Dirigent: Lightweight Serverless Orchestration

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

François Costa
ETH Zürich
francois.costa@inf.ethz.ch

Mihajlo Djokic
ETH Zürich and IBM Research Europe
djokicm@ethz.ch

Michal Friedman
ETH Zürich
michal.friedman@inf.ethz.ch

Ana Klimovic
ETH Zürich
aklimovic@ethz.ch

Abstract

While Function as a Service (FaaS) platforms can initialize function sandboxes on worker nodes in 10-100s of milliseconds, the latency to schedule functions in real FaaS clusters can be orders of magnitude higher. We find that the current approach of building FaaS cluster managers on top of legacy orchestration systems like Kubernetes leads to high scheduling delay at high sandbox churn, which is typical in FaaS clusters. While generic cluster managers use hierarchical abstractions and multiple internal components to manage and reconcile state with frequent persistent updates, this becomes a bottleneck for FaaS, where cluster state frequently changes as sandboxes are created on the critical path of requests. Based on our root cause analysis of performance issues in existing FaaS cluster managers, we propose Dirigent, a clean-slate system architecture for FaaS orchestration with three key principles. First, Dirigent optimizes internal cluster manager abstractions to simplify state management. Second, it eliminates persistent state updates on the critical path of function invocations, leveraging the fact that FaaS abstracts sandboxes from users to relax exact state reconstruction guarantees. Finally, Dirigent runs monolithic control and data planes to minimize internal communication overheads and maximize throughput. We compare Dirigent to state-of-the-art FaaS platforms and show that Dirigent reduces 99th percentile per-function scheduling latency for a production workload by 2.79× compared to AWS Lambda and can spin up 2500 sandboxes per second at low latency, which is 1250× more than with Knative.

1 Introduction

Serverless computing — in particular, Function as a Service (FaaS) — is an appealing paradigm of cloud computing as it raises the level of abstraction to the cloud and alleviates users from the burden of explicitly managing server resources [66]. However, ease of use is not enough. A FaaS platform must execute functions in securely isolated environments (i.e., sandboxes) with low latency and maximize function execution throughput per machine for cost-efficiency [34].

While initializing function sandboxes on *worker nodes* takes 10-100s of milliseconds[1] with today's FaaS worker system software [34, 37, 43, 60, 74, 80, 81], we find that the

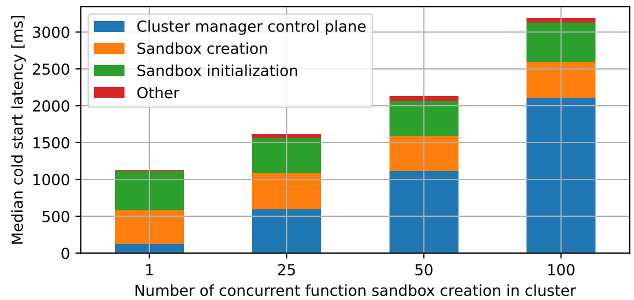

Figure 1. End-to-end latency breakdown of cold invocation bursts in Knative. Sandbox creation involves sequentially creating two containers: user code container and its sidecar. Sandbox init is the time it takes to pass health probes.

end-to-end latency to initialize function sandboxes is often one or more orders of magnitude higher in operational FaaS environments. This is because initialization involves more than creating and starting a sandbox on a worker node. First, the cluster manager receiving function invocations must schedule the sandbox to be created on a particular worker node. Then, after the sandbox is ready, the cluster manager must plug it into the cluster so that it starts receiving traffic. While scheduling a single sandbox at a time can be relatively quick, we find that scheduling delay dominates when the cluster manager concurrently schedules many sandboxes.

Figure 1 shows how the end-to-end function initialization latency — and in particular the latency contribution of the cluster manager — scales as we vary the number of concurrent sandbox creations in the Knative Serving[2] [19] FaaS platform. The cluster manager adds 2 seconds of delay when it concurrently schedules 100 sandboxes in a burst. In Figure 2, we perform a similar experiment on AWS Lambda [4]. While we cannot measure the cluster manager component of latency for proprietary FaaS platforms, Figure 2 confirms that the same symptoms are present: end-to-end latency increases as we scale concurrent cold starts. This is problematic because multi-tenant, production FaaS workloads [68] require over 300 sandbox creations per second on average, with bursts as high as 8000 (see §2.1), as FaaS applications consist of many short-lived, sporadically invoked functions [51, 79].

[1]We assume that function container images are cached on worker nodes.

[2]We refer to Knative Serving simply as Knative from now on.

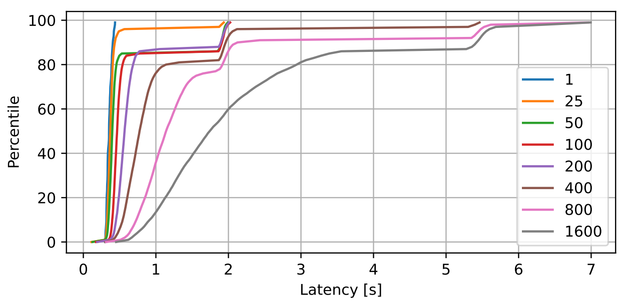

Figure 2. AWS Lambda end-to-end latency CDFs with different cold start bursts of hello-world functions. We pre-cache container images, based on insights from Brooker et al. [37].

.

So where does FaaS scheduling overhead come from and what can we do about it? We identify software bloat due to the current approach of building FaaS cluster managers on top of legacy orchestration systems that were originally designed to manage long-lived, stateful datacenter applications. We take Knative [19] as a representative FaaS cluster manager, used in open-source FaaS frameworks like vHive [76] and a commercial FaaS offering from Google [8]. Like many FaaS cluster managers [3, 13, 22, 28], Knative relies on Kubernetes (K8s) [23] to deploy sandboxes on worker nodes, monitor and manage cluster state, and recover from component failures. Knative adds invocation-based autoscaling on top of K8s, such that sandboxes can scale (potentially down to zero) for each function based on its invocations. Knative uses the K8s API to represent a sandbox as a Pod with a Service Endpoint, belonging to a ReplicaSet, managed as a Deployment. Under the hood, K8s runs a separate controller to manage the state associated with each of these abstractions. Each controller periodically executes a state reconciliation loop [71], which involves watching for updates and writing updates to a strongly consistent persistent database. Hence, creating a single sandbox involves tens of RPCs and sequential database updates in the cluster manager. With high sandbox churn in FaaS clusters, long queuing delays arise. Although serverless scheduling research has focused on scheduling *policies* [33, 45, 52, 58, 65, 69], we find that the mechanisms for propagating policy decisions from the cluster manager to worker nodes are a bottleneck.

We propose *Dirigent*, a fundamentally new system architecture for cluster management, specialized for FaaS. Dirigent exposes the same user API as current FaaS platforms, i.e., users register and invoke functions. However, instead of orchestrating sandboxes under the hood with a legacy system like K8s, Dirigent adopts a clean-slate design that removes software bloat. Since Dirigent is designed to orchestrate independent short-lived functions, for which the number and location of sandboxes in the cluster are abstracted from users, it does not need all features of generic cluster managers that enable users to deploy stateful, interdependent application sandboxes. The FaaS paradigm enables Dirigent to simplify state management and relax exact state recovery in favor of optimizing scheduling throughput. Autoscaling and placing sandboxes at high throughput is critical for short-lived, sporadically invoked functions, whereas generic cluster managers do not optimize for this as sandbox creation is amortized and off the critical path for traditional, long-lived applications. Dirigent still matches the fault tolerance guarantees of today's FaaS platforms while reducing recovery times.

We design Dirigent with three key principles. First, Dirigent *simplifies cluster management abstractions* to minimize the volume and inter-dependence of cluster state compared to systems with hierarchical abstractions (e.g., ReplicaSets, Deployments). Second, Dirigent *eliminates persistent state updates on the critical path of function invocations*. Although this means that Dirigent may not always restore a cluster to an identical state after a component fails, it is suitable for FaaS as the cluster manager abstracts sandboxes from users and continuously autoscales them. For fault tolerance, Dirigent replicates its control plane and data plane to maintain an operational cluster. Finally, Dirigent redesigns the cluster manager system architecture with a *monolithic control plane* to minimize RPC overheads and a *monolithic data plane* to reduce hops on the critical path of warm invocations.

We show that Dirigent supports 2500 cold starts per second, 1250× more than Knative. For the Azure Functions trace, Dirigent reduces per-function scheduling latency at the 99th percentile by 403× compared to Knative and 2.79× compared to AWS Lambda. We also show Dirigent can quickly recover from control plane, data plane, and worker node failures. We plan to open source Dirigent.

