2019 IEEE 12th International Conference on Cloud Computing (CLOUD)

Spock: Exploiting Serverless Functions for SLO and Cost Aware Resource Procurement in Public Cloud

Jashwant Raj Gunasekaran, Prashanth Thinakaran,
Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, Chita R. Das
Computer Science and Engineering, Pennsylvania State University, University Park, PA
{*jashwant, prashanth, kandemir, bhuvan, kesidis, das*}*@cse.psu.edu,*

***Abstract*—We are witnessing the emergence of elastic web services which are hosted in public cloud infrastructures. For reasons of cost-effectiveness, it is crucial for the elasticity of these web services to match the dynamically-evolving user demand. Traditional approaches employ clusters of virtual machines (VMs) to dynamically scale resources based on application demand. However, they still face challenges such as higher cost due to over-provisioning or incur service level objective (SLO) violations due to under-provisioning. Motivated by this observation, we propose *Spock*, a new scalable and elastic control system that exploits both VMs and serverless functions to reduce cost and ensure SLO for elastic web services. We show that under two different scaling policies, *Spock* reduces SLO violations of queries by up to 74% when compared to VM-based resource procurement schemes. Further, *Spock* yields significant cost savings, by up to 33% compared to traditional approaches which use only VMs.**

***Keywords*-serverless; FaaS; cost-aware; SLO; autoscaling;**

I. INTRODUCTION

The advent of public clouds in the last decade has led to the proliferation of web services due to economies of scale. These web services typically range from simple mailing service to multi-tier data analytics. Further, the popularity of deep learning based models and their multi-faceted application to different domains has lured application developers to create, train and host these models as a web service. These machine learning (ML) web services such as voice, text, video and image recognition allow [1, 2] end-users to submit queries via web server interface.

ML inference services are pre-trained models hosted as a web service. These inference requests are stateless and are often user-facing. Hence, the service is administered under a strict SLO with tight response times, typically under 500-1000ms [3]. Thus, high availability in terms of the resource is quintessential [4]–[6], which has to be ensured by the scaling policy.

Today, resources are typically procured and scaled automatically, in units of VMs [7]. Since new VMs may take up to a few to several minutes to start [8], these auto-scaling solutions are susceptible to being wasteful due to over-provisioning (provision more VMs to reduce future SLO violations) or suffer from poor performance due to under-provisioning (provision lesser VMs but incur SLO violations) [9]–[11]. These problems become particularly prominent and difficult to address effectively during periods of poor workload predictability (e.g., flash crowds) [12] when VM addition/removal may only be done reactively.

In this paper, we investigate the potential of recently emergent "serverless computing" [13] offerings from public cloud providers to overcome this shortcoming of VM-based auto-scaling. This is an appealing idea because serverless products tend to have much lower start-up latencies than VMs as they use containers to host application requests. Specifically, we look at Function as a Service (FaaS) offerings [14]–[17] that allow a tenant to simply provide code for functions that would be executed in response to specified events within a cloud provider-managed container runtime. Functions are charged on a per-invocation basis without having to pay for over-provisioned resources as in the case of VMs. To a limited extent, the provider may take charge of auto-scaling the resources allocated to an executing function. This may relieve the tenant of the complexity of designing an auto-scaling technique. More generally, the claimed benefit of FaaS offerings over infrastructure-as-a-service (IaaS) offerings, such as VMs, is that they relieve the tenants of the need to manage the runtime [18] and the tenants pay only for used resources.

In this paper, we explore *how FaaS offerings can alleviate the shortcomings of VM-based auto-scaling for ML inference services, while at the same time minimizing the provisioning costs.* We make an interesting observation that deploying the entire application as serverless functions is not cost efficient in several scenarios discussed in the paper. To this end, we design *Spock*, which uses serverless functions prudently along with the existing VM-based *auto-scaling* mechanism; thereby reducing the SLO violations without increasing the cost of deployment. In the context of this paper, we focus specifically on ML based web services but our proposed idea can also be extended to any stateless web-service which exhibits workload dynamism.

Towards this, we make the following contributions:

1) We deploy an elastic ML inference service using serverless functions and characterize their cost benefits against a VM-based deployment for varying user demands.
2) We present *Spock*, the first elastic control system which takes advantage of serverless functions along with VMs to reduce SLO violations by peak shaving the requests.
3) We perform a comprehensive analysis of *Spock* compared to two resource procurement schemes combined with two different auto scaling policies to reduce their cost in terms of cloud bill and reduce SLO violations.
4) We show that *Spock* reduces SLO violations of ML

2159-6190/19/$31.00 ©2019 IEEE
DOI 10.1109/CLOUD.2019.00043

inference queries by 65%-74%, when compared to using only VMs under two scaling policies. Further, *Spock* yields significant cost savings by up to 33% compared to traditional resource procurement schemes.

II. BACKGROUND AND MOTIVATION

In this section, we provide background about the relevant resource offerings in public cloud which are used to host an ML based service. Next, we describe the key ideas motivating the need for exploiting different service offerings to optimize for overall costs[1].

A. ML Inference Service

ML-based service [19] is a popular class of web services that lets clients support end-to-end training of ML models and it enables users to issue inference queries (as shown in Figure 1). There are several applications that can make use of such inference models in an elastic fashion. Generally, these queries are response time sensitive with desired latencies in the 100 millisec - 1 sec range [3]. A client, in this case, has several options in resource procurement to host the service.

The most popular approach currently for an ML service provider is to use VMs to run inference tasks and employ well-known auto-scaling techniques to match VM capacity to resource needs. On the contrary, hosting is also possible using serverless functions (like lambda [2]) [20] where clients need not perform explicit resource management unlike the case of VMs. There are other commercial offerings from cloud providers like Azure MLaaS [21], and AWS Sagemaker [22], to name a few, that have complex billing by making users pay for compute time and underlying resources (VMs) to host the service. In this paper, we limit our discussion to VMs and lambdas. We further discuss how the deployment can scale for both VMs and lambdas in detail.

