USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

KubeDirect: Unleashing the Full Power of the Cluster Manager for Serverless Computing

Sheng Qi, Zhiquan Zhang, Xuanzhe Liu, and Xin Jin, *Peking University*

https://www.usenix.org/conference/nsdi26/presentation/qi

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

KUBEDIRECT: Unleashing the Full Power of the Cluster Manager for Serverless Computing

Sheng Qi† Zhiquan Zhang‡ Xuanzhe Liu† Xin Jin†

† *School of Computer Science, Peking University*
‡ *School of Electronics Engineering and Computer Science, Peking University*

Abstract

FaaS platforms rely on cluster managers like Kubernetes for resource management. Kubernetes is popular due to its extensible state-centric APIs and modular architecture. However, to scale out a burst of FaaS instances, message passing becomes the primary bottleneck as controllers have to exchange extensive state through the API Server. Existing solutions opt for a clean-slate redesign of cluster managers, at the expense of ecosystem compatibility and substantial engineering effort.

We present KUBEDIRECT, a Kubernetes-based cluster manager for FaaS. We find that there exists a common *narrow waist* across FaaS platforms that allows us to achieve both efficiency and external compatibility. The narrow waist has a sequential structure that obviates the need for a single source of truth, allowing us to bypass the API Server and perform lightweight direct message passing. However, our approach introduces distributed and ephemeral state across controllers, making it challenging to enforce end-to-end semantics without centralized coordination. KUBEDIRECT performs novel state management that leverages the narrow waist as a *hierarchical write-back cache*, ensuring consistency and convergence to the desired state. KUBEDIRECT can seamlessly integrate with Kubernetes, adding ∼150 LoC per controller. KUBEDIRECT can reduce serving latency by 26.7× over Knative, and has similar performance as the state-of-the-art clean-slate platform Dirigent.

1 Introduction

Serverless computing, or Function-as-a-Service (FaaS), has gained significant traction in modern cloud applications [37, 55, 58, 64, 67, 71, 79, 86, 87, 91, 95, 102]. Users decompose applications into fine-grained *functions*, and the FaaS platform automates the deployment process [59–61, 99, 100]. To utilize the cloud infrastructure, FaaS platforms typically rely on low-level cluster managers [21, 83, 93, 94]. For example, the Kubernetes cluster manager has given rise to Knative [18], OpenFaaS [28], Fission [12], and many others [20, 30].

Kubernetes has become the de facto choice for FaaS platforms due to its state-centric design principle that enables great extensibility [42, 89, 90]. It represents the cluster state as a set of *API objects* [21] stored in etcd, a persistent key-value store [11]. It distributes the cluster management logic across a set of *controllers* that collaborate through a pub-sub service offered by the *API Server* (the etcd frontend). Controllers subscribe to changes on certain API objects, take actions accordingly (e.g., scheduling), modify other objects to update the cluster state, and finally publish the objects to the API Server to trigger dependent controllers. To extend Kubernetes, developers can define custom object APIs and corresponding controllers that translates them to and from the built-in APIs. Today, Kubernetes has fostered a rich ecosystem that manages networking, monitoring, storage, and security [14, 16, 31, 32], providing critical services to production-grade systems.

However, extensibility comes at the cost of efficiency due to the *indirect* write-notify round trips to the API Server between controllers. We find that while controllers are fast with their internal logic (orders of milliseconds), they spend substantial time on *message passing* when exchanging a large amount of state. Consequently, provisioning hundreds of FaaS instances takes tens of seconds in Kubernetes (§2). However, massive upscaling is prevalent in FaaS due to its burstiness [82], e.g., there can be 16K instance creations per minute in the Azure Functions trace (§2). Moreover, upscaling happens on the critical path of request handling, where excess requests are queued until new instances become ready (i.e., "cold starts" [98]). The overhead is further amplified due to most requests being short-lived [84]. In contrast, creation of the container itself only incurs sub-second or even sub-millisecond latency [36, 49, 63, 76, 96].

Existing solutions opt for a clean-slate redesign of FaaS platforms as opposed to extending legacy cluster managers [44, 46, 51, 85, 91, 101]. In pursuit of efficiency, they fundamentally diverge from the state-centric and modular design philosophy of Kubernetes, which limits their compatibility with the existing ecosystem. On the one hand, basic functionalities like scheduling and routing, whereas well-developed in Kubernetes, have to be built from scratch. On the other hand, critical extensions like service meshes [16] or monitor-

ing tools [31] cannot enjoy push-button deployment as they depend on the API pub-sub service, necessitating non-trivial manual integration.

In this paper, we show that existing cluster managers can be optimized for FaaS without compromising compatibility. Our insight is that there exists a *narrow waist* consisting of several well-known controllers (Figure 1) that are always present in Kubernetes-based FaaS platforms, covering the common functionality of scaling out FaaS instances. Upstream to the narrow waist are platform-specific controllers that translate user-facing APIs and policies to the narrow waist. Downstream are data plane load balancing components that subscribe to the output of the narrow waist, i.e., the readiness of FaaS instances. The fact that the upstream is offline and the downstream read-only implies that (1) performance-wise, the narrow waist is the primary source of cold start latency; (2) implementation-wise, the narrow waist is cleanly separated from the outside, allowing for optimizations without affecting external compatibility.

To optimize the scaling process, we bypass the API Server bottleneck with lightweight *direct* message passing through the narrow waist. Our insight is that the API Server is redundant in two aspects that make our approach practical. First, it persists every state transition in the cluster to etcd and enforces exact state recovery across failures. However, instances halfway through provisioning are naturally *fungible*. Because they are not associated with any physical resources or requests, they are free from side effects and can be safely replaced across failures. Moreover, controllers are designed as state machines that continuously query whatever the current cluster state and drive it to the desired one, making them tolerant to rollbacks where missing instances can be recreated as needed. Second, the API Server is responsible for resolving conflicts and serializing concurrent updates to the same object. However, the scaling process follows a *stage-wise* pattern where controllers sequentially decide the desired state of API objects, allowing for conflict-free collaboration.

A strawman solution of direct message passing is to send the API objects *as is* between controllers. However, there is still substantial redundancy: each controller only updates a few object attributes, e.g., the *Scheduler* only sets the target node, whereas the rest are static or predetermined. We therefore decouple the dynamic and static attributes and differentiate the APIs used in message passing and internal control loops with *dynamic materialization*. The sender only transmits the *delta* attributes for efficiency, and the receiver assembles the dynamic and static attributes in-memory, converting to standard API objects for transparent processing.

While it is straightforward to use P2P communication to avoid central bottlenecks, it is challenging to systematically apply this idea while maintaining consistency and fault tolerance. Originally, the API Server serves as the single source of truth of the cluster state to all controllers. KUBEDIRECT, in contrast, must reconcile the *ephemeral* state introduced by direct message passing across distributed controllers, in the absence of *centralized coordination*. We note that the sequential structure of the narrow waist is analogous to the chain of storage servers in the Chain Replication protocol (CR) [81]. However, CR assumes homogeneous backups that only perform replication, whereas controllers in KUBEDIRECT are heterogeneous state machines with active state transitions, leading to novel challenges.

First, because controllers can perform non-idempotent operations, e.g., scheduling depends on the varying cluster load, and also in an asynchronous fashion, naively propagating the upstream state like CR can lead to inconsistencies. Our insight is that we should instead model the narrow waist as a *hierarchical write-back cache*, where we opportunistically forward the upstream state transitions towards the tail, and handle downstream transitions and/or failures as cache invalidations to the upstream. We design hard invalidation for joining crash-restarted controllers back to the chain, and lightweight soft invalidation for live controllers.

Second, KUBEDIRECT should adhere to the conventions on instance lifecycle in Kubernetes, especially for instance termination, i.e., the transition to the *Terminating* state is *irreversible*. This calls for separate handling of termination and provisioning, because we cannot rollback the former like we would the latter. However, termination turns out to be *idempotent*. We therefore introduce a special type of *Tombstone* object and perform CR-style replication along the aforementioned opportunistic forwarding pipeline, which implements asynchronous termination, e.g., downscaling. On top of that, we implement synchronous termination, e.g., preemption, by blocking on downstream invalidations.

We implement a prototype of KUBEDIRECT on top of Kubernetes and the Knative FaaS platform [18]. We change only $\sim$150 LoC per controller in the narrow waist, and support external extensions out-of-the-box. We verify KUBEDIRECT's convergence to the desired cluster state with TLA+. In summary, we make the following contributions.

- With the design and implementation of KUBEDIRECT, we affirm that legacy cluster managers can be seamlessly optimized for FaaS without compromising compatibility.
- We propose on-demand materialization for direct message passing that achieves both efficiency and transparency.
- We ensure consistency across controllers and convergence like Kubernetes with pairwise state management.
- Experiments show that KUBEDIRECT improves function serving latency by 26.7$\times$ over Knative, and has similar performance as the state-of-the-art system Dirigent.

