Understanding the Neglected Cost of Serverless Cluster Management

Lazar Cvetković
ETH Zürich
lazar.cvetkovic@inf.ethz.ch

Rodrigo Fonseca
Azure Systems Research
fonseca.rodrigo@microsoft.com

Ana Klimovic
ETH Zürich
aklimovic@ethz.ch

Abstract

Serverless computing enables the cloud platform to optimize resource management under the hood to improve performance and resource-efficiency. However, today's serverless cluster managers are designed by simply retrofitting legacy workload orchestration systems, despite the unique characteristics of serverless workloads. We study Knative-on-K8s as a representative state-of-the-art cluster manager for serverless and show that it can cause second-scale delays and contribute to over 65% of end-to-end latency for function invocations experiencing cold starts. These overheads occur when the cluster experiences high sandbox churn, which is common in production serverless deployments. We analyze the root cause of current cluster manager overheads for serverless workloads and propose a set of design principles to improve end-to-end latency and peak throughput by rethinking the cluster manager system architecture.

CCS Concepts: • **Computer systems organization → Cloud computing**.

Keywords: Cloud computing, Serverless Computing, Cluster management

ACM Reference Format:
Lazar Cvetković, Rodrigo Fonseca, and Ana Klimovic. 2023. Understanding the Neglected Cost of Serverless Cluster Management. In *4th Workshop on Resource Disaggregation and Serverless (WORDS '23), October 23, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605181.3626286

1 Introduction

Serverless computing — also known as Functions-as-a-Service (FaaS) — is an increasingly popular paradigm of cloud computing, in which users develop fine-grained functions, while the platform automatically manages the resources needed to execute functions according to application load. Compared to the traditional Infrastructure-as-a-Service (IaaS) model of cloud computing, in which users select and rent virtual machines to run their workloads, serverless computing makes the cloud easier to use for developers and gives the cloud platform more control of the infrastructure, enabling performance and energy-efficiency optimizations under the hood.

The key component responsible for automating and optimizing resource management is the cluster manager. A FaaS cluster manager is responsible for *autoscaling* function sandboxes based on load, *placing* function sandboxes across nodes, and *load-balancing* function invocations across sandboxes available in the cluster.

The design of today's state-of-the-art FaaS cluster managers stems from legacy cluster managers or container orchestrators, such as Kubernetes (K8s). However, we find that these systems struggle to address the unique challenges imposed by the FaaS workloads. As FaaS applications typically consist of many short-lived functions whose invocation patterns are difficult to predict [26, 34], the cluster manager must frequently create and tear down function sandboxes. The cluster manager must maintain low latency, particularly when sandbox creation is on the critical path of a function invocation (we refer to this as a *cold start*). However, Figure 1 shows that cold latency increases drastically as we scale the number of concurrent sandbox creations in the cluster, primarily due to cluster manager overhead. We use Knative-on-K8s [4, 5] as a representative FaaS cluster manager, as it is the foundation for widely-used commercial and open-source serverless platforms [2, 30]. In our 10-worker-node experiment, the cluster manager becomes a bottleneck for cold function invocations, attributing as much as 65% of end-to-end latency

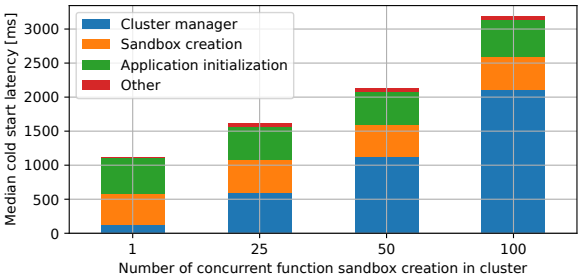

Figure 1. End-to-end latency breakdown of cold function invocation. The cluster manager adds high overhead when many concurrent cold starts.