2 Background and Motivation

We outline the requirements for FaaS cluster management (§2.1) and analyze the fundamental mismatch between these requirements and K8s, which is used today as the foundation in many FaaS platforms (§2.2). We also discuss alternative cluster managers and why are not suitable for FaaS (§2.3).

2.1 FaaS Cluster Management Requirements

1) Low latency scheduling. Since serverless functions are often short-lived (e.g., half of the functions in Azure trace execute within a second [68]), the cluster manager must schedule functions with low latency (ideally <10s of ms) on the critical path. Scheduling in the context of FaaS requires three aspects: *autoscaling* (i.e., creating and tearing down) sandboxes per function based on invocations, *placing* sandboxes across workers to optimize performance and resource efficiency, and *load-balancing* invocations across sandboxes.

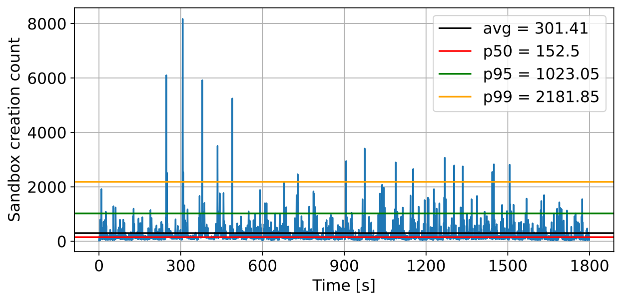

Figure 3. Rate of sandbox creation over time in a 30-minute window (after 10-min warmup) of the 70K function Azure trace [68], simulated on a 1000 worker-node cluster with default Knative scheduling policies. Each sandbox processes 1 request at a time, the default for FaaS platforms [14, 32].

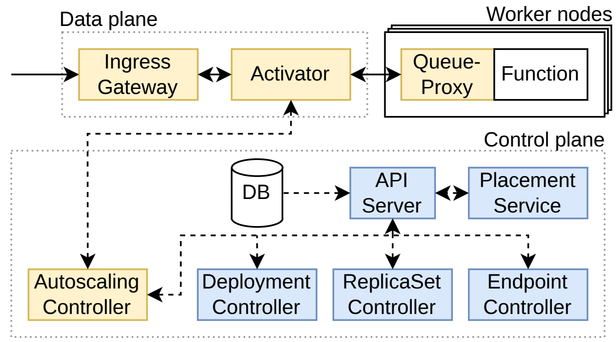

Figure 4. Knative system architecture, which builds on K8s. This diagram is simplified, showing only key components which all run as independent microservices. K8s components are blue, while yellow components are added by Knative.

2) High throughput scheduling. Due to bursty and unpredictable function invocations, as well as the high cost of DRAM needed to keep sandboxes warm in the cluster, the cluster manager must be able to create and tear down function sandboxes at high throughput while maintaining low scheduling latency. Figure 3 plots the number of sandbox creations in the Azure trace over a 30-minute time window when simulating the trace on a 1000-node cluster with the default autoscaling, load-balancing, and placement policy in Knative [20, 21, 27]. The cluster manager creates 300 sandboxes per second on average, with bursts of thousands of sandboxes per second. Even if we configure the scaling policy to have infinite keep-alive (i.e., never downscale functions to zero sandboxes after an invocation), the sandbox creation rate remains 229 sandboxes per second on average and 1551 per second at the 99th percentile — this is due to cold starts for functions being invoked for the first time. This is in contrast to traditional cluster managers, which do not optimize sandbox autoscaling throughput since sandboxes are often pre-deployed off the critical path of requests and sandbox creation is amortized for traditional, long-lived applications.

3) Fault tolerance. We distinguish between component-level and request-level fault tolerance. The FaaS cluster manager must provide *component-level* fault tolerance, i.e., ensure the platform remains operational and able to serve new invocations despite worker, data plane, or control plane node failures. The platform should minimize the impact of component failures on the end-to-end invocation latency.

Request-level fault tolerance concerns requests that are *in-flight* in the cluster when a failure occurs. Though desirable, existing FaaS platforms generally do not provide request-level fault tolerance. For synchronous function invocations — where the client blocks until receiving a response — state-of-the-art FaaS platforms rely on users to re-invoke a function [5, 14, 56] in case an invocation is lost (e.g., if a worker node fails in the middle of execution). Some FaaS platforms, like AWS Lambda, also support asynchronous invocations with a persistent queue that buffers invocations and can retry invocations in case of timeouts to provide at-least-once invocation guarantees. Since a function may get invoked (and partially executed) more than once, FaaS platforms advise users to write idempotent functions [35, 57].

Non-requirements. A FaaS cluster manager does not expose the exact number and location of application sandboxes to end-users. It also does not need to support direct communication between sandboxes [54]. This means that in case a particular sandbox fails, it is not necessary to restore the cluster to an identical state. Redeploying function sandboxes is acceptable and straightforward as FaaS functions are independent and stateless, in contrast to generic applications which may have complex workflow chains and whose components spread across different sandboxes may have complex inter-communication patterns.

2.2 The Kubernetes – FaaS Mismatch

We now discuss the mismatch between the FaaS cluster manager requirements in §2.1 and K8s-based cluster managers, which are commonly used in today's FaaS platforms, including Knative [19], OpenWhisk [3], OpenFaaS [28], Fission [13], Kubeless [22], and Google Cloud Run for Anthos [8]. K8s-based cluster managers can ensure component-level fault tolerance for FaaS (Requirement 3). However, we find that cluster managers that build on generic K8s API abstractions and inherit K8s microservice-based system architecture are unfit for low-latency and high-throughput FaaS workload scheduling (Requirements 1 and 2).

We take Knative [19] as a representative FaaS platform, as it is open-source and widely used [7], both in research [76] and commercially [8]. Figure 4 shows the Knative system architecture and how it builds on K8s components and concepts. The K8s API [24] provides concepts, such as Deployments, ReplicaSets, and Endpoints, which can be used to monitor and control cluster state at different levels of abstraction. For example, a Pod (the minimal scheduling unit in K8s) can be horizontally scaled as a ReplicaSet, a low-level K8s object that ensures a specified number of replicas are running at all times. K8s can manage ReplicaSets with a higher-level object, a Deployment, which provides additional features like rolling updates and rollbacks. K8s stores state for all objects in the cluster in a strongly-consistent database. K8s also implements multiple controllers that run reconciliation loops for objects like Deployments and ReplicaSets to converge the actual system state to the desired state. To use K8s as the underlying resource orchestrator for FaaS, Knative extends K8s with an additional set of controllers to implement invocation-based autoscaling. The Knative autoscaling controller supports scaling a function to zero sandboxes at low load. In contrast, the default K8s Horizontal Pod Autoscaler cannot scale a function to zero, i.e., has no support for cold starts but scales sandboxes based on generic metrics like CPU and memory utilization [25]. Knative also adds a component to buffer requests for cold starts (*Activator*) and a per-Pod sidecar component (*Queue-Proxy*) to throttle the number of concurrent requests each Pod can process.

While the K8s API provides convenient abstractions and the K8s architecture is modular and extensible, we find that implementing a FaaS cluster manager on top of K8s has high performance overhead. For example, Figure 5 shows the cumulative distribution of Knative scheduling latency when running a 500-function sample of the Azure production trace [68] on a 93 worker-node cluster. Scheduling has long tail latency. One third of functions experience an average scheduling latency greater than or equal to 100 seconds, whereas many functions only execute for several milliseconds (see Figure 9 for per-function slowdown).

To understand the root cause of this latency overhead, we analyze which function invocations experience high scheduling delays. We find it is functions invoked while the cluster manager is orchestrating a large number of concurrent sandbox creations. Figure 1 confirms Knative cluster manager latency increases significantly when the cluster experiences multiple concurrent cold starts. We validate our findings by running cold start invocation microbenchmarks on Google Cloud Run for Anthos [8], a commercial Knative offering. We see similar latency patterns as we scale cold start invocations.