2 Background and Motivation

2.1 Kubernetes Basics

Kubernetes is highly extensible and supports a large ecosystem of extensions. The CNCF Artifact Hub alone hosts 15.6K extensions [8]. An extension is a set of custom APIs and con-

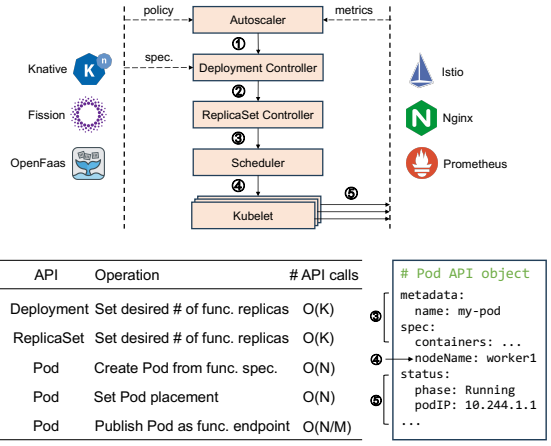


Figure 1: The *narrow waist* of Kubernetes-based FaaS platform and the scaling critical path. We consider scaling N Pods for K ReplicaSets in an M-node cluster.

trollers that collaborate with the Kubernetes core. Examples include Istio for networking [16], Prometheus for monitoring [31], Redis for storage [32], and Jenkins for CI/CD [17].

The rich ecosystem makes Kubernetes a natural choice for developing FaaS platforms. We analyze the architecture of three popular Kubernetes-based FaaS platforms: Knative [18], OpenFaaS [28], and Fission [12]. In spite of the diverse user APIs and policies, they share a common *narrow waist* of controllers that implements the basic functionality of scaling out FaaS instances as shown in Figure 1–2.

Figure 1 illustrates the API objects (capitalized) and controllers (italicized) in the narrow waist. Pod is the basic unit of scheduling, which specifies several containers to serve as one FaaS instance. ReplicaSet manages a group of Pods that share a common template. Deployment is a higher-level abstraction of ReplicaSet that implements versioning and rolling updates across versions; it is the Kubernetes-equivalent of a FaaS function. In case of upscaling, the critical path consists of the following steps. ① The *Autoscaler* computes the desired number of instances based on runtime metrics. It scales a Deployment by updating its `replicas` field. ② The *Deployment controller* selects the ReplicaSet of correct version and in turn sets its `replicas` field. ③ The *ReplicaSet controller* creates a set of new Pods to match the desired scale. ④ The *Scheduler* assigns them to cluster nodes by updating the `nodeName` field of each Pod. ⑤ The *Kubelet* on each node filters the locally assigned Pods and forwards them to the sandbox runtime. Finally it marks the Pods ready by populating their `status` fields and exposes them to the data plane.

As shown in Figure 2, upstream to the narrow waist are platform-specific controllers that perform *offline* configurations, and downstream are load balancing components that are *read-only* to the Pod API. Therefore, in terms of performance, the narrow waist is the primary source of latency. In terms of implementation, the differences in upstream configurations are normalized by the Deployment API and the common API

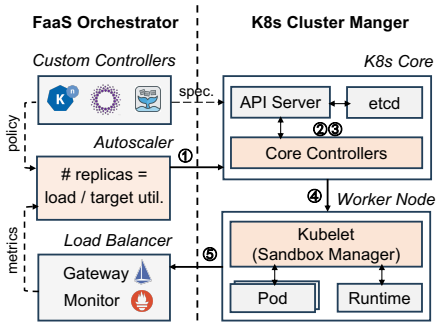


Figure 2: Architecture of Kubernetes-based FaaS platform. We highlight the narrow waist in orange. ① to ⑤ are from Figure 1 and are indirect calls via the API Server.

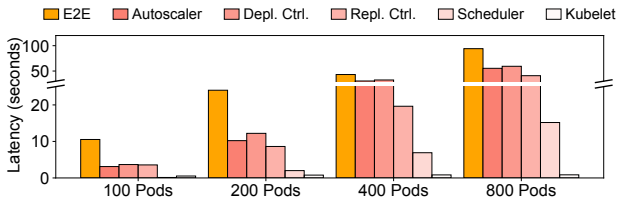
call ①, and the choices of downstream data plane by ⑤. For example, Knative [18] exposes the Knative Service API to end users and internally translates it to Deployment through the Knative Serving controller. Its data plane, based on Istio [16], monitors the Pod API for routable endpoints.

Note that unlike the *Autoscaler*, the standalone services like Istio [16], Nginx [27], or Prometheus [31] in the data plane are not part of the FaaS platform codebase. Therefore, our prototype of KUBEDIRECT optimizes ① to ④ and leaves ⑤ to the API Server, in favor of compatibility As we show in the next section, ⑤ is not the key bottleneck because it is amortized across all *Kubelets* and nodes in the cluster.

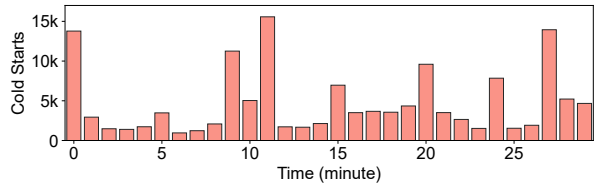
2.2 The Message Passing Bottleneck

While the state-centric APIs allow for great extensibility, it complicates the message passing between controllers. Previous works [7, 46, 47, 101] have identified several factors that make message passing expensive: (1) the amount of data exchanged, with an average of 17KB per object [46], (2) serialization/deserialization, and (3) persisting to etcd. Consequently, to avoid overwhelming the API Server and etcd, we find that Kubernetes rate-limits individual controllers in issuing API calls [2], resulting in poor performance when they need to pass a large number of objects to their downstreams. While simply relaxing the rate limits is known to cause stability issues [1, 7], tuning this configuration turns out to be labor-intensive [3–5, 62], with a strong dependency on the actual workload and hardware. Note that batching API calls will not help, as it simply evades the rate limiter but does not reduce the actual load on the API Server and etcd. In this paper, we aim to design an efficient message passing mechanism that does not require intricate tuning or fundamental refactoring of the Kubernetes architecture.

To understand the status quo overhead, we measure the end-to-end (E2E) upscaling latency with varying number of Pods in an 80-node Kubernetes cluster, and break down the latency across the controllers in the narrow waist. Figure 3a shows that the controllers face non-trivial bottlenecks except the *Kubelets*. We find that they are fast with their internal logic (100s of milliseconds) but spend most of the time passing



(a) The overhead of upscaling.



(b) The cold start rate in the Azure trace.

Figure 3: The gap between Kubernetes and serverless.

objects to the next controller (①–④ in Figure 1). Although they work in a pipelined fashion, the end-to-end latency is still dominated by the slowest stage. The *Kubelets* are more scalable because they are only responsible for the local subset of Pods (⑤). While it is possible to replicate and shard the other controllers in a similar fashion, doing so requires non-trivial refactoring of the Kubernetes codebase and even more tuning, which contradicts our design goal of minimum intrusion and maximum reuse.

The message passing bottleneck makes Kubernetes inadequate for bursty FaaS workloads where massive scaling is prevalent. Figure 3b shows the cold start rate in the Azure Functions trace [84] under a conservative 10-minute instance keepalive policy. There can be 16K cold starts in a minute, far beyond the capability of the Kubernetes control plane.

2.3 Opportunities for Efficient Message Passing

Bypassing the API Server. We note that traversing the API Server is not a fundamental requirement for the narrow waist. First, the API Server persists every state transition to etcd. However, a closer look at ①–④ in Figure 1 shows that they belong to two categories that both do not require persistence. (1) ①② are *level-triggered*; the desired number of replicas is repeatedly computed in each interation of the autoscaling loop, without needing to memorize the last decision. (2) ③④ work with *uninstantiated* Pods that only declare their resource requirements and yet free from side effects, which can be replaced across failures. Second, the API Server is responsible for resolving conflicts and serializing concurrent updates to the same object. However, the *sequential* structure of the narrow waist allows for conflict-free, one-to-one message passing. That is, each controller can *exclusively* decide the desired state of objects in the current stage (i.e., one writer) and pass each to a *single* downstream (i.e., one reader). For example, although the *Scheduler* is the upstream to all *Kubelets*, each Pod is only relevant to its designated *Kubelet*. Therefore, bypassing the API Server is practical for the narrow waist.

Eliminating the redundancy in messages. A strawman solution is to send the API objects *as is* between controllers. However, as per §2.2, prior works also identify the amount of data exchanged and serialization/deserialization as a problem [7,46,101]. Quantitatively, we find that it incurs 20%-35% overhead (§6.3). Our key observation is that most API calls in Figure 1 only update a few attributes, whereas the rest are static or predetermined, suggesting substantial redundancy. For example, when creating a Pod (③), its specification is copied from the template attribute of its parent ReplicaSet. To eliminate the redundancy, we can decouple the dynamic and static attributes and differentiate the APIs used in message passing and control loops, so that we can achieve efficiency for the former and transparency for the latter.

2.4 Challenges for State Management

The key problem of direct message passing is that it introduces a set of ephemeral objects across controllers. Unlike Kubernetes, KUBEDIRECT cannot rely on the API Server as the single source of truth. Instead, it must secure a consistent view of the cluster state across controllers in the absence of *centralized coordination*. Inspired by the sequential structure of the narrow waist, we identify an opportunity to reduce the problem of *global* consensus to *pairwise* agreements, similar to the Chain Replication protocol (CR) [81]. CR is designed to manage a chain of storage servers. It receives update requests at the head and replicates them in-sequence down the chain for linearizable reads and writes. When a server crashes, CR simply removes it from the chain and reconnects adjacent servers. However, KUBEDIRECT fundamentally differs from CR as controllers are distinct state machines that can progressively update the state of objects, rather than backups that strictly follow the primary.