WORDS '23, October 23, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0250-1/23/10…$15.00
https://doi.org/10.1145/3605181.3626286

and adding up to *seconds* of delay. This is problematic as production FaaS clusters experience hundreds of cold starts per second [26]. While prior work optimizes sandbox creation and initialization latency on worker nodes [14, 7, 35, 22, 30], little attention has been paid to minimizing the overhead of orchestrating a large number of sandboxes across a FaaS cluster under high churn.

In this paper, we analyze the root cause of state-of-the-art FaaS cluster overheads and propose a set of new design principles to address current bottlenecks. By analyzing the fine-grained breakdown of function latency as we sweep the rate of sandbox creations in a Knative-on-K8s FaaS cluster (§3), we conclude that sandbox orchestration overhead does *not* come from poor autoscaling, placement, or load-balancing policies. Instead, *propagating policy decisions* in the complex system architecture of the cluster manager is what introduces significant delays. For example, Knative-on-K8s involves multiple microservice components on the critical path of sandbox creation and placement. Components transfer large nested JSON blobs over gRPC to a centralized API server that persists state to a strongly consistent database. The centralized API server node, which serializes/deserializes gRPC payloads from multiple components to update the database, saturates CPU resources and leads to high queuing delays observed in Figure 1.

Drawing insights from our analysis of Knative-on-K8s overhead, we propose three key design principles for a new cluster manager system architecture tailored for FaaS (§4). First, we propose to minimize the number of RPCs involved in the critical path of sandbox creation and placement, by designing the cluster manager with a monolithic control plane, as opposed to a complex set of intertwined microservices. Second, to avoid the serialization overheads of large nested JSON blobs, we propose to simplify the cluster state format by storing small flat objects in key-value stores or even relational tables optimized for fast, frequent updates [9] . Our third design principle is to centrally manage state in concurrent in-memory data structures. We argue that only a fraction of cluster state needs to be persisted to a database to enable recovery from component failures, while the rest can be reconstructed. Minimizing the volume of state that needs to be persisted improves latency by minimizing the time spent waiting for state updates to persist to storage on the critical path of sandbox creation.

2 Background

We discuss the requirements for a FaaS cluster manager (§2.1), challenges (§2.2), and related work (§2.3).

2.1 FaaS Orchestration Requirements

While cluster managers designed for IaaS are primarily responsible for *placing* coarse-grained virtual machines onto physical servers [33, 16, 15, 11], a FaaS cluster manager has additional responsibilities, as it needs to further hide resource management complexity from users. In addition to placing function sandboxes across nodes, a FaaS cluster manager must *autoscale* function sandboxes in response to incoming requests (including scaling from zero and handling high bursts) and *load-balance* requests across sandboxes. The cluster manager's control plane must efficiently implement autoscaling and placement policies to respond to high request churn, which involves frequently creating and tearing down sandboxes for short-lived functions. The cluster manager's data plane must steer requests to available sandboxes at low latency and high throughput. Both the data and control planes must be fault tolerant and optimized for high performance and resource efficiency.

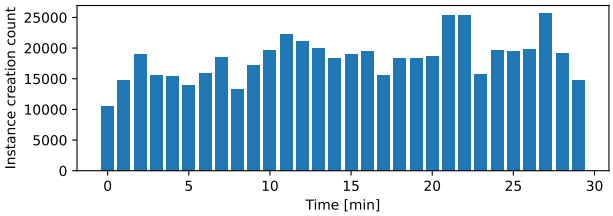

Figure 2. Sandbox creations per minute in 30-min window of Azure trace (Day 0, Hour 8).

2.2 FaaS Orchestration Challenges

FaaS applications have unique characteristics that impose three key challenges for cluster management.

Manage high volume of fine-grained state. Compared to IaaS, where users tend to run monolithic long-running workloads, FaaS applications consist of many fine-grained functions that are chained together and/or invoked in parallel. For resource efficiency, functions must be densely packed per node, with each function executing in a separate sandbox for secure isolation. While a 10K-node IaaS cluster may manage ~100K virtual machines, a 10K-node FaaS cluster would likely manage 10s of millions of function sandboxes [7]. Due to the finer granuarlity of execution, the cluster manager must also manage state at finer granularity, including per-sandbox state (e.g., network endpoints) and per-function state (e.g., autoscaling metrics). Managing state at this scale is challenging while maintaining low latency access times.

