ENSURE: Efficient Scheduling and Autonomous Resource Management in Serverless Environments

Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla,
Hima Upadhyay, Anshul Gandhi
PACE Lab, Department of Computer Science, Stony Brook University

***Abstract*—An imminent challenge in the serverless computing landscape is the escalating cost of infrastructure needed to handle the growing traffic at scale. This work presents ENSURE, a function-level scheduler and autonomous resource manager designed to minimize provider resource costs while meeting customer performance requirements. ENSURE works by classifying incoming function requests at runtime and carefully regulating the resource usage of colocated functions on each invoker. Beyond a single invoker, ENSURE elastically scales capacity, using concepts from operations research, in response to varying workload traffic to prevent cold starts. Finally, ENSURE schedules requests by concentrating load on an adequate number of invokers to encourage reuse of active hosts (thus further avoiding cold starts) and allow unneeded capacity to provably and gracefully time out. We implement ENSURE on Apache OpenWhisk and show that, across several serverless applications and compared to existing baselines, ENSURE significantly improves resource efficiency, by as much as 52%, while providing acceptable application latency.**

I. Introduction

Serverless computing is an emerging paradigm for running user-specified functions on provider resources with virtually unlimited scalability [2]. In serverless computing, the user is responsible for writing the code and packaging it, and the cloud provider is responsible for provisioning and maintaining the infrastructure, including host servers, needed to execute the user code/program [12]. The pay-per-use pricing for functions, currently at about 20 cents per million invocations (per AWS Lambda [2]), makes serverless a lucrative choice for end-users. There is consensus in the computing community that several classes of applications, including MapReduce frameworks, will eventually run seamlessly on serverless platforms [12].

An imminent challenge in this computing landscape is the escalating cost of resources (or infrastructure) needed to handle the growing serverless traffic, as alluded to by the recent study from RISELab [12]. The specific problem we consider in this paper is *how should user functions be scheduled and resource managed on bare-metal servers to minimize the provider's expenses at scale while providing acceptable latencies?* Addressing this problem requires both (i) efficient placement of the incoming workload on hosts to minimize the provider's capital expenses, and (ii) elastic scaling of the serverless platform capacity to minimize the provider's operating expenses in the presence of dynamic workload traffic. In serverless environments, the provider is responsible for making both these decisions.

There are several issues that make it difficult to achieve high resource efficiency in serverless environments:

- *Diverse applications:* Serverless computing has attracted interest from different application communities, including high-performance computing [20]. Consequently, serverless applications can differ substantially in their resource consumption patterns, ranging from short-lived scripts to embarrassingly parallel services [8], [12], [2]. This diversity necessitates application-specific scheduling and colocation to minimize resource contention across application classes.
- *Cold start latency:* Serverless functions are typically hosted on containers, which help in packaging the application and ease the function invocation process. When processing requests from a new application on a host, the following steps are involved: (i) launching a new container, (ii) setting up the runtime environments, and (iii) application-specific initialization; the latency for these steps is collectively termed as cold start latency [25], [12]. In our experiments, the cold start latency typically ranges from 3 to 6 seconds; by contrast, function execution can take as little as 0.5 seconds [25]. While recent advances, such as AWS Firecracker [2], reduce the latency of launching a new container (or microVM), the application and runtime initialization latencies still remain, and can be substantial [12], [14].
- *Resource efficiency at scale:* Serverless computing platforms, such as AWS, typically tout their ability to seamlessly handle any scale of workload requests [2]. To maintain high resource efficiency, however, the platform capacity (containers and hosts) must scale out and scale in elastically with the workload demand, including bursts of function requests [14]. Unfortunately, this can lead to frequent cold starts. Further, typical load balancing solutions, such as Round Robin or Least Connections, are designed to spread the load among available hosts, and so naturally tend to use all available hosts, resulting in potential under-utilization.

Existing approaches, such as schedulers designed for VM placement or web load balancers, are not well suited for serverless scheduling. The former requires specification of the resource request (e.g., number of cores), which is not an input that a serverless user needs to provide. The latter assumes that any server can execute an incoming request, whereas in serverless computing, specific servers that are already "warm" (have an active container of that application) are preferred to prevent cold starts. While container schedulers, such as Borg, may appear to be well suited for serverless workloads, they are not necessarily designed for short-lived functions, and can have task placement latencies as high as 25s [23]; by

contrast, serverless functions typically have latencies ranging from milliseconds to few seconds [12].

We present ENSURE, an EfficieNt SchedUling and autonomous REsource management framework for serverless computing designed to minimize provider expenses while providing acceptable request latencies. The design of ENSURE is guided by the need to achieve high resource efficiency within *and* across containers. Within a container, ENSURE takes into account the diverse resource consumption and lifetime patterns of serverless functions by classifying them into categories. ENSURE then scalably determines function placement based on the inferred class of the incoming function. To provide acceptable latencies, ENSURE mitigates the resource contention between colocated, and possibly diverse, functions by dynamically regulating their cpu-shares at runtime.

To achieve resource efficiency beyond a single container, ENSURE relies on two key components: (i) FnScale, our autoscaling component that dynamically scales the number of containers and hosts in response to changes in workload traffic, and (ii) FnSched, our scheduling component that carefully distributes function requests across hosts to minimize SLO violations. FnScale works by proactively maintaining a few additional containers and hosts that can seamlessly handle workload variations while preventing cold starts. Using concepts from operations research, we carefully choose the amount of additional capacity to (theoretically) ensure low request latency and high resource efficiency.

FnSched concentrates load on adequate number of hosts by greedily "packing" requests on a host before moving on to another host. To avoid latency violations, FnSched maintains a moving window of observed latencies to determine how aggressively each host can be packed. A key advantage of this greedy packing approach is that requests are often scheduled on non-idle hosts, thereby preventing cold starts. Another advantage of the packing approach, which we formally prove via queueing theoretic arguments, is that it allows unneeded hosts to "time out", thus naturally scaling in resources.

We implement a prototype of ENSURE on Apache OpenWhisk [3] and evaluate its performance on a 34-VM serverless cluster in AWS. Our experimental results show that ENSURE provides acceptable latencies across a diverse set of serverless applications, unlike the existing resource managers in OpenWhisk and Kubernetes. Our multi-host evaluation, using time-varying (trace-driven) traffic, highlights the efficiency of ENSURE; compared to existing scheduling policies, ENSURE handles the incoming traffic with 18%–52% fewer hosts and with about 60% fewer cold starts.

II. Background and Motivation

In a serverless computing platform, the user writes a cloud function in a high-level language and creates the trigger (events from the back-end, http end-points) to run the function [2]. The serverless provider is then responsible for the infrastructure that will execute the user function.