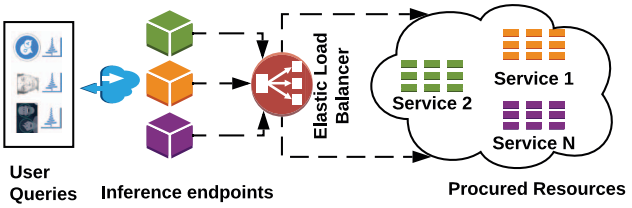


Figure 1: System architecture for an ML web service application that handles user queries for different inference models. The query is received by inference endpoint and forwarded to dedicated back-ends that execute the inference and sends the output to the user.

B. A Representative VM Autoscaler

The service described above is typically hosted as VMs, where the resource manager anticipates the resource demands and provisions the VMs accordingly. To meet this changing demand, the resource manager allocates from a static pool of VMs. Public Cloud Providers (PCPs), like

[1]From here on-wards we refer to costs in terms of cloud bill.
[2]We use the terms "serverless functions" and "lambda" interchangeably.

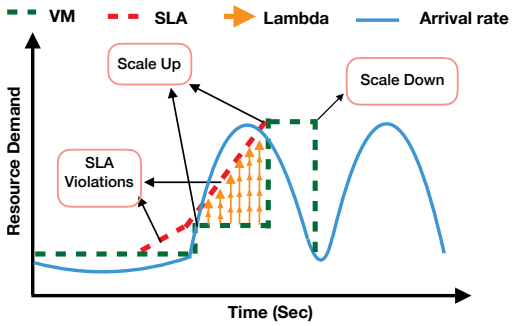


Figure 2: A snapshot of peak shaving by using lambda functions along with VM-based *autoscaling*. The plot shows the resource demands over time while using the Lambda functions to bridge the lapse of VM acquisition and cold-start latency, thus reducing SLO violations during the VM scale-out.

Amazon, provide mechanisms with resource auto-scaling enabled such as AWS *autoscaling* [23]. During the setup for *VM* instances, the user specifies an instance type and availability zone and sets a scaling target in terms of different metrics (e.g., min/max CPU utilization). If the scaling target is reached, *autoscaling* acquires or releases instances to reach the new target. We consider a representative *autoscaler* as our "baseline", which is similar to AWS *autoscaling* [23]. Our *autoscaler* will *reactively* scale resources every second, if the request demand cannot be met. As seen in Figure 2, as the resource demand surges (shown in blue), based on the *autoscaler*, VMs are spawned to react to the demand (depicted in green). However, VMs have a start-up latency on the order of hundreds of seconds [24]–[26], there is an intermittent lapse of availability of the service. This gap is bridged by the use of lambdas (depicted in orange), which typically have start-up latencies that are an order of magnitude lesser than VMs. We explain this further in the next section. As the resource demand drops, the VMs are scaled down to avoid resource over-provisioning leading to cost savings.

C. Serverless Functions

Serverless function-based models have shown several advantages over traditional VMs due to their minimal provisioning overheads. Specifically, the fundamental advantages of serverless functions over VMs are as follows: (i) cost efficiency at scale due to event-driven billing where the client only pays for the compute time they consume and (ii) their ability of scaling almost instantly and automatically. This is possible because the back-end framework uses containers to host the function invocations on a ready VM, thus having minimal start-up latency compared to traditional VM with boot times. Serverless functions are billed based on the memory footprint [27] and function invocation count which is based on query arrival frequency. However, in terms of cost, the VMs are also billed on a per-second basis similar to lambda [28]. Therefore, the real advantage of a lambda function comes due to the reduced start-up latency. Note

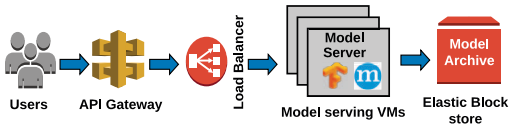
that, serverless functions have two different types of start-up latency, (i) coldstart: which is the process of launching a new function instance by spawning a new container and (ii) warmstart: which is reusing an existing containers. Though, container coldstart latency are significantly longer than warmstart (average of 1-10s) [29], they are still a magnitude lesser compared to VMs.

The properties of serverless function discussed above benefit certain classes of applications better than others. Specifically those with event-driven behavior, composed of stateless, short running and agile queries. For example, a face/object detection function can be triggered by the occurrence (arrival) of an event such as an image from the client, following which a new lambda function can be spawned to execute inference and send the result back to the user. Based on the changes in resource demand, application providers need not define *autoscaling* policies as these functions can scale "automatically". In this work, we try to exploit this property which can be used alongside VM *autoscaling*. As shown in Figure 2, the sudden surge of resource demand is intermittently handled by the lambda functions until the new VMs are up to meet the demand to handle the requests. This results in two benefits: (i) reduction in SLO violations during a request surge, and (ii) reduce intermittent over-provisioning of VMs.

III. Characterizing Cost for *VM* and *Serverless*

In order to understand the cost of deployment at scale for an ML web service (as shown in Figure 1), we characterize the service on both VMs and serverless functions. Our experiments use the AWS Lambda [30], however we believe our approach is also applicable to other serverless-based offerings. Also, we do not consider spot VM instances, in order to ensure availability and avoid potential overheads from VM preemption and request migration.

(a) Serverless based deployment.

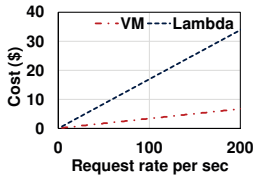


(b) VM based deployment.

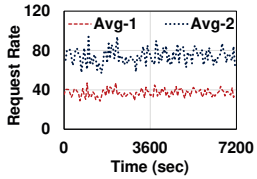
Figure 3: Framework to deploy an ML based web-service using serverless functions (lambda) and using VMs.

A. Serverless-Based Deployment

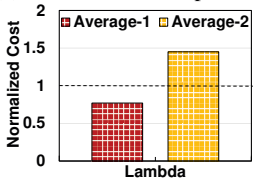
Figure 3a shows the overall framework to deploy ML inference using lambda functions. Users submit a query (image or voice) to a front end (inference end point). The input is sent to an elastic storage (e.g. S3 bucket) [31], which in turn triggers an event and invokes the corresponding



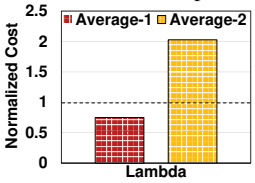
(a) Constant Arrival pattern.