Low latency updates due to high sandbox churn. As serverless functions have highly bursty invocation patterns and short execution times [26, 34], the cluster manager must frequently create and tear down function sandboxes, resulting in frequent cluster state updates. For example, Figure 2 simulates[1] a 30-min window of the Azure Functions trace [26] and shows that the cluster manager must create 18.3K sandboxes per minute (i.e. 305 per second), on average. We also measure the fraction of function invocation requests for which sandbox creation is on the critical path, since low latency

[1]The experiment simulates trace execution on a 1000-node cluster with a cluster manager architecture resembling Knative-on-K8s and using its default autoscaling, load-balancing, and placement policies.

cluster state updates are particularly important for these "cold" requests. Assuming a 1-min keep-alive for function sandboxes (default in Knative-on-K8s and similar to [26]), we find that 52% of functions always experience a cold start, whereas for 8% of functions, every other invocation is a cold start. While prior work proposes autoscaling policies to decrease the probability of cold starts [24, 26, 27], these approaches add extra memory pressure and cost. In addition to minimizing the frequency of cold starts, the cluster manager must minimize cold start delays.

Fault tolerance and cluster state consistency. FaaS clusters are inherently distributed systems prone to component failures. Hence, the cluster manager must be able to recover from control plane and data plane failures, while providing fault tolerance features with minimal overhead. To avoid the latency overhead of persisting and replicating state on the critical path of sandbox creation, it is important to determine which state can be maintained as soft state that is simply reconstructed after a failure.

2.3 Current Cluster Managers

Kubernetes-based cluster management. Container orchestration systems are commonly the basis of FaaS cluster managers, as they provide mechanisms and policies for autoscaling, placing, and load-balancing sandboxes. Kubernetes (K8s) [5] is an open-source, state-of-the-art container orchestrator, which stems from Google's internal cluster manager, Borg [32, 29]. Although originally designed to manage long-running containerized applications, K8s is highly modular and general-purpose. Hence, K8s is the foundation for many serverless platforms, such as OpenWhisk [1], vHive [30], OpenFaas [6], and CloudRun for Anthos [2], which is Google's commercial serverless offering that runs a FaaS software layer called Knative [4] on top of K8s. We use Knative-on-K8s as a representative open-source cluster manager for FaaS, as it is used for commercial serverless offerings [2] and in popular open-source FaaS platforms, like vHive [30]. We analyze its performance limitations in §3 and find similar performance issues in other K8s-based FaaS platforms like OpenWhisk.

Cluster management beyond Kubernetes. Other cluster managers, such as Mesos [17], YARN [31], Borg [32, 33], Mercury [20], Omega [25], Twine [28], Sparrow [23], Quasar [12], Paragon [11], Tarcil [13], and Apollo [8] have been designed to efficiently share of a large cluster of machines between different applications. Traditional cluster managers lack native autoscaling features required for serverless computing and often sacrifice responsiveness to achieve scalability (e.g., all inter-component communication piggybacks on heartbeats that are sent every few seconds), making them unfit for subsecond-scale elasticity in FaaS workloads. These systems also typically target workloads running for more than 100s of seconds with a lower degree of collocation compared to densely-packed serverless functions [26]. While much related work on cluster management focuses on improving the quality of scheduling policies [15, 18], we argue that more emphasis needs to be placed on the system architecture design and implementation to guarantee low latency at high load and high churn in serverless environments.

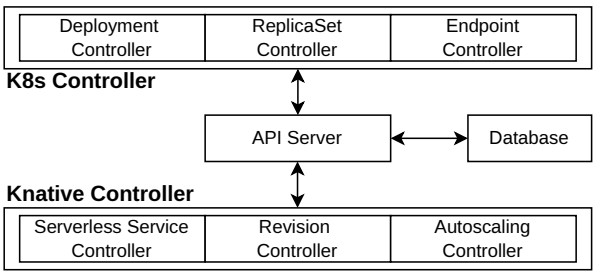

Figure 3. Simplified diagram of Knative-on-K8s.