Figure 1 illustrates a typical serverless platform environment. The incoming user function is sent to the scheduler via an HTTP front-end; we use the terms function, application request, and workload interchangeably. The scheduler then chooses an *invoker* on which the function can run; the invoker, or host, is typically a server or VM. Depending on the application, a function may use additional back-end services, such as a database, object store, etc. Typically, the serverless platform uses containers or other sandboxing approaches to host and execute the function on an invoker [25]; we assume the use of containers, similar to Google Cloud Run [10] and Apache OpenWhisk [3]. Thus, an invoker is a VM or server which hosts the containers of serverless functions.

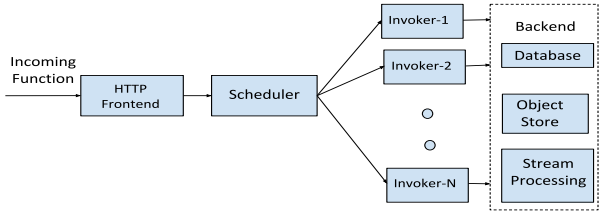


Fig. 1. Illustration of a typical serverless platform.

A key concern in the serverless environment is the ability to **maintain acceptable latency** for function requests. Prior work has shown that cold starts can significantly contribute to application latency, by as much 500ms–10s [12], [25], which is on the order of runtime of typical serverless applications [25], [14]. Our experiments on AWS Lambda (see Section IV-B for experimental setup) show that user applications can experience 20–120 cold starts per hour at moderate loads. Thus, to maintain acceptable performance, providers must carefully schedule user requests among invokers to minimize cold starts.

A competing objective for providers is the need to **maintain resource efficiency as serverless workload scales**. Unlike in VM cloud provisioning, users in the serverless environment are not required to fully specify their resource requirements; this complicates the provider's job of efficiently provisioning resources. For example, in AWS Lambda and Google Cloud Functions [25], the user can only specify a memory requirement (and not CPU requirement), which the provider internally uses to also allocate CPU resources proportionally. To obtain superior performance, users may resort to overprovisioning memory. As user workload demand scales, this behavior can result in severe underutilization of provider resources. Thus, to maintain resource and cost efficiency, providers must carefully manage the CPU and memory allocation of applications.

Complicating the above requirements is the fact that serverless workload demand can be quite variable. A recent Microsoft Azure study [14] found that the inter-arrival time distribution of serverless applications can be quite variable, necessitating a dynamic and elastic autoscaling solution.

III. Design of ENSURE

The design of ENSURE, and its key benefits, are rooted in the core principle of resource efficiency. ENSURE efficiently manages resources, both within containers and across containers, by more effectively "packing" function requests at each invoker; the key challenge here is to schedule requests *without*

	Application	Class	Runtime	Average CPU	Peak CPU	Mem usage	N/W usage
TRAIN	Image Resizing (IR)	ET	0.66 s	8.6%	10.3%	24.4 MB	0.8 MB/s
	Streaming Analytics (SA)	ET	0.73 s	12.4%	14.2%	21.5 MB	0.1 MB/s
	Nearest Neighbor (NN)	MP	8.2 s	68.5%	100%	126.4 MB	3.3 MB/s
	Comp. Fluid Dynamics (CFD)	MP	20.1 s	88.3%	100%	201.7 MB	2.2 MB/s
TEST	Email Gen (EG)	ET	0.64 s	19%	31%	45 MB	0.15 MB/s
	Stock Analysis (ST)	ET	0.78 s	14%	26%	30 MB	0.75 MB/s
	File Encrypt (FE)	ET	0.71 s	18%	33%	50 MB	1.3 MB/s
	Sentiment Review (SR)	ET	1.03 s	18%	35%	91 MB	0.8 MB/s
	Sorting (SO)	MP	44.9 s	90%	100%	318 MB	1.5 MB/s
	Matrix Multiply (MM)	MP	19.85 s	85%	100%	60 MB	0.65 MB/s

TABLE I
CHARACTERIZATION OF THE (TRAINING AND TEST SET) SERVERLESS APPLICATIONS USED IN OUR EXPERIMENTS.

significantly impacting function latency. We specifically design ENSURE to avoid cold starts (thus maintaining acceptable latency) and allow for more agile resource provisioning (to achieve high resource efficiency). We first describe how ENSURE achieves high resource efficiency at the container level, and then describe how ENSURE autoscales the number of containers and invokers to achieve resource efficiency at scale.

We consider a serverless environment as illustrated in Figure 1 with a tier of homogeneous invoker nodes. We consider multiple applications, possibly belonging to different users, that issue function requests; thus, functions can be heterogeneous in nature. As is typically the case (e.g., for AWS Lambda [2] and OpenWhisk [3]), we assume that the (peak) memory requirements of the function are provided as input by the user. Each invoker hosts application-specific containers, that in turn serve incoming functions for their application.

In serverless platforms, the user expects a certain level of acceptable service from the provider, in terms of the latency of executing the user function. We capture this requirement as a Service Level Objective (SLO) which dictates the allowable latency degradation for a user function, ***latencyThd***. Specifically, if the latency of function execution when run in isolation on a dedicated host is ***isoLatency***, then the SLO dictates that the latency in the serverless environment should be no more than $(isoLatency \cdot latencyThd)$.

A. *Resource management within a container*

Since the peak memory requirements of a container are known a priori, we only launch a container on an invoker that has enough spare memory to accommodate the container. By contrast, the CPU availability for a container may change abruptly over time (as colocated functions complete and new ones arrive), necessitating a more dynamic CPU resource management approach; in our experiments with serverless functions, we find that CPU is often the source of contention.

To regulate the CPU usage of containers colocated on the same processor core, ENSURE leverages *cpu-shares* [6], which specifies the relative share of CPU time available to the container; cpu-shares is a soft limit enforced only when CPU cycles are constrained. We allow multiple containers to share the same processor core to improve resource efficiency.

In general, the CPU requirements of a function depend on the underlying application. An embarrassingly parallel application will require many more CPU cycles compared to a short-lived function. Consequently, a long-running application may be able to tolerate a few milliseconds of CPU contention without much impact on latency, unlike a short-lived, CPU-intensive application. Thus, ENSURE takes into account the nature of the application when regulating its cpu-shares.

Classifying serverless applications. We classify applications into two categories – Edge Triggered (ET), and Massively Parallel (MP) – based on their runtime and resource usage. ET applications are typically short-lived and/or triggered based on events, e.g., streaming analytics and IoT back-end [2]. MP applications are resource intensive and are typically embarrassingly parallel, e.g., data mining [5] and MapReduce [18].