The fundamental bottleneck is the complex critical path of sandbox creation in Knative, as the system relies on multiple K8s-based controllers to reconcile desired and actual cluster state. While computing the desired state (i.e., executing the autoscaling and placement algorithms) is fast, reconciling the cluster state is highly inefficient for several reasons. First, by design, K8s components cannot exchange information directly, even if they run in the same process. The K8s controllers can only exchange information through synchronous read-modify-write sequences to a centralized cluster state database, etcd [11]. Hence, creating a new sandbox in the

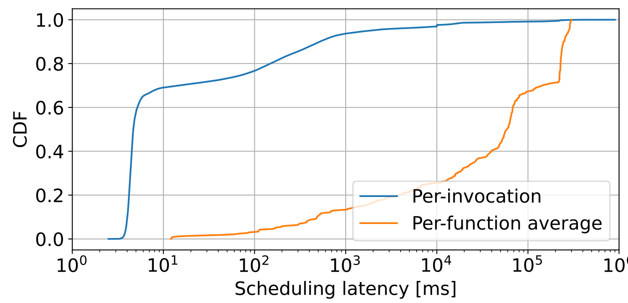

Figure 5. CDF of per-invocation scheduling latency and per-function mean scheduling latency when executing 500-function Azure trace [68, 75] on a 93-worker cluster.

cluster involves multiple RPCs between controllers and the database front-end (the API server). These operations are not commutative and hence impede scalability [39]. Second, the volume of state exchanged in RPC calls is large as K8s manages state with key-value pairs that average 17kB in size in our experiments and are represented as deeply nested trees. As a result, we find the API server spends significant CPU cycles on data serialization. When invoking cold starts at a steady rate, we find Knative can only support 2 cold starts per second before scheduling latency saturates (see Figure 7) due to the API server saturating CPU resources. Finally, K8s serializes and persists cluster state updates with strong consistency. While serializing and persisting updates enable restoring the cluster state to an identical state as before a failure occurred, it limits sandbox creation throughput.

To avoid high scheduling tail latency, we could reduce concurrent cold starts per cluster by deploying functions across independent sub-clusters. However, supporting the median cold start rate in Azure trace shown in Figure 3 (152 sandbox creations per second) would require spreading invocations across ~90 separate sub-clusters, each managed by a separate Knative instance. Running separate cluster managers requires extra resources, introduces an extra hop for all requests, and reduces global visibility of the load across machines, which can degrade scheduling decision quality [64].

To test whether our findings generalize to other K8s-based cluster managers besides Knative, we also experimented with OpenWhisk [3]. We also tested bypassing high-level K8s abstractions, such as Deployments and ReplicaSets, and instead directly created and managed Pods with K8s. In both cases, we still observe high cold start latency with concurrent cold starts, confirming that even creating and tearing down the minimal type of K8s objects (Pods) has high overhead at the high churn rate required by FaaS applications.

We also observe that increasing concurrent sandbox creations significantly impacts AWS Lambda cold start latency (Figure 2), however, we do not have access to the platform's cluster manager implementation to analyze the root cause.

Feature of K8s-based FaaS system design that contributes to high scheduling latency	Insight for Dirigent design
Managing a large volume of state for many, hierarchical abstractions in K8s (e.g., Deployments, ReplicaSets).	Simple internal cluster management abstractions.
Persisting and serializing each cluster state update on the critical path of cold function invocations.	Persistence-free latency-critical operations, relaxing exact cluster state reconstruction as it is abstracted from FaaS users.
Microservice-based control plane with RPC communication between components.	Monolithic control plane.
Per-sandbox sidecars on workers for concurrency throttling.	Monolithic data plane for request throttling.

Table 1. Dirigent's design principles, based on insights from our performance issues analysis in K8s-based FaaS systems.

2.3 Related Work

Alternative cluster managers. Cluster manager design is an active research area, with many alternatives to K8s [38, 67]. However, data center cluster managers [29, 36, 40–42, 47–49, 53, 62, 73, 77, 78] are typically designed to orchestrate long-living applications. Sandbox creation is not on the critical paths of request, and can be amortized by their long lifetimes. FaaS, in contrast, has much shorter container lifetimes and higher churn. To orchestrate thousands of nodes, systems such as Mesos and YARN [48, 77] embed all inter-component communication into periodic heartbeats. However, long heartbeat periods lead to poor responsiveness, which is unsuitable for FaaS workloads. Quincy [49] and Firmament [47] focus on scheduling policy design and explore the tradeoff of computational efficiency vs. decision quality, but ignore how the cluster manager system architecture affects decision propagation speed in the cluster. Sparrow [62] improves scalability by decentralizing scheduling, however, trading off global knowledge of the load on each worker node can degrade decision quality [64].

Many prior works explore complementary, such as reducing interference between the co-located workloads [40–42]. Mercury [53] explores tradeoffs for collocating long-running analytic jobs with latency-critical workload. Omega [67] explores tradeoffs between centralized and distributed scheduler designs. DCM [72] proposes a new cluster management architecture. However, its goal is to simplify scheduling policy implementation and debugging for developers, by enabling declarative SQL queries to a relational cluster state database. Sieve [71] addresses debugging challenges with state reconciliation systems like K8s to improve reliability.

Cluster management for FaaS. The closest to our work is a study characterizing the gap between FaaS research and real-world systems, which also identifies high cold start latency when scheduling many sandboxes [51]. We provide a root-cause analysis and solution with Dirigent. Ilúvatar [44] focuses on reducing warm start scheduling overheads originating on worker nodes, whereas we focus on alleviating bottlenecks in the cluster manager control plane. Most work on FaaS orchestration has focused on autoscaling, load-balancing, and placement policies to reduce the frequency and overhead of cold starts, improve end-to-end performance, and resource efficiency [33, 46, 52, 58, 62, 65, 68, 69]. However, these works build on top of existing FaaS cluster manager system architectures, in which the state management performance bottlenecks described in §2.2 remain.

Adapting K8s. Some works have adapted K8s for different use cases. KOLE [83] adapts K8s for the edge environment and manages to scale K8s to 1M nodes but at the expense of abolishing dynamic Pod creation and scheduling, which is not suitable for FaaS. K3s [17] is a lightweight K8s for IoT and edge environments. Although the single-process version of K8s is easy to deploy, we observed the system suffers from many of the same performance issues as the baseline K8s. Faasd [12] targets single-node resource-constrained edge setups, while we target FaaS cloud clusters.

3 Dirigent Design Approach

To address the scheduling overheads in state-of-the-art FaaS platforms, we propose *Dirigent*, a new cluster manager catered for FaaS. Dirigent maintains the same serverless end-user API as today's FaaS platforms (i.e., users register and invoke functions) such that applications designed for AWS Lambda or Knative can seamlessly run on Dirigent. Under the hood, to meet the performance and fault tolerance requirements of FaaS applications (discussed in §2.1), Dirigent orchestrates function sandboxes with a clean-slate cluster manager design as opposed to building on top of a legacy cluster manager like K8s, which we saw in §2.2 is the bottleneck for FaaS.

Dirigent's design is based on principles that address the performance issues we identified in K8s-based FaaS cluster managers (Table 1). We discuss each principle below.

Simple internal abstractions. In contrast to Knative, which uses a plethora of K8s objects (Deployments, ReplicaSets, Endpoints, Revisions, Routes, etc.) for FaaS orchestration, Dirigent's control plane orchestrates only four fundamental object types shown in Table 2. The *Function* abstraction represents a function that a user registers with a particular name, container image URL, and port to expose. Dirigent

Abstraction	Associated State	Persisted
Function	Name	✓
	Image URL	✓
	Port to expose	✓
	Scheduling configuration	✓
	Scheduling metrics	
Sandbox	Name	
	IP address	
	Port on worker node	
	Worker node ID	
DataPlane	IP address	✓
	Port	✓
WorkerNode	Name	✓
	IP address	✓
	Port	✓

Table 2. Dirigent's cluster management abstractions and the associated state maintained by the control plane for each.

saves this information as a recipe to create sandboxes for the function. Dirigent also keeps track of per-function scheduling configurations (e.g., autoscaling knobs, resource quotas, placement constraints) and monitors per-function scheduling metrics, such as the number of inflight requests for the function. The *Sandbox* abstraction (analogous to Pod in K8s) represents information about the sandbox state on a worker node, such as the sandbox name, IP address, dedicated port, and the name of the worker node it resides on. The control plane also keeps track of *DataPlane* and *WorkerNode* objects, which store the IP addresses and ports so the control plane can reach them (e.g., to trigger sandbox creation on worker nodes and notify data planes of the new sandboxes available in the cluster for request load-balancing).

Minimizing the number of internal abstractions minimizes the amount of state that Dirigent needs to maintain, improves resource efficiency, and avoids double bookkeeping and its associated consistency overheads. Moreover, it reduces the number of state updates needed whenever the autoscaling algorithm triggers a sandbox creation or teardown. In Knative, a sandbox creation triggers updates to multiple hierarchical objects (e.g., Deployment, ReplicaSet, Endpoint, Routes) via their associated state reconciliation controllers. On the other hand, Dirigent simply updates a single Sandbox object and forwards data planes an updated list of sandboxes.