3 Understanding Current Cluster Manager Overhead

We study the end-to-end latency breakdown for cold and warm requests to understand the cluster manager's contribution to latency and current scalability limits.

Experiment setup. Our experiment setup consists of a 13-node cluster of x170 machines on CloudLab [3], each running stock Ubuntu 20.04 on a 10-core 2.4 GHz Intel E5-2640 CPU with SMT disabled, 64 GB DRAM, and a 25 Gb/s NIC. We use Knative v1.3.0 running on top of Kubernetes v1.23.5, with containerd v1.6.2 as the sandbox environment. We dedicate one node in the cluster for Kubernetes-specific components (placement service, upscale controllers, cluster state database) and another node for Knative-specific components (autoscaling and load balancing services). The load generator runs on a separate node, while the remaining ten nodes are worker nodes used solely for running functions. We use the default Knative-on-K8s scheduling policies for autoscaling, load balancing, and placement. Out-of-the-box Knative-on-K8s requires extensive configuration for performance and cannot be used to conduct fine-grained latency breakdown directly. Hence, we tune configuration knobs (e.g., increase components QPS limits) and instrument system components to capture fine-grained timestamps. We also prune the critical path of sandbox creation to contain only fundamental functionality (e.g., we disable authentication token volume automount).

3.1 Cold Requests

Cold start latency breakdown. Figure 1 breaks down the latency of "hello world" function invocations where sandbox creation is on the critical path (i.e., cold starts).

Sandbox creation and *sandbox initialization* latency in Figure 1 corresponds to actions that take place on worker nodes. Sandbox creation is the time it takes to create a function sandbox (i.e., K8s Pod) and its network setup,

with the former being the dominant source of latency. In Knative-on-K8s, each pod consists of two containers, which are created sequentially: the container running the user's function and the sidecar used for throttling requests to the main container. We assume container images are prefetched on each worker node. Sandbox initialization is the time it takes to execute user-level initialization code for the two containers, i.e., start a gRPC server and pass the K8s readiness probe to notify the cluster manager data plane that the sandbox is now ready to receive requests. As seen in Figure 1, sandbox creation and initialization dominate cold start latency when the cluster has few sandbox creations. Much prior work has focused on sandbox boot time optimization [22, 14, 21]. However, we observe that when many sandbox creations occur concurrently, the cluster manager accounts for up to 65% of end-to-end latency.

Takeaway 1: *The state-of-the-art cluster management system for FaaS becomes a bottleneck when the cluster experiences high function sandbox churn.*

The *cluster manager* latency in Figure 1 includes all actions from when an invocation arrives at the cluster to when sandbox creation starts on a selected worker node. This involves running the autoscaling algorithm, running the placement algorithm to select a worker node for the sandbox, and updating cluster state. By breaking down the cluster manager latency, we find that the large delay at high sandbox churn comes from inefficient propagation of policy decisions across multiple components in the complex cluster manager system architecture.

Figure 3 shows a simplified diagram of the Knative-on-K8s architecture, which consists of multiple controllers. Knative-on-K8s associates multiple K8s resource types (e.g., `Deployment`, `ReplicaSet`, and `Endpoint`) with each function sandbox. K8s maintains a separate reconciliation controller for each resource type, which iteratively tries to converge the actual state to a desired state. K8s takes a level-based (vs. edge-triggered) approach to this reconciliation. This is good for robustness, but comes at the expense of latency and is at odds with the event-driven nature of FaaS. While all controllers run in the same process, these components cannot communicate directly via shared memory, because K8s imposes a strong consistency requirement over all cluster state. Hence, controller reconciliation actions involve many round trips to the cluster state database via the API Server, which serves as the database frontend. Jeffery et al. [19] found that a simple operation sequence (e.g., create a deployment with 3 sandboxes, upscale it to 10, downscale it to 5, and delete the deployment) involves approximately 3K operations to the database. Furthermore, each write to the K8s `etcd` database is a quorum-based blocking operation. Overall, the controller reconciliation pipeline can be modeled as a series of queues. The more sandboxes are created, the more actions are triggered in the cluster, saturating CPU resources and leading to the high queuing delays seen in Figure 1.

