for the VM-based resources, because they can take up to several minutes to start and be ready to serve application traffic. In the current implementation, we introduce the notion of an *epoch* that represents a configurable unit-time (*e.g.*, 10 seconds or 1 minute). The LB continually reports the number of requests received in an *epoch* to the

TM, which uses this information to estimate the future load. Currently, the TM keeps track of the Exponentially Weighted Moving Average (EWMA) and sample deviation of requests received in previous epochs as given in Equations (6) and (7), where $reqs_{curr}$ is the number of requests received in the current epoch, α and $\beta \in [0, 1]$ are configurable based on how quickly a user wants the system to react in the face of traffic variations. Our experiments have shown that EWMA and sample deviation of the number of requests track well the traffic variation[6] as shown in Figure 3.

$$avg = (1 - \alpha) \times avg + \alpha \times reqs_{curr} \tag{6}$$

$$std = (1 - \beta) \times std + \beta \times |reqs_{curr} - avg| \tag{7}$$

The TM reports the avg and std values to the *Scaling Manager* every K epochs. The *Scaling Manager* then makes scaling decisions as explained next.

B. Scaling Manager

The Scaling Manager (SM) is a crucial component of our LIBRA architecture. It periodically receives the traffic statistics from the *Traffic Monitor* and orchestrates resources in the form of VMs and serverless functions to serve application requests.

As one of the design goals of LIBRA is to keep the serverless load below a certain threshold (CIP) to avoid overpaying, and send the maximum stable load to provisioned VMs for their cost effectiveness, our SM provisions VM (IaaS) resources that can handle a request rate equal to $(avg{+}\phi{\cdot}std)$, with remaining requests directed to run as serverless functions (FaaS). $\phi \in \mathbb{R}$ is a configurable parameter of the LIBRA system *and is discussed later in Section III-D.* Our experiments (cf. Sections IV and V) have shown that any traffic above $(avg + \phi \cdot std)$ is either transient or cheaper to be served by serverless functions. LIBRA provisions VMs cautiously based on the estimated demand given by $(avg + \phi \cdot std)$ so as to avoid either under-provisioning VMs and then suffering from higher startup-delays while spinning up additional VMs, or over-provisioning VMs and paying unnecessary cost due to VM under-utilization.

Algorithm 1 describes how the SM procures VM resources for IaaS. In line 1, the function $get\_active\_vms()$ returns the current number of active VMs. Line 2 checks if the average request rate is below the CIP threshold (obtained through cost analysis). If it is, LIBRA shuts down and deallocates all the currently provisioned VMs (line 3), as the current demand can be met cost-effectively using only serverless functions. In line 6, the algorithm calculates the number of requests (r_{max}) that a VM with given resources can serve while meeting the SLA. In the case of homogeneous VM resources and consistent workload for each request, r_{max} can be calculated by using Equation (5). In the case of variable workload, adaptive controllers (e.g., PID [26]) can be used to set r_{max}. A proportional–integral–derivative (PID) controller can adapt r_{max} based on the error between the target and measured response/service time. In lines 7-8, we obtain the number of VM instances that are needed to cater to a demand $vm_{reqs} = avg + \phi \cdot std$, as instantaneous requests beyond that value are considered transient and will be handled by serverless functions. The functions add_vms and $remove\_vms$ (lines 11 and 13) implement the VM-cloud (IaaS) interface to allocate or deallocate VMs to achieve the desired $num\_instances$ (line 8). The initial provisioning of VMs is performed based on user configurations similar to other autoscaling services [27]. Moreover, if the decision is to add more VMs, the *Scaling Manager* waits until the VMs are in ready state before sending a vm_{reqs} update to the *Load Balancer*.

[6]A LIBRA user can employ other traffic prediction models as well.

Algorithm 1 LIBRA's Scaling Algorithm

```
Input:
$avg$, $std$: EWMA and sample deviation reported by the Traffic Monitor (TM)
$VM_{res}$: resources available in each VM instance
$req_{res}$: resources required to serve one application request
$req_{time}$: average request service time
$cip$: Cost Indifference Point
$\phi$: number of sample deviations beyond average demand
Output:
$vm_{reqs}$ // request rate that provisioned VMs can handle

1: $active\_vms$ = $get\_active\_vms()$
                    // returns the number of active VMs
2: if $avg < cip$ then
3:     $remove\_vms(active\_vms)$
                    // removes all VM instances
       $vm_{reqs}$ = 0
4:     return
5: end if
6: $r_{max}$ = $vm\_capacity(VM_{res}, req_{res}, req_{time})$
        // get maximum number of requests a VM can serve
7: $vm_{reqs}$ = $avg + \phi \cdot std$
8: $num\_instances$ = $\lceil (vm_{reqs}/r_{max}) \rceil$
9: $vm\_diff$ = $num\_instances$ - $active\_vms$
10: if $vm\_diff > 0$ then
11:     $add\_vms(vm\_diff)$ // adds VM instances
12: else
13:     $remove\_vms(vm\_diff)$ // removes VM instances
14: end if
```