Table I shows the runtime and resource usage characteristics of the (training and test set) applications we experiment with in this paper; we describe these applications later in Section IV-B. The values are obtained by averaging the results over 3 runs of 5 minutes each. We see that different applications can have very different runtimes and resource usages. For several applications, the average, and even peak, cpu usage is often much below 100%, motivating the need to colocate containers on the same processor core to improve cpu utilization.

We find that applications can be easily classified into the MP category by using simple threshold rules on their runtime and/or average or peak CPU usage. For example, based on the training set applications, we can classify an application as MP or ET depending on whether or not it satisfies the condition $runtime > 5s \;\; \&\& \;\; avg\ CPU > 50\%$. Such a rule would result in perfect classification accuracy for the test set as well. We note that decision tree classifiers can be used to more formally classify applications into ET and MP.

In practice, when requests from a new application are first encountered, ENSURE executes them on containers that are not colocated with other applications. The resulting request latency should be close to the application's $isoLatency$, which, along with its observed resource usage characteristics, is used to classify the application. We note that, for a given application, the input data sizes may be different, resulting in a range of $isoLatency$ values; we address this issue by further classifying applications, as discussed in Section IV-E.

Regulating cpu-shares. Since different classes of applications have very different resource usage and runtime characteristics,

it is important to regulate the cpu-shares of application-specific containers colocated on the same core to mitigate resource contention. Commonly employed approaches to set cpu-shares either set the shares to a fixed value (e.g., the default Linux policy sets cpu-shares for every container to 1024) or set the shares proportional to a container's memory requirements, such as the cpu-shares policy used by AWS Lambda [13], [25] and OpenWhisk [3]. In fact, AWS's proportional policy encourages users to overprovision memory to accelerate their function, leading to memory under-utilization. Clearly, such static and ad-hoc approaches are not well suited for serverless applications since they exhibit a diverse range of resource usage; further, as new requests arrive and existing ones complete, the cpu-shares values must be dynamically updated.

ENSURE's dynamic and application-aware cpu-shares regulation policy works as follows. At runtime, ENSURE monitors the average latency of applications over a moving-window (10 requests, in our case). If the latency for an application starts to approach the SLO ($isoLatnecy \cdot latencyThd$), ENSURE increases its cpu-shares. In particular, ENSURE increases an application's cpu-shares if its moving-average latency exceeds $isoLatnecy \cdot updateLatencyThd$, where $updateLatencyThd < latencyThd$; the $updateLatencyThd$ acts as an early warning sign to prevent SLO violations. For example, in our experimental evaluation, we set $latencyThd = 1.15$ (meaning no more than 15% degradation) and set $updateLatencyThd = 1.10$. The exact SLO threshold is a parameter and can be tuned per the service provider's objectives; ENSURE's design is not specific to a given SLO threshold, and the SLOs can be application-dependent.

ENSURE increments cpu-shares of all containers of the application in steps of $cpuSharesStep$, to ensure that the cpu allocation of containers is increased gradually. If the increase in cpu-shares will exceed the total cpu-shares of the invoker (1024 cpu-shares per core), ENSURE rebalances the shares from the other application containers. After increasing cpu-shares for an application, ENSURE waits for $numUpdatesThd$ iterations (or requests) before evaluating the application's latency again, to ensure that the newly set cpu-shares value has taken effect. We perform a sensitivity analysis of the various algorithm parameters in Section IV-D.

B. Resource management across containers and invokers

To avoid violating performance SLOs in the presence of dynamic workload demand, ENSURE elastically scales the serverless platform capacity (number of invokers and containers). Simultaneously, in the presence of multiple invokers, ENSURE carefully schedules incoming function requests across invokers to facilitate elastic scaling while regulating the number of requests sent to an invoker.

The solution architecture of ENSURE is illustrated in Figure 2. The two core components that handle resource management across invokers are ENSURE's scheduling component, FnSched, and ENSURE's elastic scaling component, FnScale. At a high-level, incoming function requests are received by FnSched, which determines which invoker will serve each request. At the chosen invoker, an application-specific container is launched, if it is not already present, to execute the incoming request. The resource management at the container is handled via the cpu-shares regulation policy discussed in the previous subsection. To gracefully scale in capacity, FnScale employs a simple timeout-based approach. In parallel, the FnScale component periodically determines how many containers and invokers are required to maintain acceptable function latency, and scales out the required additional capacity, as needed.

We now discuss the FnSched and FnScale components.

1) Scheduling requests across invokers via FnSched

FnSched aims to pack (or concentrate) load on just the adequate number of invokers, allowing the additional unneeded invokers to idle (and eventually be turned off). The key challenge with this packing approach is to prevent overloading of the invokers to avoid SLO violations.

Operating zones for an invoker: To avoid SLO violations, FnSched tracks the load at each invoker, classifying them into different operating zones based on the moving average latency of application-specific requests over a window; in our implementation, the window spans the last 30 requests received at the invoker. When taking the moving average, we ignore the latency of cold start requests as this increased latency is not due to resource contention at the invoker but due to lack of warm containers. Note that the operating zones are defined for each application, and so the latency tracking is done per application as well.

Ideally, request latency should be between $isoLatency$ (the latency of request when run in isolation) and $isoLatency \cdot latencyThd$ (upper bound on allowable latency, beyond which an SLO violation occurs); in our implementation, we set $latencyThd = 1.15$, meaning that we allow at most 15% latency degradation beyond $isoLatency$. Avoiding SLO violations in a timely manner requires us to closely track the increase in application latency. To obtain an actionable signal which highlights this increase in latency, we divide the allowable latency range, $latencyRng = isoLatency \times (latencyThd - 1)$, into four quarters. If the moving average latency for an invoker is in the first quarter of the allowable range ($isoLatency$ to $isoLatency + 0.25 \cdot latencyRng$), we consider the invoker to be in Safe zone. Likewise, if the moving average latency is in the second, third, and fourth quarter of the allowable range, we consider the invoker to be in Prewarning, Warning, and Unsafe zones, respectively. The division of zones allows FnSched to efficiently decide on how many more requests can be sent to each invoker.

FnSched scheduling: FnSched starts by statically indexing all invokers from 1 to, say, $numInvokers$. To pack requests on invokers, FnSched attempts to schedule an incoming request at the lowest indexed invoker that is not in Unsafe zone, and has sufficient memory to host a new container, if needed. For example, if invoker 1 and invoker 2 are in Unsafe zone, and invoker 3 is in Prewarning zone, the next incoming request will be scheduled at invoker 3.