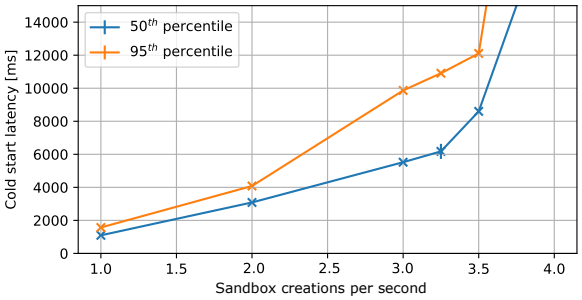

Figure 4. Cold start latency vs. sandbox churn.

Takeaway 2: *The cluster manager bottleneck comes from a complex critical path of sandbox creation that involves multiple controller reconciliation loops and persisting all state to a strongly consistent database. The bottleneck arises due to the cluster manager's system architecture design, not its resource management policies. The system architecture requires a redesign to meet the high sandbox churn and low latency needs of FaaS applications.*

Cluster manager scalability limit. We conduct an additional series of experiments in which we vary the steady rate of sandbox churn in the Knative-on-K8s cluster and measure end-to-end latency in Figure 4 to find the latency saturation point. Latency saturates at a sandbox churn rate of merely 3.5 sandbox creations per second. At this point, the API server process saturates the 16 CPU cores on the K8s master node. The API server spends over 43% of CPU cycles on JSON serialization and object diff/merging and 17% on Go garbage collection.

In attempt to increase K8s master node throughput, we tried moving Knative's components to a dedicated set of nodes from K8s components, but we observed the same performance. We also experimented with different APIs exposed by K8s. While Knative uses the Deployment API, one can also directly create new sandboxes with the K8s Pod API. However, we only observed marginal performance improvements, suggesting that the overheads are more fundamental to the internal design of K8s. Prior work has re-architected the communication mechanism between the API server and worker nodes in K8s to enable scaling clusters to millions of edge worker nodes [36], however this came at the expense of renouncing many K8s features that are essential for FaaS, such as dynamic scheduling of sandboxes.

Takeaway 3: *There is a large gap between the steady rate of sandbox churn that Knative-on-K8s can support (less than 4 sandbox creations per second in Figure 4) and the average sandbox churn that production FaaS clusters should support to meet the requirements of real workload traces (100s of sandbox creations per second [26]).*

Generalizing beyond Knative-on-K8s. We validate our results by running the same experiments on Google Cloud Run for Anthos [2], a commercial cloud platform that runs Knative on managed K8s instances. We find that managed K8s exhibits comparable latency (slightly

higher as not all knobs are exposed to optimize the system configuration). We also run our experiments on OpenWhisk [1] and observe similar bottlenecks.

Sandbox teardown. While sandbox creation is on the critical path for many function invocations, sandbox teardown happens off the critical path. We find that teardown can take up to 400 ms and primarily involves actions on worker nodes. As FaaS worker nodes need to support a high degree of sandbox collocation for resource efficiency, frequent teardowns can negatively affect latency and throughput by competing for CPU cycles on worker nodes.

3.2 Warm Requests

Thus far, we have analyzed the cluster manager's impact on end-to-end latency for cold starts. We now explore its contribution to warm request latency. When an invocation arrives in Knative-on-K8s and there is already an available sandbox to process the request, the cluster manager adds a 4 ms to 6 ms delay to function execution time, at 50^{th} and 95^{th} percentiles, respectively. This time is spent on routing the request in the data plane to load-balance invocations across sandboxes. Since the load balancer caches an up-to-date list of existing sandboxes in the cluster, it requires no interaction with the cluster manager control plane to route requests. Hence, warm requests are not affected by queuing in the cluster control plane at high sandbox churn.