(b) Variable Arrival pattern.



(c) Caffenet model.



(d) ResNet-200 model.

Figure 4: (a) Cost of VM and lambda for a constant request rate executing a single inference type; (b) two different average request rate for a constant peak request rate (140 req/sec); cost of executing (c) Caffenet and (d) ResNet-200 in lambda for the two arrival rates shown in (b), normalized to the cost of executing them in VM.

Query Type	Memory Required (GB)	Memory Allocated (GB)	Average Execution (ms)	Requests per vCPU for VMs
Caffenet	1.024	3.072	300	4
Googlenet	0.456	2.048	450	3
Squeezenet	0.154	2.048	130	6
Resnet-18	0.304	3.072	320	3
Resnet-200	1.024	3.072	956	1
Resnext-50	0.645	3.072	560	2

Table I: Query description for VM and lambda functions. The 2nd column is actual memory consumed by every model. 3rd column shows the memory allocated for lambda functions. 5th column shows the requests per vCPU for VMs, executing in parallel.

lambda function. The lambda function fetches the pre-trained model from an existing S3 bucket, runs the inference, and pushes the output to the user. Every input is registered as an event, thus leading to a separate function invocation, and each lambda independently processes ML inference requests. Note that, there is an initial read latency to fetch the ML model from S3 (average of 5s) which is significantly reduced during further invocations for the same model.

B. VM-Based Deployment

Figure 3b shows the overall framework for a VM-based deployment to host ML inference service. A cluster of EC2 instances are deployed as a model serving cluster. The pre-trained model is fetched into the EC2 instance using an EBS (Elastic Block Storage) volume. Note that, depending on the size of an instance, the number of parallel inference executions per instance can be different. We characterize the degree of parallelism for 6 different ML models used for image inference in terms of virtual cpus (vCPUs) as shown in Table I. We use a c4 large instance (2vCPU and 4GB memory) which has enough memory to accommodate the largest ML model used in our experiments.

C. Comparing Cost of Deployment

The overall cost difference between the VM-based or lambda-based deployment depends upon how frequent the

resources are used. In order to model the time-varying nature of the resource demands, we designed a load generator which is discussed in Section V). This load generator is modeled to mimic the real-world datacenter's event arrival rate. The resource demands, in terms of memory, varies for each query as they are associated with different ML models, as shown in Table I. We build a cost analysis model for each ML inference individually for VMs and lambdas. The cost of for VMs is straightforward as VMs are billed by the seconds of usage (with a one minute minimum) based on the instance type used (vCPUs and memory capacity). But for lambda functions, the cost depends on the following: (i) the number of times a lambda function is invoked (N) (ii) the execution time of every lambda function. (E in seconds), and (iii) the memory allocated to the function by the application (M in GB). Based on the above metrics the cost of a lambda (C) is calculated using Equation 1.

$$C = (M * E * 1.67 * e^{-5} * N) + (N * 2.4 * e^{-7}) \quad (1)$$

The constants above are modeled after Amazon Lambda function's pricing [27]. For our proposed scheme,to enable a fair comparison, we consider memory allocation (as shown in Table I) for an *iso-performance* case where execution time is same for both VM and lambda based deployments.

We calculate a per-unit cost for one inference query on both VMs and lambdas for a constant memory over a period of 60 minutes. We observe that, the cost of executing an inference for a single image using ResNet-200 which consumes 1GB memory would be $0.172 using lambda but $0.0519 on VM. It is observed that lambda functions are more expensive compared to VMs on per unit basis. However, for web services the requests are periodic with sustained arrival rates.

Figure 4a shows the cost for deploying a Caffenet inference model [32] either using VM or lambda over the fixed period of 2 hours for increasing arrival rates. It can be clearly seen that the per unit cost of memory is still higher for lambda compared to VM over a fixed load. Given this model, it can be seen that the cost increases linearly and lambda is always expensive for such fixed load. However, for a time varying workload, the queries do not follow a constant arrival rate.

We model the load generator after the real-world traces described in section V. As shown in Figure 4b, queries arrive over a 2-hour period for varying arrival rates. The maximum arrival rate is 140 requests/sec while the moving average is plotted for every 30 second interval. The Figures 4c and 4d show the cost of execution using lambda normalized to VMs. It is seen that the lambdas functions are expensive in the case of higher average request rate. On the other hand, if the average request rate is lower, then the lambda functions are cheaper compared to VMs as they had to be provisioned for the peak rate (to satisfy 100% SLO) for the entire duration. Therefore, based on the gap between average to peak arrival rate, either VM or lambda can be cheaper.

In case of varying resource demands, it is not feasible to provision the VMs for the peak demands. At the same time, the peak request rate cannot be predicted during a lot of scenarios like a flash-crowd [33, 34]. Further, during any mis-predictions, VM start-up latency (as described in Section II) would further lead to severe SLO violations [12, 35]. Lambda functions, on the other hand, can start up much faster compared to VMs and bridge the gap in such cases. Thus, we summarize our findings as follows:

1) It is non-trivial to predict the peak request rate at any given time period.
2) Provisioning VMs for the peak demands would always lead to higher cost of deployment. While, under provisioning VMs leads to severe SLO violations for queries.
3) Using serverless functions exclusively to deploy the web service would overcome the SLO violation problem. However, it is not cost effective.

Hence, there is an inherent need for a "scalable elastic control system" which exploits lambda to be used along with existing VM-based *autoscaling* mechanism, especially in case of request surges (peaks). We propose *Spock*, which performs SLO and cost aware resource procurement, while ensuring the SLO for stateless elastic web services.

IV. Spock Design and Implementation

We describe the overall design of *Spock* in Figure 5 along with the details of individual components and the scaling policies used.