To maintain acceptable latencies, FnSched limits the num-

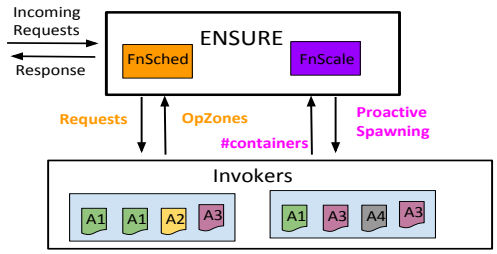


Fig. 2. Illustration of FnSched and FnScale's role in ENSURE. A1–A4 represent different applications running in containers within the invokers.

ber of simultaneous requests of an application that can be handled by any invoker; we refer to this limit as *capacity*. Since the operating zone of an invoker is representative of its current load level, *capacity* is zone-dependent. For Safe and Prewarning zone invokers, we set *capacity* for an application to be the number of containers of that application; however, to limit cpu contention, the *capacity* value is not allowed to exceed the number of cores on the invoker. Thus, for any invoker and application, $capacity = min(\text{number of application containers}, \text{number of cores})$.

For a Warning zone invoker, we set $capacity = 1$ to allow only one more request to be scheduled to this invoker. In our previous example, if invoker 2 had instead just entered the Warning zone, the next incoming request would be scheduled at invoker 2, but a subsequent request would be scheduled at invoker 3. An advantage of this $capacity = 1$ setting is that it allows us to track the latency at such Warning invokers by gradually advancing the moving window by 1. For an Unsafe invoker, we set $capacity = 0$; after a backoff period (of 2s, in our implementation), we move the Unsafe invoker to the Warning zone and reset its moving window latency average (so that the appropriate operating zone can be inferred).

In summary, *FnSched schedules the next incoming request at the lowest indexed non-Unsafe invoker that has fewer than capacity inflight requests on it*. A key advantage of FnSched's lowest-indexed-based packing scheduling is that by preferring lower numbered invokers, ENSURE tends to reuse previously used invokers, thus avoiding cold starts. If all invokers are either in Unsafe zone or already have *capacity* requests, then the incoming request is scheduled at the least loaded (in terms of number of inflight requests) invoker, potentially overriding the *capacity* at that invoker. Note that if a request is scheduled at invoker i, the next incoming request may get scheduled at invoker $j < i$ if invoker j moves into the Safe or Prewarning zone in the inter-arrival time between the requests.

Once scheduled, the chosen invoker executes the request on an existing idle container of the incoming application, if it exists, else it launches (or wakes up) such a container and executes the request. Note that, for FnSched, the Safe and Prewarning zones have the same role. However, for the elastic scaling component, which we discuss next, the differentiation between Safe and Prewarning zones is important.

2) Elastically scaling serverless capacity via FnScale

The second core component of ENSURE is its elastic capacity manager, FnScale, which helps achieve resource efficiency at scale. Intuitively, while scaling out, FnScale aims to maintain some additional capacity to handle bursts of workload demand. While scaling in, FnScale deactivates (or turns off) unneeded capacity if it has been idle for some time.

Scaling out container capacity: To minimize SLO violations, FnScale scales out container capacity via the "square-root staffing" policy, which ensures that the probability of an incoming request not finding an available container is arbitrarily small. For reference, we state the corresponding theoretical result from the queueing theory literature below.

Theorem 1 (Square-Root Staffing Rule)**.** *[11, Theorem 15.2] Given an M/M/k with arrival rate λ and server speed μ and $R = \lambda/\mu$, where R is large, let k^*_α denote the least number of servers needed to ensure that the probability of queueing, $P_Q < \alpha$. Then $k^*_\alpha \approx R + c\sqrt{R}$, where c is the solution to the equation $\frac{c\Phi(c)}{\phi(c)} = \frac{1-\alpha}{\alpha}$, where $\Phi(\cdot)$ denotes the c.d.f. of the standard Normal and $\phi(\cdot)$ denotes its p.d.f.*

Theoretically (under the M/M/k modeling assumptions), the square-root staffing policy [11, Theorem 15.2] ensures that the probability of an incoming request not finding an available container (or "server", as in the Theorem) is arbitrarily small, thus providing low request latency. Despite the theoretical setting, the square-root staffing rule has been shown to work well in realistic settings, for example, in prior data center capacity provisioning research [9], [4].

In practice, we find that $c = 1$ works well for our experiments, and so FnScale maintains, at all times, an additional $\lceil\sqrt{R}\rceil$ containers. For a well-provisioned system, R is the average number of active servers/containers. In our implementation, we use the number of containers being actively employed as a proxy for R; we mark a container as active if it served at least one request in the past 5 seconds. Note that the R value is tracked for each application, allowing us to transparently scale capacity for all applications. The additional $\lceil\sqrt{R}\rceil$ containers are launched, in first-fit manner, on the lowest indexed invokers (as these are likely to handle much of the load because of our packing-based scheduling). We periodically send a heartbeat request to these newly launched containers to keep them warm. Note that this is different from prewarm containers that OpenWhisk maintains at all times which are generic language runtimes (currently only node.js containers are spawned) and would require application specific libraries to be loaded before they can service function requests.

The additional $\lceil\sqrt{R}\rceil$ containers maintained by FnScale serve as a buffer of warm containers in case of an abrupt load spike. There could, of course, be cases where the load spike is so severe that additional containers will have be launched, as directed by FnSched when it looks to schedule incoming requests on available containers. However, as we discuss at the end of this subsection, the number of containers eventually does return to (almost) $R + \lceil\sqrt{R}\rceil$. In our implementation, we check for container scale-out periodically once every 5secs.

Scaling out invoker capacity: If the required number of containers (say, $R + \lceil\sqrt{R}\rceil$) cannot be accommodated on

the existing number of invokers, then we add the required number of invokers. Note that, in our design, an invoker can accommodate at most $capacity$ requests, and thus at most $capacity$ containers. For example, if $capacity = 4$, and we currently have only one invoker (with index 1) with $R = 3$ active containers, then, by the square-root staffing policy, we need to launch $\lceil\sqrt{R}\rceil = 2$ additional containers. Since the one invoker will only support $capacity = 4$ containers, we scale-out to a second invoker (assigning it the index 2) and launch the 5th container on that invoker.

Launching (or turning on) a new invoker may involve some startup delay, which could impact ENSURE's ability to handle bursts of requests. In fact, the additional containers launched on the new invoker will first experience a cold-start, further limiting ENSURE's ability to quickly scale out. To prevent such delays and cold starts, we proactively launch a new invoker (if sufficient invokers are not available) and the required additional containers on it every time an invoker enters the Prewarning zone. This proactive approach significantly reduces the cold-starts experienced by application requests.

Scaling in capacity: Finally, to scale-in unneeded capacity, e.g., when load decreases, we use a simple timeout approach and deactivate (or turn off) idle containers and invokers after a timeout period (60s, in our implementation). For containers, the timeout policy does not impact the $R + \lceil\sqrt{R}\rceil$ required containers since we keep the $\lceil\sqrt{R}\rceil$ containers warm (non-idle) by sending periodic heartbeat messages. The timeout-based policy is decentralized and easy to implement, yet provides useful theoretical performance guarantees when combined with our packing-based scheduling, as we show below.