Takeaway 4: *Warm function latency is not noticeably affected by sandbox creations for other functions in the cluster, due to clean separation of data and control planes.*

4 Towards a New Cluster Manager for Serverless

To address the fundamental inefficiencies uncovered in §3 about the design of state-of-the-art K8s-based cluster managers, we propose three design principles for a clean-slate cluster manager architecture tailored for FaaS.

From microservice to monolith. While microservice architectures have advantages — such as higher modularity and parallelism with component replication — they also add overheads, as we saw with Knative-on-K8s. By aggregating logically related components in the cluster manager, we can use lock-free in-memory concurrent data structures [10] for communication rather than generic updates/watches to a cluster database over gRPC. Hence, we propose to fuse control plane components (autoscaling service, placement service, control loop handlers) as a monolithic process to reduce the latency and complexity of the critical path for sandbox creation. We still propose to separate the data plane (which forwards invocation requests to worker nodes when a sandbox is available) and the control plane (which implements autoscaling and placement algorithms and manages cluster-wide state for fault tolerance).

Simplified cluster state format. The K8s approach of persisting cluster state updates in a format of ~15 kB deeply-nested JSON blobs consumes high storage capacity at FaaS cluster scale and consumes extensive CPU cycles for data serialization. The deeper the nesting in the schema, the higher the (de)serialization overhead with Golang native JSON serialization library. To address these issues, we propose to simplify the cluster state format. We find that not all the state that Knative-on-K8s manages is essential in the FaaS context. We estimate that the essential per-function state fits within 1 kB, on average, and per-sandbox state within 16 bytes. This makes it feasible to keep all state in memory for clusters with millions of functions and sandboxes.

In-memory state with limited persistence. In contrast to the heavyweight fault tolerance mechanisms in K8s, which involve synchronously persisting every cluster state change to a replicated database, we propose to manage cluster state in-memory and persist to replicated storage only the most fundamental data, such as function definitions and component connection state. Other state can be reconstructed. For example, if the primary control plane node fails, a new control plane node can take over and notify all worker nodes to forward their list of running sandboxes so that it can reconstruct this state. If the whole serverless cluster fails, function definitions can be reloaded from the disk on recovery and the system can scale from zero as if there were no sandboxes running beforehand. These optimizations apply in the context of FaaS and allow us to optimize latency on the critical path of sandbox creation during normal operation. Another potential optimization we plan to explore is whether some persisted state can be replicated asynchronously outside of the critical path instead of enforcing strong consistency for all state, as some duties of a FaaS cluster manager may be feasible to implement on eventually consistent (stale) data [19]. For example, distributed schedulers like Sparrow [23] manage to maintain high decision quality while operating on partial data for decision making. Hence, it is worth exploring which state can be stale (and how stale) to reduce cluster manager overheads without compromising decision quality. We also plan to explore cluster state store designs (e.g., key-value stores, relational database).

5 Conclusion

Today's cluster managers, such as Knative-on-K8s, add seconds of delay to function invocations when many sandboxes need to be created at once — a common case in FaaS clusters. These overheads come from system architecture inefficiencies, as multiple software components communicate by exchanging messages via strongly consistent cluster database, resulting in high CPU contention and queuing delays. Instead of retrofitting existing systems, we propose to redesign FaaS cluster manager based on the lessons learned from this study.

References

[1] Apache OpenWhisk. Available at https://openwhisk.apache.org/.

[2] Cloud Run for Anthos. Available at https://cloud.google.com/anthos/run.

[3] Cloudlab. Available at https://www.cloudlab.us/.

[4] Knative. Available at https://knative.dev/.

[5] Kubernetes. Available at https://kubernetes.io/.

[6] OpenFaaS. Available at https://www.openfaas.com/.

[7] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (2020), pp. 419–434.

[8] Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., and Zhou, L. Apollo: Scalable and coordinated scheduling for Cloud-Scale computing. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014).

