Escra: Event-driven, Sub-second Container Resource Allocation

Greg Cusack, Maziyar Nazari, Sepideh Goodarzy, Erika Hunhoff,
Prerit Oberai, Eric Keller, Eric Rozner, Richard Han
University of Colorado Boulder, Boulder, CO USA

***Abstract*—This paper pushes the limits of automated resource allocation in container environments. Recent works set container CPU and memory limits by automatically scaling containers based on past resource usage. However, these systems are heavy weight and run on coarse-grained time scales, resulting in poor performance when predictions are incorrect. We propose Escra, a container orchestrator that enables fine-grained, event-based resource allocation for a single container and distributed resource allocation to manage a collection of containers. We find resource allocation can easily adapt to sub-second intervals within and across hosts, meaning operators can cost-effectively scale resources without performance penalty. Escra integrates into and manages two types of containerized applications – microservices and serverless functions. In microservice environments, our evaluations show fine-grained and event-based resource allocation can reduce application latency by up to 96.9% and increase throughput by up to 3.2x when compared against the current state-of-the-art. Escra can increase performance while simultaneously reducing 50th and 99th%ile CPU waste by over 10x and 3.2x, respectively. In serverless environments, according to our evaluation, Escra can reduce an application's CPU reservations by over 2.1x and memory reservations by more than 2x while maintaining similar end-to-end performance.**

I. Introduction

Containerized infrastructure is quickly becoming a preferred method of deploying applications. The light-weight nature of containers coupled with rich orchestration systems enable a new way to design automated operations that are integrated with development workflows. In these deployments, per-container resources limits are used to prevent interference between containers and unchecked resource usage.

Setting container resource limits is a trade-off between application performance and efficient use of underlying system resources. When resource limits are set low to prioritize efficient resource use, applications will experience an increased number of CPU throttles and out-of-memory (OOM) events. Throttles slow processing and OOMs kill containers which results in degraded application performance. When resource limits are set high to prioritize application performance, resources are underutilized which increases deployment cost. Developers pay the cost when cloud providers charge tenants based on resources reserved [1]–[3]. Cloud providers pay the cost in cases where developers are charged by usage, such as in serverless computing [4]–[7].

Due to this trade-off, setting accurate limits is important. In practice, it is also difficult [1], [8]–[11][1]. Using profiling to characterize application resource requirements will only result in accurate estimates if there is a representative workload. As workloads are often dynamic, the resources needed will change over long timescales (diurnal patterns, gradual changes in application popularity, etc.) and short timescales (bursts, failures of coupled systems, etc.). Since creating an accurate estimate of resource requirements is so complex, developers and operators often resort to over provisioning resources. This results in underutilized deployments which are often observed by datacenter operators [10], [12]–[14].

Recent work has addressed some of these challenges [1], [9] by leveraging machine learning to predict future needs and then automatically scaling container resource limits based on those predictions. These works eliminate the developer burden of setting resource limits but are constrained to using coarse-grained intervals (e.g., several minutes) to set resource limits. Coarse-grained intervals are required because the system has to learn enough information to be able to predict resource use. This is a poor fit for some workloads with short-lived containers, such as in serverless systems [15]–[18]. Coarse-grained intervals also increase the odds of misprediction since the dynamics of applications can change throughout an interval. Thus, these works still contend with the performance and efficiency trade-off.

In this paper, we argue the performance and efficiency trade-off can be avoided by using a *fine-grained, event-based resource allocation* scheme. To this end, we introduce *Escra*: a fine-grained, event-based resource allocation infrastructure for single containers and distributed resource allocation capable of managing containers' resources across multiple nodes. We find resource allocation can easily adapt to sub-second intervals within and across hosts, allowing datacenter operators to cost-effectively scale and assign resources without performance penalty. This scheme has numerous benefits. Instead of a container being killed when it reaches an OOM event, an *event-based* system can catch the event and scale the container dynamically. Instead of making conservative allocations in order to avoid performance degradation over coarse-grained time intervals, a *fine-grained* system can always aim to right-fit allocations to current resource demands, and can quickly react to instances of CPU throttling.

Escra's design consists of a logically centralized controller that administers resource allocations to containers across servers. Each server is instrumented with kernel hooks and runs an agent process to apply resource decisions and report container usage to the controller. A *Distributed Container* abstraction enforces per-application resource isolation and allows

[1]The aggregate CPU utilization at Twitter is <20% but the reservations reach up to 80%. Memory utilization is only slightly better at 40-50% but the reservations still greatly exceed the usage [10].

containers belonging to the same tenant to share compute resources across hosts on the order of milliseconds. The contributions of our work are as follows:

- We expose fine-grained telemetry data from Linux's Completely Fair Scheduler (CFS) [19]. This allows Escra to quickly track and react to actual resource needs, resulting in both high performance (low latency and high throughput) *and* low cost (minimal slack).
- We implement event-based memory scaling, which allows Escra to increase a container's memory upon an OOM event rather than allow the container to be killed.
- We show Escra is effective by comparing slack, latency, and throughput performance to recently proposed systems. We reduce application latency by up to 96% while increasing throughput up to 3.2x over a state of the art container orchestrator. These low latency and high throughput rates are achieved while simultaneously reducing the median CPU and memory slack by over 10x and 2.5x, respectively. We show the overhead from the central controller is minimal.
- We show Escra reduces slack and CPU/memory reservations in serverless applications without increasing application latency, potentially reducing cost to both the developer and the infrastructure provider.

II. Related Work

Current container orchestration systems (Kubernetes [20], Borg [21], Mesos [22]) set static container resource allocations. Here we present recent works that instead dynamically scale containers and discuss the limitations of these systems.

Vertical Pod Autoscaler (VPA) VPA is a Kubernetes project that implements automated container scaling through a threshold-based scaling mechanism [23]. VPA sets a target resource utilization and an upper and lower bound on that utilization. When the container usage hits the upper threshold, VPA scales the container up. When the lower bound is hit, VPA scales the container down. VPA also has the capability to enforce per-application limits via Resource Quotas [24]. A resource quota is a hard resource limit on the aggregate compute usage across all or a subset of deployments or services in a Kubernetes namespace.

Limitations of VPA VPA sets the upper and lower limit scaling bounds far apart. Since scaling a container requires a container restart, VPA only scales a container at most once per minute. The loose scaling-bound limit and infrequent container scaling results in high slack which translates to decreased cost-efficiency.

Autopilot Autopilot is a proprietary Google project that addresses the low cost-efficiency of static container deployments [1]. Autopilot runs a control loop that collects both per-second and five minute aggregated usage data from each container, analyzes it, and then makes a prediction on whether or not a container needs to be scaled. Autopilot uses machine learning predictions to scale container limits as frequently as every five minutes.