Result 1. *For an $M/M/\infty$ system with load R, under the timeout policy with appropriately chosen timeout value, when using the packing-based scheduling algorithm, the number of containers (or invokers) converges to $R + \sqrt{R}$.*

Proof. For an $M/M/\infty$ with load R, the number of requests in system (and hence the number of active containers) follows a Poisson distribution with mean R [11, Ch. 15.2.1]. Under the packing-based scheduling algorithm, let $I(i)$ denote the idle time (time between requests) for the container (or invoker) indexed i. Based on the underlying $M/M/\infty$ Markov chain, with arrival rate λ and service rate μ (so $R = \lambda/\mu$) we have:

$$E[I(i+1)] = \frac{1}{\lambda + i\ \mu} + \frac{i\ \mu}{\lambda + i\ \mu} \cdot (E[I(i)] + E[I(i+1)])$$

$$\implies E[I(i+1)] = \frac{1}{\lambda}\ (1 + i\ \mu\ E[I(i)]) \quad (1)$$

Since $E[I(1)] = 1/\lambda$, we have:

$$E[I(i+1)] = \frac{1}{\lambda}\left(1 + \frac{i}{R} + \frac{i(i-1)}{R^2} + \ldots + \frac{i!}{R^i}\right) \quad (2)$$

Since $E[I(i)]$ is monotonically increasing, if we set the timeout value to $E[I(R+\sqrt{R})]+\epsilon$, the $(R+\sqrt{R})$th container (or invoker) will remain on, but the $(R + \sqrt{R} + 1)$th one can time out, resulting in $R + \sqrt{R}$ containers (or invokers). □

Result 1 theoretically ensures (under the stated model assumptions) that even if the capacity does temporarily increase beyond $R + \sqrt{R}$ (for example, in response to an increase in workload), the capacity will eventually converge back to $R+\sqrt{R}$. Together with Theorem 1, we thus have the property that ENSURE will maintain adequate capacity to ensure low request latency. Since $\sqrt{R} << R$ for high loads, ENSURE can provide low request latency without requiring too much additional capacity, thus achieving good resource efficiency.

In practice, of course, we do not have an $M/M/\infty$ or even an $M/M/k$ system. However, as we show in our evaluation, the timeout-based scale-in policy together with the packing-based scheduling does result in ENSURE achieving good resource efficiency while providing acceptable function latency.

IV. Evaluation Results

We now present our evaluation of ENSURE. We start by detailing our prototype implementation of ENSURE, and then describe our experimental setup and evaluation methodology. Finally, we present our experimental evaluation results for ENSURE under the single-invoker and multi-invoker settings.

A. Implementation of ENSURE

We implement ENSURE on top of Apache OpenWhisk [3], an open-source serverless cloud platform. OpenWhisk has a REST interface to accept requests and provide a response to them. The Controller component is responsible for processing the HTTP method and executing the function on the invoker, which in turn creates the Docker container. We implement our cpu-shares algorithm from Section III-A at each invoker, and implement the FnSched scheduler and the FnScale elastic scaler at the Controller. In total, we add about 2,000 lines of Scala code to implement ENSURE (publicly available [17]).

B. Experimental Setup

We host our OpenWhisk cluster in AWS with 34 m5a.2xlarge VMs, each with 4 cores (hyper-threading disabled) and 32GB of memory. The HTTP front-end and Apache Kafka messaging service run on a dedicated VM and the remaining OpenWhisk components, including the Controller, run on a different VM. The remaining 32 VMs serve as invokers. The applications hosted on these invokers are deployed on Docker containers, with the inactivity timeout for the containers set to 1 minute. To support the applications we employ, we use the following distributed back-end services: Redis database (3 VMs), Apache Kafka (1 VM), and AWS S3 for object storage. These back-end services are adequately provisioned so as not to be the performance bottleneck.

Applications. We employ ten serverless applications in our experiments, all of which we implement in Python 3.5. Their runtime and resource usage characteristics are listed in Table I. Six of these, Image Resizing (IR), Streaming Analytics (SA), Email Gen (EG), Stock Analysis (ST), File Encrypt (FE), and Sentiment Review (SR), are classified as edge triggered (ET). The remaining four, Nearest Neighbor (NN), Computational Fluid Dynamics (CFD), Sorting (SO), and Matrix Multiply (MM), are classified as massively parallel (MP). We divide the applications into training and test sets, as denoted in Table I, for sensitivity analysis and parameter tuning.

The application names are representative of their functionality. For example, Image Resizing (IR) accepts an input image, which is read from a distributed file system, resizes it into three different image sizes, and stores them back into the distributed file system. Likewise, Nearest Neighbor (NN), whose MPI implementation we port from Rodinia [5], finds the k-nearest neighbors for an unstructured data set. All ET applications are modeled from the Lambda reference architecture examples [2]. Where applicable, the applications use Object Store (AWS S3), to read and store the data, and Apache Kafka and Redis database (self-hosted on VMs), respectively, to read and store streaming data. All applications use 256 MB of reserved memory, expect CFD and Sorting, which use 512 MB memory.

C. Evaluation Methodology

Metrics. We use two key metrics to evaluate the performance of different scheduling policies. All reported evaluation metrics are averaged across 5 runs of each experiment.

1) *Latency degradation*: this is the average percentage latency degradation of an application compared to its standalone latency, $isoLatency$. We set the SLO to be 15% over the $isoLatency$; thus, $latencyThd = 1.15$ (see Section III).
2) *Invokers used*: this is the time-average number of invokers used by a scheduler over the duration of the experiment, and acts as a proxy for the provider's expenses. An invoker is considered to be "in use" if it has at least one live container. Since all invokers are homogeneous in our setup, invoker count serves as a proxy for the cost metric.

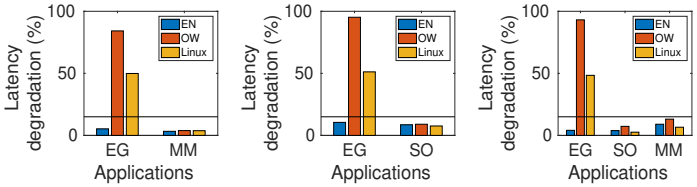
Workload and traces. We define a base load of 1× for each application as the maximum request rate (with deterministic inter-arrival time) for which the average latency does not exceed $isoLatency$, when that application is run in isolation. For single invoker evaluation (Section IV-D), the experiments have a warm up phase wherein we spawn a container per core per application, followed by 5 minutes of constant load (1.5–2× for ET applications and 1× for MP application). For multi-invoker evaluation, we use synthetic and real-world traces to drive the time-varying load, as detailed in Section IV-E.