In addition to managing fewer objects, Dirigent also minimizes the state stored per object. For example, by tailoring Dirigent's state management for the FaaS use case, we store the sandbox state in 16 bytes, compared to a K8s Pod resource definition, which we find can be as big as 17 KB. We find Knative uses K8s abstractions to store large function-related metadata in YAML format as raw Unicode text. This data includes annotations and labels, environment variables, sandbox state transition timestamps, and control messages. The schema features many long keys, which amplify serialization overheads. In Dirigent, we adopt a minimalist metadata and storage schema and store state in a serialized binary format.

Persistence-free latency-critical operations. To minimize function scheduling latency, Dirigent avoids persisting cluster state on the critical path of function invocations. The last column in Table 2 shows the subset of the state that the Dirigent control plane persists and which state is stored in-memory only. The Sandbox abstraction state and Function scheduling metrics are not persisted, as these are updated on the critical path of sandbox creation (i.e., cold starts). In contrast, in platforms like Knative, K8s mandates that every change to the cluster state (e.g., adding a Pod to a Deployment) is persisted in a centralized, strongly database. While this enables K8s to restore the cluster to an identical state as before a component failure, it is not necessary in the context of FaaS as the number and location of sandboxes for a function are abstracted from end-users and from other sandboxes. If worker nodes fail, the invocation-based autoscaling algorithm will restore affected sandboxes to the appropriate level and sandboxes can be placed on different workers with different IP addresses. If the control plane fails and a standby replica takes over, it will construct in-memory state of sandboxes in the cluster by requesting information from worker nodes (WorkerNode IP addresses and port numbers are persisted). While relaxing state reconstruction may not be suitable for a generic cluster manager, Dirigent still satisfies the component-level fault tolerance requirements of FaaS platforms. We discuss Dirigent's fault tolerance for a variety of component failure scenarios in §4.2. Meanwhile, removing state persistence from the critical path of function invocations allows Dirigent to maximize scheduling throughput, as we will show in §5.2.

Monolithic control and data planes. Dirigent centralizes the functionality for creating and managing sandboxes into a monolithic control plane and the functionality for routing, throttling, and buffering function invocations into a monolithic data plane. Dirigent's monolithic architecture contrasts to systems like Knative and OpenWhisk, which inherit the microservice architecture of K8s in which multiple components run as separate services that communicate via RPCs. Dirigent's monolithic control and data planes allow simpler deployment and management, fewer leader elections, and faster recovery time on crashes. In Dirigent's control plane, modules such as the state manager, health monitor, autoscaler, and placer exchange information through fast in-memory channels and atomic primitives. The monolithic data plane allows Dirigent to minimize infrastructure tax on warm starts, compared to Knative's approach of deploying separate Queue-Proxy sidecars per function sandbox for request buffering and throttling. Abolishing sidecars leads to

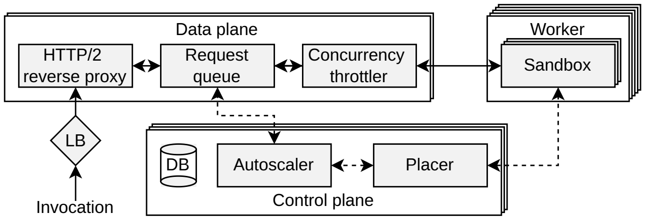

Figure 6. System diagram of Dirigent cluster manager.

faster sandbox startup time, better monitoring over invocations from data planes, less resource usage, and a shorter invocation critical path. We still separate the control and data planes, such that we can scale data planes independently based on the warm invocation load while maintaining stable control plane performance for cold starts.

4 Dirigent Implementation

4.1 System Overview

System architecture. Figure 6 shows Dirigent's system architecture. The *control plane* is responsible for monitoring cluster components, autoscaling, placing sandboxes on worker nodes, and persisting cluster state. The *data plane* load balances incoming invocations to worker nodes, buffers invocations waiting for a sandbox, and limits the number of requests a sandbox processes in parallel (concurrency throttling). Each data plane maintains a per-function endpoint list for load-balancing. The front-end *load balancer* (LB in Figure 6) spreads incoming invocations across data plane replicas by hashing the function ID. This ensures that all invocations of a particular function end up on the same data plane and allows centralized tracking of the number of in-flight requests for each function. *Worker nodes* are responsible for executing function invocations and creating/destroying sandboxes when instructed by the control plane.

Dirigent API. Table 3 shows Dirigent's end-user API (see Client caller rows), which corresponds to the APIs of FaaS platforms like AWS Lambda and Knative. The other rows in the table show the internal calls supported between the Dirigent control plane, data plane, and worker nodes.

Life of an invocation. A function invocation arrives in Dirigent through the front-end load balancer (LB) and reverse proxy. If there is a sandbox to handle the invocation (i.e., a *warm start*), the data plane picks a sandbox that will execute the invocation (load-balancing), ensures the sandbox has an available processing slot (concurrency throttling), and proxies the request to the worker node. If no sandboxes are available to process a request on its arrival (i.e., *cold start*), the invocation waits in a data plane's request queue until at least one sandbox becomes available. In the meantime, the data plane periodically sends autoscaling metrics to the control plane. The autoscaler determines the number of sandboxes needed to serve the traffic. The placer chooses a node that will spin up new sandboxes, and orders worker nodes to

Caller	Operation	Callee
Client	**(De)-Register function**	**CP**
	Invoke function	**DP**
Data plane (DP)	(De)-Register data plane	CP
	List registered functions	CP
	Send scaling metric	CP
	Send heartbeat	CP
Control plane (CP)	Add/remove function	DP
	Add/remove LB endpoint	DP
	Create/Kill sandbox	WN
	List sandboxes	WN
	Vote for leader election	CP
Worker node (WN)	(De)-Register worker	CP
	Send heartbeat	CP

Table 3. Dirigent API. Bold operations are exposed to users. Others are internal calls between Dirigent components.

create them. Once a sandbox is created, the worker daemon issues health probes to ensure the sandbox is booted and ready to handle the traffic. Once the sandbox passes a health probe, the worker daemon notifies the control plane, which then broadcasts endpoint updates data plane(s). The data plane dequeues the request and handles it as a warm start. Requests leave the system in the reverse direction and pass through the same data plane to reach the client.

Synchronous vs. asynchronous invocations. Dirigent supports both operation modes. Users specify the mode in the request header and submit the request as described above. Asynchronous calls pass through an additional queue between the front-end load balancer and the reverse proxy which submits requests and monitors invocation status and can be configured to re-invoke functions on timeouts. In this paper, we focus on synchronous calls as asynchronous ones are not supported by all FaaS platforms (e.g., Knative).

4.2 Fault Tolerance

A key design principle in Dirigent (§3) is minimizing state persistence, particularly on the critical path of invocations. We discuss how Dirigent matches the fault tolerance guarantees of today's FaaS platforms while persisting less state than K8s-based FaaS platforms. While K8s persists each state update to enable exact recovery in case of failure, specializing the cluster manager design for FaaS enables relaxing exact state reconstruction. For example, if a serverless cluster fails, the sandboxes can be created on different worker nodes, as the IP addresses are not exposed to end-users. The cluster does not even need to recover the same number of sandboxes as the demand may have changed in the meantime. We now discuss how Dirigent handles component-level and request-level fault tolerance.

4.2.1 Component-level fault tolerance.

These failures occur because Dirigent's component(s) or nodes running them crash. For high availability (HA), Dirigent runs multiple control planes and data planes. Only one control plane is the leader that serves requests, whereas others are on standby. Dirigent optimizes its components' startup time (see §5.4) and implements a restart policy on component failure. We now discuss how Dirigent handles component crashes.

Control plane. While the control plane is down no new sandboxes can be spawned, whereas warm functions remain unaffected, provided the data plane remains alive. The control plane recovers by fetching all *DataPlane* and *WorkerNode* abstractions from the persistent storage and uses this information to reestablish connections with these components. The control plane then retrieves all *Function* abstractions from the database and forwards relevant metadata to data planes for load-balancing. At this point, the cluster can start receiving new invocations, despite the scale of all functions being zero. As there are scenarios where the control plane was the only component that crashed, worker daemons can supply the control plane with a list of sandboxes they run. The control plane merges this information asynchronously, as it arrives, with its internal list and notifies data planes of changes. This feature allows sandbox reuse, while at the same time reducing performance degradation. Dirigent does not downscale these recovered sandboxes for one autoscaling time window (60 s by default) as the autoscaling metrics, which were lost on failure, take time to repopulate.

Data plane. On recovery, the data plane re-establishes connection with the control plane and pulls the list of registered functions and relevant metadata for load-balancing.