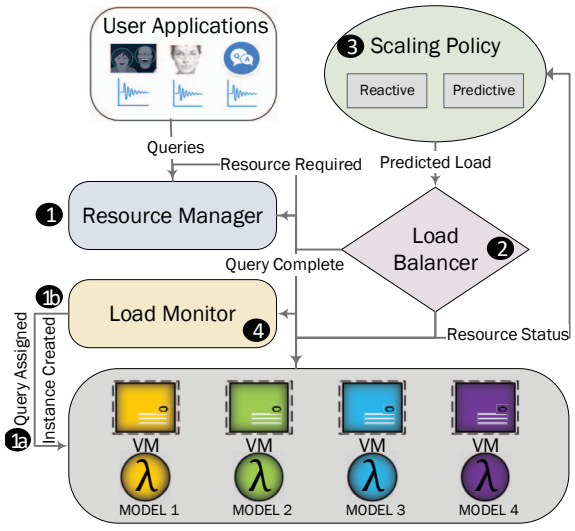


Figure 5: A high level view of *Spock*.

A. Spock Design Components.

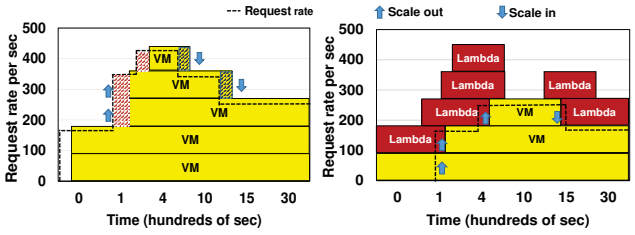
1) Resource Manager: The resource manager (RM) ❶ is the primary entity which handles query assignment ❶a, instance (resource) creation ❶b and termination. As discussed

in the previous section, we use AWS EC2 VMs and AWS lambda functions as our two main resource types. Based on the input from the load balancer (LB) ❷ which determines the resources required, RM initiates the procurement of additional instances. For each individual query type based on the request, it spawns the VM instances or lambda functions needed to meet the demand. During a scale-in decision from the LB, it terminates VM instances, whereas for lambda functions the scale-in is automatic.

2) Load Balancer: Load balancer ❷ uses the predicted load from the scaling policy to decide the resources required to be active over each monitoring period. It decides the scale-in or scale-out of resources while balancing the load between VMs and lambdas.

3) Scaling Policy: The scaling policy ❸ at each time determines the amount of resources expected to be active to serve the incoming requests. Since the load monitor provides resource usage information, users can define any scaling policy and plug it into the framework. The scale-out policy for both VMs and lambda are discussed in Section IV-B.

4) Load Monitor: Load monitor ❹ governs the active VM and lambda resources. The scaling policy initiates requests to the load monitor to query for metrics such as request served over given time, average utilization of VMs, etc. The load monitor queries individual VMs for required metrics in each monitoring period and performs bookkeeping for future policy decisions.



(a) Only VM based allocation. (b) *Spock* resource allocation.

Figure 6: Resource provisioning of *Spock* compared to only employing VMs. The red shaded region indicates SLO violations due to start-up of VMs. The blue shaded lines indicate temporary over-provisioning of VMs.

B. Spock Resource Procurement Scheme

The ML web service provider sets the SLO in terms of seconds. Based on how strict or loose the SLO is, the resource manager procures the resources (either VM or lambda). A group of queries (ML inference) are submitted to the cluster every second, requesting any of the ML models shown in Table I. *Spock* uses dedicated instance pools to serve individual query type as seen from the Figure 5. It is essential to have dedicated resource pools for each ML service model. This is because: (a) it simplifies the VM load balancing policy and (ii) it also reduces the VM startup time overheads. Since, loading all the models in one VM type would lead to start-up delays, in turn, leading to unfair comparison against model-specific lambda functions. When a request is submitted, the RM first tries to schedule it on any available VM which has free slots (we refer to free slots as free vCPUs). If a free VM is not available, it redirects the request to be run in a new lambda function. We discuss two different scale-out policies that are leveraged by *Spock* to procure resources.

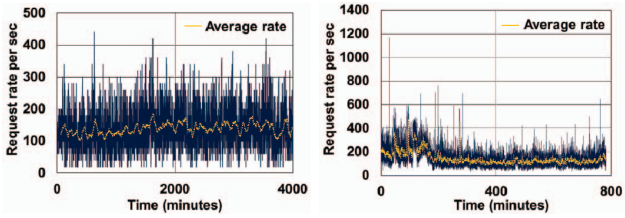
1) Reactive Scale-out: Whenever the RM cannot find a free slot in a VM to server a new request, it redirects the request to execute in a lambda function. Then, based on the request rate, the scaling policy decides to scale the number of VM instances required to satisfy the current load. Until the required number of VMs instances are spawned, any query which cannot be accommodated on VM is redirected to lambda functions. This is a straightforward way to reduce the SLO violation of queries due to VM start-up latencies. Similarly, for a scheduling decision which happens every second, the scaling policy *reactively* scale-out the number of VM instances. It is to be noted however that, this policy is susceptible to two vulnerabilities: (i) for time-varying client workloads the demand at time t1 need not be the same for time t2, and (ii) the interval between t1 and t2 could be potentially large or small depending on the characteristics of the workload. Hence, *reactively* scaling the VM resources at every time step is not a feasible solution, and may lead to intermittent over provisioning of VMs. This, in turn, results in higher cost of resource procurement. We discuss in detail about this scheme in Section VI.

2) Predictive Scale-out: As discussed in Section III it is non-trivial to predict the peak arrival rate of requests at a given time t. To overcome this problem, we design a predictor which can predict the average request rate at any time t for the next t + $t_{startup}$ time, where $t_{startup}$ is the start-up latency of VM. We use a moving window linear regression (LR) model to predict requests t+$t_{startup}$ into the future. The window size is set to 500s based on empirical analysis [3]. This enables us to provision VMs in advance to serve the average request rate while the short lived request surges can be served using lambda function. The LR predictor accurately predicts the average request rate while minimizing the overheads of the prediction policy.