D. Evaluating Single Invoker Scheduling

We start by evaluating ENSURE under a single invoker. Recall from Section III-A that the key challenge for single invoker scheduling is the resource (cpu) contention among functions. In each experiment, we compare ENSURE with the following:

1) *OpenWhisk default*: OpenWhisk sets cpu-shares for each container proportional to its requested memory capacity [3]. Specifically, if m is the memory requested by a container, and M and C are respectively the total memory and cpu-shares capacity of the invoker, the container is provided $C \cdot m/M$ cpu-shares. This policy will maximize the number of containers that can be spawned in a host, since both memory and cpu are being proportionally allocated.
2) *Linux default*: Linux's cpu-shares policy allocates every container with one CPU core (1024) worth of cpu-shares.

Sensitivity analysis. ENSURE's cpu-shares algorithm (Section III-A) has three tunable parameters – $numUpdatesThd$, $cpuSharesStep$, and $maxCpuShares$. Based on an empirically conducted sensitivity analysis (see our preliminary work [22]), we choose the following values for these parameters: $maxCpuShares$ of 768 for ET and 256 for MP; $numUpdatesThd$ of 5 for ET and 3 for MP; $cpuSharesStep$ of 128 for ET and 64 for MP.



(a) EG with MM (b) EG with SO (c) EG with SO and MM

Fig. 3. Latency degradation when colocating applications on a single invoker.

Colocation results. Figure 3 shows our results when colocating EG with MM, SO, and MM+SO. In all cases, ENSURE easily satisfies the 15% latency degradation SLO (denoted by the horizontal line). By contrast, the SLO is severely violated for EG under both comparison baselines, and by as much as 95% under the OpenWhisk baseline.

Under ENSURE, the two applications quickly stabilize at their respective maximum cpu-shares. Subsequently, when there is contention, EG gets higher preference (since it has a $maxCpuShares$ value of 768 compared to 256 for the MP application) and so its latency degradation is minimized. Under the Linux baseline, all applications are allocated the same cpu-shares value, making the short-lived EG application vulnerable to resource pressure from the longer-running, cpu-intensive MP applications. Under OpenWhisk, which allocates cpu-shares proportional to the memory requirements of the application, EG gets equal preference when colocated with MM, but gets lower preference when colocated with SO (since SO uses 512MB of memory compared to the 256MB used by EG and MM). EG is thus unable to avoid cpu contention with the MP application, resulting in severe latency degradation.

Finally, Figure 3(c) shows our results when EG is colocated with both, SO (on 2 cores) and MM (on the remaining 2 cores). Again, ENSURE consistently meets the SLO requirements of all applications. By contrast, the latency degradation of EG is much higher under the Linux and OpenWhisk baselines.

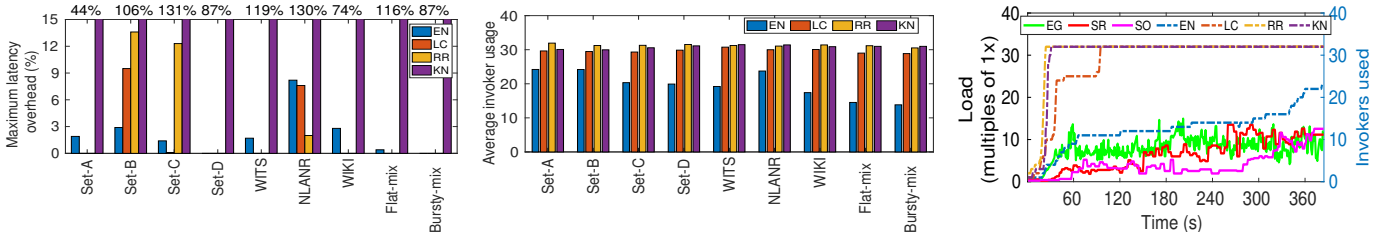
We also experimented with other potential colocation combinations using other test set applications; results are qualitatively similar, and are thus omitted in the interest of space.

E. Evaluating Multi-Invoker Scheduling

We now evaluate the performance of ENSURE in the more challenging and practical multi-invoker scenarios. The objective for multi-invoker scheduling is to achieve high resource efficiency by minimizing the average number of invokers employed while ensuring acceptable application latencies.

To evaluate the benefits of ENSURE, we compare its performance with three baseline schedulers:

1) *Round Robin (RR)* aims to distribute the load by sending successive requests to different invokers in a cyclic manner.



(a) Maximum (across all applications) latency increase under different schedulers.

(b) Time-averaged number of invokers used under different schedulers.

(c) Timeline view of load and resource scaling in our experiments for Bursty-mix workload.

Fig. 4. Performance evaluation of different schedulers under various workload scenarios.

2) *Least Connections (LC)* sends the incoming request to the least loaded (fewest in-flight requests) invoker. Archipelago's scheduling policy [21] closely resembles LC.
3) *Knative (KN)* [1] does container placement by filtering the invokers which can accommodate the container, scoring them (based on resource availability), and picking the highest scored invoker (breaking ties randomly). Knative scales capacity to maintain an upper bound on the moving average (over 60s) number of in-flight requests at each container, with the caveat that if the load doubles in a shorter panic-window (of 6s), then additional containers are added to handle the burst. We use the default settings of Knative in our experiments with the exception of the `container-concurrency-target` parameter, which we set to 1 to improve application latency by disallowing simultaneous requests at a container.

For ENSURE and all baselines (except Knative, which employs Linux's default cpu-shares policy), we employ the cpu-shares algorithm from Section III-A with the parameters identified by our sensitivity analysis.

Results. We employ synthetic and real-world traces to evaluate our multi-invoker scheduling. Our synthetic traces start from 1× load, gradually increase to a peak load, and then gradually decrease to 1× load (resembling an inverted V shape). Using synthetic traces, we consider four workload scenarios:

1) ***Set-A*** employs all four test set ET applications (EG, ST, FE, and SR) and drives the load using synthetic traces with a peak load of 16× for each application.
2) ***Set-B*** is the same as *Set-A* but with an additional MP application (SO) which also has a peak load of 16×.
3) ***Set-C*** uses two ET applications (EG, ST) with 32× peak load and one MP application, MM, with 16× peak load.
4) ***Set-D*** uses EG, 64× peak load, and MM, 16× peak load.