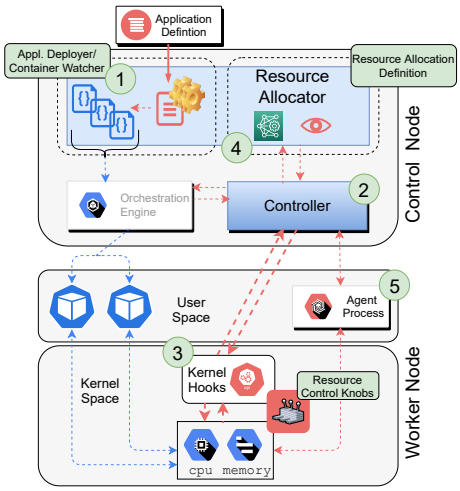


Fig. 1: Escra Architecture

Limitations of Autopilot While Autopilot provides an automated mechanism to set limits, it does so at coarse-granularity which causes cost-efficiency and performance issues for two reasons. First, Autopilot's heavyweight algorithm and periodic control loop prevent it from quickly responding to changes in workloads. As a result, resource predictions are forced to at least match the maximum predicted usage over the next allocation period (Autopilot uses a default 5-minute period). This leads to unnecessary slack. Second, because Autopilot relies solely on prediction, it is unable to correct inaccurate predictions even when resources are available. Inaccurate predictions can cause unnecessary OOMs and CPU throttles.

Firm Firm also uses machine learning to improve containerized application performance and cost-efficiency [9]. While Firm does attempt to minimize CPU reservations, Firm's primary objective is to reduce service-level objective (SLO) violations. Firm minimizes SLO violations by intelligently multiplexing compute resources to optimize the critical path of an application. Firm is similar to Autopilot because it does not require a pod restart to scale container CPU resources and can update container limits automatically.

Limitations of Firm Firm does not implement seamless or automatic *memory* scaling, requiring users to set static limits. Firm shares Autopilot's limitations regarding performance and cost-efficiency issues as both frameworks feature a coarse-grained, ML-based feedback loop.

III. Introducing Escra

Escra is a container resource allocation system that achieves high performance, cost-efficiency, and strong isolation. Escra automatically scales containers in a fine-grained manner, while providing strong isolation via a new abstraction called a Distributed Container. A Distributed Container allows containers belonging to the same tenant to dynamically share resources across multiple compute nodes while capping the overall aggregate resource usage for a given application or tenant.

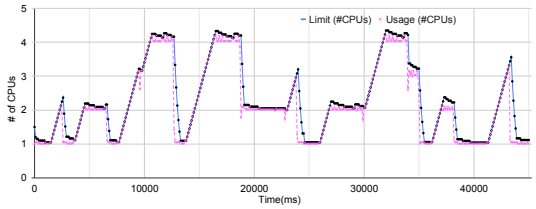


Fig. 2: Escra's CPU tracking ability under a dynamic workload

Figure 1 shows a high-level view of the four key components in the Escra architecture. The Application Deployer and Container Watcher (①) take a set of YAML files describing a set of Kubernetes deployments, services, and containers. The Application Deployer interfaces with the Kubernetes API to deploy containers. The Container Watcher monitors Escra containers and enables newly-deployed containers to start streaming fine-grained telemetry to the Controller. The logically-centralized Controller (②) handles the unique, fine-grained telemetry sent from the kernel via kernel hooks on workers (③). These kernel hooks obtain fine-grained scheduler data that is not available in user-space. The centralized controller model has been shown to be scalable, as evidenced by production systems for datacenters [25] and geo-distributed network services [26]. The Resource Allocator (④) ingests telemetry from the Controller and makes per-container resource allocation decisions. Finally, similar to Kubernetes's per-node kubelet [20], an Agent is run on each host (⑤). The Agent handles resource updates sent from the controller and can dynamically scale both CPU and memory container limits without restart on the order of 100s of microseconds. A complete description of Escra's architecture follows in Section IV. In this section, we describe Escra's unique ability to make scaling decisions on a fine-grained timescale and in an event driven manner.

To illustrate the benefits of fine-grained container resource allocation, we deployed and loaded a container with sysbench [27], saturating 1-4 CPUs at any one time. The trace of the application execution with Escra is shown in Figure 2. Escra is able to track the exact needs on a very short timescale by proactively adjusting resources as well as relatively adjusting resources based on information collected during each CPU scheduling period and at OOM events. The implication of this fine-grained right-sizing is that Escra (1) significantly reduces slack and (2) simultaneously improves performance as applications are being allocated the resources they need rather than being throttled or killed due to OOMs. The remainder of this section provides further insights into how Escra achieves fine-grained resource allocation.

Per-period CPU Telemetry and Dynamic Reallocation Fine-grained telemetry data is required to minimize slack via fine-grained resource allocation. Our initial analysis of systems that aggregate CPU and memory data (cAdvisor [28], Prometheus [29], and Kubectl [20], etc.) found they suffer from inefficiencies stemming from reliance on coarse-grained timescales. Allocating resources quickly is not useful if based on usage data that is stale or aggregated at insufficient levels. Our goal is to obtain near-instant usage information so Escra never operates on stale data. In order to obtain fine-grained CPU data, we add kernel hooks into Linux's Completely Fair Scheduler. Upon deployment of each container, the Agent process creates a kernel socket for the container to use to report its metrics to the controller. To implement fine-grained telemetry, containers report their per-period runtime statistics to the controller at the end of each period. The telemetry data consists of the container's cgroup ID, whether the container was throttled in the last period, and the amount of unused runtime in that period.

The Resource Allocator ingests raw container metrics and uses two windowed statistics to track unused runtime and the number of throttles. The Resource Allocator uses these statistics to update per-container limits as often as every 100ms. The goal is to proactively update limits in order to keep the container limits just above container usage at all times. We seamlessly update a container's CPU quota through RPCs to the container's host Agent Process, similar to [9].

Reactive Memory Reclamation and Reallocation upon OOM Events Escra monitors container memory usage and can seamlessly scale memory limits via two custom system calls that hook into Linux's memory cgroup structure.[2] One unique opportunity of fine-grained allocation is the ability to react to OOM events. To achieve this, a kernel hook is added in Linux's memory allocation function, `try_charge()`, to catch a container after it exceeds its memory limit and right before it gets OOMed. We add this hook to combat inaccurate predictions within autoscalers. For example, VPA [23] and Autopilot [1] scale containers at most once a minute and once every five minutes, respectively. There is a chance a container will OOM in between allocation decisions. Our kernel hook lets a container contact the controller for more memory prior to getting killed. While this is a reactive mechanism for memory scaling, the request lookup penalty is orders of magnitude faster than a container crash and restart.

One exciting aspect of this OOM-preventing kernel event is the Resource Allocator can determine how to allocate additional memory resources depending on the state of the node and the application. If there is available memory on the node, the Allocator can simply scale the needy container up. If the node is under memory pressure, the Controller can launch an aggressive memory reclamation process that reclaims memory from containers on the node with high slack. Not only will this free up memory for the container in need, but it also increases node utilization, reduces slack, and improves cost-efficiency.

Proactive Periodic Memory Reclamation In order to reduce memory slack, the Escra Controller periodically contacts the Escra Agent on each worker node, asking the Agent to reduce the memory limits of each container on the Agent's node.

[2]Docker supports seamless container scaling [30], but Kubernetes does not.