From Figure 6a, we can see that VM based allocation would lead to severe SLO violations during a request surge (red shaded region). Due to the delayed scale-in of VMs, the pending queries are still being served (blue shaded region) during the scale-out phase. On the other hand, Figure 6b shows that, *Spock* efficiently executes requests in lambda during a request surge and thus avoiding the spawning additional VMs. *Spock* spawns additional VMs only based on the output from the predictor. Thus, predictive policy accurately captures the average resource demands for a given time period and in case of resource demand surges, it handles the queries through lambda functions. This reduces the SLO violations of queries while avoiding VM over provisioning.

3) Scale-in: Subsequent to scaling-out resources, it is also important to terminate idle instances to reduce cost of deployment. Using the load monitor, *Spock* constantly monitors the load of hosted VM resources for a given time. The LB decides to scale in the VM resources after three minutes of idle usage (as proposed by Gandhi et.al. [36]) in order to prevent early termination of instances in case of short term request rate fluctuations.

4) Runtime Packing: The RM decides to schedule requests on existing VMs, only if they have free resources to accommodate. The primary objective of RM should be to minimize the cost in terms of total VM active time. As we know, the total number of queries that each vCPU can handle in parallel (shown in Table I), we order the VMs by least available free vCPU (as opposed to most available vCPUs). This is because the busy VMs would remain active and it potentially enables underutilized VMs to become idle which can be terminated eventually. The potential drawback is that, more VMs would be terminated, and during a request surge, they might need to be re-spawned. Since we inherently use VMs to serve the average request rates, the peaks are efficiently handled using lambda functions.



(a) Berkeley trace. (b) WITS trace.

Figure 7: Traces used by the load generator. A moving window average is denoted by the yellow line. We use first 4000 minutes of Berkeley trace and 800 minutes of Wits trace.

V. IMPLEMENTATION METHODOLOGY

1) Simulator: We built a high fidelity event-driven simulator to evaluate the benefits of *Spock*. The simulator takes input from the load generator (explained further in Section V-2), which uses real-world trace for request arrival time generation. Each request is derived from a pool of pre-trained ML inference models for image classification (shown in Table I). We use *mxnet* [37] ML framework to deploy and run inference on the models. Note that, each of these ML models consumes different amount of memory, depending on the underlying model architecture. These model have been studied in the past in detail. and are trained in our study using the *imagenet* dataset [38]. The simulator accounts for VM start-up latency (60s to 100s) as observed on AWS [8, 39]. We limit our experiments to use instances from the same EC2 [28] family (c4-large, c4-xlarge, c4-2xlarge) to enable fairness. For lambda functions, we account for coldstart latency for the first invocations, but, during further invocations, we mitigate coldstarts by enabling warmstart through injecting a lambda function every 20 minutes, equivalent to the peak arrival rate in the past 500s window. This is also based on the observation made by Wang et al., [29] that lambda functions are kept warm for at least 20 minutes from the start time and the median cold start latency in AWS is within 200ms. We also account for the additional invocations in our cost calculation. Primarily, we limit our instance allocation region to US-east-1 only. This avoids unnecessary costs resulting from data transfer across regions.

2) Load Generator: We use traces from Berkeley [40] and WITS [41], which are given as input to the load generator. WITS trace has a large variation in peaks compared to the Berkeley trace. The requests arrive every second over the entire trace duration (show in Figure 7). All traces are scaled to have an average of 130 requests per second in order to generate sufficient load for the experiments. We associate each request with an image to run ML inference, which is selected randomly from the pool of ML models. The images are picked from a subset of *imagenet* data set. In this way, we mimic a real-world scenario where the web service engine receives different requests for different models every second. We consider two different workload mixes by inter-mixing the different models based on varying memory requirements.

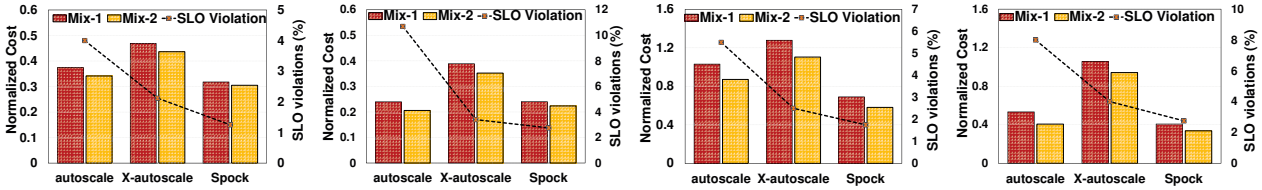
3) Assumptions: We set the SLO for every request to a maximum of 1s. This includes the end-to-end request latency (including model fetch time from S3 for lambda). Apart from the compute and invocation costs associated with lambda and VMs, there are also additional storage costs. We incur storage cost for EBS (elastic block store) in VMs and read/write request cost for remote store used along with lambda. From our initial characterization (Section III), these costs were similar for both VMs and lambdas. Hence, they do not affect our cost calculation. Each VM contains 2 to 8 vCPUs, and it can service requests proportionate to the number of requests each vCPU can handle in parallel (shown in Table I). Also each VM can queue any number of requests.

VI. EVALUATION AND RESULTS

We evaluate our results by comparing the cost and SLO of *Spock* for two workload mixes against the following resource procurement schemes: (i) deploying the web service application via lambda; (ii) deploying the same via VMs with *autoscaling* (as defined in Section II). We name this scheme as *autoscale*; (iii) using conservative over-provisioning with *autoscaling* i.e., acquire 1.5 times more resources than required. This is commonly used in *autoscaling* to reduce SLO violations. We name the last scheme X-autoscale. With respect to these resource acquisition schemes, we perform evaluations using two scale-out policies (discussed in Section IV) namely (i) *reactive* and (ii) *predictive*. The simulations were run over 5 iterations for consistency.

A. Results Analysis

Figures 8a and 8b, shows the cost savings and reduction in SLO violations for the Berkeley trace across two workload



(a) Berkeley under Reactive. (b) Berkeley under Predictive. (c) Wits under Reactive. (d) Wits under Predictive.

Figure 8: Cost savings and percentage of SLO violations for Berkeley and Wits trace. The cost is normalized to the cost of execution using only lambda. The resource procurement schemes are shown in-y axis.