C. Load Balancer

The Load Balancer (LB) receives requests from the end-users and forwards them to the appropriate resources, either VMs (IaaS) or serverless (FaaS). It also keeps track of the requests received in an epoch and periodically notifies the *Traffic Monitor*. Moreover, whenever the *Scaling Manager* makes a scaling decision, it reports the new value of vm_{reqs} to the *Load Balancer* as the *Scaling Manager* provisions VMs to accommodate a request rate of vm_{reqs}. From queuing theory [28], to ensure stable (predictable) performance and small queuing delays, the request rate to the provisioned VMs should be lower than the service rate given by the VM provisioned rate of vm_{reqs}. This keeps the aggregate utilization of the provisioned VMs below one. Consequently, our LB directs only a fraction ρ of the request rate, *i.e.*, $\rho \cdot vm_{reqs}$ to the VM resources. *This fraction ρ of provisioned VM capacity that can be used to serve requests is a configurable parameter of LIBRA and discussed in detail in Section III-D*.

The LB adopts a forwarding approach that directs requests to VMs (IaaS) first, which has two key benefits: 1) The VMs are already in ready state and will not incur any cold-start delays, and 2) ready VMs are cheaper compared to serverless.

D. LIBRA Parameters

Our LIBRA approach has the following configurable parameters that an administrator can tweak to maximize their gain whether it is performance, cost, or both. We studied the behavior of these parameters in simulation and experimentally, and here we briefly summarize the effect of the following parameters and their recommended settings.

1) EWMA Weights: The *Traffic Monitor* in LIBRA uses EWMA to monitor the average rate of requests and sample deviation. The weights α and β given to the most recent number of requests observed over the current epoch are configurable parameters. A high weight value can lead to a quick response to a sudden increase in demand, resulting in over-provisioning of VM resources if the increase were transient. On the other hand, a low weight value can lead to a slow response to a sudden increase in demand, resulting in under-provisioning of VM resources if the decrease were persistent, which increases the usage of serverless functions and results in a higher cost.

2) Scaling Decision Interval: The *Scaling Manager* makes scaling decisions every K epochs. A smaller value of K (*e.g.*, less than the startup delay of VMs) can result in back-to-back scaling decisions without waiting for the system to react and reach equilibrium. A large value of K results in longer intervals between scaling decisions, hence slower adaptation leading to missing potential cost savings. The scaling decision interval should be larger than the startup delay of the VM instances being used. This is because when the *Scaling Manager* makes the decision to scale out, it waits for the newly added VM instances to be in ready state before informing the *Load Balancer* of the newly provisioned VM capacity. In our experiments we set the scaling decision interval to be three times the VM startup delay.

3) VM Utilization: As described in Section III-C, the *Load Balancer* only utilizes a certain proportion (ρ) of the provisioned VMs. The goal is to make sure that the VMs serve the requests without SLA violations (cf. Section III-C). In our experiments we set this parameter ρ to 80%.

4) Traffic Estimation: To estimate traffic demand, LIBRA uses the EWMA and sample deviation to obtain $(avg + \phi \cdot std)$, where $\phi \in \mathbb{R}$ and its value depends on the fluctuations in demand. In LIBRA, IaaS resources are provisioned for $(avg + \phi \cdot std)$ demand. Hence, a higher value of ϕ will cause more aggressive provisioning of VMs, which can potentially lead to VM under-utilization. On the other hand, a lower value of ϕ can lead to more FaaS usage which can potentially lead to higher cost as serving requests by FaaS is expensive.

In the next two sections, we evaluate LIBRA using simulations and on Amazon AWS.

IV. Simulation Model & Results

LIBRA closely monitors the demand for an application and consequently provisions VM resources, while the transient spikes and small portion of the demand are served by serverless functions. This approach results in little to no SLA violations while also reduces the cost of cloud usage for the tenant. To evaluate the long-term efficacy of LIBRA, we modeled [7] various cloud services after Amazon Web Services (AWS) [30]. These include IaaS, FaaS, Load Balancer, and Autoscaler. Using real traces, we evaluated LIBRA against different approaches: VM over-provisioning, FaaS-only, and provisioning of VM resources using the autoscaler.

A. Modeling Cloud Services

We modeled IaaS and FaaS (and related services) after AWS EC2 and Amazon Lambda, respectively.

1) IaaS: Our modeled IaaS has various resource types to offer for application deployment. Different VM instance types have different cold-start delay depending on the size of the instance and resource such as memory. Moreover, our pricing model follows the AWS EC2 pricing model, where users are charged based on partial usage, *i.e.* on seconds basis as specified in [20]. Any instance can host a pre-defined number of requests based on the resources available in the instance and desired SLA. Hosting more requests on an instance can lead to performance degradation and potential SLA violations. The usage cost is calculated according to Equation (3).

Load Balancer: Production-ready applications typically use more than one VM. Incoming requests are distributed among them in a Round Robin fashion. If all the VM instances already have a pre-defined number of requests running, any subsequent request is queued and served as soon as any instance can accommodate it.