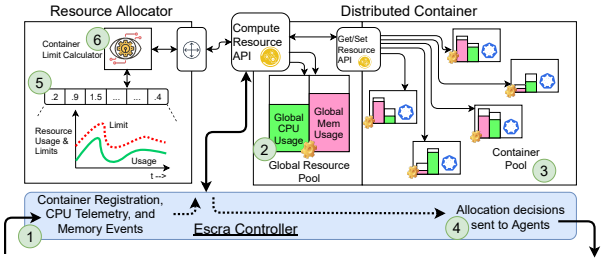


Fig. 3: Escra Controller, Resource Allocator, and Distributed Container

The Agent checks the usage and the limit of each container it manages. If the limit of a container exceeds the usage of the container by more than Δ bytes, then the Agent shrinks the container's memory limit such that the memory limit minus the memory usage equals Δ bytes. Each Agent then reports back the total reclaimed memory from its containers to the Escra Controller. The Controller can then give the reclaimed memory to other containers experiencing memory pressure.

IV. Escra Architecture

This section describes the architecture of Escra, our container orchestrator built with Kubernetes, that implements (i) automated container limit settings, (ii) seamless container scaling, (iii) fine-grained resource allocation, and (iv) dynamic, per-tenant resource sharing and collective resource limits enforced at runtime. Escra implements these features using fine-grained telemetry, event-based memory scaling, aggregated application-wide resource limits, and a centralized Controller and Resource Allocator.

A. Application Deployer & Container Watcher

The Application Deployer ingests a Distributed Container configuration as a set of YAML files (Figure 1, ①) describing a set of containers, and maximum CPU and memory limits. The maximum CPU and memory limits represent the limit on the aggregate usage of all containers in the application (Figure 3, ②). Prior to deploying the containers via Kubernetes, the Deployer sends the global application limits to the Controller. This lets the Resource Allocator (Figure 1, ④) know the total maximum usage of the containers in the deployment. Once the Deployer sends the application limits to the Controller, the Controller is ready to accept network connections from each container.

Initial limits are set to bootstrap containers when they first deploy but these limits will be changed by the Controller at runtime. The Deployer sets each container's CPU and memory limit to: $\frac{global\_cpu\_limit}{\#\ containers}$ (1) $\frac{global\_mem\_limit * \sigma}{\#\ containers}$ (2) where σ is a configurable parameter representing the percentage of the global application memory limit to be withheld for containers that experience OOM events.

The Container Watcher integrates with Kubernetes to detect container creation. Upon detection, the Watcher notifies the Agent (Figure 1, ⑤) running on the same host of the existence of a new container.

B. Kernel Hooks

Escra uses kernel hooks to enable fine-grained telemetry and trap OOMs. After an Agent is notified that a new container has deployed, the Agent invokes a custom syscall that carries out three tasks, each implemented via kernel hooks (Figure 1, ③). First, the syscall creates a TCP kernel socket to message the Controller (Figure 1, ②) and informs the Controller of the container's existence. The per-container TCP kernel socket will persist for the life of the container. Once the Controller registers the new container, it updates the container's CPU and memory limit based on the global application limits and current application resource use.

Next, the syscall modifies the container's underlying Linux CPU and memory cgroup structures to enable fine-grained telemetry and event handling. For CPU, the syscall hooks into Linux's Completely Fair Scheduler to extract runtime data to stream to the Controller. At the end of each period, the hook writes the container's cgroup quota, unused runtime (the `runtime` variable in the CFS Bandwidth kernel structure), and whether the container was throttled in the last period into a shared FIFO buffer in the kernel[3].

After the hook finishes writing its data into the buffer, the cgroup's runtime is refilled and the next period begins. Per-container kernel threads consume statistics from the FIFO queue and send the queued CPU statistics over UDP to the Controller. Along with the container's quota and runtime remaining, the CPU statistic message also includes a tag letting the Controller know what container the incoming statistic refers to. The hook will report statistics once per-period for the life of the container.

To handle OOM events, the syscall adds a kernel hook in the memory cgroup structure (`mem_cgroup`) for the container. If a container exceeds its memory limit, before it is killed this kernel hook forwards the OOM event to the Controller over the existing TCP kernel socket that was previously used during container initialization. If memory is available in the global application pool, the container can increase its memory limit and continue running.

C. Controller

The Controller brings all of the system components together and coordinates their interactions. Figure 3 shows a more detailed view of the Controller, Resource Allocator, and the Distributed Container abstraction.

When containers register themselves with the Controller upon deployment, the Controller creates a logical container object and adds it to a pool of the other Escra containers within the application (Figure 3, ②). The logical pool of Escra containers is used to maintain an updated view and status (resource usage, limit, etc) of the containers it is managing.

[3]Note that per-period unused runtime is not available in userspace and while one could interpret similar data from the `cpuacct` cgroup subsystem, `cpuacct` was never designed for accuracy and was initially designed as a way to showcase the capabilities of cgroups [31].

Once all containers are deployed and registered with the Controller, the Controller becomes responsible for several additional tasks. The Controller is responsible for launching a periodic memory reclamation process, handling fine-grained telemetry data from all containers, and handling memory requests from containers under memory pressure (Figure 3, ①). The Controller is also responsible for carrying out allocation decisions made by the Resource Allocator (Figure 3, ④). Note, the Controller is *not* responsible for CPU and memory allocation decisions.

The Controller launches a periodic reclamation loop that reclaims excess reserved but unused memory from each container in the cluster. Every 5 seconds, the controller sends a request to each Escra Agent, requesting the Agent to reduce the memory limit of each Escra container, $C(i)$, and send back the amount the container was resized by γ. This resized value is the amount of memory reclaimed from that specific container. The reclaim process is as follows. The Agent reduces the memory limit on a container if:

$$C(i)_l > C(i)_u + \delta$$

where $C(i)_l$ and $C(i)_u$ are the container's memory limit and usage, respectively, and δ is a tunable parameter that represents the memory reclamation "safe margin." If the condition above is satisfied, the container's limit is updated via: $C(i)'_l \leftarrow C(i)_u + \delta$, otherwise, the container limit is left unchanged. We empirically set our safe margin to 50 MiB. The amount of reclaimed memory is measured as:

$$\gamma \leftarrow C(i)_l - C(i)'_l$$

where $C(i)'_l$ is the resized container limit and γ is the amount of reclaimed memory. Therefore, for each container that is resized, the Agent passes back to the controller γ bytes of memory. The Escra Controller then adds γ bytes of memory into the global memory pool via: $global\_mem\_limit \leftarrow global\_mem\_limit + \gamma$. The Controller passes all CPU telemetry data, memory requests, and reclaimed memory to the Resource Allocator.

D. Resource Allocator

The Resource Allocator is the lightweight decision-making component that determines the containers whose resources should be allocated to or reclaimed from. The Resource Allocator is composed of three key components. First, it has a global resource pool for both CPU and memory. For CPU and memory, it keeps track of the maximum application limit ②, the total allocated resources, and the total unallocated (or available) resources ⑥. Second, the Allocator collects fine-grained CPU telemetry data from the Controller and uses a lightweight algorithm to make decisions on whether or not to scale up or scale down individual container CPU limits ⑤. Third, the Allocator consumes *out-of-memory* events sent from the Controller and, based on the globally available memory, increases the memory limit of memory-pressured containers. If a container is not using up to its allocated resource limit, the Resource Allocator will take away those excess resources. However, the Allocator is designed to quickly give back resources to containers when needed.