Worker node/daemon. The worker node is considered healthy and schedulable as long as the control plane receives periodic heartbeats from it. Otherwise, the control plane notifies data planes not to forward new requests to the affected worker and reruns autoscaling to spin up sandboxes on other nodes. The worker node continuously monitors sandbox processes and notifies the control plane of crashes.

4.2.2 Request-level fault tolerance.

Cluster manager component failures may lead to invocation failures. For example, if a worker node fails, all invocations executing on that node will also fail. If a data plane fails, all inflight requests in that data plane will be terminated, as connections to clients are lost. Dirigent provides no request-level fault tolerance guarantees for synchronous invocations, which is also the case with the Knative, OpenWhisk [30], and commercial FaaS platforms such as AWS Lambda and Azure Functions [5, 6, 14]. We discuss future research opportunities for improving FaaS request-level fault tolerance in §6.

4.3 Implementation Details and Limitations

We implement Dirigent in approximately 11.3K lines of Go code. Communication between system components shown in Figure 6 happens via gRPC calls that are invokable at any time, rather than through periodic heartbeats like in Mesos and YARN. Dirigent uses RAFT [61] for control plane leader election and relies on *systemd* to monitor Dirigent component health and restart a failed process. Dirigent uses Redis [31] to persist system state, which we replicate and collocate with control planes. When a control plane leader changes, the Redis master also changes.

Concurrency. System components use readers-writer locks for all critical sections with a transactional state update. On hot-paths, we use lock-free data structures where possible. The communication between different control plane modules such as placer and autoscaler uses Go channels.

Worker node software stack. We implement Dirigent with two different sandbox runtimes: containerd [10] and Firecracker [34, 37] with and without microVM snapshots. Integrating additional sandbox runtimes only involves extending a three-call interface. Each worker node maintains a local container image and snapshot cache to reduce image pulling. Because of Linux network stack performance issues on parallel network interface creations [59, 74], each worker node maintains a pool of pre-created recyclable network configurations along with pre-configured iptables rules to allow quick network allocation to a newly created sandbox.

Scheduling policies. Dirigent implements and uses Knative's default scheduling policies across all three scheduling dimensions (autoscaling, load balancing, and placement). The autoscaling algorithm scales the number of sandboxes per function based on the number of in-flight requests for each function [20]. The load-balancing algorithm forwards invocations to least-loaded sandboxes [21]. The placement policy favors nodes with the least utilized resources, while aiming to balance resource utilization across CPU and memory [27]. Dirigent also supports Hermod [52] and CH-RLU [46] scheduling policies, though they are unused in evaluation in §5 to ensure a fair comparison to Knative. Implementing new scheduling policies and metrics in Dirigent reduces to extending the relevant Golang interfaces in the control plane (for autoscaling and placement policies) and in the data plane (for load-balancing policies), recompilation, and redeployment. Knative also requires recompilation, repackaging, and redeployment of its autoscaling, load-balancing, or placement service containers to add new policies and metrics.

Operations and monitoring. Dirigent components expose global and per-function metrics (e.g., the number of inflight requests, queue depth, and number of successful invocations) via HTTP, similar to Knative. Dirigent is equipped with logging infrastructure that reports important events in the cluster, eases debugging, and can be used to break down end-to-end function latency. Dirigent's logging and monitoring infrastructure provides a foundation for building fine-grain resource accounting and billing services.

Limitations. Dirigent does not currently support function versioning and partial traffic steering to different function versions, which is supported in Knative. This can be implemented in Dirigent by extending function and sandbox abstractions with a version number and by adding a versioning-aware load balancing policy in the data plane. Cluster manager features like QoS support and remote log fetching are not yet integrated into Dirigent but can be added. We emphasize that Dirigent is an alternative to FaaS cluster managers. It is not intended as a replacement for a general-purpose cluster manager as it does not support naming/discovery services for coordination between sandboxes or provide strict guarantees for state reconstruction upon failures as K8s.

5 Evaluation

We evaluate Dirigent to answer the following key questions:

- What is the throughput of Dirigent's control plane, i.e., what is the system's peak sandbox creation rate?
- What is the Dirigent's data plane throughput, i.e., how many warm requests can Dirigent serve per second?
- How does Dirigent improve end-to-end function latency and cluster resource utilization for FaaS production workload compared to state-of-the-art systems?
- How effectively does Dirigent handle control plane, data plane, and worker node failure scenarios?

5.1 Experimental Methodology

Baselines. We compare Dirigent to two open-source K8s-based FaaS platforms: Knative [19] and OpenWhisk [3]. We briefly experimented with OpenFaaS [28] as another K8s-based baseline, but we found that the community version is not competitive as it only supports up to 15 functions and lacks critical features like scale-to-zero and concurrency throttling. We compare Dirigent's end-to-end performance to a state-of-the-art commercial platform, AWS Lambda [4].

Hardware setup. We run Dirigent and the open-source baseline systems on a 100-node xl170 Cloudlab cluster [9]. Each node is an Intel Xeon E5-2640 v4 @ 2.4 GHz CPU with 10 physical cores, 64GB of DRAM, and an Intel DC S3520 SSD. All nodes run Ubuntu 20.04. Nodes are connected in groups of 40 machines with 25 Gbps links to Mellanox 2410 leaf switches and groups connect to a Mellanox 2700 spine switch with 100 Gbps links. For AWS Lambda experiments, we register functions in the us-east-1 region and invoke functions from T3 EC2 instances in the same region.

Software setup. We run Knative v1.13.1 [19] with Istio v1.20.2 [16] and OpenWhisk v1.0.1 [3]. Both baselines run on top of Kubernetes v1.29.1 [23]. We use containerd v1.6.18 [10] as the sandbox manager. Dirigent also supports snapshot-enabled Firecracker v1.7.0 [34] sandboxes. Firecracker microVMs run Linux kernel v4.14. For the persistent data store, Dirigent uses Redis v7.2.0 [31] in append-only mode with fsync enabled at each query. We use HAProxy v2.4.24 [15] with keepalived v2.2.8 [18] as a highly-available front-end load balancer. We configure sandboxes to handle only one request at a time, similar to commercial cloud offerings [14, 32]. We employ the same scheduling policies in Knative and Dirigent (§4.3), and prefetch container images and VM snapshots on each worker node. In AWS Lambda experiments, we do container image prefetching by employing the zero unique chunk technique from [37].

In all baselines, we run the Dirigent control plane and data plane components in high-availability (HA) mode. For each component, there are three replicas, each of which runs on a dedicated node. We co-locate the front-end load balancer with the data planes and run the InVitro [75] load generator on a separate machine in the cluster.

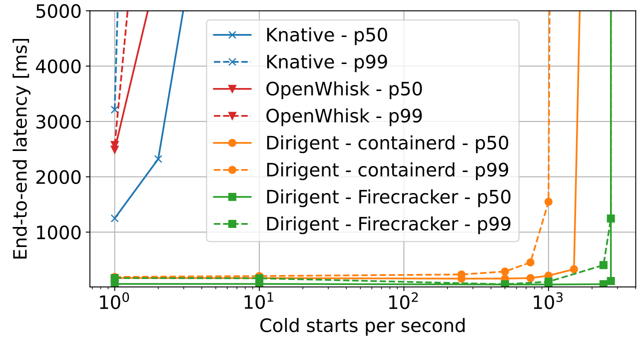

Figure 7. Cold start performance.

5.2 Microbenchmarks

We first analyze cluster manager latency, peak throughput, and scalability by invoking hello-world functions. We run cold start microbenchmarks to stress test the cluster manager control plane. We run warm start microbenchmarks to stress test the cluster manager data plane.

5.2.1 Cold Start Performance.

Peak sandbox creation throughput. Figure 7 shows the p50 and p99 end-to-end latency as we sweep the number of cold start invocations per second in the 93 worker-node cluster. Dirigent sandbox creation throughput with containerd saturates at 1750 cold starts per second. The bottleneck is not the Dirigent control plane, but rather kernel lock contention for sandbox creation, network interface configuration, and iptables rule updates on containerd worker nodes. To saturate the Dirigent control plane, we optimize the worker node software stack by running functions in Firecracker microVMs booted from snapshots. Dirigent with Firecracker microVMs achieves a peak throughput of 2500 cold starts per second. At this load, the Dirigent control plane CPU utilization is still only 55% and the bottleneck lies in acquiring locks for updates to shared data structures used for autoscaling. In contrast, cold start latency with Knative and OpenWhisk saturates at significantly lower load (below 2 cold starts per second!), due to high CPU utilization on the K8s API Server which is processing many RPCs from controller components and serializing large volumes of data for state updates to the etcd database. Note that compared to the experiment in Figure 1, where we invoked bursts of specific size and reported the median latency for invocations in that burst, here we invoke functions at a steady rate. Overall, Dirigent enables 1250× higher sandbox creation throughput than the K8s-based cluster managers. This is critical as FaaS clusters in production experience bursts in which thousands of sandboxes must be created within a second (recall Figure 3).