Autoscaler: Our modeled autoscaler works similarly to Amazon EC2 Auto Scaling [27] and allows users to define auto-scaling policies such as scale-in/scale-out thresholds, scaling groups, and minimum/maximum number of instances. Moreover, our autoscaler can use a threshold on metrics, such as average memory utilization or request count on each instance, to make scaling decisions.

2) FaaS: To model FaaS, we deployed various types of application function on Amazon Lambda and found the relationship between the configurable resources (*e.g.*, memory) and execution time. This follows an exponentially decay model as given by Equation (1). Other approaches (*e.g.*, [15], [22]) have reported similar execution patterns. We also use Amazon

[7]The LIBRA simulator and AWS code is available at [29].

Lambda's pricing model where, based on the configured resources and execution time, usage cost is calculated according to Equation (2).

3) Simulation Parameters: For our evaluation, we chose the "large" EC2 instance type *m4.large*, which has 8.0 GB of memory and 0.1 dollars per hour cost. We ran multiple instances of type *m4.large* and noted that the provisioning time (cold-start delay) to obtain an instance is about 100 seconds. Each request should have at least 512MB of memory to complete in one second (SLA) on both IaaS and FaaS [8]. For LIBRA parameters, we used 0.2 as the EWMA weight, 300 seconds as the scaling decision interval (which is $3\times$ the cold-start of the VM instance being used), VM utilization threshold ρ = 80%, and traffic estimation parameter $\phi = 1$.

B. Log Traces

We used WITS [33], Berkeley [34], and synthetic traces to evaluate the efficacy of LIBRA. These traces have also been used to evaluate similar approaches [12], [35]. We utilize a 12-hour long segment from the traces to generate the workload for our simulation. Each request is assumed to have a constant and equal service (execution) time when served on either the IaaS or FaaS based implementation. Our simulator takes into account the cold-start of VMs under IaaS. We also assume that the serverless functions under FaaS have a minimal cold-start delay that would not affect the performance of an application with relatively high popularity as shown in [11]. While LIBRA produced similar results for all traces, due to space limitation, we only present results for the WITS trace. Figure 3 shows the 12-hour snippet of a WITS trace of the number of requests per epoch (second), along with the EWMA. Traces were generated by counting the number of HTTP requests during every second in TCP dumps available at [33]. Despite the high variation of the demand, EWMA accurately tracks the dynamicity of the trace.

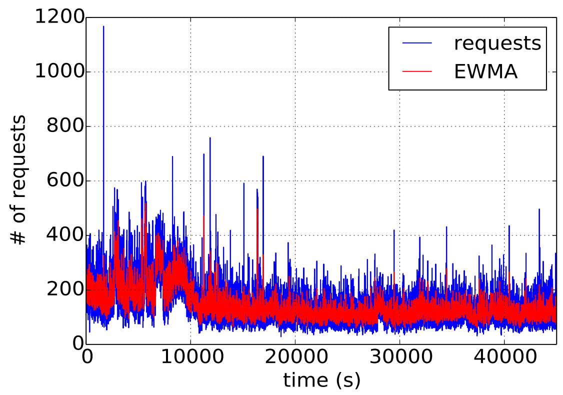


Fig. 3: WITS trace and EWMA

C. Resource Provisioning & Deployment Policies

As discussed in Section III, LIBRA's main goal is to utilize both FaaS and IaaS to minimize the overall cost while at the same time meeting the performance (execution time) requirement of the application. LIBRA leverages the best of both cloud services: the quick provisioning time of serverless functions and the low cost of provisioned VM resources. We compare LIBRA's balanced approach to the following policies:

1) Over-provisioning (MAX): Cloud applications have strict SLAs and not meeting their performance constraints can result in bad user experience and potentially loss of revenue. To avoid potential SLA violations, the tenant could opt to over-provision the VM resources. We simulate this scenario by scanning the whole demand trace and provisioning the VM resources based on the *maximum* number of requests received during a second (*i.e.*, the peak rate). While such approach would avoid SLA violations, allocated VM resources will be underutilized and the client would incur higher costs.

2) Autoscaling (AUTO): Autoscaling is a popular service provided by major cloud providers. In Autoscaling, the performance of the currently allocated resources (*e.g.*, VMs), is monitored based on some metric. The performance metrics can vary based on the cloud provider, but some examples include memory/CPU utilization or request/connection count on each host. If the metric exceeds a certain threshold, new resources are added to the system to avoid potential overloading of current resources and subsequent degradation of performance. If the metric falls below a certain threshold, resources are removed to avoid under-utilization.

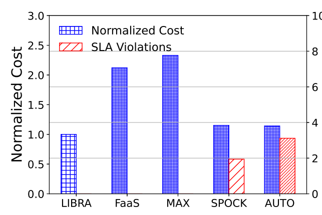
3) Spock: Previous approaches, such as Spock [12] and MArk [2] reduce SLA violations due to VM start-up delays by directing demand to serverless functions while VMs are being provisioned. Unlike LIBRA, Spock-like schemes do *not* consistently and simultaneously use serverless functions to serve transient demand and reduce overall cost. We simulated this by directing the *excess* portion of the demand to FaaS *during* scale-out events.