For realistic arrival patterns, we use trace snippets from WITS [24], NLANR [15], and Wikipedia [7], appropriately scaled for our setup, and consider the following scenarios:

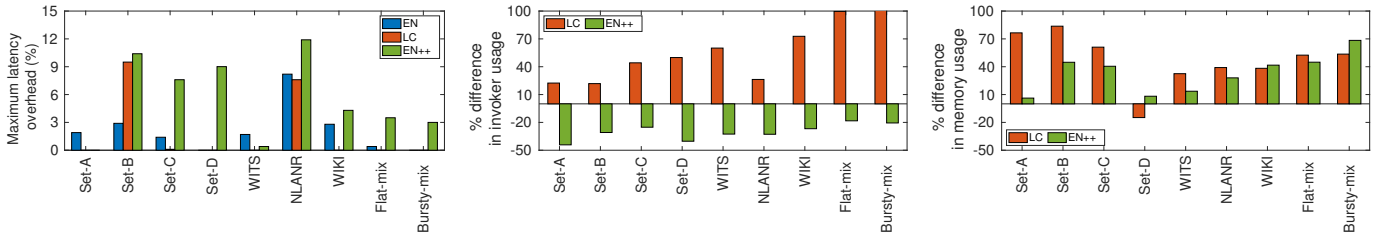
1) We employ ***WITS*** traces to drive the load for all four test set ET applications with a peak load of 18×.
2) ***NLANR*** traces drive the load for all four test set ET applications with 22× peak load and an MP application (SO) with 13× peak load.
3) ***WIKI*** traces drive the load for two test set ET applications (EG, ST) with 16× peak load and SO with 14× peak load.
4) ***Flat-mix*** uses relatively stable request rate traces from all three trace families to drive the load for two test set ET applications (EG, FE) with peak load of 15× and SO with peak load of 12×.
5) ***Bursty-mix*** uses bursty traces from all trace families to drive the load for two test set ET applications (EG, SR) with peak load of 14× and SO with peak load of 12×.

Figure 4(a) shows the latency degradation for each workload scenario under all four schedulers. For ease of presentation, we only show the maximum latency degradation from among all applications employed in each scenario. We see that ENSURE (abbreviated as EN) easily meets the 15% latency target for all workload scenarios, highlighting the efficacy of EN under gradual synthetic load changes and realistic load fluctuations. In fact, the degradation is less than 3% for all scenarios, except for NLANR (8.2%), which has some abrupt load changes. RR and LC also meet the 15% latency target for all workload scenarios, but have relatively higher degradations for Set-B and Set-C workloads, in addition to NLANR. In general, LC has lower latency degradation than RR.

However, Knative (KN) does not meet the latency target in any case, with the maximum latency degradation ranging from 44%–132%; this is clearly much higher than the 15% degradation target. Worse, Knative also drops 1–2% of the EG application requests in all scenarios, and 5–13% of the SO application requests for the scenarios which employ SO.

Figure 4(b) shows the time-average number of invokers employed for each scenario under all four schedulers. For every scenario, ENSURE uses the least number of invokers. Across all synthetic trace workloads, ENSURE uses, on average, 25%, 30%, and 27% fewer invokers, respectively, compared to LC, RR, and KN. The savings afforded by EN for the trace-driven workloads are higher — 40%, 43%, and 43%, respectively, compared to LC, RR, and KN. This shows that EN autoscales the invokers much better than the other baselines, especially for the workload fluctuations experienced by real-world applications. In terms of other baselines, LC, RR, and Knative all have similar invoker usage, with LC being the most resource efficient among them. Combined with the latency degradation results, we conclude that LC is slightly superior to RR, and both are vastly superior to Knative (which drops requests and still has a similar invoker usage).

The resource savings achieved by ENSURE validate the theoretical resource efficiency properties enabled by FnScale.



(a) Maximum (across apps) latency increase. (b) Invoker usage compared to ENSURE. (c) Memory usage compared to ENSURE.

Fig. 5. Performance of ENSURE++ and Least-Connections compared to ENSURE.

Consider the timeline view of resource usage in Figure 4(c) (corresponding to the Bursty-mix workload); the relative application load is depicted by the solid lines (left axis) and the invoker usage is shown with dotted lines (right axis). We see that while RR, LC, and KN clearly overprovision the number of invokers, ENSURE (shown as EN) uses much fewer invokers. Importantly, the invoker count under EN closely follows the overall application load (which increases with time), highlighting the agility of EN.

Another important aspect of our design is that ENSURE achieves acceptable latency *despite* packing requests in fewer invokers. A key reason for this is the proactive spawning done by FnScale, which significantly reduces the number of cold starts experienced under ENSURE. Compared to LC and RR, ENSURE has 64% and 60% fewer cold starts, respectively. In general, bursty workload traffic necessitates more cold-starts. For example, all policies incurred 5%–10% additional cold-starts under the Bursty-mix workload compared to Flat-mix.

Tuning the resource efficiency tradeoff. Serverless providers may have varying requirements and priorities when it comes to the tradeoff between, say, latency degradation and resource efficiency. ENSURE has been purposely designed with tunable parameters to allow such tradeoffs.

As one specific example, consider the case where a thinly provisioned serverless provider has higher preference for invoker efficiency given the scarcity of servers at their disposal. In this case, the provider can tune the $capacity$ parameter in FnSched (see Section III-B1) to allow more containers to be packed at each invoker. In particular, we implement a variation of ENSURE, which we refer to as ENSURE++ (or EN++), in which the $capacity$ for Safe and Prewarning invokers is increased to number of cores (from the previous value of $capacity = min(\text{number of application containers}, \text{number of cores})$).

Figures 5(a) and 5(b) shows the latency degradation and increase in invoker usage (relative to ENSURE) for EN++ and LC; we ignore RR and Knative since LC and EN are clearly superior to them, as established by our previous results. We see that EN++ provides additional invoker usage savings over EN, to the tune of 30%; compared to LC, we find that EN++ reduces the number of invokers used by almost 54%, averaged across all workload scenarios. However, this enhanced resource efficiency comes at the expense of an increase in latency degradation, though the degradation is still within the allowed 15% threshold. Compared to EN, EN++ increases the (absolute) latency degradation by around 3.5%.

Finally, there is another tradeoff that can be explored – the tradeoff between different types of resource usage. For example, the average number of containers in use can serve as a proxy for the memory cost, in addition to the invoker usage metric which serves as a proxy for the overall resource cost to the provider. Figure 5(c) shows the increase in memory usage incurred by EN++ and LC compared to the memory usage of EN. We see that, while EN++ reduces invoker usage compared to EN, it increases memory usage (by about 33%). This is because, by design, ENSURE++ deploys more containers in invokers, hurting memory usage. Interestingly, ENSURE is more memory efficient than LC; on average, LC uses 47% more memory than ENSURE. Note that this 47% is higher than the 33% memory increase incurred by ENSURE++, making ENSURE++ more memory efficient than LC.