1) Dynamic CPU Allocation

The CPU allocation algorithm consumes CPU telemetry data sent from each container across all nodes in order to share CPU allocations across nodes and remain under the maximum CPU limit (Ω_l). At the end of the container's running period t, the Resource Allocator consumes a runtime statistic from the Controller. The runtime statistic for a container i during period t ($C(i)[t]$) includes the container's quota ($C(i)_q[t]$) in ms, the amount of unused runtime ($C(i)_q[t] - C(i)_u[t]$) in ms, and whether the container was throttled ($C(i)_{th}[t]$) in the last period t.

The Resource Allocator uses two sliding windowed statistics that track (i) the excess runtime a container has at the end of each period and (ii) if a container was throttled during the last period. Based on these windowed statistics, the Resource Allocator determines whether a container needs or has excess CPU runtime and updates container quotas. A container's quota (or limit) during period t is increased if $C(i)_{th}[t] = 1$ and will be increased for the following period $t+1$ via:

$$C(i)_q[t+1] = C(i)_q[t] + \frac{\sum_{t=0}^{n} C(i)_{th}[t]}{n} * (\Omega_l - \sum_{i=0}^{\lambda} C(i)_q[t]) * \Upsilon$$

where $\frac{\sum_{t=0}^{n} C(i)_{th}[t]}{n}$ is the windowed statistic measuring the average number of throttles over the last n container periods, $\sum_{i=0}^{\lambda} C(i)_q[t]$ is the unallocated CPU runtime for the entire application, λ is the number of containers in the application, and Υ is a tunable parameter that affects the rate at which a container's CPU quota is scaled.

A container's quota during period t is decreased if $C(i)_q[t] - C(i)_u[t] > \gamma$, where γ is a tunable parameter that adjusts how quickly containers' quotas should be scaled down. A container's quota for period $t+1$ is scaled down via:

$$C(i)_q[t+1] = C(i)_q[t] - \frac{\sum_{t=0}^{n} (C(i)_q[t] - C(i)_u[t])}{n} * \kappa$$

where $\frac{\sum_{t=0}^{n} (C(i)_q[t] - C(i)_u[t])}{n}$ is the windowed statistic measuring the average runtime remaining during the last n container periods, and κ is a tunable parameter that affects the rate at which container's are scaled down. We empirically found that systems with high variance in CPU usage between periods performed better with a larger Υ and a smaller γ and κ.

2) Dynamic Memory Allocation

Here we dive into the Resource Allocator's algorithm for handling *out-of-memory* events received from containers and ensuring the proper sharing of memory resources across an application. The Allocator allocates additional memory to containers under memory pressure and reclaims memory from containers that are not using the memory allocated to them.

The Allocator consumes *out-of-memory* events that are sent from a container just before the container is killed for

exceeding its memory limit. Upon receiving an *out-of-memory* event from a container $C(i)$, the Allocator checks if there is unallocated memory available in the global resource pool. If there is no available memory (all global memory has been allocated to containers), the Allocator tells the Controller to reclaim unused memory from other containers in the application (described in Section IV-C). We implement *out-of-memory* events in Escra this way to avoid killing a container for exceeding its memory limit when available memory in the application exists.

If the Controller is able to reclaim memory from other containers in the application, the Allocator will allocate a fixed number pages of memory to $C(i)$ by invoking the Agent to update the $C(i)$'s memory limit. If the Allocator is unable to reclaim any memory from other containers, $C(i)$ is killed by the operating system (as is standard).

E. Integrating Escra With Serverless Frameworks

Escra's fine-grained approach to resource allocation is well suited to serverless environments due to the high degree of multitenancy in serverless systems as well as the short-lived nature of serverless functions. Since functions have short execution times (90% execute in under 1 minute) [15], coarse-grained resource management solutions are insufficient for serverless workloads. Since Escra is fine-grained and designed for use with containers, it is compatible with serverless frameworks that use containers to isolate serverless functions.

We choose OpenWhisk [32], an open-source serverless platform, as an example to illustrate how Escra may be integrated with serverless frameworks. In our configuration, OpenWhisk is deployed via Kubernetes and serverless functions (termed *user actions*) are run in pods. Each pod is deployed as part of the Kubernetes `openwhisk` namespace. Treating OpenWhisk as a single application, one can use the `openwhisk` namespace and invoker `containerPool` memory limit to set global application memory in Escra. We modified pod affinity to ensure OpenWhisk infrastructure was deployed on dedicated infrastructure nodes so there would be no resource contention between architectural components and user actions. There is no global invoker CPU limit in OpenWhisk however one can set memory and CPU to scale linearly which allows one to indirectly set a global CPU limit. Escra does not delay container creation in OpenWhisk because the connection between a container and the Controller does not block the container from beginning to execute. Escra already interfaces with Kubernetes so no further modifications are needed for a minimal integration that allows all user action pods to benefit from resource sharing and reclamation.

V. Implementation

We implement Escra in a total of 14.1k SLOC. The Controller and Resource Allocator are written in C++ and utilize gRPC to communicate with the Deployer, Watcher, and Agents (all written in Go). The Deployer sits on top of Kubernetes and integrates with the Kubernetes deployer API via client-go [33] to deploy Escra containers. Docker is used as the underlying container runtime.The Container Watcher integrates with the Kubernetes Work-queue API and communicates with the Agent via gRPC as well.

Escra worker nodes run on a custom Linux kernel based on the Linux kernel 4.20.16. The custom kernel includes a hook in the CFS cgroup subsystem and in the memory management subsystem. The kernel also includes a custom message structure used for CPU telemetry reporting and memory requests from the Controller. The rest of the kernel modifications include approximately 1,500 SLOC spread across six kernel modules that implement limit resizing functionality and the CPU telemetry implementation.

VI. Evaluation

The goal of Escra is to automatically and seamlessly achieve high performance, cost-efficiency, and isolation. As fine-grained allocation is a key capability of Escra, the first goal of our evaluation is to show how much Escra's highly reactive decision making process is able to improve both performance and cost-efficiency in comparison to common practice (static allocation) and a state-of-the-art system (Autopilot). Our second goal is to show how Escra can reduce the overall reservation requirements for serverless applications, while maintaining application performance; this has the potential to reduce cost for both the application owner and the infrastructure provider.

A. Experimental Setup

Experiment clusters are created using Cloudlab [34] resources consisting of a control node and worker nodes. Along with the default Kubernetes components, the control node runs the Escra Deployer, Watcher, Controller, and Resource Allocator. Each worker node runs an instance of the Escra Agent.

Microservice Benchmark Applications We first evaluate Escra on a set of four microservice applications running across three worker nodes and one control node where each node consists of two Intel Xeon Silver 4114 10-core 2.20 GHz CPUs, 192GB of ECC DDR4-2666 memory, and a dual-port Intel X520-DA2 10Gb NIC. We set κ to 0.8, γ to 0.2, and Υ to 20 in the Resource Allocator for all experiments.