4) FaaS only: The application is deployed on a serverless platform and all requests are served by serverless functions.

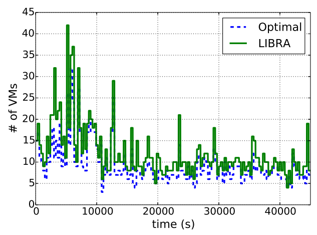
D. Simulation Results

1) Cost and SLA violations: Figure 4a compares the cost and performance of the aforementioned resource provisioning policies with LIBRA. The x-axis represents the different approaches described above, the y-axis (left) represents the incurred cost normalized to that of LIBRA, and the y-axis (right) reports the percentage of SLA violations. LIBRA's cost also includes the cost of the VM instance used to deploy the LIBRA Gateway (cf. Figure 2). As expected, over-provisioning (MAX) leads to zero SLA violations, but incurs the highest cost. The autoscaling approach (AUTO) reduces cost significantly but introduces significant SLA violations. This is due to the fact that VMs have high cold-start delays, and while a new VM is being set up to share the demand, existing VM instances get saturated, which leads to performance degradation and SLA violations. We note that we have experimented with various thresholds for the utilization metrics used by autoscaling. However, we have observed that either the system performs better at the expense of a much

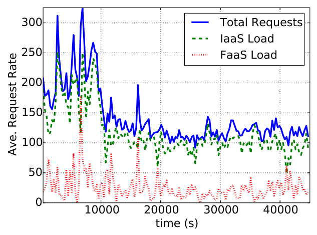
[8]For IaaS, a user can specify resources for each container executing the request as documented in [31], [32].



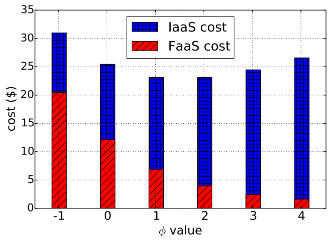
(a) LIBRA and other resource provisioning policies



(b) LIBRA's VM provisioning



(c) LIBRA's request distribution among available resources



(d) Cost for different values of ϕ

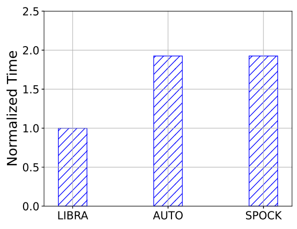
Fig. 4: Cost and performance of LIBRA in simulated cloud environment

higher cost or the system has lower cost at the expense of a much worse performance (higher SLA violations). Compared to autoscaling, Spock reduces SLA violations by more than 35% at about the same cost. Notice that consistent with the original study [12], Spock reduces SLA violations but does not completely eliminate them. LIBRA yields the lowest cost – 15% less cost than autoscaling/Spock and cuts the cost by more than half compared to serverless-only (FaaS) or over-provisioning (MAX). In our simulation, we assume that a serverless function has resources configured correctly so that each request always meets the SLA [15]. Thus, a serverless-only deployment yields zero SLA violations albeit at a higher cost. Similarly, LIBRA yields zero SLA violations. However, LIBRA reduces the overall cost by always directing a portion of the demand to VMs that are provisioned to meet the SLA, while the rest of the demand is directed to serverless functions that are also configured to meet the SLA.

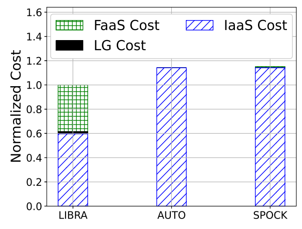
2) *VM Provisioning & Request Distribution:* LIBRA's main goal is to cautiously provision resources in the VM cloud (IaaS) to avoid under/over-utilization while simultaneously serving low-rate and sudden spikes in demand using serverless functions (FaaS). Figure 4b shows how LIBRA accurately tracks the incoming load, provisions VM resources, and avoids over-provisioning. This is observed by the similar behavior of LIBRA in terms of the number of VMs provisioned (green solid curve) throughout the duration of the simulation and the *ideal* (offline) case (blue dashed curve) of provisioning the number of VM instances assuming perfect knowledge of future demand. The points on both curves represent scaling decisions taken every $K = 300$ seconds (cf. Section III-D).

Figure 4c shows the average rate of requests for the portion of the demand forwarded to the VM instances (IaaS) and the rest of the demand directed to serverless functions (FaaS) every 300 seconds. We observe that a consistent majority of the load is served by VMs (IaaS) whereas a small amount of the load with temporary peaks is handled by serverless (FaaS).