Cold start latency breakdown. Figure 7 also shows that Dirigent's cold start latency is lower than K8s-based systems *even at low load* (e.g., 1 cold start per second). We analyze the breakdown of unloaded cold start latency in Knative and Dirigent. Knative is slow at booting new sandboxes (~400 ms) since in addition to the user container, it creates a queue-proxy sidecar container on the worker node for each user function container. The sidecar buffers requests to the user container. These two containers are created sequentially and need to pass the readiness probe checks, which we find takes ~500 ms after both containers are created. In contrast, Dirigent buffers requests in per-function queues in data plane nodes and therefore does not need to boot queue-proxy sidecars on worker nodes in the critical path. This significantly reduces sandbox creation and readiness wait latency. Dirigent also has lower control plane latency due to minimal state updates on the critical path of sandbox creation. Dirigent with Firecracker snapshot microVMs further reduces unloaded cold start latency as it reduces sandbox creation and network configuration latency on worker nodes.

Dirigent optimization breakdown: To understand which aspects of Dirigent's design contribute most to performance benefits, we repeat the cold start throughput sweep experiment with a modified version of Dirigent that persists all state in Table 2, including sandbox state. Persisting sandbox state in the control plane introduces a write to persistent storage on the critical path for cold starts, which decreases Dirigent's peak cold start throughput to 1000 cold starts per second, and p99 latency surges at 500 cold starts per second. This confirms that avoiding persistent state updates on the critical path of cold start requests is a performance-critical design decision. In §5.4 we will show that the design decision does not sacrifice fault-tolerance, as Dirigent can still reconstruct sandbox state efficiently from worker nodes in case of control plane failures. We also confirm that simply fusing K8s components (which avoids RPCs between controllers) is not sufficient to eliminate performance issues in K8s-based cluster managers. We deploy Knative on top of K3s [17], which is a monolithic implementation of K8s within a single process. We observe only marginally higher peak cold start throughput than Knative, indicating that the state management and state persistence design decisions in Dirigent are much more performance-critical than its monolithic control plane. Dirigent's monolithic control plane is still useful as it simplifies the system design and deployment.

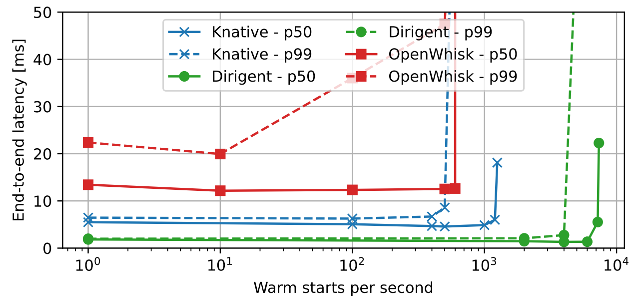

Figure 8. Warm start performance.

5.2.2 Warm Start Performance.

To stress-test the cluster manager data plane, we now consider only warm starts, i.e., invocations for which a sandbox is already available in the cluster and the control plane is not on the critical path. Figure 8 shows the p50 and p99 end-to-end latency as we sweep warm start throughput. Dirigent can sustain 4000 warm invocations per second with a median latency of 1.4 ms and a p99 latency of 2.5 ms. The components that contribute to the warm start latency are the front-end load balancer, Dirigent's HTTP proxy service and request throttler on data plane nodes, and iptables address translation on worker nodes. At the peak warm start throughput, Dirigent data plane nodes become a bottleneck as threads contend for locks to update data structures storing load balancing metadata. In contrast, Knative achieves a peak throughput of only 1200 warm starts per second with a median latency of 7 ms, as the activator and queue-proxy components in Knative add some additional latency. OpenWhisk's high latency originates from its architecture, i.e., Apache Kafka [2] and CouchDB [1] being on each invocation's critical path [44].

5.2.3 Scalability to More Worker Nodes.

We explore how cluster manager performance scales to more worker nodes. Knative documentation recommends clusters with up to 5K nodes [26]. To evaluate Dirigent's scalability, we sweep cold start throughput as we increase the number of worker nodes. Since we do not have a sufficiently large cluster, we use our 100-node cluster and run multiple worker daemons per physical machine. Each worker daemon sends heartbeats to the control plane and sleeps for 40 ms upon receiving a sandbox creation request, which corresponds to the median Firecracker microVM creation time from snapshots. We find Dirigent latency and peak throughput match the one in Figure 7 when cold starts are distributed across up to 2500 worker nodes. With more worker nodes, peak cold start throughput starts to degrade (e.g., with 5000 workers, we observe peak throughput of 2000 cold starts per second) due to lock contention for updates to shared data structures that monitor the health of sandboxes in response to heartbeats.

5.2.4 Function Registration Performance.

To be invoked, a function must first be registered with the cluster manager. Although done once per function, fast registration

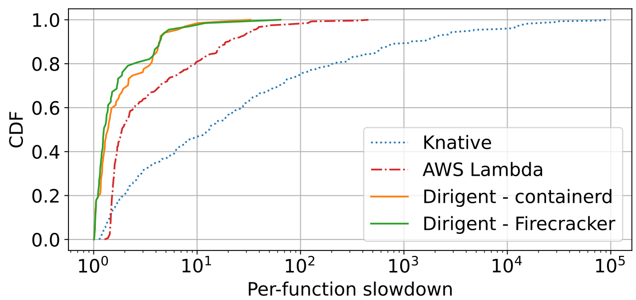

Figure 9. Per-function slowdown CDF for Azure 500 trace.

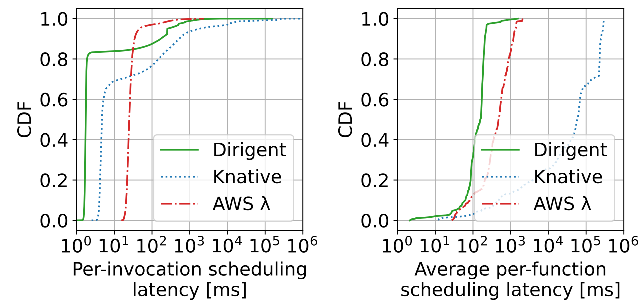

Figure 10. Scheduling latency for Azure 500 function trace.

enables users to quickly deploy applications with hundreds of functions. In AWS Lambda, we observe registering 500 functions can take hours. Knative does this work in roughly 18 minutes, whereas in Dirigent, it takes 1 second. In Knative, it takes ~770 ms to register a single function in an empty cluster, but this latency grows the more functions there are in the system. This is because Knative ascribes multiple abstractions to each function on registration (e.g., routes, revisions, services) and synchronizes ingress controllers. In contrast, registering a function in Dirigent takes 2 ms on average, as it only involves persisting function specification into the database and propagating metadata to data planes.

5.3 End-to-End Performance on Azure Trace

We now measure end-to-end performance on a FaaS production workload trace from Microsoft Azure [68] that contains 70K functions invoked over two weeks. We use InVitro [75] to obtain a representative trace sample that can run on our 100-node cluster. We extract a 30-minute time window starting in the middle of the trace (8th hour of day 6) and sample 500 functions trace with 168K invocations. We also test Dirigent with a larger trace containing 4K functions and 3.33M invocations. Functions execute the SQRTSD x86 instruction for a number of loop iterations based on the function execution time distribution in the trace. We run experiments for 30 minutes and discard the first 10 minutes as a warm-up.

We measure scheduling latency and per-function slowdown. Slowdown is the end-to-end latency of the invocation in the FaaS cluster divided by the function's execution time on a dedicated worker node with no cluster scheduling overhead. Since the execution times of different functions in the trace can vary by orders of magnitude, we group by function and report the geometric mean slowdown per function. We also evaluate resource efficiency by measuring cluster CPU and memory usage. Since OpenWhisk performance is worse than Knative for both cold and warm starts in §5.2, we do not include it here, but we compare to AWS Lambda.

Function latency analysis. Figure 9 shows Dirigent significantly reduces per-function slowdown compared to state-of-the-art systems. While the median function slowdown is 1.87 in AWS Lambda and 13.2 in Knative, it is only 1.38 with Dirigent. Dirigent especially reduces scheduling overheads at the tail, i.e., reduces p99 function slowdown by 6.89× compared to AWS Lambda and by over three orders of magnitude compared to Knative. While slowdown quantifies the impact the cluster manager has on end-to-end latency (which also depends on the function's execution time), Figure 10 shows the raw scheduling latency CDFs for the same experiment, both per-invocation and per-function average scheduling latency. Note the log scale. Dirigent reduces the median and p99 per-function scheduling delay by 3.07× and 2.79× compared to AWS Lambda, respectively. Dirigent reduces the p99 per-function scheduling delay by 403× compared to Knative.