We summarize two major challenges. First, certain non-idempotent controller operations, e.g. Pod scheduling, make it unsafe to perform CR-style fast-forwarding (§4.1). Instead, KUBEDIRECT must judiciously chooses between propagating or rolling back the state of controllers. Second, to seamlessly integrate with the ecosystem, KUBEDIRECT should comply with the rules of Pod lifecycle. Specifically, the transition to the *Terminating* state must be *irreversible*, requiring that termination be handled differently from upscaling, which are free from such constraints as per §2.3. Moreover, termination can be asynchronous, for downscaling, or synchronous, for preemption by high-priority services. KUBEDIRECT should correctly handle these nuances in direct message passing.

3 KUBEDIRECT Overview

The core of KUBEDIRECT is a library for controllers to exchange and reconcile their local state in a distributed manner, providing efficiency, transparency, and consistency. We build a prototype of KUBEDIRECT by integrating with the Knative [18] FaaS platform, but our library is applicable to other platforms and essentially to any chain of controllers.

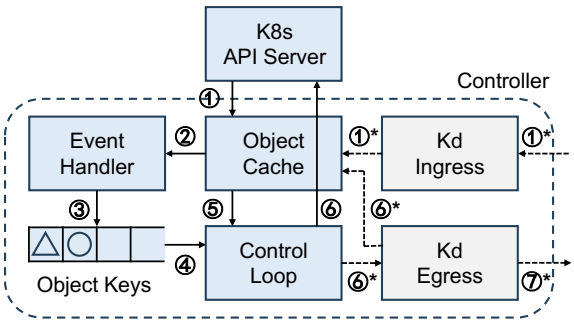


Figure 4: KUBEDIRECT's (Kd) integration with Kubernetes (K8s) controllers. The added steps are labeled with asterisks.

To let KUBEDIRECT manage the scaling of a FaaS function, users simply add a special annotation to the matching Deployment object; they can switch back to standard Kubernetes by removing the annotation. KUBEDIRECT follows the same critical path as in Figure 1, but performs direct message passing under the hood.

3.1 Architecture

We start with the standard architecture of a Kubernetes controller and then explain how KUBEDIRECT can be seamlessly integrated, as illustrated in Figure 4.

Basic architecture. Kubernetes controllers adhere to a *uniform* state-centric architecture [89,90]. A controller maintains a local cache that ① subscribes to the API Server to receive update notifications on certain objects of interest. The notifications then ② trigger a set of event handlers that decide which objects need to be reconciled and ③ push the object keys to a work queue. The main control loop then ④ dequeues the keys, ⑤ fetches the corresponding objects from the cache, and takes actions accordingly, e.g., creating or scheduling Pods. Finally it ⑥ exports its updates to the API Server.

KUBEDIRECT integration. KUBEDIRECT adds a pair of ingress and egress modules for ephemeral objects, parallel to the existing API pub-sub workflow. Specifically, we connect two adjacent controllers with a TCP-based bidirectional link. The downstream link forwards the desired state, while the upstream link sends feedback signals such as cache invalidations (§4.2). We integrate KUBEDIRECT into the basic architecture with the following steps, marked with asterisks; the circled numbers show which steps in the original workflow they are parallel to. ①* The ingress converts KUBEDIRECT's messages to standard API objects with dynamic materialization and merges them into the cache, *transparently* triggering the internal control loop. ⑥* The egress intercepts outbound API objects, ⑦* converts them to KUBEDIRECT's messages, and sends them to the downstream. Note that prior to step ⑦*, the egress can immediately populate the local cache with the latest state. This is because each controller can solely decide the state of the object it manages per the sequential structure of the narrow waist.

```
// May start with objID in case of external pointer
type KdKey {
    string attrPath
}
type KdValue union {
    string value
    KdKey ptr
}
type KdMessage {
    string objID
    dict[KdKey, KdValue] attrs
}
// Example message from Scheduler to Kubelet
KdMessage Pod {
    objID: "podX",
    attrs: {
        "spec": "replicasetY.spec.template.spec",
        "spec.nodeName": "worker1",
    }
}
```

Figure 5: The minimal message format in KUBEDIRECT.

3.2 Dynamic Materialization

Minimal message format. KUBEDIRECT decouples the dynamic and static attributes of API objects and only passes the dynamic ones for efficiency, using the minimal message format in Figure 5. *Dynamic materialization* is the process of translating to and from standard API objects to provide transparency to the control loop. Specifically, a KUBEDIRECT message is set of key-values pairs; the key references a certain attribute, whereas the value can be an arbitrary literal, representing dynamic attributes, or an external key that points to some static attribute in another object. For example, Figure 5 shows a message from the *Scheduler* to the *Kubelet* representing "PodX" on node "worker1". It includes a external pointer to the "replicasetY" object. The *Kubelet* can retrieve the ReplicaSet from its local cache, copy its template, and use it to construct the target Pod.

Extensibility. Because the Kubernetes APIs have a well-defined schema, controllers can use reflection [13] to decode KUBEDIRECT messages such that they remain loosely coupled. Our handshake protocol (§4.2) further allows joining new controllers into the narrow waist, e.g., one after the *Scheduler* that sets node-specific variables for Pods.

Complexity. In terms of the upscaling process in Figure 1, KUBEDIRECT has the same algorithmic complexity as ①–⑤, but significantly reduces the constant factor. Apart from bypassing the API Server, our minimal message format uses up to 64B per object, whereas the size can be up to 17KB in Kubernetes [46]. KUBEDIRECT can further reduce the message passing overhead by batching messages.

4 State Management

This section presents the state management of KUBEDIRECT that maintains consistency across controllers and ensures end-to-end semantics of instance lifecycle. We start with an anal-

ysis of the problem in §4.1 and introduce our mechanisms for instance provisioning in §4.2 and termination in §4.3. We discuss the correctness of our design in §4.4.

4.1 Problem Analysis

As per §2.3, the *Autoscaler* and the *Deployment controller* are level-triggered and idempotent, for which fast-forwarding suffices. The focus of this section is reconciling the state of Pod objects across the *ReplicaSet controller*, the *Scheduler*, and the *Kubelets*, in the context of upscaling. We discuss downscaling and termination in §4.3, which have more subtle semantics. Also note that although it currently takes only three stages of controllers to manage Pods, our analysis and approach applies to arbitrary numbers of sequential stages.

Why is fast-forwarding unsafe? While having a similar sequential structure, Chain Replication (CR) [81] is not applicable to the narrow waist, where controllers can update the desired state independently and non-idempotently, rather than replicating the upstream. Compounded with the asynchrony between controllers, naive fast-forwarding can lead to unexpected anomalies.

Anomaly #1. Suppose a *Kubelet* temporarily disconnects from the *Scheduler*. Meanwhile, it evicts a Pod due to resource contention. However, the *Scheduler* later reconnects, finds the Pod missing, and fast-forwards it again. The *Kubelet* could instantiate the terminated Pod again with a different IP. This is not allowed in Kubernetes because it violates the rules of instance lifecycle (§4.3), and could have unexpected effects over external controllers outside the narrow waist.

Anomaly #2. Suppose the *Scheduler* restarts after a crash but only reconnects with a subset of *Kubelets*. It is possible that a particular Pod is cached both at the *ReplicaSet controller* (the upstream) and the unreachable *Kubelet* (the downstream), but not at the *Scheduler* itself. The *ReplicaSet controller*, unaware of the Pod's placement by the *Scheduler* prior to the crash, fast-forwards the Pod again. The *Scheduler* could then assign it to a different node, leading to undefined behavior.

Why assume a hierarchical write-back cache? We draw an important conclusion from the anomalies: due to the state being mutable through the narrow waist, *the downstream becomes the single source of truth*. CR assumes the opposite where the head server in the chain acts as the primary. Therefore, upstream controllers must make decisions based on the downstream state, rather than naively fast-forwarding their own. It follows that the problem is similar to managing a *hierarchical write-back cache*. Specifically, the upstream controller opportunistically issues writes to the downstream, which acknowledges them but does not guarantee that they will be applied due to active cancellation or passive failures. Our job is to reflect the downstream changes to the upstream as *cache invalidations*. We note that our analogy is practical because (1) the sequential structure of the narrow waist implies that the downstream holds all the state the upstream needs to know; (2) controllers are designed to be tolerant of asynchronous notifications, be it cache invalidation or API Server feed, and will continuously reconcile whatever the observed cluster state to match the desired one.

4.2 Hierarchical Write-Back Cache

We build the hierarchical cache on top of the bidirectional links between adjacent controllers (§3.1). We consider two types of cache invalidations:

- **Hard invalidation** occurs whenever the upstream disconnects and reconnects with the downstream. It *atomically* resets the upstream state to the downstream's before resuming its control loop.
- **Soft invalidation** occurs between a live and connected pair of controllers where the downstream *incrementally* informs the upstream of its state changes, e.g., when the *Scheduler* sets the target node for a Pod.

In brief, our goal is to use a combination of hard and soft invalidations to ensures a consistent view of the cluster across controllers, in spite of their asynchrony. We defer the proofs of correctness to §4.4.

Specifically, hard invalidation is to provide a consistent initial state across controllers, such that subsequent state exchanges, either forward or backward, can be incremental. Hard invalidation also provides a uniform solution to controller or network failures, which is necessary because it can be difficult to proactively distinguish between the two in distributed systems [56, 69, 70, 72, 78, 97, 103].

We implement soft invalidation with dynamic materialization, similar to the forward message passing (§3.2). We now present the design of the handshake protocol that implements hard invalidation.

The handshake protocol. Figure 6 shows the pseudocode of the handshake protocol. The protocol is initiated by the upstream controller (as the client) towards the downstream (as the server). Once complete, it returns a connection handle for subsequent state exchanges.