3) *VM Uptime & Cost Breakdown:* In Figure 5a, we compare the total uptime of all VMs used to run the application under various approaches. LIBRA cuts the VM uptime by half compared to autoscaling and Spock. This is because (1) LIBRA only scales out when the demand persists for longer time. LIBRA is able to identify transient demand and avoid reacting to it by using serverless functions (FaaS) rather than adding new VM instances (IaaS); and (2) LIBRA can add any arbitrary number of VMs to the active VMs when scaling out, while autoscaling adds or removes a user-configured number of instances (referred to as "scaling group"). Figure 5b compares the cost breakdown across autoscaling, Spock, and LIBRA. Autoscaling has zero FaaS cost since VMs are the only resources used to serve the application demand. Spock employs FaaS when scaling out, only to hide VM startup delay, so the FaaS usage is really small ($\approx 1\%$). LIBRA consistently uses serverless functions to serve a portion of the demand. The FaaS cost contributes around 40% of the total cost in LIBRA's case. Despite higher FaaS cost, the overall cost of LIBRA is smallest. LIBRA intelligently uses FaaS for a portion of the demand that is either below CIP or transient, since the cost of new VM instances for that portion of the demand would have been higher. Note that the cost of LIBRA includes that of the LIBRA Gateway (LG), the added cost of running the LIBRA system.



(a) VM uptime



(b) Cost breakdown

Fig. 5: VM usage and cost breakdown

4) *Traffic Estimation:* Recall that LIBRA provisions IaaS resources for a request rate of $vm_{reqs} = (avg + \phi \cdot std)$. The actual request rate directed to the provisioned VMs is $\rho \cdot vm_{reqs}$ ($\rho < 1$), while the remaining requests (in each epoch) are directed to FaaS where they are served within the SLA but at a higher cost. The value of ϕ can be adapted based on the particular fluctuations in demand. Tuning ϕ affects the cost but not the performance of an application. This is because the SLA is met whether LIBRA directs the request to the *provisioned* VMs or to FaaS, however FaaS is more costly. For the WITS traces used in our evaluation, the effect of different values of ϕ on the cost is shown in Figure 4d. We observe that a lower value of ϕ leads to more FaaS usage and hence higher overall cost, whereas a higher value of ϕ causes over-provisioning of VMs, which leads to VM under-utilization and

hence higher cost. Here, $\phi = 1$ gives the least cost.

V. LIBRA on AWS

To validate our simulation results from Section IV, where LIBRA was shown to be effective in reducing both cloud-usage cost and SLA violations, we implemented LIBRA to perform load balancing for an application deployed on Amazon AWS Cloud, *i.e.* EC2 for IaaS and Lambda for FaaS.

A. Application

The application is an image manipulation application written in Python. On Amazon Lambda, we are able to deploy the application as a single function. While we expect similar results for multi-function applications, our choice of single-function application is inspired by many use cases such as ML inference models [2], [12], [22], IoT and computer vision applications [36] which can be usually deployed as a single function. To deploy the application on EC2 VM instances, we used a python multi-threaded HTTP server library. We ran it on a *t3.medium* [20] EC2 instance type with Ubuntu Server 18.04 LTS operating system, two 2.5 GHz vCPU, and 4 GB memory.

B. Application Profiling & Lambda Resources

As described in Section III-B, LIBRA's *Scaling Manager* uses the maximum number of requests, r_{max}, that a VM instance can handle, to calculate the required number of instances. To obtain r_{max} for this evaluation, we deployed our application on a *t3.medium* instance, profiled its performance, and obtained r_{max} that meets the SLA. We take the SLA to be one second of execution time serving a request. Profiling an application on a given VM instance is a one-time task and a developer can perform this prior to production deployment. For FaaS deployment, we invoked the function with various memory configurations and picked the memory setting that gave the least cost while meeting the SLA.

C. Setup & Implementation

To obtain a consistent network environment for our evaluation on AWS, we deployed an application client on an EC2 instance, which will generate HTTP requests for the application. We compare LIBRA to the same four resource provisioning and deployment strategies described in Section IV-C. We needed to modify the implementations of LIBRA, Autoscaling, and Spock, as follows:

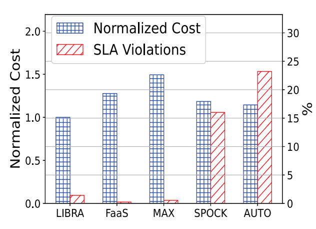
1) LIBRA on AWS: We deployed the LIBRA Gateway (LG) on an EC2 instance of the type *t2.micro*. The LG distributes requests between IaaS based resources (EC2 VMs) and FaaS (AWS Lambda functions). Within IaaS, we used the AWS Application Load Balancer (ALB) [37] to distribute requests among the active VM instances evenly, *i.e.* in a Round Robin fashion.

2) Autoscaling on AWS: We used EC2's autoscaling service, which is a threshold-based scaling service. The AWS CloudWatch Alarm was used to monitor *RequestCountPerTarget* metric to make scaling decisions. Again, ALB was used to distribute requests to the VMs.