The functions that experience the highest slowdown in Dirigent are those with the shortest execution time (i.e., below 10 ms) as these functions are the most sensitive to scheduling overheads and sandbox creation delays. Meanwhile, the functions with the highest slowdown in Knative and AWS Lambda experiments are predominantly functions whose individual invocations are greatly spread out over time but occur during times in the trace when the cluster experiences the most cold starts. We find some functions in the trace are repeatedly invoked in unison (due to timer-based invocation triggers [68]) with long periods, resulting in large cold start bursts in the cluster. These bursts lead to high scheduling latency in AWS Lambda and Knative, whereas Dirigent handles much higher cold start throughput. For the Azure 500 function trace experiment, Knative's median per-invocation scheduling latency is 4.67 ms and 59.59 s at the 99th percentile. In contrast, Dirigent's median scheduling latency is 1.74 ms and 1.13 s at the 99th percentile. Dirigent with Firecracker has a bit longer per-function slowdown tail as some functions are never invoked during the warm-up period, hence the first cold start is not from a microVM snapshot.

Sandbox creation count. We also notice that Dirigent creates fewer sandboxes throughout the experiment even though it uses the same autoscaling algorithm and metrics as Knative. During the experiment, Knative spawned 2930 sandboxes, whereas Dirigent created only 713 sandboxes for the same workload trace. To understand this discrepancy, we need to delve into the functioning of the Knative autoscaling algorithm. Knative's autoscaler monitors the number of inflight requests, which encompasses both those actively being processed within pods and those queued. The desired number of pods is directly proportional to the inflight request count. Intuitively, when a queue forms, the autoscaler initiates new pod creations proportionally to the queue length. However, due to a lengthy scale-up delay within Knative, the queue continues to grow during the scale-up process, prompting the creation of even more pods. In contrast, Dirigent exhibits a more responsive behavior. When a queue starts to form, the Knative autoscaling algorithm starts creating pods, and Dirigent promptly scales the number of ready pods to the desired state of the autoscaler, leading to a near-immediate depletion of the queue. This swift response translates to a significantly reduced number of pods being provisioned overall.

Resource utilization. We observe that the Dirigent control plane node only uses 3% of CPU cycles on average, whereas the Knative control plane CPU is consistently above 75% utilized as it struggles to handle cold start bursts. Hence, Dirigent provides higher scheduling performance while also consuming fewer CPU resources for the control plane than Knative. Worker nodes memory in Knative and Dirigent is utilized 4.62% and 3.1%, respectively.

Larger trace. While the sampled Azure trace with 500 functions is the biggest trace we can run with Knative before we start observing high invocation failure rates due to timeouts, the trace does not saturate the same hardware cluster orchestrated by Dirigent. Hence, we run a larger Azure trace sample with 4000 functions and 3.33M invocations. We compare Dirigent to AWS Lambda. With this trace, Dirigent utilizes 70% of CPU resources on worker nodes and achieves p50 and p99 slowdowns of 2.14 and 15.4, respectively. On the other hand, AWS Lambda's p50 and p99 slowdowns are 70 and 11631, respectively. Finally, Dirigent experiences a negligible invocation failure rate, while in the AWS Lambda, 14% of invocations experience timeout.

5.4 Fault Tolerance

We now analyze the impact of component failures. We measure average function invocation slowdown over time for the Azure 500-function workload, while triggering failures.

Control plane failure. Figure 11, shows how the slowdown of function invocations varies over time before and after we fail the control plane leader for Dirigent and Knative. A control plane failure impacts performance by adding a queuing delay for cold starts. Cold starts must be buffered in the data plane until the control plane becomes operational again and can schedule a sandbox creation or until a previously busy sandbox for the function on a worker node becomes available in the cluster. Dirigent achieves a lower per-invocation slowdown for invocations issued at the moment of failure and manages to stabilize the slowdown quicker than Knative. The performance improvements of

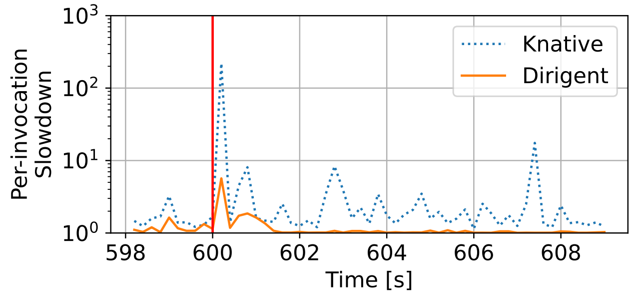

Figure 11. Control plane fault tolerance. The vertical red line shows when the failure occurs.

Dirigent stem from the monolithic control plane architecture which only requires a single leader election, taking up to 10 ms. In contrast, Knative elects a leader for each control plane microservice followed by a component registration with the API Server, which can take several seconds.

Data plane failure. When a data plane fails, all inflight requests also fail, as connections to clients are terminated. We fail one data plane replica and monitor the invocation failure rate. We observe it takes 2 seconds for the failure rate to drop back down to zero after a data plane failure in Dirigent, compared to 15 seconds in Knative. In Knative, it takes more time for the front-end load balancer to detect a data plane failed, after which it stops routing new requests to the affected component. Since Knative's data plane is not a monolith as in Dirigent, the recovery time is dominated by the Istio gateway, the slowest component to restart.

Worker daemon failure. When the worker daemon on a node fails, the worker can no longer respond to any control plane commands, including starting or tearing down sandboxes. This leads to a higher slowdown on cold invocations, while warm invocations remain affected. We failed 47 out of 93 worker daemons in the cluster while monitoring the slowdown of functions invoked during worker downtime. Dirigent achieves a peak per-invocation slowdown of 2.7, which is 10× lower than Knative, as Dirigent can efficiently create new sandboxes on non-affected nodes and because it has shorter worker daemon recovery time.

Concurrent component failures. Dirigent remains operational as long as one control plane replica is elected as a leader and at least one data plane is operational. In case of concurrent component failures, the recovery time will be dominated by the slowest component to recover, as components can recover in parallel.

6 Future Directions

By enabling orders of magnitude higher sandbox creation throughput than existing platforms, Dirigent can provide a foundation for future research in FaaS systems. We are currently exploring how to schedule function workflows in Dirigent by using data planes as workflow orchestrators. We also aim to explore how to provide at-least-once or exactly-once request-level guarantees and quantify their cost at scale [50, 63, 70, 82]. We are adding support for more sandbox runtimes [55] and scheduling policies. Another important question is how to manage container image caching at scale [37].

7 Conclusion

Dirigent is a new customized cluster manager for serverless. In contrast to the state-of-the-art approach of building FaaS cluster managers on top of legacy cluster managers like Kubernetes, Dirigent presents a clean-slate system architecture, simple abstractions, and lightweight persistence for state management to eliminate the performance bottlenecks of K8s-based cluster managers in high-churn FaaS environments. We show that Dirigent can pin up 2500 sandboxes per second at low latency, which is 1250× more than Knative. Dirigent achieves 6.89× lower 99th percentile per-function slowdown and 403× lower 99th percentile per-function scheduling latency compared to Knative on a production Azure trace while maintaining 25× lower control plane CPU utilization on average. Dirigent also improves recovery times from component failures compared to Knative.