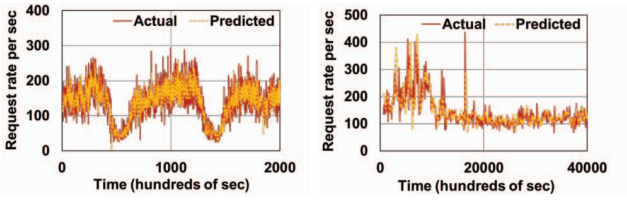
mixes. The cost is normalized to the resource procurement scheme that only uses lambda. We report both the cost savings and SLO as an average of the two workload mixes. The results show that *Spock* reduces the cost and SLO violations compared to *autoscale* and X-autoscale under both the scaling policies for both the workload mixes. For the Berkeley trace, *Spock*, when compared to *autoscale*, reduces SLO violations by 68% and 74% for both the reactive and predictive scale-out policies, respectively. At the same time, *Spock* also reduces cost by 15%, when employing the reactive policy. The cost savings are not significant for the predictive policy because it inherently avoids VM over-provisioning; but this leads to more SLO violations.

Similarly, when compared to X-autoscale, *Spock* reduces SLO violations by 41% and 21% for the two scale-out policies, respectively. The reduction is lesser when compared to the reactive policy, mainly because the additional resources provided can handle the requests during peak surge. However, these additional VMs come at a higher cost, which results in more cost savings when employing *Spock* (by 33% and 36%, respectively, for the two scaling policies).

Figures 8c and 8d, show the cost savings and reduction in SLO violations for the WITS trace. *Spock*, when compared to *autoscale*, reduces the SLO violations by 68% and 65% and reduces the cost by 33% and 18% for the two scale-out policies. The cost savings are higher when compared to the Berkeley trace because the WITS trace exhibits a large variation in peak to median ratio of request rates. Recall from Section IV that reactive *autoscaling leads to intermittent over-provisioning of VMs, which in turn results in higher cost. Similar to Berkeley, the cost savings is higher (up to 61%) when compared to X-Autoscale; though the reduction in SLO violations is low. In general,* Spock with predictive scale-out is the best with the least cost and minimal SLO violations for both traces tested. This is because, as seen in Figure 9, the predictive scheme can closely predict the average rate in the 500s window. Due to this, requests arriving in the peak are efficiently handled by lambdas (a.k.a. peak shaving).

B. Breakdown of Benefits

Intuitively, compared to *autoscale*, *Spock* is expected to be more expensive because it uses lambda along with VMs during a scale-out phase. However, in the *autoscale* policy,



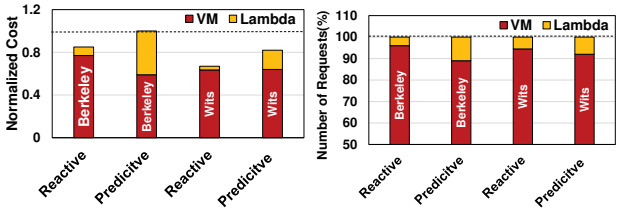
(a) Berkeley trace. (b) WITS trace.

Figure 9: LR based prediction of average request rates.

during the scale-out, requests get queued on existing VMs until new VMs are spawned. This results in VMs staying active for a longer duration. Since *Spock* uses lambda during scale-out, it helps in reducing the queuing of queries to existing VMs. This in turn enables early scale-in of unused VMs spawned during the scale-out period.

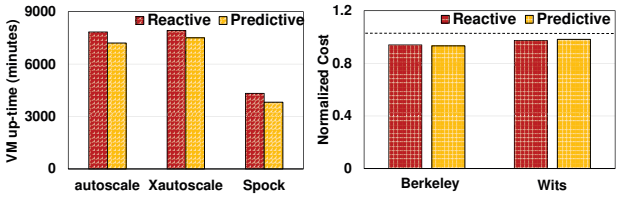
Figure 10a shows the breakdown of cost for both VM and lambda using *autoscale* policy. The total cost is *normalized* to the cost of executing using only VMs. It can be seen that *Spock* reduces cost of VMs by 23% and 40% for the Berkeley trace under the two scaling policies, respectively. However, the overall savings is only 15% for reactive scaling and nearly no savings for the predictive scaling. This is because there is an additional 15% and 41% cost added by lambdas. The cost added by lambda is higher for the predictive scale-out because the Berkeley trace exhibits more frequent peaks, and thus results is less cost savings. In contrast, for the WITS trace, the cost added by lambda is significantly lower because peaks occur infrequently. However, the overall cost savings is higher (by 20%) because the peak to average ratio is high.

Figure 10c shows the total VM active time for *autoscale*, X-autoscale and *Spock* under the two scaling policies. It is evident that *Spock* significantly reduces the total number of active VMs for all scenarios which is due to early scale-in of VMs. Figure 10b shows the percentage of total requests handled in VM and lambda for both traces. The number of requests in lambda is more for the predictive scale-out when compared to the reactive scale-out. This is because the predictive scale-out can fairly predict the average request rate in the arrival distribution. This is evident from Figure 9. Thus, VMs efficiently handle all requests in the average arrival rate, whereas request that arrive in sudden bursts are offloaded to lambda. On the contrary, since more VMs are spawned in the reactive scale-out scheme, the number of



(a) Cost breakup of VM and lambda normalized to *autoscale*

(b) Request partitioned across VM and lambda for *Spock*.



(c) Total VM active time for all three schemes.

(d) Cost of ideal VM scale-out normalized to Spock.

Figure 10: Breakdown of benefits from *Spock*.

requests offloaded to lambda is less.

In the case of workloads, where there is no sudden request surge, and peaks are fairly predictable, *Spock* would still work but the benefits in terms of SLO violations and cost will be subdued. We also compare *Spock* to an "ideal" scheme. The ideal scheme assumes the VMs to have start-up latencies similar to lambda functions. With this assumption, the ideal scheme will not have as many SLO violations compared to traditional VMs and will also avoid over-provisioning. Figure 10d shows the *normalized* cost of the ideal scheme compared to *Spock* under the two scaling policies. *Spock* is very close to the ideal scheme (difference of less than 7%) in terms of cost and at the same time achieves low SLO violation (maximum of 3%).