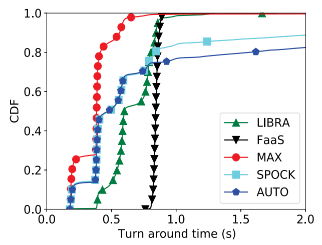
3) Spock on AWS: An alarm from AWS CloudWatch triggers the scale-out or scale-in events. When a scale-out event is triggered, new VM instances are provisioned. During this VM provisioning time, the system sends the extra requests, that cannot be served by the active VM instances, to the serverless functions. Once the new VM instances are ready, all requests are forwarded to the VMs. Thus, Spock attempts to use serverless functions (FaaS) only to hide VM startup delays.

D. Results & Discussion

For our AWS experiments, to reduce real load, we use a scaled-down version of the first 1800 seconds of the WITS trace shown in Figure 3. In particular, we reduce the rate of requests by a factor of 16.



(a) Cost and SLA violations for LIBRA vs. other policies



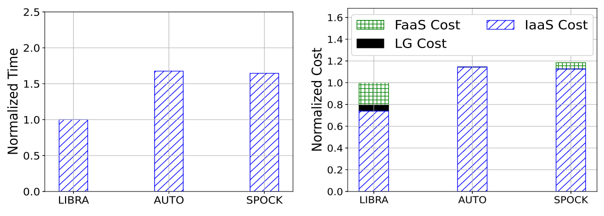
(b) Response time distribution

Fig. 6: Performance of LIBRA on AWS

1) Cost and SLA Violations: We compare the cost and performance of LIBRA versus other resource provisioning and deployment strategies. The results are consistent with our simulation results. Figure 6a shows that LIBRA yields the lowest cost with very low amount of SLA violations. LIBRA reduces the SLA violations (by more than 85%) and cost (up to 20%) when compared to auto-scaling and Spock. LIBRA's cost also includes the cost of deploying the LIBRA Gateway on an EC2 instance. Max-provisioning and serverless-only deployment yield the lowest SLA violations but incur up to 50% increase in cost. We observe that FaaS, LIBRA, and max-provisioning, all have a little amount of SLA violations. This is because unlike our simulation model, in a real setup, factors such as co-location, cold-starts for serverless functions, and underlying resource contention for VMs, can introduce slight variation in the performance of an application. For LIBRA, a lower value for the VM utilization parameter ρ (discussed in Section III-D3) can mitigate these SLA violations.

While Spock reduces SLA violations by 40% compared to autoscaling, about 15% of the requests fail to complete within the SLA. This can be explained by Spock's reactive scheme, where scaling out is triggered when VM resources are saturated, resulting in SLA violations. On the other hand, LIBRA avoids saturating the VM instances by directing excess load to serverless functions. If the load is not transient and the demand stays higher for a longer period, LIBRA increases the number of VM instances at the next scaling decision.

Figure 6b shows the CDF of completion time of each request. The over-provisioning policy gives the best performance in terms of keeping execution time really low, as expected. FaaS has the most consistent performance, *i.e.* the completion time is always between 0.8s to 1s. This is due to the fact that each request is executed in a dedicated sandbox environment with dedicated resources, such as memory, so the chance of performance fluctuation is lower. The performance of a serverless function is primarily affected by cold-start delay [22], which is negligible and can be as low as 10s of milliseconds for serverless functions written in Python [11].



(a) VM uptime (b) Cost breakdown

Fig. 7: VM usage and cost breakdown

2) VM Uptime & Cost Breakdown: Figure 7 confirms the benefit of LIBRA by illustrating the cost breakdown. This is consistent with our simulation results. Figure 7a shows the uptime of VM instances used to run the application for Spock, Autoscaling, and LIBRA. LIBRA is able to closely monitor the demand and provision required VM resources without over-provisioning, which results in lower overall VM uptimes and cost. Figure 7b shows the cost breakdown of these approaches. LIBRA has the lowest overall cost (including the cost to deploy LG), with lowest IaaS cost but highest FaaS cost. LIBRA uses serverless functions more consistently and effectively, *i.e.* for transient demand or portion below CIP, which results in higher usage of FaaS, and lower usage of IaaS.

E. Performance Overhead & LIBRA Scalability

LIBRA works as a legacy load balancer and directs requests to appropriate resources, introducing overhead no more than legacy load balancers. At the same time, other LIBRA operations, *i.e.* scaling and traffic monitoring, occur in the background and do not impact the real-time processing of requests. The LIBRA Gateway has a small computational footprint, and hence a small cost. Various components of the LIBRA Gateway can be implemented as serverless functions, where scalability is taken care of by the cloud provider. Alternatively, the LIBRA Gateway can be implemented as one service and deployed using VM instances and the application administrator can rely on the cloud provider's scaling services such as AWS auto-scaling.

VI. Related Work

Serverless computing has been extensively studied to deploy cloud applications, such as ML applications [2], [12], [38], video/data processing [3], [4], [39], IoT [40], scientific workflows [5], biomedical applications [41]–[43] and to solve various mathematical and optimization problems [44]–[46].