The microservice applications represent a set of four interactive, real-world benchmarks: (1) *Media Microservice* [35] - 32-containers - a microservice similar to IMDB where users can search, review, rate, and add films, (2) *Hipster Shop* [36] - 11 containers - an online shopping microservice consisting of standard browsing and purchasing of various items, (3) *Train Ticket* [37] - 68 containers - a microservice that simulates a train ticket booking service consisting of searching, booking, modifying tickets, (4) *Teastore* [38] - 7 containers - simulates an online teastore where users can browse and purchase hundreds of various teas.

For each microservice experiment we load the microservice with one of four workload distributions: an Alibaba datacenter trace [39], a bursting request rate, an exponentially distributed request rate, and a fixed request rate. The Fixed load sends re-

quests at a constant 400 requests per second. The Exponential (Exp) workload sends requests in an exponential distribution with $\lambda = 300$. The Burst workload sends a fixed 50 req/s with an additional 10s exponential burst of requests where $\lambda = 600$ every 20 seconds. Finally, the Alibaba workload is sped up by 10x and sends requests at rates anywhere from 56-548 req/s.

Evaluation Metrics Below is a list of metrics used in this section (derived from [1]) and their respective definitions:

- **Absolute Slack** - The container's CPU or memory limit minus the container's CPU or memory usage.
- **Application Throughput** - Measured in successful requests/sec.
- **Application 99.9%ile Latency** - Measured as the 99.9%ile end-to-end latency.

Autopilot Implementation Autopilot is not open-source so we implemented a recreation of Autopilot's [1] ML recommender to compare against Escra. The Autopilot ML recommender is inspired by a multi-armed bandit problem in which an agent tries to use the best set of arms to maximize the total reward gain over time. Some parameters used in the Autopilot algorithm are manually tuned by their engineers (w_o, w_u, etc). As they did not specify what values they used for these parameters, we tuned them to values that resulted in the best performance.

Note that Autopilot defaults to a 5-minute update period: deciding whether to update a container's limits every 5 minutes. We tested Autopilot's update period at 60s, 30s, 10s, and 1s and saw finer-grained update periods achieve better performance. Hipster-shop's throughput with Autopilot at 1, 10, 30, and 60 second update periods degrades from 422 req/s to 382 req/s to 279 req/s to 108 req/s, respectively. While we don't know how practical it is to run Autopilot's ML at that granularity at scale, we show comparisons against 1s intervals as a best case for Autopilot.

B. Performance - Cost-Efficiency Trade-off

Intuitively, there exists a trade-off in performance and cost-efficiency. One can allocate a large amount of resources to eliminate any possible performance penalty (measured in throughput and latency), but this leads to poor cost-efficiency (measured in terms of slack). In contrast, we can significantly under-allocate resources and improve the cost-efficiency, but we pay the price in performance. We further examine this trade-off in the context of both common practice (Static allocations) and state-of-the-art (Autopilot), and illustrate that Escra achieves better performance and cost-efficiency than each system, and that each system compromised on one of the metrics.

We estimated the resources needed for the Media Microservice from the Deathstar Benchmark [35]. We profiled each container and measured maximum CPU and memory usage. We then ran the application in an underutilized (limits set at 0.75x the profiled max), a best-estimate (set at 1.0x), and a safe buffer (set at 1.5x) case. For each case, we measure the end-to-end performance (latency and throughput) and slack (CPU cores allocated minus cores used, and MiBs allocated minus MiBs used). As expected, performance increased (i.e., latency decreased and throughput increased) with more resources allocated; however, slack (resource wastage) also increased. We find the 1.5x allocation level illustrates a sufficient buffer and as such, use that setting for evaluating the trade-offs in comparison to Autopilot and Escra.

App Comp.	Avg. Δ Latency	Avg. Δ Tput.	Avg. Δ 50% CPU Slack	Avg. Δ 99% CPU Slack	Avg. Δ 50% Mem. Slack	Avg. Δ 99% Mem. Slack
Static vs. Escra	38.0%	25.4%	81.3%	74.2%	55.0%	95.9%
Autopilot vs. Escra	36.1%	54.5%	78.3%	78.6%	26.7%	68.9%

TABLE I: Average Performance increase and Average Slack reduction for both CPU and memory between Static and Escra and Autopilot and Escra. Escra improves performance, while significantly reducing slack

To evaluate, we deployed each microservice and loaded them using the workload generation-based benchmarking tool wrk2 [40] with the four different workloads. Each application is evaluated when managed by 1.5x Static Limits, Autopilot and Escra. This setup allows us to measure both the latency and throughput rate to quantify the performance in each approach, and the slack to quantify the cost-efficiency of each approach. Figure 4 shows the resulting *change* in latency and *change* in throughput between Autopilot and Escra and Static Limits and Escra for all four applications and workload distributions. Table I summarizes our results and is broken down in the subsequent sub-sections.

C. Static Allocation vs. Escra

We first look at the change in both latency and throughput between a statically allocated application and an application deployed with Escra. Table I show that on average, Escra decreases latency by 38% and increases throughput by 25.4% compared to statically allocated applications. Escra can achieve these performance numbers with an average 50%ile and 99%ile CPU slack improvement of 81.3% and 74.2%, respectively. Escra also decreases 50%ile and 99%ile memory slack by 55% and 95.9%, respectively.

In an ideal world, we would not see a performance improvement from Escra over a statically deployed application allocated 1.5 times the peak measured resource usage; the static deployment would never experience any throttles or OOMs. However, this result is a testament to how difficult it is for developers to set resource limits on containers [1], [8]–[11]. Not only is it hard to profile containers, since you never know what the workload rate is truly going to be, but also the tools to measure resource usages (especially for CPU) tend to aggregate over seconds to minutes, smoothing out usage spikes [28], [29], [41].

We break down Train-Ticket with Fixed and Teastore with Alibaba experiments in the following paragraphs to help illustrate Escra's ability to achieve both high performance and cost efficiency.