Specifically, the downstream controller listens for incoming connections and responds with its local state. Because it is the source of truth in our hierarchical cache analogy, it immediately finishes its part of the handshake.

The upstream controller, after receiving the downstream state, can operate in two modes. (1) In recover mode, the controller has crash-restarted and has empty local state. It therefore applies the downstream state as is. (2) In reset mode, the controller has non-empty local state and must actively resets it (line 7). Specifically, it computes the change set between its local state and the downstream's. There are two cases for each local object. If the object is also present in the downstream, it is simply overwritten and marked dirty. Otherwise, it is marked as invalid but not immediately removed from local state; it is hidden from the internal control loop

```
def ClientHandshake(cache):
    conn = tcp.connect()
    server_state = conn.recv()
    if cache.is_empty(): # recover mode
        cache.set(server_state)
    else: # reset mode
        change_set = cache.diff(server_state)
        cache.set(server_state)
    # use soft invalidation to propagate change_set
    return conn, change_set

def ServerHandshake(cache):
    conn = tcp.accept()
    serer_state = cache.get_state()
    conn.send(serer_state)
    return conn
```

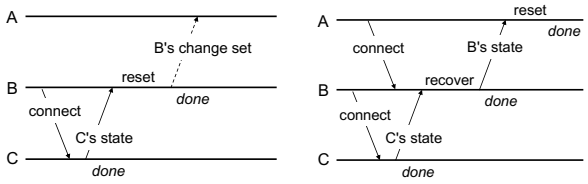
Figure 6: Pseudocode of the handshake protocol.

such that it is equivalent to being deleted. After completing the handshake, the controller will use soft invalidation to notify its further upstream of the marked objects in either case. Because soft invalidation is asynchronous and may overlap with the control loop, retaining the set of objects with the invalid mark allows the controller to ignore any incoming updates for those objects. These objects can be discarded once the further upstream acknowledges them. Eventually, the marked change set will be propagated to all upstreams. Note that this is guaranteed even though soft invalidations are best-effort; should any upstream crash, it would have to populate its state *"the hard way"*, which would surely incorporate all changes with respect to the downstream source of truth.

Atomicity. The handshake protocol should atomically reset the upstream cache. This is straightforward for one-to-one connection. A special case arises for the one-to-many connections from the *Scheduler* to the *Kubelets*. For atomicity, we open connections concurrently and grant a grace period in which all *Kubelets* should respond. If some fail to do so in time, the *Scheduler* performs *cancellation* (detailed in §4.3) by marking these nodes as invalid and draining all KUBEDIRECT-managed Pods on them.

Overhead. Our approach has little extra overhead compared to Kubernetes, since the API Server also needs to populate or refresh the state of controllers while using the raw API objects rather than our lightweight dynamic materialization. KUBEDIRECT also employs another optimization: when running in reset mode, the upstream only requests the *version number* of objects from the downstream in the first round; only the change set is exchanged in the second round. As we show next, the key difference with Kubernetes is that KUBEDIRECT may incur multiple hops under concurrent failures, but this is not a major concern because the narrow waist is shallow. We validate the efficiency of our approach in §6.3.

Autonomous recovery. The handshake protocol provides a uniform solution to controller or network failures. Figure 7 shows an example. Network disconnections can be fixed with



(a) Handling B–C disconnection. (b) Handling B's crash failure.

Figure 7: Failure handling using the handshake protocol.

a single round of handshake in reset mode. Crash failures can be fixed in two rounds, first with the downstream in recover mode, then with the upstream in reset mode. Following this downstream-first rule, the protocol naturally extends to multi-point failures. Moreover, we can use it to join new controllers into the narrow waist, similar to handling crash failures.

4.3 Instance Lifecycle Management

This section presents our design to support downscaling in the hierarchical cache and enforce the high-level yet more subtle requirement: the state transitions observed by each controller should adhere to the conventions on Pod lifecycle. This is important because KUBEDIRECT is designed to seamlessly integrate with the Kubernetes ecosystem.

State diagram. We outline a simplified state diagram of Pod lifecycle. A Pod starts in the *Pending* state, and transitions to *Running* when it becomes ready. During this process, it may enter the *Terminating* state, which should be irreversible per the Kubernetes convention. A *Terminating* Pod will be eventually removed from the cluster state.

Provisioning vs. Termination. Without loss of generality, we consider two types of events that trigger lifecycle transitions: Pod provisioning and Pod termination. We identify a fundamental difference in their semantics that calls for separate handling. Pod provisioning is well-suited to the opportunistic state forwarding and inherently tolerant to sporadic downstream reset, while Pod termination is not. This is because resetting can have different implications in the two cases. For Pod provisioning, resetting leads to the loss of certain assumed Pods, which can be considered as the transition from *Pending* to *Terminating*. However, for Pod termination, resetting a Pod that is assumed to be *Terminating* could "revive" it, e.g., when the decision to terminate is dropped somewhere downstream, which is undefined behavior.

Consequently, it turns out that the upstream *once again holds the source of truth* in the context of *active* Pod termination. Note that we distinguish *active* termination from the *passive* case, e.g., Pod lost due to node crashing, because the former involves controllers assuming certain transitions should happen, while the latter can simply inform controllers of the status quo. We also note that termination is inherently *idempotent*, i.e., it either removes an existing Pod or becomes a no-op, which adheres to the assumption of the Chain Replication Protocol (CR) [81].

Replicating Tombstones. Based on the above observation, we introduce a special type of API object internal to the narrow waist—the Tombstones. A Tombstone contains the identifier of a certain Pod, which marks it for *best-effort* termination within the controller's *current session*, lasting until the controller crashes. During the current session, the controller performs CR-style replication of the Tombstones along the opportunistic forwarding pipeline. The only possibility of losing a Tombstone is when *every controller the Tombstone has been replicated to crashes*, implying that the termination, whereas best-effort, has a high possibility of success. A controller can stop replicating a Tombstone when the referenced Pod is not locally present. This is well-defined based on the fact that new Pods are always created upstream and propagated downstream. The controller then issues soft invalidations to the upstream to trigger cascade garbage collection of both the Pod and its Tombstone.

Practical examples. We demonstrate how to handle real scenarios of active termination based on Tombstones.

- **Downscaling** occurs when the *ReplicaSet controller* receives a smaller desired scale. It selects a set of Pods to terminate, creates Tombstones, and replicates them downstream. It does so in an *asynchronous* fashion, i.e., it continues to process subsequent scaling requests, but uses the Tombstones to track the Pods awaiting termination, to avoid unnecessary thrashing.
- **Preemption** can occur at the *Scheduler* or the *Kubelets*. Because the *Kubelets* are at the tail of the narrow waist, Tombstones are unnecessary, and invalidations alone suffice. The *Scheduler* preempts in a similar way as downscaling, except that it must do so in a *synchronous* fashion. This is because there could be a high-priority Pod whose placement is conditioned on the termination of the victim Pod. Therefore, the *Scheduler* waits for the invalidation signal from the downstream *Kubelet*.
- **Cancellation** occurs in a special case where the *Scheduler* connects with only a subset of *Kubelets*. The *Scheduler* could be running, in which case it wants to cancel the Pods on the unreachable node; or have crash-restarted, as in *Anomaly #2* (§4.1), in which case it does not know the Pods running on the unreachable node, and wants to drain the node to enforce a consistent view of the cluster state. Due to the disconnection, KUBEDIRECT cannot rely on direct message passing to inform the target *Kubelet* of its decision. Instead, the *Scheduler* marks the corresponding Node API object as invalid through the API Server. The target *Kubelet* is expected to drain all KUBEDIRECT-managed Pods once it sees the mark. Meanwhile, the *Scheduler* can safely assume that the related Pods are irreversible terminated, and can proceed to send invalidation signals.

Remark. Tombstones are similar in functionality to the invalid marks in the handshake protocol. However, we decouple the two because they are propagated in the opposite direction, follow different sources of truth, have different lifetimes, and require separate handling in case of failures.

4.4 Correctness

Like Kubernetes, KUBEDIRECT has the same guaranteed convergence to the desired state, i.e., it ensures that eventually there will be the desired number of Pods running in the cluster. Because reasoning over such a complicated distributed system involves numerous nuances, we use TLA+ to verify the end-to-end properties of KUBEDIRECT in our technical report [80]. Nevertheless, to help understand the correctness of KUBEDIRECT, we highlight two important properties.

The Safety Invariant. *Let P be a predicate over the cluster state, e.g., Pod X is assigned to node Y, Pod Z is in Terminating, etc. If P holds at a suffix of the sequence of controllers, then it eventually holds at all upstreams.*

The Liveness Assumption. *The narrow waist becomes totally connected infinitely often and sufficiently long for a round of end-to-end message passing.*

As described in §4.2, KUBEDIRECT achieves the *safety* invariant through the combination of soft and hard invalidation—soft invalidation incrementally ensures consistency between controllers while they are connected, and hard invalidation forcefully establishes a consistent basis upon reconnection. The *liveness* assumption ensures that changes to the cluster state at any controller can be eventually propagated through the entire narrow waist. Combining the two implies that controllers cannot diverge indefinitely, such that they can correctly reconcile and amend the cluster state in spite of failures. Furthermore, we enforce Pod lifecycle as a restriction on the transitions allowed during this process, guaranteeing end-to-end semantics.