CherryPick [24] helps the developer find the best VM instance type using statistical learning techniques. Previous approaches use various ML and learning-based approaches to configure serverless functions [15], [47], [48] to optimize cost and performance. Costless [38] decomposes a serverless application across edge and core cloud to minimize cost and meet performance constraints. CloudCmp [49] performs comprehensive measurement studies on various commercial clouds so as to find a suitable cloud provider for a given application. Similarly, cluster management systems, such as Google Borg [50] and Mesos [51], orchestrate and allocate resources in the cloud for various applications. Google has developed a machine type recommendation system [52] that helps a user find the suitable instance type that maximizes resource utilization. Moreover, most of the commercial cloud providers provide autoscaling services (reactive in nature) to manage the virtual resources, *e.g.,* AWS's EC2 autoscaling [27] and Google autoscaling [53]. Similar to LIBRA, other approaches use various prediction models to predict the demand for an application and orchestrate cloud resources [54]–[58]. The aforementioned approaches only manage resources within one type of service, while LIBRA orchestrates resources across two different services, *i.e.*, an IaaS and a FaaS. Our analysis extends the cost analysis in [59] of using FaaS or IaaS for an application by introducing the dependency of execution time on the resources allocated.

Previous works have used serverless functions to hide VM startup delays for ML and other applications while scaling out VM-based resources [2], [12], [13]. In contrast, LIBRA uses serverless functions as an alternative cloud service to run applications, and based on the demand, can decide to use either one or both services to optimize *both* cost and performance.

VII. Conclusion and Future Work

We presented LIBRA, a load balancing approach that simultaneously uses both IaaS and FaaS cloud services to cater to the dynamic demand of applications. Our approach can be employed by cloud providers as a value-added service or used by end-users directly. Our evaluation of LIBRA in simulations and on AWS shows its clear advantage over other resource provisioning policies in reducing both cost and SLA violations.

In this paper, LIBRA utilizes IaaS and FaaS services available from one cloud provider in support of single-function applications. Future work includes extending LIBRA to accommodate applications that consist of chains of functions, leveraging spot instances, integrating with platforms such as Kubernetes, and utilizing services from multiple cloud providers.

Acknowledgement

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

References

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, 2019.

[2] C. Zhang, M. Yu, W. Wang, and F. Yan, "MArk: Exploiting Cloud Services for Cost-Effective, SLO-Aware Machine Learning Inference Serving," in *USENIX ATC*, Renton, WA, 2019.

[3] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *ACM SoCC*, 2018.

[4] M. Zhang, Y. Zhu, C. Zhang, and J. Liu, "Video Processing with Serverless Computing: A Measurement Study," in *ACM NOSSDAV*, Amherst, MA, 2019.

[5] 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*, 2017.

[6] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless Computing: A Survey of Opportunities, Challenges and Applications," *arXiv preprint arXiv:1911.01296*, 2019.

[7] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[8] M. Yan, P. Castro, P. Cheng, and V. Ishakian, "Building a chatbot with serverless computing," in *ACM 1st International Workshop on Mashups of Things and APIs*, 2016, p. 5.

[9] 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.

[10] J. Spillner, "Transformation of Python Applications into Function-as-a-Service Deployments," *arXiv preprint arXiv:1705.08169*, 2017.

[11] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking Behind the Curtains of Serverless Platforms," in *2018 USENIX ATC*, Boston, MA.

[12] J. R. Gunasekaran, P. Thinakaran, M. T. Kandemir, B. Urgaonkar, G. Kesidis, and C. Das, "Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud," in *IEEE CLOUD*, 2019.

[13] J. H. Novak, S. K. Kasera, and R. Stutsman, "Cloud Functions for Fast and Robust Resource Auto-Scaling," in *COMSNETS*, 2019.

[14] "AWS Lambda Pricing," https://aws.amazon.com/lambda /pricing/, 2020.

[15] N. Akhtar, A. Raza, V. Ishakian, and I. Matta, "COSE: Configuring Serverless Functions using Statistical Learning," in *IEEE INFOCOM*, 2020.

[16] "IBM Function Pricing," https://cloud.ibm.com/functions/learn /pricing, 2020.

[17] "Google Function Pricing," https://cloud.google.com/functions/ pricing, 2020.

[18] "Azure Function Pricing," https://azure.microsoft.com/en-us/pricing/details/functions/, 2021.

[19] "Microsoft Azure," https://azure.microsoft.com/en-us/services/virtual-machines/, 2020.

[20] "EC2 Pricing," https://aws.amazon.com/ec2/pricing/on-demand/, 2020.

[21] "IBM Cloud," https://www.ibm.com/cloud/pricing, 2021.

[22] V. Ishakian, V. Muthusamy, and A. Slominski, "Serving deep learning models in a serverless platform," in *IEEE IC2E*, 2018.

[23] A. Allen, *Probability, Statistics, and Queueing Theory*, ser. Computer Science and Scientific Computing. Elsevier Science, 1990.