Train-Ticket with Fixed Workload Figure 4 shows that

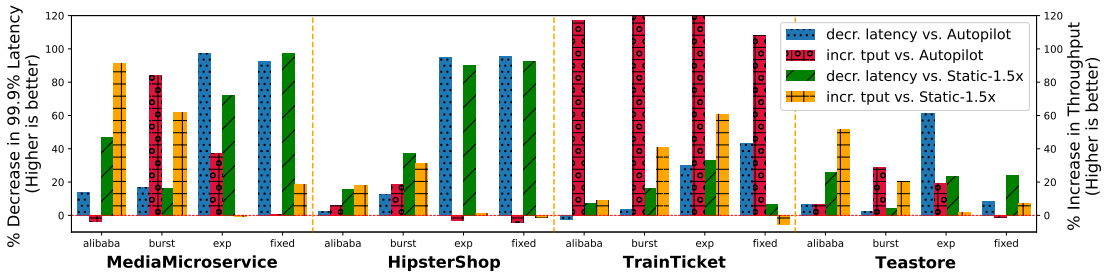
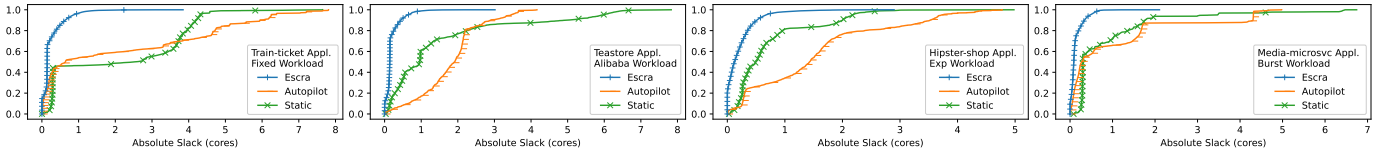
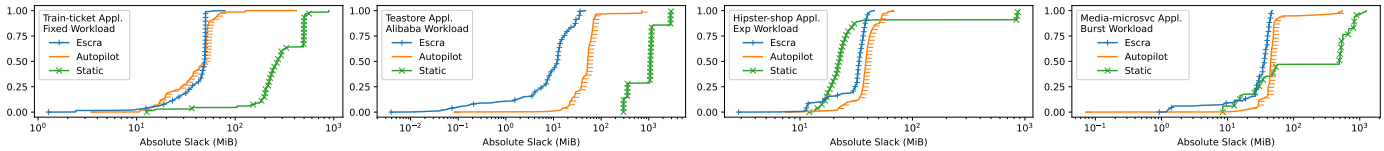


Fig. 4: Change in 99.9% Latency and Throughput between Autopilot, the 1.5x measured peak static allocation and Escra. Note: Train Ticket with Burst and Exp workloads experienced a throughput increase of 134% and 324% respectively but are cut off at the top of the figure



(a) Train-Ticket - Fixed (b) Teastore - Alibaba (c) Hipster-Shop - Exp (d) Media-Microservice - Burst

Fig. 5: CPU slack CDFs comparing Escra, Autopilot, and a statically deployed system across the Media, Hipster Shop, Train Ticket, and Teastore Microservices with various workloads



(a) Train-Ticket - Fixed (b) Teastore - Alibaba (c) Hipster-Shop - Exp (d) Media-Microservice - Burst

Fig. 6: Memory slack CDFs comparing Escra, Autopilot, and a statically deployed system across the Media, Hipster Shop, Train Ticket, and Teastore Microservices with various workloads. Note the x-axis is a log scale

Train-ticket with Fixed performs slightly worse with Escra than with the Static allocation, seeing a 5.5% decrease in throughout. Looking at the slack in Figures 5a and 6a, 50% of the time, the Static allocation has over 2.5 cores of CPU slack and 256MiB of memory slack. At the same time Escra has a 50% CPU slack of 0.14 cores (a 17.9x improvement) and memory slack of 49MiB. This experiment shows the tradeoff the static deployment makes, sacrificing significant cost-efficiency for a slight performance increase.

Teastore with Alibaba Workload Escra improves latency and throughput of Teastore by 25.7% and 51.6%, respectively. Figures 5b and 6b show while Escra is able to increase performance, it can do so while reducing 50%ile and 99%ile CPU slack by over 81% and 74% respectively, while also significantly reducing memory slack.

D. Autopilot vs. Escra

Autopilot aims to reduce slack, but tries to find a balance where it doesn't sacrifice too much performance via an ML-like approach. However, Table I shows on average, Escra decreases latency by 36.1% and increases throughput by 54.5% compared to Autopilot. Table I also shows Escra's average 50%ile and 99%ile CPU slack improvement over Autopilot is 78.3% and 78.6%, respectively. Escra also decreases 50%ile and 99%ile memory slack by 26.7% and 68.9%, respectively. Once again, we dive into two experiments below to further detail how Escra can achieve both high performance and high cost efficiency.

Hipster Shop with Exp Workload In a few cases, Autopilot gets some performance improvements over Escra since it trades for performance gains at the cost of slack. Autopilot increases the throughput of Hipster-shop compared to Escra by 3.16%. However, Figures 5c and 6c show Autopilot over allocates resources, with the median slack greater than 1.43 cores and 20% of allocations over 2.38 cores. For Escra, the median slack is 0.12 cores (an 11.6x decrease) with an 80%ile CPU slack of 0.35 cores.

Media-Microservice with Burst Workload Figure 4 shows Autopilot degrades Media Microservice with Burst throughput and increases its latency. This indicates that Autopilot fails to quickly react to rapid and significant changes in CPU workloads and memory usages, resulting in low slack but higher latency and lower throughput. For the same application and workload, Escra is able to not only increase latency and throughput performance by 16.6% and 84.3%, but also able to reduce slack over Autopilot. Escra has a 99%ile slack less

than 66% of a core and a 99%ile memory slack of 46MiB.

E. Takeaways

Table I, Figure 4, and the four cases above show Escra rarely performs worse than Static allocation and Autopilot, but when it does, the performance degradation is small and the slack savings are significant. When Escra outperforms the Static allocation and Autopilot, Escra does so with significantly reduced slack, proving that Escra is able to achieve both high performance and high cost efficiency. One of the key reasons for Escra's high performance is that Escra is able to greatly reduce OOMs. In all 32 experiments, Escra experienced zero OOMs, while Autopilot had up to 8 OOMs in a single experiment.

F. Serverless

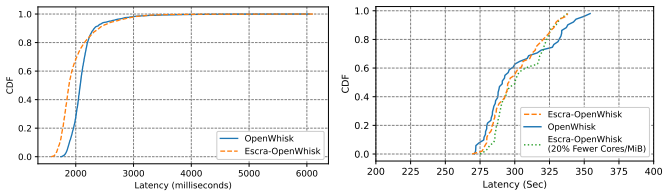
This section shows how Escra integrates with OpenWhisk [32] by benchmarking two applications: ImageProcess and GridSearch. We run ImageProcess with one control node, three worker nodes, and two nodes reserved for serverless infrastructure (i.e., OpenWhisk and databases). The GridSearch application runs with one additional worker node. Each node is composed of two Xeon E5-2650v2 8-core 2.6 Ghz CPUs, 64GB of DDR-3 memory, and a dual-port Intel X520 10Gb NIC. We set κ to 0.8 and γ to 0.2 for both applications and Υ to 35 for ImageProcess and 20 for GridSearch in the Resource Allocator. For both applications OpenWhisk is configured to create each user action pod with 1 vCPU for CPU request and limit, and 256 MiB of memory.

Serverless Benchmark Applications ImageProcess is a single-function application inspired by the image processing application in [42]. The function reads an image from a database, processes image metadata, creates a thumbnail, and writes the thumbnail to the database. Our workload is simple: an ImageProcess request is sent every 0.8 seconds over 10 minutes. We perform four iterations of the experiment for a total of 3k invocations for each test case. At the beginning of each experiment, we ensure there are no ImageProcess pods running (to ensure initial cold starts).