5 Implementation

We implement KUBEDIRECT based on Kubernetes v1.32.0, changing ∼150 LoC per controller in the narrow waist. We deploy Knative on top of KUBEDIRECT and retrofit its *Autoscaler*. As a proof of compatibility, the KUBEDIRECT-based Knative can directly work with the Prometheus [31], Kourier [19], and Istio [16] extensions for monitoring and networking. We open-source KUBEDIRECT at `https://github.com/TomQuartz/kubedirect-ae`.

Exclusive ownership. KUBEDIRECT has *exclusive* ownership over the ephemeral state in the narrow waist. Pods are protected because they remain hidden until the very end. ReplicaSets and Deployments are not. KUBEDIRECT guards their `replicas` fields using Kubernetes admission control [26]. External updates to the guarded fields will be rejected; non-essential fields such as annotations are unaffected.

Pod discovery. One can already obtain the routable FaaS endpoints by subscribing to the Pod API that the narrow waist publishes to. In practice, Kubernetes also offers the Service

Cluster Manager	Control Plane	Sandbox Manager
K8s	K8s	Kubelet
Kd	Kd	Kubelet
K8s+	K8s	Dirigent's
Kd+	Kd	Dirigent's

(a) Microbenchmarks

FaaS Platform	FaaS Orchestrator	Cluster Manager
Kn/K8s	Knative	K8s
Kn/Kd	Knative	Kd
Dr/K8s+	Dirigent's	K8s+
Dr/Kd+	Dirigent's	Kd+

(b) End-to-End experiments

Figure 8: Baselines in our evaluation.

API [25] that selects certain endpoints and abstracts them with a single static IP address. Specifically, the *Endpoints controller* monitors the Service selector and finds matching Pods. It then publishes the list of endpoints through the Endpoints API to the per-node *Kube-proxies*, which handles address translation. This process also incurs many API calls. However, we note that the Endpoints are *read-only* transformations of Pods. Therefore, we optimize the *Endpoints controller* to directly stream the Endpoints to the *Kube-proxies*.

High-availability. KUBEDIRECT is compatible with the high-availability (HA) setup of Kubernetes where controllers are replicated in primary-back mode [15]. This is because a controller may become operational only if it wins the leader election, such that our assumption of a sequential structure still holds. The new leader should run the handshake protocol upon takeover.

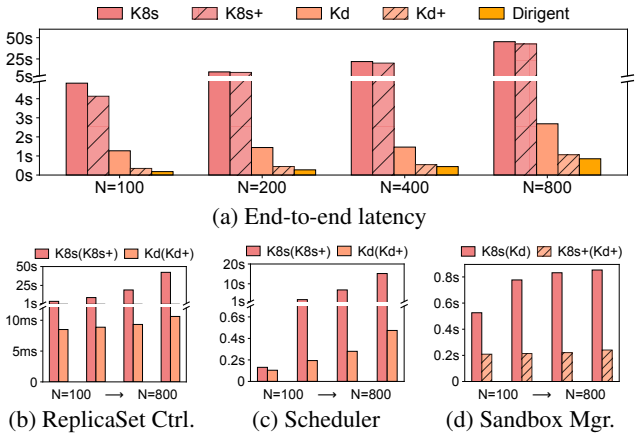
6 Evaluation

Methodology. We set up an 80-node cluster of xl170 instances on CloudLab [9]. Each node has ten Intel E5-2640v4 2.4 GHz CPU cores, 64 GB RAM, and 25 Gbps interconnect.

Our primary baselines are (1) the Kubernetes (K8s) v1.32.0 cluster manager, which is the codebase of KUBEDIRECT (Kd), (2) Knative v1.15.0, a popular Kubernetes-based FaaS platform, and (3) Dirigent, the state-of-the-art clean-slate FaaS platform. As per Figure 2, a FaaS platform generally consists of the FaaS orchestrator and the base cluster manager; the latter consists of the control plane and the sandbox manager (e.g., *Kubelet*) on worker nodes. To understand the benefits of KUBEDIRECT, we set up different combinations of FaaS orchestrators and sandbox managers as control variables as shown in Table 8. Note that it is possible to do so because KUBEDIRECT focuses on message passing itself and does not depend on specific implementations.

For the cluster manager (Figure 8a), Kd is the optimized K8s with direct message passing. We also implement K8s+ and Kd+ by replacing the *Kubelet* part of K8s and Kd with Dirigent's custom sandbox manager. We run microbenchmarks on all variants to evaluate KUBEDIRECT's primitives.

For the FaaS platform (Figure 8b), we deploy Knative on K8s, denoted as Kn/K8s, and on Kd, denoted as Kn/Kd. We also port Dirigent's orchestrator to K8s+ and Kd+, denoted as Dr/K8s+ and Dr/Kd+; Dr/K8s+ and Dr/Kd+ replace Dirigent with the respective control planes while retaining the orchestrator and sandbox manager. We run end-to-end workloads on these baselines. For fair comparison, we compare Kn/Kd with Kn/K8s, and Dr/Kd+ with Dr/K8s+ and Dirigent.



(a) End-to-end latency

(b) ReplicaSet Ctrl. (c) Scheduler (d) Sandbox Mgr.

Figure 9: Upscaling latency for varying # of Pods.

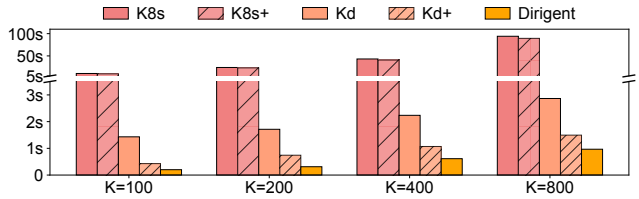
6.1 Microbenchmarks

We scale out N Pods for K FaaS functions in an M-node cluster, and evaluate the scalability of the baselines over the three dimensions. We use a strawman *Autoscaler* that issues one-shot scaling call per function (Deployment). We measure the latency between the scaling call and all Pods being ready, and also the time each controller spent during this process.

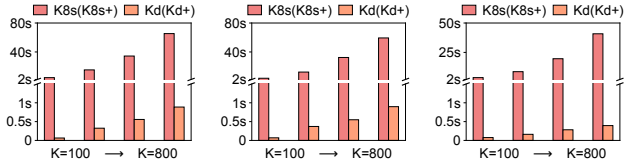
***N*-scalability (Pods)**. We set K to one, M to 80, and vary N from 100 to 800 Pods. As shown in Figure 9a, Kd is 3.7–16.9× faster than K8s, and Kd+ is 11.9–40.0× faster than K8s+. K8s+ is marginally faster than K8s despite using Dirigent's optimized sandbox manager, because the bottleneck is in the control plane. The benefit of a faster sandbox manager is only perceptible when the control plane is fast enough—Kd+ achieves the same sub-second latency as Dirigent.

We further break down the latency for the *ReplicaSet controller*, the *scheduler*, and the sandbox manager in Figure 9b–9d. For the first two, K8s is equivalent to K8s+, and Kd to Kd+, because they are of the same control plane respectively. For the sandbox manager, K8s is equivalent to Kd, and K8s+ to Kd+, due to the respective common data plane. Kd improves K8s by two orders of magnitude for the *ReplicaSet controller*, and one for the *scheduler*. The improvements come from two aspects. First, Kd is free from the API rate-limiting in K8s. Second, Kd's message passing itself has sub-millisecond latency, whereas it takes 10–35 milliseconds in K8s. Consequently, KUBEDIRECT approaches the raw performance of the internal control loops, adding up to 10% overhead in our measurements. The sandbox manager (Figure 9d), in contrast, is relatively more scalable whether optimized or not.

We do not break down for the *Autoscaler* or *Deployment controller* because we set $K = 1$ such that they both issue a single scaling call, with no scalability issue. We study their performance next while we vary the number of functions K.



(a) End-to-end latency



(b) Autoscaler (c) Deployment Ctrl. (d) ReplicaSet Ctrl.

Figure 10: Upscaling latency for varying # of functions.

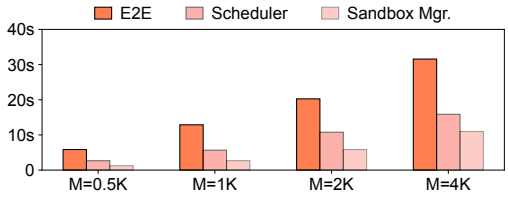


Figure 11: Upscaling latency for varying # of nodes.

***K*-scalability (Functions)**. We set M to 80, and vary K from 100 to 800 functions with one Pod per function, i.e., $N = K$. We omit the breakdown for the *scheduler* and the sandbox manager because there is no difference from the previous case in their perspective. As shown in Figure 10a, Kd is 7.4–32.8× faster than K8s, and Kd+ is 22.7–59.8× faster than K8s+. K8s (K8s+) experiences higher latency in the *Autoscaler* and the *Deployment controller* (Figure 10b and 10c) than that in Figure 9, because it has to scale up the same number of Pods while doing it on a per-function basis. KUBEDIRECT improves the former by 39.5–70.3× and the latter by 31.2–50.1×. In terms of the *ReplicaSet controller* (Figure 10d), K8s's performance is similar to that in Figure 9; Kd's latency is increased by $60 - 350$ milliseconds because per-function scaling results in less batching. However, the impact on the end-to-end latency is marginal due to inter-controller pipelining.