[24] 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 *NSDI*, Boston, MA, 2017.

[25] S. A. Greenlaw, E. Dodge, C. Gamez, A. Jauregui, D. Keenan, D. MacDonald, A. Moledina, C. Richardson, D. Shapiro, and R. Sonenshine, *Principles of Economics 2e*, 2nd ed. Openstax, 2017.

[26] G. F. Franklin, D. J. Powell, and A. Emami-Naeini, *Feedback Control of Dynamic Systems*, 4th ed. USA: Prentice Hall PTR, 2001.

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

[28] C. Ayimba, P. Casari, and V. Mancuso, "Adaptive Resource Provisioning based on Application State," in *International Conference on Computing, Networking and Communications (ICNC)*, 2019, pp. 663–668.

[29] "LIBRA Code," https://github.com/aliraza0337/LIBRA, 2021.

[30] "Amazon Web Services," https://aws.amazon.com/, 2020.

[31] "Kubernetes per Request," https://kubernetes.io/docs/concepts/ configuration/manage-resources-containers/, 2021.

[32] "Docker Resources," https://docs.docker.com/config/containers/ resource_constraints/, 2021.

[33] "WITS Trace," https://wand.net.nz/wits/catalogue.php, 2020.

[34] "Berkeley Trace," https://www.comp.nus.edu.sg/∼cs5222/simulator /traces/berkeley/index.htm, 2021.

[35] "Spock Simulator," https://github.com/jashwantraj92/spock, 2020.

[36] "Amazon Rekognition Image," https://github.com/aws-samples/amazon-rekognition-image-for-box-skills, 2021.

[37] "AWS Load Balancer," https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html, 2019.

[38] T. Elgamal, "Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement," in *IEEE/ACM SEC, Bellevue, WA*, 2018.

[39] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *SOCC*, New York,, 2017.

[40] D. Pinto, J. P. Dias, and H. Sereno Ferreira, "Dynamic Allocation of Serverless Functions in IoT Environments," *IEEE EUC, Bucharest, Romania*, 2018.

[41] B. Lee, M. Timony, and P. Ruiz, "DNAvisualization.org: a serverless web tool for DNA sequence visualization," *Nucleic Acids Research*, vol. 47, 06 2019.

[42] D. Kumanov, L.-H. Hung, W. Lloyd, and K. Y. Yeung, "Serverless computing provides on-demand high performance computing for biomedical research," *arXiv preprint arXiv:1807.11659*, 2018.

[43] L.-H. Hung, D. Kumanov, X. Niu, W. Lloyd, and K. Y. Yeung, "Rapid RNA sequencing data analysis using serverless computing," *bioRxiv*, 2019.

[44] A. Aytekin and M. Johansson, "Harnessing the Power of Serverless Runtimes for Large-Scale Optimization," *arXiv preprint arXiv:1901.03161*, 2019.

[45] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless Linear Algebra," in *11th ACM SoCC*, Virtual Event, USA, 2020.

[46] S. Werner, J. Kuhlenkamp, M. Klems, J. Müller, and S. Tai, "Serverless Big Data Processing using Matrix Multiplication as Example," in *IEEE International Conference on Big Data (Big Data)*, 2018, pp. 358–365.

[47] L. Schuler, S. Jamil, and N. Kühl, "AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments," *arXiv preprint arXiv:2005.14410*, 2020.

[48] S. Eismann, L. Bui, J. Grohmann, C. L. Abad, N. Herbst, and S. Kounev, "Sizeless: Predicting the optimal size of serverless functions," *arXiv preprint arXiv:2010.15162*, 2020.

[49] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," in *ACM IMC*, Melbourne, Australia, 2010.

[50] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale Cluster Management at Google with Borg," in *EuroSys, Bordeaux, France*, 2015.

[51] 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 *NSDI, Boston, MA*, 2011.

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

[53] "Google Auto Scaling," https://cloud.google.com/compute/ docs/autoscaler, 2018.

[54] W. Fang, Z. Lu, J. Wu, and Z. Cao, "RPPS: A Novel Resource Prediction and Provisioning Scheme in Cloud Data Center," in *IEEE SCC, Honolulu, HI*, 2012.

[55] N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting," in *IEEE CLOUD, Washington, USA*, 2011.

[56] L. Aniello, S. Bonomi, F. Lombardi, A. Zelli, and R. Baldoni, "An architecture for automatic scaling of replicated services," in *NETYS*, 2014.

[57] A. Y. Nikravesh, S. A. Ajila, and C. Lung, "Towards an Autonomic Auto-scaling Prediction System for Cloud Resource Provisioning," in *IEEE/ACM SEAMS, Firenze, Italy*, 2015.

[58] A. H. Mahmud, Y. He, and S. Ren, "BATS: Budget-Constrained Autoscaling for Cloud Performance Optimization," in *IEEE MASCOTS*, Atlanta, GA, 2015.

[59] "Economic of Serverless," https://www.bbva.com/en/economics-of-serverless/, 2021.