Extension to multiple input sizes. Thus far we assumed that the runtimes of an application, when adequately resource provisioned, should be similar across invocations. However, this may not be the case if, for example, the input data is vastly different across invocations. We consider the File Encrypt application and create three different sets of input files to be encrypted, each with a different file size: 0.9MB, 1.8MB, and 3.6MB. The average $isoLatency$ across multiple (150) invocations of the FE application for each input file size is shown in Figure 6. We make two important observations here. First, there is negligible variation in runtimes between different invocations for the same input size, as evidenced by the near-zero standard deviation range around the three individual bars. Second, the runtime can indeed change with the input data.

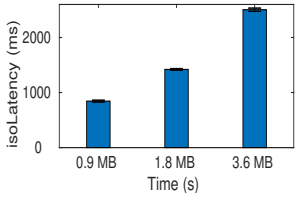


Fig. 6. $isoLatency$ for different input sizes of File Encryption.

We perform a simple experiment where all three input file size sets are made available to FE, and the overall load scales from $1\times$ to $16\times$ and back. Based on the input data size, we classify, at runtime, different invocations of FE into three classes, each with its own $isoLatency$ (obtained during the first few request executions). This can be easily done at the invoker, by the provider, by monitoring the data transfer over the network during a request execution. ENSURE treats each class as a different application, and attempts to maintain application latency for each class close to its $isoLatency$.

The resulting latency degradation for each class is below the 15% degradation target. Thus, by also classifying applications based on their input data sizes, we can extend ENSURE to applications with diverse inputs.

V. Prior Work

Serverless platform characterization: In FaasProfiler [19], the authors perform a microarchitectural characterization of the FaaS (Function as a Service) platform. The study attributes cold-starts and resource contention as the major sources of performance degradation for serverless applications; these are exactly the factors that ENSURE attempts to mitigate. A performance study was conducted by Wang et al. [25] on the serverless offerings of AWS, Azure and Google. The study finds that AWS uses a bin-packing-like strategy to maximize VM memory utilization and that severe contention between functions can arise in AWS and Azure. A similar characterization was conducted by Lloyd et al. [13], revealing that container initialization burdens serverless computing platforms. The authors found that extra infrastructure is provisioned to compensate for initialization overhead of cold service requests, motivating the need for resource efficiency.

Scheduling: In Archipelago's serverless scheduling scheme [21], which is similar to our LC baseline, a new container is placed on a node which has the fewest number of containers of that application. This is done to ensure that application latency is low and to ensure that application container coverage is well balanced across hosts. In stark contrast to the design choice made by Archipelago [21], we show that it is indeed possible to achieve acceptable latency while concentrating load on fewer invokers by carefully monitoring request latency (via operating zones, in our case). In fact, concentrating load also naturally scales in capacity by allowing unneeded nodes to time out.

Prior work has inferred that AWS Lambda's (closed-source) scheduling employs bin packing to pack requests from the same user on an invoker [25], [13]. However, AWS statically allots the cpu (proportional to memory request), resulting in potential under-utilization of memory; by contrast, ENSURE independently and dynamically sets cpu-shares. As shown in Section IV-D, ENSURE provides superior performance compared to static or proportional cpu-shares policies.

Kubernetes can schedule serverless containers (currently available as Knative service), but it requires the scaling policy parameters to be specified by the user. By contrast, ENSURE monitors resource usage and autonomically scales resources (invokers) to meet SLO requirements, without introducing too much complexity in the scheduling design. Further, as we show in our evaluation, Knative often drops function requests under high load and has high invoker usage, unlike ENSURE.

Existing request-level schedulers, such as Sparrow [16], typically consider requests that have similar execution patterns. By contrast, serverless requests can take anywhere from a few milliseconds (ET applications) to multiple seconds or even minutes (MP applications) to complete. As such, ENSURE also regulates the resource usage of the requests at a fine-grained level (via cpu-shares) to mitigate contention.

VI. Conclusion

This paper presents the design and evaluation of ENSURE, an efficient scheduler and resource manager for serverless platforms. Unlike existing solutions designed for serverful clouds, ENSURE is specifically tailored to avoid fine-grained resource contention between diverse application functions and prevent container cold starts. The design of ENSURE, grounded in queueing theoretic concepts, enables high resource efficiency without compromising on application latency.

Acknowledgment

This work was supported by NSF grants 1717588 and 1750109, and the AWS Cloud Credits for Research program.

References

[1] Knative. https://knative.dev, 2020.
[2] Amazon Web Services. *Serverless Computing*, 2020.
[3] The Apache Software Foundation. *Apache OpenWhisk*, 2020.
[4] Baarzi, A. F., et al. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *ACM SoCC* (2019).
[5] Che, S., et al. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC '09* (2009).
[6] Docker. *Specify container resource*, 2020 (accessed May 11, 2020).
[7] G Urdaneta et al. Wikipedia workload analysis for decentralized hosting. *Elsevier Computer Networks* (2009). http://www.globule.org/publi/WWADH_comnet2009.html.
[8] Gan, Y., et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS '19* (2019).
[9] Gandhi, A., et al. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst.* (2012).
[10] Google Cloud Platform. Cloud Run. https://cloud.google.com/run, 2020.
[11] Harchol-Balter, M. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013.
[12] Jonas, E., et al. Cloud programming simplified: A berkeley view on serverless computing. Tech. rep., UC Berkeley, 2019.
[13] Lloyd, W., et al. Serverless computing: An investigation of factors influencing microservice performance. In *IC2E 2018* (2018).
[14] M. Shahrad et al. Serverless in the wild:characterizing and optimizing the serverless workload at a large cloud provider. In *arXiv* (2020).
[15] National Laboratory for Applied Network Research. *Anonymized access logs*, 1995.
[16] Ousterhout, K., et al. Sparrow:distributed,low latency scheduling. In *SOSP'13* (2013).
[17] PACE Lab, Stony Brook University. ENSURE Serverless Scheduling. https://github.com/PACELab/ENSURE, 2020.
[18] Pu, Q., et al. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *NSDI'19* (2019).
[19] Shahrad, M., et al. Architectural implications of function-as-a-service computing. In *Proceedings of the IEEE/ACM MICRO* (2019).
[20] Shankar, V., et al. numpywren: serverless linear algebra. *CoRR* (2018).
[21] Singhvi, A., et al. Archipelago: A scalable low-latency serverless platform. In *arXiv* (2019).
[22] Suresh, A., et al. Fnsched: An efficient scheduler for serverless functions. In *WOSC 19* (2019).
[23] Verma, A., et al. Large-scale cluster management at google with borg. In *EuroSys '15* (2015).
[24] WAND Network Research Group. WITS: Waikato Internet Traffic Storage. http://www.wand.net.nz/wits/index.php, 2019.
[25] Wang, L., et al. Peeking behind the curtains of serverless platforms. In *USENIX ATC'18* (2018).