***M*-scalability (nodes)**. Finally, we evaluate KUBEDIRECT's scalability in large clusters. We vary the number of nodes M from 500 to 4000 and scale up five Pods per node. Because we do not have a large enough real cluster, we set up fake nodes with simulators for the *Kubelets*. Figure 11 shows that Kd can scale up 20K Pods in 30 seconds. We see a substantial increase in the latency of the *Scheduler* because it has to consider more nodes during scheduling; KUBEDIRECT's message passing adds up to 6% overhead. The sandbox manager also experiences higher latency due to the ∼20K concurrent API calls as the Pods get published. Note that each sandbox manager still follows its API rate limit; the high load on the API Server is a problem inherent to large Kubernetes clusters [7], which limits the maximum size to 5000 nodes [6].

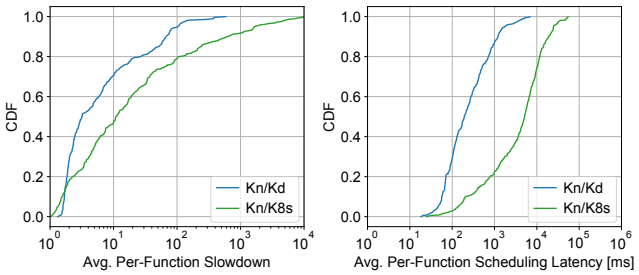


Figure 12: End-to-end performance on Knative-variants.

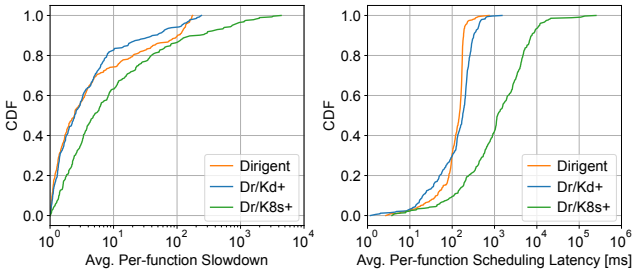


Figure 13: End-to-end performance on Dirigent-variants.

Downscaling. We also evaluate the downscaling performance of KUBEDIRECT under the same setups. Because the number of API calls or messages required is approximately the same as upscaling, we observe similar characteristics in performance. Specifically, for K-scalability, Kd is 6.9–30.3× faster than K8s, and Kd+ 16.8–45.2× faster than K8s+.

6.2 End-to-End FaaS Workload

We run the same FaaS workload as Dirigent (available in its artifact [10]). Specifically, it is a 30-minute clip of the Microsoft Azure Functions trace [84] with 500 functions and 168K invocations. The durations of invocations are sampled based on the percentiles given in the trace; to handle an invocation, a function instance busy loops with the SQRTSD x86 instruction for the requested duration.

In line with Dirigent, we consider two performance metrics: the average request slowdown and the average request scheduling latency. The former is the end-to-end latency divided by the requested execution time, and the latter is the time from the invocation arrival to the beginning of its processing by some instance. Because the execution times and invocation frequencies of different functions in the trace can vary by orders of magnitude, we group the metrics by function and plot the overall CDF.

Figure 12 and 13 shows the results for baselines in Table 8b. For the Knative-variants, Kn/Kd improves the median (p99) slowdown by 3.5× (19.4×) and the median (p99) scheduling latency by 26.7× (10.3×) compared to Kn/K8s. For the

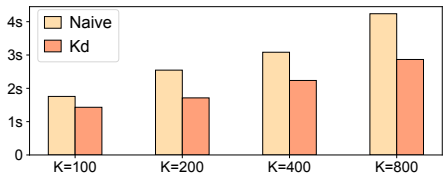
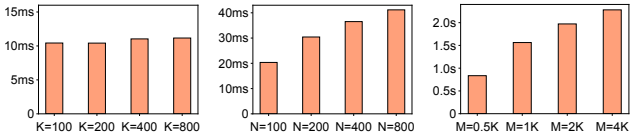


Figure 14: Benefits of on-demand materialization.



(a) Autoscaler (b) ReplicaSet Ctrl. (c) Scheduler

Figure 15: Failure handling with hard invalidation.

Dirigent-variants, Dr/Kd+ improves the median (p99) slowdown by 2.0× (10.4×) and the median (p99) scheduling latency by 6.6× (134×) compared to Dr/K8s+. Despite using the Kubernetes code base and retaining compatibility, Dr/Kd+ achieves similar performance as Dirigent.

We find that the long tails of Kn/K8s and Dr/K8s+ come from periodic bursts of *cold* functions, which are infrequent as individuals but tend to arrive simultaneously [46, 84]. This is not a serious problem for Kn/Kd, Dr/Kd+, or Dirigent, because the control plane can efficiently absorb the bursts; instead, the tails of these baselines are caused by functions with very short durations. We also observe a 67% reduction in the number of cold starts when replacing K8s with Kd. This is due to the autoscaling policy of Knative and Dirigent that computes the desired replicas count based on the number of inflight requests. Faster upscaling can effectively reduce the queuing effect and prevent the *Autoscaler* from desperately scaling up even more replicas.

6.3 Ablation Study

Dynamic materialization. To understand the necessity of dynamic materialization, we compare it with naive direct message passing that sends full API objects, which avoids the overhead of persistence but not serialization and deserialization. We use the *K*-scalability setup in §6.1. Figure 14 shows that the naive approach incurs 20–35% higher latency than KUBEDIRECT, which justifies our design choice.

Failure handling with hard invalidation. As per §4.2, KUBEDIRECT recovers from controller or network failures via hard invalidation, i.e., the handshake protocol. To measure its overhead and the impact on failure recovery, we reset the local state in each controller to force handshakes as if in crash-restarts We use the *K*-scalability setup §6.1 to populate the cache of the *Autoscaler* and *Deployment controller*, and use the *N*−, and *M*-scalability setups for the *ReplicaSet controller*, and *Scheduler* respectively. Figure 15 shows the result. The *Autoscaler* and *Deployment controller* (not shown) have similar and negligible overhead; this is because they are level-triggered and idempotent, so we can skip cache rollbacks altogether. The *ReplicaSet controller* has sub-linear overhead because Pods can be exchanged in batches. The *Scheduler* also has sub-linear overhead because it handshakes with the *Kubelets* in parallel.

Termination with soft invalidation. We measure the overhead of soft invalidations and its impact on *synchronous* termination, e.g., preemption, which needs to wait for downstream signals (§4.3). Specifically, one hop of soft invalidation takes 0.5–1.2 milliseconds, similar to forward message passing. The end-to-end preemption latency, including two hops and the processing at the *Kubelet*, is 6.2–13.4 milliseconds, compared to the 10–35 milliseconds latency of a standard API call.

7 Discussion

Deploying KUBEDIRECT. KUBEDIRECT cannot be deployed as a bolt-on extension to Kubernetes because the *Scheduler* and the *Kubelet* must be aware of the ephemeral Pods to correctly allocate resources. However, we do support a rollout deployment by gradually replacing the controllers with KUBEDIRECT's, in the *reverse* order of the narrow waist. Because KUBEDIRECT is fully compatible with the Kubernetes APIs, the downtime can be kept to a minimum. We also expect long-term compatibility with future Kubernetes releases, because the narrow waist has been stable through many years of development.

Admission control and multi-tenancy. An important caveat is that bypassing the API Server also implies bypassing its authentication [23], authorization [24], and admission control [22] stages that could validate and amend the API requests. Therefore, KUBEDIRECT is intended for trusted-tenant clusters, where the FaaS orchestrator, as the first-hand user of the cluster manager, should only make valid requests that do not require additional policy enforcement. Because the end users, whereas untrusted, can only interact with the orchestrator, malicious requests can be rejected at an earlier stage. For example, the orchestrator can enforce the resource quota per the end user's namespace and validate the Pod specifications, before submitting scaling requests to KUBEDIRECT. Also note that while KUBEDIRECT offers a fast path for trusted workloads, the standard path with full admission semantics is still available to handle untrusted tenants.

Webhook and extension points. Webhook [26] is a special type of admission controller that executes as HTTP callbacks, whereby users inject custom validation and mutation logic besides the built-in policies. We plan to support webhooks in future work. Specifically, users register webhooks as usual and let the API Server "push down" the HTTP endpoints to the ingress/egress modules in KUBEDIRECT (§3), which then invoke them on behalf of the API Server.

Observability. The ephemeral Pods in KUBEDIRECT are invisible to external controllers until they exit the narrow

waist, which implies that API-based monitoring tools cannot observe intermediate events such as Pod creation and scheduling. However, as discussed above, administrators can still insert webhook-based monitors into the narrow waist.

Horizontal scaling. KUBEDIRECT vertically scales the capability of a single cluster. An orthogonal approach is to horizontally scale out multiple sub-clusters to amortize the overhead. However, based on Figure 3, handling the cold starts of the Azure trace requires tens of sub-clusters, leading to management burdens, load imbalance, and resource fragmentation [46]. While KUBEDIRECT can also scale horizontally, the fact that it is 30× faster than Kubernetes means it needs much fewer sub-clusters for the same throughput. Another approach is to replicate and shard the controllers in the narrow waist (§2.2). However, doing so requires non-trivial code changes, even more configuring and tuning, and risks load imbalance and conflicts across shards, which contradicts our design goal of minimum intrusion and maximum reuse.

Stability. While KUBEDIRECT increases the throughput of scaling, the *Kubelets* still need to call the Kubernetes API. However, this will not compromise the stability of the API Server because the *Kubelets* still follow the API rate limits.

Availability. Although KUBEDIRECT requires direct connections between controllers, it typically offers the same availability as the decoupled approach of Kubernetes. In both systems, a controller can only make substantial progress when its desired state can be processed by its downstream. Also note that disconnections between controllers will not affect the progress of non-KUBEDIRECT managed objects, because we enforce a clean separation of ownership.