VII. Related Work

A. VM procurement and auto-scaling

There are several research works in the past that optimize for the VM provisioning cost. Typically these works broadly fall in to two categories as follows, (i) tuning the auto-scaling policy to optimize for type of VM procurement based on changing needs (Spot, On Demand, etc.,) [39, 42]–[46], (ii) predicting the peak loads and offer proactive provisioning based auto-scaling policy [3, 12]. Since all the VM offerings are billed at an hour granularity in the majority of these price models, their auto-scaling policy often fails to optimize for the case of intermittent load surges while end up paying the higher procurement costs. In contrast to these related work, *Spock* leverages two different types of service deployment which in turn exploits two different service offerings leading to overall cost-savings.

The most relevant work to *Spock* is FEAT, proposed by Novak et al. [47]. Both FEAT and *Spock* consider exploiting serverless functions to efficiently autoscale VMs, while *Spock* is specifically catered towards elastic ML-based web services. In addition *Spock* supports applications with multiple request types.

B. Serverless Computing

Despite its recency, serverless computing has already been used in different fields like Internet of Things [48, 49] and edge computing [50, 51], data parallel frameworks [52, 53], real-time video processing [54]–[57], and system security [58]. However, the recent work on serverless can be classified broadly into two categories.

1) Serverless application design: Firstly, To decompose a monolithic application into interconnected microservices to use the functional execution framework. Major body of related work [52, 54, 59] in serverless research proposes techniques to redesign the applications to take advantage of serverless functions owing to reduced management overheads. Villamizar et al. [60] compare the infrastructure costs of running application as microservices using IaaS or serverless platform without considering the mix of both in terms of cost. While Elgamal et al. proposed [61] function fusion and placement to reduce serverless function costs. However, they do not consider the case of changing resource demands which *Spock* uses to its advantage (as proposed in Section III).

2) Serverless framework optimization: Further there are several research works [29, 62, 63] that propose to optimize for lambda cold starts and address several other performance limitations of using serverless framework. Container initialization and package dependencies are also common causes for container cold-start, which is addressed in [64]. We fine-tune *Spock* framework based on their findings to keep the containers warm that further reduces SLO violations.

VIII. Conclusion

In this paper, we identify the shortcomings of using existing VM-based *autoscaling* mechanisms and its inabilities to guarantee the SLO at a given cost budget for an elastic web service. We propose *Spock*, a scalable elastic control system which exploits serverless functions along with VMs *autoscaling*. We evaluate *Spock* against two different *autoscaling* mechanisms under reactive and predictive scaling policies. *Spock* reduces the SLO violations by up to 74% while reducing the cost by up to 36% when compared to VM only resource procurement scheme. We further plan to deploy *Spock* in public cloud systems and evaluate for real-time workloads from different application domains.

Acknowledgment

We are indebted to Ashutosh Pattnaik, Anup Sarma, Haibo Zhang, Cyan Mishra, Vivek Bhasi, and Prasanna Rengasamy for their insightful comments on several drafts of this paper. Also, this research is partially supported by NSF grants #1629129, #1763681, #1439021, #1629915, #1439057, #1626251, #1526750 and we thank NSF Chameleon Cloud project CH-819640 for their generous compute grant.

REFERENCES

[1] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars, "Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 223–238.

[2] P. V. Rengasamy, H. Zhang, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "CritICs Critiquing Criticality in Mobile Apps," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 867–880.

[3] A. Gujarati, S. Elnikety, Y. He, K. S. McKinley, and B. B. Brandenburg, "Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. New York, NY, USA: ACM, 2017, pp. 109–120.

[4] J. A. Hoxmeier and C. Dicesare, "System response time and user satisfaction: An experimental study of browser-based applications," 2000.

[5] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *IEEE Computer*, vol. 40, pp. 103–105, 10 2007.

[6] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das, "Phoenix: A constraint-aware scheduler for heterogeneous datacenters," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017, pp. 977–987.

[7] C. Wang, A. Gupta, and B. Urgaonkar, "Fine-grained resource scaling in a public cloud: A tenant's perspective," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, June 2016, pp. 124–131.

[8] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," in *2012 IEEE Fifth International Conference on Cloud Computing*, June 2012, pp. 423–430.

[9] H. Liu and S. Wee, "Web server farm in the cloud: Performance evaluation and dynamic architecture," in *Cloud Computing*, M. G. Jaatun, G. Zhao, and C. Rong, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 369–380.

[10] G. Galante and L. C. E. d. Bona, "A survey on cloud computing elasticity," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, Nov 2012, pp. 263–270.

[11] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A cost-aware elasticity provisioning system for the cloud," in *2011 31st International Conference on Distributed Computing Systems*, June 2011, pp. 559–570.

[12] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Trans. Auton. Adapt. Syst.*, vol. 3, no. 1, pp. 1:1–1:39, Mar. 2008.

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

[14] "Amazon machine learning - predictive analytics with aws." https://aws.amazon.com/machine-learning/ , February 2018.

[15] "Google cloud functions." https://cloud.google.com/functions/docs/, February 2018.

[16] "Microsoft azure functions." https://azure. microsoft.com/en-us/services/functions/, February 2018.

[17] "Watson machine learning." http://datascience.ibm.com/features machinelearning , February 2018.

[18] Y. Yamasaki and M. Aritsugi, "A case study of iaas and saas in a public cloud," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 434–439.

[19] Y. Yao, Z. Xiao, B. Wang, B. Viswanath, H. Zheng, and B. Y. Zhao, "Complexity vs. performance: Empirical analysis of machine learning as a service," in *Proceedings of the 2017 Internet Measurement Conference*, ser. IMC '17. New York, NY, USA: ACM, 2017, pp. 384–397.

[20] "ML on lambda," https://medium.freecodecamp.org/what-we-learned-by-serving-machine-learning-models-using-aws-lambda-c70b303404a1, 2019.

[21] "Azure Machine Learning as a Service." https://azure.microsoft.com/en-us/pricing/details/machine-learning-service/ , February 2018.