GridSearch is a traditional approach for tuning hyperparameters in classifiers. This batch-like application [43] uses ˜115 serverless function pods to classify an Amazon product review dataset using scikit-learn [44] and tunes the classifier hyperparameters using the GridSearch algorithm. Each function is charged with completing tasks until all 960 tasks are completed. GridSearch uses the Lithops framework [45] for orchestration. We set Lithops's serverless backend to use OpenWhisk and Lithop's storage backed to use a Redis instance.

Evaluation Metrics Below are the metrics used in the evaluation of the serverless benchmarks:

- **Aggregate Limits** - Since it is common in serverless systems to bill based on total usage, and serverless providers have a strong incentive to pack as many functions as possible per server, instead of CPU/memory usage per pod we focus on the aggregate of the containers' CPU and memory limits.



(a) ImageProcess request latency (b) GridSearch application latency

Fig. 7: Serverless latency CDFs

- **Application Latency** - Measured in end-to-end latency per request (ImageProcess) or job (GridSearch)

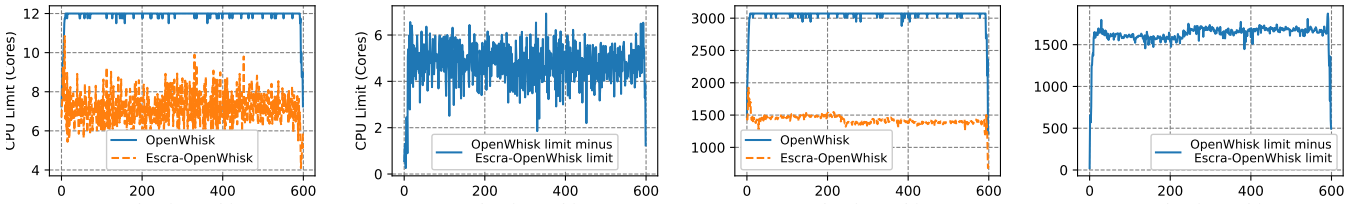
G. OpenWhisk vs. Escra + OpenWhisk

Performance We first consider ImageProcess performance for OpenWhisk alone and OpenWhisk + Escra. Figure 7a shows that, up to the 80th%ile, OpenWhisk + Escra sees modest performance gains over OpenWhisk alone, while the overall 99th%ile latency remains similar for both. The average invocation latency with OpenWhisk + Escra is 1.99 seconds as opposed to 2.12 seconds with OpenWhisk alone. Unlike other applications tested with Escra, ImageProcess requires Escra to handle a variable number of pods as the number of application pods at the start of each benchmark iteration is zero. The similarity in tail latency between OpenWhisk alone and OpenWhisk + Escra indicates that Escra is capable of supporting the dynamic scale-up of application pods needed in serverless environments.

To obtain a CDF of GridSearch's application latency, we ran GridSearch on: (1) OpenWhisk alone, (2) OpenWhisk + Escra with the same amount of resources allocated as in the OpenWhisk alone experiment, and (3) OpenWhisk + Escra with 80% of the application resource limits allocated compared to OpenWhisk alone. We ran the application 50 times for each setup. Interestingly, we observe the same average latency ($\sim$300 seconds) when we run GridSearch by allocating equal resources to OpenWhisk and Escra + OpenWhisk (cases 1 and 2) and only 1% higher average (303 seconds) for case 3, showing Escra can allocate fewer resources to an app and maintain similar performance. As is indicated in Figure 7b, Escra + OpenWhisk outperforms OpenWhisk alone at 99%ile and has lower tail latency.

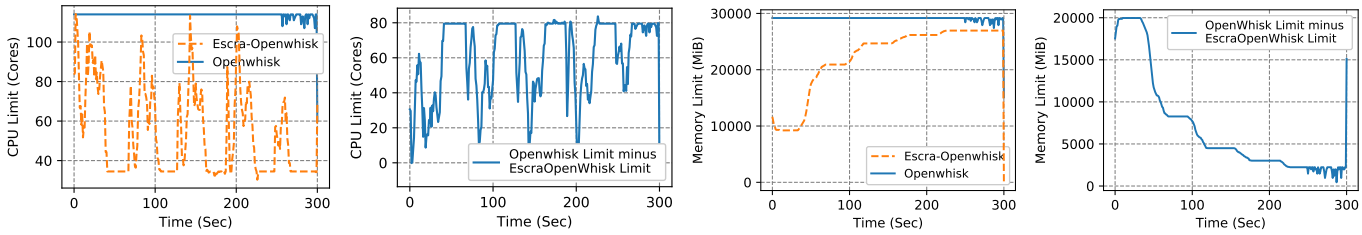
Efficiency Figure 8 shows aggregate CPU and memory limits for OpenWhisk and OpenWhisk + Escra for ImageProcess. On average, OpenWhisk + Escra sets the limit at 7 vCPU whereas OpenWhisk's static allocation scheme results in a limit of 12 vCPU, resulting in a savings of approximately 5 vCPU for identical workloads. For memory, the difference in the limit averages around 1550 MiB.

According to Figure 9, OpenWhisk allocates 113 vCPUs for GridSearch on average. On the other hand, Escra + OpenWhisk was able to reduce the vCPU allocation to 53 vCPUs. For memory, on average, OpenWhisk sets the application ag-



(a) Image Process CPU (b) Image Process CPU Savings (c) Image Process Mem (d) Image Process Mem Savings

Fig. 8: Aggregate Memory and CPU limits averaged per second over four test iterations. We highlight the difference (savings) between OpenWhisk's limits and OpenWhisk + Escra's limits with the savings graphs.



(a) GridSearch CPU (b) GridSearch CPU Savings (c) GridSearch Mem (d) GridSearch Mem Savings

Fig. 9: Aggregate Memory and CPU limits over 5 minutes of running Serverless GridSearch application. We highlight the difference (savings) between OpenWhisk's limits and OpenWhisk + Escra's limits with the savings graphs.

gregate limit to 29087 MiB while Escra + OpenWhisk is able to run the same GridSearch application with an application limit of 22264 MiB. On average, Escra + OpenWhisk saves 60 vCPUs and roughly 7 GiB of memory space.

H. Takeaways

From the ImageProcess and GridSearch benchmark, Escra only minimally effects function latency while providing significant resource savings on static CPU/memory limits. In sum, Escra increased efficiency while maintaining performance. ImageProcess in particular shows that Escra is able to handle a dynamic and rapid increase in number of application pods. Besides, GridSearch application statistics showcases how Escra can help running batch-like, data intensive, long-running applications with fewer resources allocated to them with no harm in latency.

I. Escra MicroBenchmarks and Overheads

Why a 100ms Report Period? Escra uses a 100ms CPU telemetry report frequency for two main reasons. First, 100ms complements the default Linux CFS period. Second, we measured the 99% end-to-end latency performance across various report frequencies every 50ms from 50ms to 200ms. Collecting CPU statistics at the end of every period (100ms) and reporting them directly to the controller resulted in the lowest application latency.

Escra Network Overhead Escra sends usage statistics over UDP to the Controller and the Controller launches RPC calls to the Agent Process to update container limits. The peak network overhead measured for 32 containers is 12.06Mbps. Since the majority of the bandwidth usage comes from the per-container CPU telemetry, we expect the network overhead to scale linearly with the number of containers managed.