General applicability and design principles. While KUBEDIRECT focuses on optimizing Kubernetes alone, its general applicability is founded on the wide popularity of the latter, whose rich features and ecosystem have fostered a large number of cloud services. In terms of design principles, our techniques are useful under two preconditions: (1) disaggregated cluster state; (2) sequential control plane dependencies. (1) has been adopted and developed by the Borg [42], Omega [83], Kubernetes [21] lineage at Google. We also note that (2) is in fact a consequence of (1): to avoid conflicting updates, it is natural to perform stage-by-stage processing.

8 Related Work

Cluster managers are responsible for governing the cloud infrastructure [29,41,48,53,54,57]. Modularity and extensibility are important design principles. Kubernetes [21], Borg [94], Twine [93], Omega [83], and vSphere [33] adopt state-centric APIs to break down the control plane into loosely coupled controllers. However, they are traditionally designed for long-running workloads where orchestration can be amortized over time [41,54], contradicting the bursty and short-lived nature of FaaS [66]. KUBEDIRECT presents a practical design to bridge this gap.

FaaS orchestration is an active research area [43,58,95,102]. Many works present new autoscaling, load-balancing, and placement algorithms and realize them on top of existing cluster managers [34, 37, 40, 64, 67, 86, 87]. Dirigent [46] and others [44,51,75,85,91] identify the mismatch between FaaS and legacy managers and propose novel clean-slate architectures. KUBEDIRECT is inspired by their observations and boils down the problem to message passing. It retains existing architecture and APIs for compatibility.

State machine replication is a common approach in building fault-tolerant distributed services [35,38,39,45,50,52,68,73,77]. It ensures that commands are executed in a consistent order across replicas. KUBEDIRECT is particularly inspired by the Chain Replication protocol [81] that assumes a sequential structure like the narrow waist. However, KUBEDIRECT is faced with a novel and more complicated problem where the state machines are heterogeneous and can progressively modify their state.

Speculative execution is a key idea behind KUBEDIRECT's fast upscaling, where controllers opportunistically drive the cluster to its desired state instead of committing step by step. This technique has been used in a wide context ranging from distributed systems [52, 65, 73], FaaS serving [88], to large language model inference [74, 92]. The idea is to replace expensive operations with cheap ones that do not *guarantee* but *very likely* provide the desired property, and implement safeguard mechanisms to ensure correctness.

9 Conclusion

KUBEDIRECT retrofits Kubernetes to handle bursty FaaS workloads. The common narrow waist structure across FaaS platforms allows us to achieve both efficiency and compatibility. We employ lightweight direct message passing between controllers to bypass the API Server bottleneck. In spite of the asynchrony of controllers, ephemerality of their state, and the non-idempotence of their operations, we implement a hierarchical write-back cache through the narrow waist that provides a consistent view of cluster state. In addition, we specifically handle termination to comply with the convention on instance lifecycle. KUBEDIRECT has similar performance as the state-of-the-art system Dirigent.

Acknowledgments. We thank our shepherd, Atul Adya, and the anonymous reviewers for their insightful feedback. We also thank Ke Zhang for his help on the implementation and evaluation of KUBEDIRECT in an early stage. This work was supported in part by the National Natural Science Foundation of China under Grant 62325201 and the Scientific Research Innovation Capability Support Project for Young Faculty under Grant ZYGXQNJSKYCXNLZCXM-I1. Xin Jin and Xuanzhe Liu are the corresponding authors. Sheng Qi, Zhiquan Zhang, Xuanzhe Liu, and Xin Jin are also affiliated with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education.

References

[1] etcd out-of-memory issue. https://github.com/kubernetes/kubernetes/issues/18266, 2018. Accessed 2025-04-06.

[2] Kubernetes client qps limits. https://github.com/kubernetes/kubernetes/issues/14955, 2018. Accessed 2025-04-06.

[3] Kubernetes scalability tuning over 10k nodes. https://alibaba-cloud.medium.com/how-does-alibaba-ensure-the-performance-of-system-components-in-a-10-000-node-kubernetes-cluster-ff0786cade32, 2019. Accessed 2025-04-06.

[4] Kubernetes scalability tuning over 4k nodes. https://medium.com/paypal-tech/scaling-kubernetes-to-over-4k-nodes-and-200k-pods-29988fad6ed, 2022. Accessed 2025-04-06.

[5] Kubernetes scalability tuning over 1k nodes. https://medium.com/@nixndme/how-to-scale-kubernetes-to-1000-nodes-lessons-from-a-leading-ai-and-tech-project-a857c3e3663d, 2023. Accessed 2025-04-06.

[6] Considerations for large clusters. https://kubernetes.io/docs/setup/best-practices/cluster-large/, 2024. Accessed 2025-04-06.

[7] Kubernetes api server performance issue in a large-scale cluster. https://github.com/kubernetes/kubernetes/issues/123986, 2024. Accessed 2025-04-06.

[8] Artifact hub. https://artifacthub.io/, 2025. Accessed 2025-04-06.

[9] Cloudlab. https://www.cloudlab.us/, 2025. Accessed 2025-04-06.

[10] Dirigent artifact. https://github.com/eth-easl/dirigent, 2025. Accessed 2025-04-06.

[11] Etcd. https://etcd.io/, 2025. Accessed 2025-04-06.

[12] Fission. https://fission.io/, 2025. Accessed 2025-04-06.

[13] Golang reflection. https://go.dev/blog/laws-of-reflection, 2025. Accessed 2025-08-20.

[14] Hcp terraform operator for kubernetes. https://github.com/hashicorp/hcp-terraform-operator, 2025. Accessed 2025-04-06.

[15] Highly available topology for kubernetes. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/, 2025. Accessed 2025-04-06.

[16] Istio. https://istio.io/, 2025. Accessed 2025-04-06.

[17] Jenkins. https://www.jenkins.io/, 2025. Accessed 2025-04-06.

[18] Knative. https://knative.dev/docs/, 2025. Accessed 2025-04-06.

[19] Kourier. https://github.com/knative-extensions/net-kourier, 2025. Accessed 2025-04-06.

[20] Kubeless. https://github.com/vmware-archive/kubeless, 2025. Accessed 2025-04-06.

[21] Kubernetes. https://kubernetes.io/, 2025. Accessed 2025-04-06.

[22] Kubernetes admission control. https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/, 2025. Accessed 2026-02-06.

[23] Kubernetes authentication. https://kubernetes.io/docs/reference/access-authn-authz/authentication/, 2025. Accessed 2026-02-06.

[24] Kubernetes authorization. https://kubernetes.io/docs/reference/access-authn-authz/authorization/, 2025. Accessed 2026-02-06.

[25] Kubernetes services and endpoints. https://kubernetes.io/docs/concepts/services-networking/service/, 2025. Accessed 2025-04-06.

[26] Kubernetes webhooks. https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/, 2025. Accessed 2026-02-06.

[27] Nginx. https://nginx.org/, 2025. Accessed 2025-08-20.

[28] Openfaas. https://www.openfaas.com/, 2025. Accessed 2025-04-06.

[29] Openstack. https://www.openstack.org/, 2025. Accessed 2025-04-06.

[30] Openwhisk. https://openwhisk.apache.org/, 2025. Accessed 2025-04-06.

[31] Prometheus. https://prometheus.io/, 2025. Accessed 2025-04-06.

[32] Redis. https://redis.io/, 2025. Accessed 2025-04-06.

[33] vsphere: Unified management for containers and vms. https://www.vmware.com/products/vsphere.html, 2025. Accessed 2025-04-06.

[34] Mania Abdi, Samuel Ginzburg, Xiayue Charles Lin, Jose Faleiro, Gohar Irfan Chaudhry, Inigo Goiri, Ricardo Bianchini, Daniel S Berger, and Rodrigo Fonseca. Palette load balancing: Locality hints for serverless functions. In *EuroSys*, pages 365–380, 2023.

[35] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *USENIX OSDI*, pages 599–616, 2020.

[36] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards {High-Performance} serverless computing. In *USENIX ATC*, pages 923–935, 2018.

[37] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *EuroSys*, pages 398–415, 2023.

[38] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual consensus in delos. In *USENIX OSDI*, pages 617–632, 2020.

[39] Mahesh Balakrishnan, Chen Shen, Ahmed Jafri, Suyog Mapara, David Geraghty, Jason Flinn, Vidhya Venkat, Ivailo Nedelchev, Santosh Ghosh, Mihir Dharamshi, et al. Log-structured protocols in delos. In *ACM SOSP*, pages 538–552, 2021.

[40] Muhammad Bilal, Marco Canini, Rodrigo Fonseca, and Rodrigo Rodrigues. With great freedom comes great opportunity: Rethinking resource allocation for serverless functions. In *EuroSys*, pages 381–397, 2023.

[41] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for Cloud-Scale computing. In *USENIX OSDI*, pages 285–300, 2014.

[42] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Communications of the ACM*, 59(5):50–57, 2016.

[43] Xiaohu Chai, Tianyu Zhou, Keyang Hu, Jianfeng Tan, Tiwei Bie, Anqi Shen, Dawei Shen, Qi Xing, Shun Song, Tongkai Yang, Le Gao, Feng Yu, Zhengyu He, Dong Du, Yubin Xia, Kang Chen, and Yu Chen. Fork in the Road: Reflections and Optimizations for Cold Start Latency in Production Serverless Systems. In *USENIX OSDI*, pages 199–218, 2025.