[22] "AWS Sagemaker." https://aws.amazon.com/sagemaker/, February 2018.

[23] "Automating Elasticity," https://d1.awsstatic.com/whitepapers/cost-optimization-automating-elasticity.pdf, March, 2018.

[24] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes, "Press: Predictive elastic resource scaling for cloud systems," in *2010 International Conference on Network and Service Management*, Oct 2010, pp. 9–16.

[25] B. Urgaonkar, P. Shenoy, and T. Roscoe, "Resource overbooking and application profiling in shared hosting platforms," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 239–254, Dec. 2002.

[26] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy, "Profiling and modeling resource usage of virtualized applications," in *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, ser. Middleware '08. New York, NY, USA: Springer-Verlag New York, Inc., 2008, pp. 366–387.

[27] "AWS Lambda," https://aws.amazon.com/lambda/pricing/, 2016.

[28] "Amazon EC2 pricing," https://aws.amazon.com/ec2/pricing/.

[29] 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.

[30] "AWS Lambda." https://aws.amazon.com/lambda/, February 2018.

[31] V. Persico, A. Montieri, and A. Pescap, "On the network performance of amazon s3 cloud-storage service," in *2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, Oct 2016, pp. 113–118.

[32] N. Liu, L. Wan, Y. Zhang, T. Zhou, H. Huo, and T. Fang, "Exploiting convolutional neural networks with deeply local description for remote sensing image classification," *IEEE Access*, vol. 6, pp. 11 215–11 228, 2018.

[33] P. Thinakaran, J. Raj, B. Sharma, M. T. Kandemir, and C. R. Das, "The curious case of container orchestration and scheduling in gpu-based datacenters," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. ACM, 2018, pp. 524–524.

[34] J. B. Kotra, H. Zhang, A. R. Alameldeen, C. Wilkerson, and M. T. Kandemir, "Chameleon: A dynamically reconfigurable heterogeneous memory system," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2018, pp. 533–545.

[35] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang, "Grandslam: Guaranteeing slas for jobs in microser-

vices execution frameworks," in *Proceedings of the Fourteenth EuroSys Conference 2019*, ser. EuroSys '19. ACM, 2019, pp. 34:1–34:16.

[36] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers," *ACM Trans. Comput. Syst.*, vol. 30, no. 4, pp. 14:1–14:26, Nov. 2012.

[37] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015.

[38] J. Deng, W. Dong, R. Socher, L. Li, and and, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009, pp. 248–255.

[39] A. Chung, J. W. Park, and G. R. Ganger, "Stratus: Cost-aware container scheduling in the public cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 121–134.

[40] "UC Berkeley Home IP Web Traces." http://www.comp.nus.edu.sg/ cs5222/simulator/traces/berkeley /index.htm.

[41] "WITS: Waikato Internet Traffic Storage," https://wand.net.nz/wits/index.php .

[42] A. Harlap, A. Chung, A. Tumanov, G. R. Ganger, and P. B. Gibbons, "Tributary: spot-dancing for elastic services with latency SLOs," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 1–14.

[43] C. Wang, B. Urgaonkar, N. Nasiriani, and G. Kesidis, "Using burstable instances in the public cloud: Why, when and how?" *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, pp. 11:1–11:28, Jun. 2017.

[44] A. Ali, R. Pinciroli, F. Yan, and E. Smirni, "Cedule: A scheduling framework for burstable performance in cloud computing," *2018 IEEE International Conference on Autonomic Computing (ICAC)*, pp. 141–150, 2018.

[45] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with processing-in-memory capabilities," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, 2016, pp. 31–44.

[46] A. Pattnaik, X. Tang, O. Kayiran, A. Jog, A. Mishra, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das, "Opportunistic computing in gpu architectures," in *Proceedings of the 46th Annual International Symposium on Computer Architecture*, ser. ISCA '19, 2019.

[47] J. H. Novak, S. K. Kasera, and R. Stutsman, "Cloud functions for fast and robust resource auto-scaling," in *2019 11th International Conference on Communication Systems Networks (COMSNETS)*, Jan 2019, pp. 133–140.

[48] "Aws greengrass." https://www.amazon.com/greengrass/.

[49] S. Zhao, P. V. Rengasamy, H. Zhang, S. Bhuyan, N. C. Nachiappan, A. Sivasubramaniam, M. Kandemir, and C. R. Das, "Understanding energy efficiency in iot app executions," July 2019.

[50] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster, "Ripple: Home automation for research data management," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 389–394.

[51] P. V. Rengasamy, H. Zhang, N. Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Characterizing diverse handheld apps for customized hardware acceleration," in *2017 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2017, pp. 187–196.

[52] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, ser. SoCC '17. New York, NY, USA: ACM, 2017, pp. 445–451.

[53] Q. Pu, S. Venkataraman, and I. Stoica, "Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, 2019, pp. 193–206.

[54] S. Fouladi, R. S. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, in *Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads.*, 2017.

[55] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: ACM, 2018, pp. 263–274.

[56] H. Zhang, P. V. Rengasamy, S. Zhao, N. C. Nachiappan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Race-to-sleep + content caching + display caching: A recipe for energy-efficient video streaming on handhelds," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. New York, NY, USA: ACM, 2017, pp. 517–531.

[57] H. Zhang, P. V. Rengasamy, N. Chidambaram Nachiappan, S. Zhao, A. Sivasubramaniam, M. T. Kandemir, and C. R. Das, "Floss: Flow sensitive scheduling on mobile platforms," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, June 2018, pp. 1–6.

[58] E. d. Lara, C. S. Gomes, S. Langridge, S. H. Mortazavi, and M. Roodi, "Poster abstract: Hierarchical serverless computing for the mobile edge," in *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, Oct 2016, pp. 109–110.

[59] L. Feng, P. Kudva, D. Da Silva, and J. Hu, "Exploring serverless computing for neural network training," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018, pp. 334–341.

[60] M. Villamizar, O. Garcs, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 179–182.

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

[62] 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.

[63] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "Sand: Towards high-performance serverless computing," in *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[64] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "SOCK: Rapid task provisioning with serverless-optimized containers," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 57–70.