Escra CPU Overhead The largest CPU consumers in Escra are the Controller/Allocator and the kernel threads running on each worker node reporting telemetry data. The Controller's biggest CPU consuming process is the memory reclamation process as it relies on cAdvisor API [28], consuming up to 85% of a core. Replacing the cAdvisor functionality with memory limit/usage system calls would greatly reduce the overhead of reading from the cAdvisor API. Without cAdvisor, the Controller and Resource Allocator use 5.7% of a core with 68 containers. For a cloud-scale analysis, we assume a separate Escra controller that manages each application (set of containers). At 5.7% of a core per controller for 68 containers, Escra controllers are able to manage 1,192 containers per core. Assuming 20 cores per node, a collection of Escra controllers can manage up to 23,859 containers/node. Note, as more containers are registered with the Controller, the mean time between subsequent container stats increases sublinearly.

VII. Discussion and Future Work

In this section, we discuss how Escra affects cloud ecosystems and may influence future work.

Multi-tenant Building a fully-fledged cluster management system that takes advantage of Escra remains future work. This paper's contribution shows fine-grained, event-driven resource allocation is possible and performs well. While Escra can effectively reduce slack and increase performance, it remains an open question in how such benefits translate to a large-

scale, complex, multi-tenant system.

Serverless Our initial implementation of OpenWhisk + Escra is naive in several ways: 1) all containers are treated as the same application; the framework would need to modify this to deploy pods in per-tenant namespaces, and 2) the OpenWhisk invoker remains unaware of the actual CPU/Memory limits being used; it would need to be modified to ingest current usage/limits from Escra. We leave these to future work.

Beyond efficiency benefit of using Escra in serverless systems, the Distributed Container abstraction may further be useful for billing and accounting in serverless systems. Many commercial frameworks set global limits on serverless applications by setting an invocation limit (i.e., the maximum number of concurrently running functions). With the Distributed Container abstraction, it would be possible to instead limit based on maximum memory or CPU limits. The study of limits and billing using Distributed Containers in serverless systems is a subject of future work.

VIII. Conclusion

This work illustrates how current orchestration systems fail to achieve both high performance and cost efficient container deployments, typically trading performance (throughput/latency) for cost-efficiency (slack) or vice versa. We motivate the need for a fine-grained and seamless container scaling orchestrator and propose a solution: Escra. Escra uses kernel hooks to generate both fine-grained telemetry and OOM handling events that allow a logically-centralized Escra Controller to allocate resources within 100s of milliseconds. As a result, Escra minimizes CPU slack by over 10x compared to our implementation of Autopilot. Escra also reduces application limits in serverless frameworks, saving more than 2x the CPU and memory resources over a standard serverless deployment. Escra's comparison to static approaches, Autopilot, and OpenWhisk deployments indicates fine-grained container scaling finds the balance between performance and efficiency while maintaining isolation.

References

[1] K. Rzadca, P. Findeisen, J. Świderski, P. Zych, P. Broniek, J. Kusmierek, P. K. Nowak, B. Strack, P. Witusowski, S. Hand, and J. Wilkes, "Autopilot: Workload autoscaling at google scale," in *Eurosys 20*.

[2] "Amazon elastic container service," https://aws.amazon.com/ecs/.

[3] "Azure kubernetes service (aks)," https://azure.microsoft.com/en-us/services/kubernetes-service/#overview.

[4] "Aws lambda," https://aws.amazon.com/lambda/.

[5] "Azure functions," https://azure.microsoft.com/en-us/services/functions/.

[6] "Cloud functions," https://cloud.google.com/functions.

[7] "Ibm cloud functions," https://www.ibm.com/cloud/functions.

[8] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics," in *NSDI 17)*, 2017.

[9] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices," in *OSDI 20)*.

[10] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," *SIGPLAN Not.*, Feb. 2014. [Online]. Available: https://doi.org/10.1145/2644865.2541941

[11] "One year using kubernetes in production: Lessons learned," https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned.

[12] M. Tirmazi, A. Barker, N. Deng, M. E. Haque, Z. G. Qin, S. Hand, M. Harchol-Balter, and J. Wilkes, "Borg: The next generation," ser. EuroSys '20.

[13] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in *2017 IEEE Big Data*.

[14] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao, "Who limits the resource efficiency of my datacenter: An analysis of alibaba datacenter traces," in *2019 IEEE/ACM IWQoS*.

[15] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *USENIX ATC 20*.

[16] "Aws lambda enables functions that can run up to 15 minutes," https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/.

[17] "host.json reference for azure functions 2.x and later," https://docs.microsoft.com/en-us/azure/azure-functions/functions-host-json#functiontimeout.

[18] "Cloud functions execution environment," https://cloud.google.com/functions/docs/concepts/exec#timeout.

[19] P. Turner, B. B. Rao, and N. Rao, "Cpu bandwidth control for cfs," in *Proceedings of the Linux Symposium*, 2010.

[20] "Kubernetes: Production-grade container orchestration," https://kubernetes.io/.

[21] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the European Conferfence on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[22] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center." in *NSDI 15*.

[23] K. Grygiel and M. Wielgus, "Kubernetes vpa," https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/.

[24] "Resource quotas," https://kubernetes.io/docs/concepts/policy/.

[25] "Vmware nsx data center," https://www.vmware.com/products/nsx.html.

[26] "Telecom at&t's paradise: 75% of telco's mpls tunnel data traffic now under sdn control," https://www.fiercetelecom.com/telecom/at-t-s-paradise-75-telco-s-mpls-tunnel-data-traffic-now-under-sdn-control.

[27] A. Kopytov, "Sysbench," https://github.com/akopytov/sysbench.

[28] "cadvisor," https://github.com/google/cadvisor.

[29] "Prometheus," https://github.com/prometheus/prometheus.

[30] "Docker," https://www.docker.com/.

[31] N. Brown, "Control groups, part 4: On accounting," https://lwn.net/Articles/606004/.

[32] "Apache openwhisk," https://github.com/apache/openwhisk.

[33] "client-go," https://github.com/kubernetes/client-go.

[34] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *ATC 19*.

[35] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS 19*.

[36] "Hipster shop: Cloud-native microservices demo application," https://github.com/Brown-NSG/microservices-demo.

[37] "Train ticket: A benchmark microservice system," https://github.com/FudanSELab/train-ticket.

[38] "Teastore," https://github.com/DescartesResearch/TeaStore.

[39] "Alibaba cluster trace program," https://github.com/alibaba/clusterdata.

[40] G. Tene, "wrk2: a http benchmarking tool based mostly on wrk," https://github.com/giltene/wrk2.

[41] "Performance co-pilot (pcp) manual," https://pcp.io/docs/index.html.

[42] T. Yu *et al.*, "Characterizing serverless platforms with serverlessbench," in *SoCC 2020*.

[43] "Hyperparameter tuning grid search example," https://github.com/lithops-cloud/applications/tree/master/sklearn.

[44] "scikit-learn: Machine learning in python," https://scikit-learn.org/stable/.

[45] "Lithops," https://lithops-cloud.github.io/.