[9] Chandramouli, B., Prasaad, G., Kossmann, D., Levandoski, J., Hunter, J., and Barnett, M. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data* (2018), SIGMOD '18.

[10] Cohen, N., and Petrank, E. Efficient Memory Management for Lock-Free Data Structures with Optimistic Access. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (2015), pp. 254–263.

[11] Delimitrou, C., and Kozyrakis, C. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS '13.

[12] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS '14, p. 127–144.

[13] Delimitrou, C., Sanchez, D., and Kozyrakis, C. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (2015), SoCC '15.

[14] Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., and Chen, H. Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (2020), pp. 467–481.

[15] Gog, I., Schwarzkopf, M., Gleave, A., Watson, R. N. M., and Hand, S. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (2016), pp. 99–115.

[16] Hadary, O., Marshall, L., Menache, I., Pan, A., Greeff, E. E., Dion, D., Dorminey, S., Joshi, S., Chen, Y., Russinovich, M., and Moscibroda, T. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 845–861.

[17] Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., Shenker, S., and Stoica, I. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (2011).

[18] Isard, M., Prabhakaran, V., Currey, J., Wieder, U., Talwar, K., and Goldberg, A. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP '09.

[19] Jeffery, A., Howard, H., and Mortier, R. Rearchitecting kubernetes for the edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (2021), EdgeSys'21, p. 7–12.

[20] Karanasos, K., Rao, S., Curino, C., Douglas, C., Chaliparambil, K., Fumarola, G. M., Heddaya, S., Ramakrishnan, R., and Sakalanaga, S. Mercury: Hybrid centralized and distributed scheduling in large shared clusters. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015).

[21] Li, Z., Cheng, J., Chen, Q., Guan, E., Bian, Z., Tao, Y., Zha, B., Wang, Q., Han, W., and Guo, M. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)* (2022).

[22] Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (2018), pp. 57–70.

[23] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), pp. 69–84.

[24] Roy, R. B., Patel, T., and Tiwari, D. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2022), pp. 753–767.

[25] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), pp. 351–364.

[26] Shahrad, M., Fonseca, R., Goiri, I., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., and Bianchini, R. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 205–218.

[27] Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M. D., Venkataraman, S., and Akella, A. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 138–152.

[28] Tang, C., Yu, K., Veeraraghavan, K., Kaldor, J., Michelson, S., Kooburat, T., Anbudurai, A., Clark, M., Gogia, K., Cheng, L., Christensen, B., Gartrell, A., Khutornenko, M., Kulkarni, S., Pawlowski, M., Pelkonen, T., Rodrigues, A., Tibrewal, R., Venkatesan, V., and Zhang, P. Twine: A unified cluster management system for shared infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020).

[29] Tirmazi, M., Barker, A., Deng, N., Haque, M. E., Qin, Z. G., Hand, S., Harchol-Balter, M., and Wilkes, J. Borg: the next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (2020), pp. 30:1–30:14.

[30] Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., and Grot, B. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 559–572.

[31] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. Apache Hadoop YARN: yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing* (2013), pp. 5:1–5:16.

[32] Verma, A., Pedrosa, L., Korupolu, M., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), pp. 18:1–18:17.

[33] Verma, A., Pedrosa, L., Korupolu, M. R., Oppenheimer, D., Tune, E., and Wilkes, J. Large-scale cluster management at

Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2015).

[34] Wang, A., Chang, S., Tian, H., Wang, H., Yang, H., Li, H., Du, R., and Cheng, Y. FaaSNet: Scalable and Fast Provisioning of Custom Serverless Container Runtimes at Alibaba Cloud Function Compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (2021), pp. 443–457.

[35] Young, E. G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing* (2019).

[36] Zhang, J., Jin, C., Huang, Y., Yi, L., Ding, Y., and Guo, F. KOLE: breaking the scalability barrier for managing far edge nodes in cloud. In *Proceedings of the 13th Symposium on Cloud Computing* (2022), pp. 196–209.