Acknowledgments

The authors would like to express their gratitude to Rodrigo Fonseca, Lalith Suresh, Timothy Roscoe, Michael Wawrzoniak, Patrick Stuedi, and Malte Schwarzkopf for providing their valuable feedback. The authors would also like to thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] Apache CouchDB. Available at https://couchdb.apache.org/.
[2] Apache Kafka. Available at https://kafka.apache.org/.
[3] Apache OpenWhisk. Available at https://openwhisk.apache.org/.
[4] AWS Lambda. Available at https://aws.amazon.com/lambda/.
[5] AWS Lambda Invocation-Level Guarantees. Available at https://docs.aws.amazon.com/lambda/latest/dg/invocation-retries.html.
[6] Azure Functions Invocation-Level Guarantees. Available at https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-error-pages.
[7] Cloud Native Computing Foundation Survey – 2020. Available at https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.
[8] Cloud Run for Anthos. Available at https://cloud.google.com/anthos/run.
[9] Cloudlab. Available at https://www.cloudlab.us/.
[10] containerd. Available at https://containerd.io/.
[11] etcd. Available at https://etcd.io/.
[12] faasd - a lightweight & portable faas engine. Available at https://github.com/openfaas/faasd.
[13] Fission. Available at https://github.com/fission/fission.
[14] Google Cloud Functions Invocation-Level Guarantees. Available at https://cloud.google.com/functions/docs/bestpractices/retries.
[15] Haproxy. Available at https://www.haproxy.com/.
[16] Istio considerations for large clusters. Available at https://www.istio.io/.
[17] K3s - lightweight kubernetes. Available at https://k3s.io/.
[18] keepalived. Available at https://www.keepalived.org/.
[19] Knative. Available at https://knative.dev/.
[20] Knative autoscaling. Available at https://knative.dev/docs/serving/autoscaling/.
[21] Knative load-balancing. Available at https://knative.dev/docs/serving/load-balancing/.
[22] Kubeless. Available at https://kubeless.io/.
[23] Kubernetes. Available at https://kubernetes.io/.
[24] Kubernetes – API Overview. Available at https://kubernetes.io/docs/reference/generated/kubernetes-api/.
[25] Kubernetes – horizontal pod autoscaler. Available at https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[26] Kubernetes considerations for large clusters. Available at https://www.kubernetes.io/docs/setup/best-practices/cluster-large/.
[27] Kubernetes placement. Available at https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.
[28] OpenFaaS. Available at https://www.openfaas.com/.
[29] Openstack. Available at https://www.openstack.org/.
[30] OpenWhisk Invocation-Level Guarantees. Available at https://github.com/apache/openwhisk/issues/5449.
[31] Redis. Available at https://redis.io/.
[32] Understanding AWS Lambda's invoke throttling limits. Available at https://aws.amazon.com/blogs/compute/understanding-aws-lambdas-invoke-throttle-limits/.
[33] Abdi, M., Ginzburg, S., Lin, X. C., Faleiro, J., Chaudhry, G. I., Goiri, I., Bianchini, R., Berger, D. S., and Fonseca, R. Palette load balancing: Locality hints for serverless functions. In *Proceedings of the Eighteenth European Conference on Computer Systems* (New York, NY, USA, 2023), EuroSys '23, Association for Computing Machinery, p. 365–380.
[34] 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.
[35] AWS Lambda. Developing for retries and failures. Available at https://docs.aws.amazon.com/lambda/latest/operatorguide/retries-failures.html.
[36] 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).
[37] Brooker, M., Danilov, M., Greenwood, C., and Piwonka, P. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 315–328.
[38] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue 14*, 1 (jan 2016), 70–93.
[39] Clements, A. T., Kaashoek, M. F., Zeldovich, N., Morris, R. T., and Kohler, E. The scalable commutativity rule: designing scalable software for multicore processors. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, Association for Computing Machinery, p. 1–17.
[40] 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.
[41] Delimitrou, C., and Kozyrakis, C. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not. 49*, 4 (feb 2014), 127–144.
[42] 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.
[43] 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.

[44] Fuerst, A., Rehman, A., and Sharma, P. Ilúvatar: A fast control plane for serverless computing. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (2023), HPDC '23, p. 267–280.

[45] Fuerst, A., and Sharma, P. Faascache: keeping serverless computing alive with greedy-dual caching. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021* (2021), T. Sherwood, E. D. Berger, and C. Kozyrakis, Eds., ACM, pp. 386–400.

[46] Fuerst, A., and Sharma, P. Locality-aware load-balancing for serverless clusters. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing* (New York, NY, USA, 2022), HPDC '22, Association for Computing Machinery, p. 227–239.

[47] 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 2016, Savannah, GA, USA, November 2-4, 2016* (2016), K. Keeton and T. Roscoe, Eds., USENIX Association, pp. 99–115.

[48] 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 Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011* (2011), D. G. Andersen and S. Ratnasamy, Eds., USENIX Association.

[49] 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.

[50] Jia, Z., and Witchel, E. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (New York, NY, USA, 2021), SOSP '21, Association for Computing Machinery, p. 691–707.

[51] Joosen, A., Hassan, A., Asenov, M., Singh, R., Darlow, L. N., Wang, J., and Barker, A. How does it function?: Characterizing long-term trends in production serverless workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023* (2023), ACM, pp. 443–458.

[52] Kaffes, K., Yadwadkar, N. J., and Kozyrakis, C. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 13th Symposium on Cloud Computing* (New York, NY, USA, 2022), SoCC '22, Association for Computing Machinery, p. 289–305.

[53] 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).

[54] Klimovic, A., Wang, Y., Stuedi, P., Trivedi, A., Pfefferle, J., and Kozyrakis, C. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (Carlsbad, CA, Oct. 2018), USENIX Association, pp. 427–444.

[55] Kuchler, T., Giardino, M., Roscoe, T., and Klimovic, A. Function as a function. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (New York, NY, USA, 2023), SoCC '23, Association for Computing Machinery, p. 81–92.

[56] Microsoft Azure. Azure functions error handling and retries. Available at https://learn.microsoft.com/en-us/azure/azure-functions/functions-bindings-error-pages.

[57] Microsoft Azure. Azure functions reliable event processing. Available at https://learn.microsoft.com/en-us/azure/azure-functions/functions-reliable-event-processing.

[58] Mittal, V., Qi, S., Bhattacharya, R., Lyu, X., Li, J., Kulkarni, S. G., Li, D., Hwang, J., Ramakrishnan, K. K., and Wood, T. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2021), SoCC '21, Association for Computing Machinery, p. 168–181.

[59] Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., and Sukhomlinov, V. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)* (Renton, WA, July 2019), USENIX Association.

[60] 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.

[61] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, 2014), USENIX Association, pp. 305–319.

[62] Ousterhout, K., Wendell, P., Zaharia, M., and Stoica, I. Sparrow: distributed, low latency scheduling. In *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP '13, Farmington, PA, USA, November 3-6, 2013* (2013), M. Kaminsky and M. Dahlin, Eds., ACM, pp. 69–84.

[63] Qi, S., Liu, X., and Jin, X. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 314–330.

[64] Rasley, J., Karanasos, K., Kandula, S., Fonseca, R., Vojnovic, M., and Rao, S. Efficient queue management for cluster scheduling. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), EuroSys '16.

[65] 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.

[66] Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N. J., Popa, R. A., Gonzalez, J. E., Stoica, I., and Patterson, D. A. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM 64*, 5 (apr 2021), 76–84.

[67] Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., and Wilkes, J. Omega: flexible, scalable schedulers for large compute clusters. In *SIGOPS European Conference on Computer Systems (EuroSys)* (Prague, Czech Republic, 2013), pp. 351–364.

[68] 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)*, USENIX Association, pp. 205–218.

[69] 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.

[70] Sreekanti, V., Wu, C., Chhatrapati, S., Gonzalez, J. E., Hellerstein, J. M., and Faleiro, J. M. A fault-tolerance shim for serverless computing, 2020.

[71] Sun, X., Luo, W., Gu, J. T., Ganesan, A., Alagappan, R., Gasch, M., Suresh, L., and Xu, T. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)* (Carlsbad, CA, July 2022), USENIX Association, pp. 143–159.

[72] Suresh, L., Loff, J., Kalim, F., Jyothi, S. A., Narodytska, N., Ryzhyk, L., Gamage, S., Oki, B., Jain, P., and Gasch, M. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 827–844.

[73] 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).

[74] Thomas, S., Ao, L., Voelker, G. M., and Porter, G. Particle: ephemeral endpoints for serverless networking. In *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020* (2020), R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds., ACM, pp. 16–29.

[75] Ustiugov, D., Park, D., Cvetković, L., Djokic, M., Hè, H., Grot, B., and Klimovic, A. Enabling in-vitro serverless systems research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless* (New York, NY, USA, 2023), WORDS '23, Association for Computing Machinery, p. 1–7.

[76] 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.

[77] 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. C., and Baldeschwieler, E. Apache hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013* (2013), G. M. Lohman, Ed., ACM, pp. 5:1–5:16.

[78] 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, EuroSys 2015, Bordeaux, France, April 21-24, 2015* (2015), L. Réveillère, T. Harris, and M. Herlihy, Eds., ACM, pp. 18:1–18:17.

[79] 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).

[80] Wanninger, N. C., Bowden, J. J., Shetty, K., Garg, A., and Hale, K. C. Isolating functions at the hardware limit with virtines. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), EuroSys '22.

[81] 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).

[82] Zhang, H., Cardoza, A., Chen, P. B., Angel, S., and Liu, V. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 1187–1204.

[83] 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.