[44] Qiong Chen, Jianmin Qian, Yulin Che, Ziqi Lin, Jianfeng Wang, Jie Zhou, Licheng Song, Yi Liang, Jie Wu, Wei Zheng, et al. Yuanrong: A production general-purpose serverless system for distributed applications in the cloud. In *ACM SIGCOMM*, pages 843–859, 2024.

[45] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *ACM SOSP*, pages 105–120, 2015.

[46] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. Dirigent: Lightweight serverless orchestration. In *ACM SOSP*, pages 369–384, New York, NY, USA, 2024. Association for Computing Machinery.

[47] Lazar Cvetković, Rodrigo Fonseca, and Ana Klimovic. Understanding the neglected cost of serverless cluster management. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless*, pages 22–28, 2023.

[48] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: reconciling scheduling speed and quality in large shared clusters. In *ACM Symposium on Cloud Computing*, pages 97–110, 2015.

[49] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *ACM ASPLOS*, pages 467–481, 2020.

[50] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In *EuroSys*, pages 178–193, 2021.

[51] Alexander Fuerst, Abdul Rehman, and Prateek Sharma. Ilúvatar: A fast control plane for serverless computing. In *IEEE HPDC*, pages 267–280, 2023.

[52] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Exploiting nil-externality for fast replicated storage. In *ACM SOSP*, pages 440–456, 2021.

[53] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. Firmament: Fast, centralized cluster scheduling at scale. In *USENIX OSDI*, 2016.

[54] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for Fine-Grained resource sharing in the data center. In *USENIX NSDI*, 2011.

[55] Jialiang Huang, MingXing Zhang, Teng Ma, Zheng Liu, Sixing Lin, Kang Chen, Jinlei Jiang, Xia Liao, Yingdi Shan, Ning Zhang, et al. Trenv: Transparently share serverless execution environments across different functions and nodes. In *ACM SOSP*, pages 421–437, 2024.

[56] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *USENIX OSDI*, pages 1–16, 2018.

[57] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *ACM SOSP*, pages 261–276, 2009.

[58] Deepak Sirone Jegan, Liang Wang, Siddhant Bhagat, and Michael Swift. Guarding serverless applications with kalium. In *USENIX Security*, pages 4087–4104, 2023.

[59] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *ACM SOSP*, pages 691–707, 2021.

[60] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *ACM SOSP*, pages 152–166, 2021.

[61] Konstantinos Kallas, Haoran Zhang, Rajeev Alur, Sebastian Angel, and Vincent Liu. Executing microservice applications on serverless, correctly. *ACM POPL*, 7(POPL):367–395, 2023.

[62] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. {SelfTune}: Tuning cluster managers. In *USENIX NSDI*, pages 1097–1114, 2023.

[63] Sumer Kohli, Shreyas Kharbanda, Rodrigo Bruno, Joao Carreira, and Pedro Fonseca. Pronghorn: Effective checkpoint orchestration for serverless hot-starts. In *EuroSys*, pages 298–316, 2024.

[64] Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. Function as a function. In *ACM Symposium on Cloud Computing*, pages 81–92, 2023.

[65] Nanqinqin Li, Anja Kalaba, Michael J Freedman, Wyatt Lloyd, and Amit Levy. Speculative recovery: Cheap, highly available fault tolerance with disaggregated storage. In *USENIX ATC*, pages 271–286, 2022.

[66] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The gap between serverless research and real-world systems. In *ACM Symposium on Cloud Computing*, pages 475–485, 2023.

[67] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. Harmonizing efficiency and practicability: Optimizing resource utilization in serverless computing with jiagu. In *USENIX ATC*, pages 1–17, 2024.

[68] Joshua Lockerman, Jose M Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, Siddhartha Sen, and Mahesh Balakrishnan. The {FuzzyLog}: A partially ordered shared log. In *USENIX OSDI*, pages 357–372, 2018.

[69] Chang Lou, Yuzhuo Jing, and Peng Huang. Demystifying and checking silent semantic violations in large distributed systems. In *USENIX OSDI*, pages 91–107, 2022.

[70] Chang Lou, Dimas Shidqi Parikesit, Yujin Huang, Zhewen Yang, Senapati Diwangkara, Yuzhuo Jing, Achmad Imam Kistijantoro, Ding Yuan, Suman Nath, and Peng Huang. Deriving semantic checkers from tests to detect silent failures in production distributed systems. In *USENIX OSDI*, 2025.

[71] Fangming Lu, Xingda Wei, Zhuobin Huang, Rong Chen, Minyu Wu, and Haibo Chen. Serialization/deserialization-free state transfer in serverless workflows. In *EuroSys*, pages 132–147, 2024.

[72] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. One-size-fits-none: Understanding and enhancing slow-fault tolerance in modern distributed systems. In *USENIX NSDI*, 2025.

[73] Xuhao Luo, Shreesha G Bhat, Jiyu Hu, Ramnatthan Alagappan, and Aishwarya Ganesan. Lazylog: A new shared log abstraction for low-latency applications. In *ACM SOSP*, pages 296–312, 2024.

[74] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. Specinfer: Accelerating large language model serving with tree-based speculative inference and verification. In *ACM ASPLOS*, pages 932–949, 2024.

[75] Viyom Mittal, Shixiong Qi, Ratnadeep Bhattacharya, Xiaosu Lyu, Junfeng Li, Sameer G. Kulkarni, Dan Li, Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. Mu: An efficient, fair and responsive serverless framework for resource-constrained edge clouds. In *ACM Symposium on Cloud Computing*, pages 168–181, 2021.

[76] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *USENIX ATC*, pages 57–70, 2018.

[77] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In *ACM SOSP*, pages 472–487, 2021.

[78] Jia Pan, Haoze Wu, Tanakorn Leesatapornwongsa, Suman Nath, and Peng Huang. Efficient reproduction of fault-induced failures in distributed systems with feedback-driven fault injection. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 46–62, 2024.

[79] Sheng Qi, Xuanzhe Liu, and Xin Jin. Halfmoon: Log-optimal fault-tolerant stateful serverless computing. In *ACM SOSP*, pages 314–330, 2023.

[80] Sheng Qi, Zhiquan Zhang, Xuanzhe Liu, and Xin Jin. Kubedirect: Unleashing the full power of the cluster manager for serverless computing, 2026.

[81] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *USENIX OSDI*, pages 91–104, 2004.

[82] Alireza Sahraei, Soteris Demetriou, Amirali Sobhgol, Haoran Zhang, Abhigna Nagaraja, Neeraj Pathak, Girish Joshi, Carla Souza, Bo Huang, Wyatt Cook, et al. Xfaas: Hyperscale and low cost serverless functions at meta. In *ACM SOSP*, pages 231–246, 2023.

[83] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *EuroSys*, pages 351–364, 2013.

[84] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *USENIX ATC*, pages 205–218, 2020.

[85] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. Atoll: A scalable low-latency serverless platform. In *ACM Symposium on Cloud Computing*, pages 138–152, 2021.

[86] Jovan Stojkovic, Nikoleta Iliakopoulou, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Ecofaas: Rethinking the design of serverless environments for energy efficiency. In *ACM/IEEE ISCA*, pages 471–486. IEEE, 2024.

[87] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Mxfaas: Resource sharing in serverless environments for parallelism and efficiency. In *Proceedings of the 50th annual international symposium on computer architecture*, pages 1–15, 2023.

[88] Jovan Stojkovic, Tianyin Xu, Hubertus Franke, and Josep Torrellas. Specfaas: Accelerating serverless applications with speculative function execution. In *IEEE HPCA*, pages 814–827. IEEE, 2023.

[89] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *USENIX OSDI*, pages 143–159, 2022.

[90] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, et al. Anvil: Verifying liveness of cluster management controllers. In *USENIX OSDI*, pages 649–666, 2024.

[91] Ariel Szekely, Adam Belay, Robert Morris, and M Frans Kaashoek. Unifying serverless and microservice workloads with sigmaos. In *ACM SOSP*, pages 385–402, 2024.

[92] Yifan Tan, Cheng Tan, Zeyu Mi, and Haibo Chen. Pipellm: Fast and confidential large language model services with speculative pipelined encryption. In *ACM ASPLOS*, pages 843–857, 2025.

[93] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, et al. Twine: A unified cluster management system for shared infrastructure. In *USENIX OSDI*, pages 787–803, 2020.

[94] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *EuroSys*, pages 1–17, 2015.

[95] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *USENIX ATC*, pages 133–146, 2018.

[96] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. No provisioned concurrency: Fast {RDMA-codesigned} remote fork for serverless computing. In *USENIX OSDI*, pages 497–517, 2023.

[97] Haoze Wu, Jia Pan, and Peng Huang. Efficient exposure of partial failure bugs in distributed systems with inferred abstract states. In *USENIX NSDI*, pages 1267–1283, 2024.

[98] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *ACM ASPLOS*, pages 335–350, 2024.

[99] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *USENIX OSDI*, pages 1187–1204, 2020.

[100] Haoran Zhang, Shuai Mu, Sebastian Angel, and Vincent Liu. Causalmesh: A causal cache for stateful serverless computing. *Proceedings of the VLDB Endowment*, 17(13):4599–4613, 2024.

[101] Jie Zhang, Chen Jin, YuQi Huang, Li Yi, Yu Ding, and Fei Guo. Kole: Breaking the scalability barrier for managing far edge nodes in cloud. In *ACM Symposium on Cloud Computing*, pages 196–209, 2022.

[102] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and cheaper serverless computing on harvested resources. In *ACM SOSP*, pages 724–739, 2021.

[103] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *ACM SOSP*, pages 131–146, 2019.