PREEMPT-FaaS: Taming Orchestration Times in Latency-Sensitive Serverless Environments

Marcello Cinque
Università degli Studi di Napoli Federico II, Italy

Luigi De Simone
Università degli Studi di Napoli Federico II, Italy

Raffaele Della Corte
Università degli Studi di Napoli Federico II, Italy

Stefano Toscano
Università degli Studi di Napoli Federico II, Italy

Abstract

The orchestration of application instances is critical for the efficient management of cloud computing platforms. Specifically, the serverless paradigm automates container spawning and de-spawning based on actual load, mitigating inefficiencies, such as over- and under-provisioning, that might compromise Service Level Objectives (SLOs). This dynamic behavior introduces significant challenges concerning initialization and termination latencies, which are exacerbated when enforcing real-time requirements in mixed-criticality systems. The existing literature already addresses key issues, such as reducing cold-start times and assuring real-time performance to deployed instances. However, container orchestration times remain an overlooked factor that can severely affect instance startup times, especially when the orchestrator is subject to intense workloads. In this paper, we present PREEMPT-FaaS, an orchestration controller that, unlike commonly adopted controllers, adopts a fixed-priority preemptive scheduling of requests to guarantee reduced orchestration times to high-priority and highly critical instances. We implemented PREEMPT-FaaS as a Rust custom controller for Kubernetes (K8s), along with a patch for Knative, a popular serverless platform built upon K8s. We perform an extensive experimental campaign of PREEMPT-FaaS, including the serving of AI workloads, such as, recurring neural networks and video analytics, showing up to ∼6× reduction of orchestration times under high load and improving end-to-end cold-start times of critical instances, with a consequent reduction of service-level latencies (up to ∼2 s reduction under stress at the 95th percentile).

2012 ACM Subject Classification Computer systems organization → Real-time system architecture; Computer systems organization → Cloud computing; Software and its engineering → Real-time systems software; Software and its engineering → Software as a service orchestration systems

Keywords and phrases Edge-Cloud, Orchestration, Containers, Mixed-Criticality, Kubernetes

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2026.22

Supplementary Material
Software (Source Code): `https://github.com/dessertlab/preempt-k8s` [12]
archived at `swh:1:dir:aa067aa2469f1a7a0ef5dad20b909e8c1453a6eb`
Software (Source Code): `https://github.com/dessertlab/knative-crd-scaled` [11]
archived at `swh:1:dir:248b9febbae1cd09735cc4c896f544e72cbed87b`
Software (ECRTS 2026 Artifact Evaluation approved artifact):
`https://doi.org/10.4230/DARTS.12.2.1`

Funding This work was partially supported by the SERENA-IIoT project, which has been funded by EU – NGEU, Mission 4 Component 1, CUP J53D23007090006, under the PRIN 2022 (MUR – *Ministero dell'Università e della Ricerca*) program (project code 2022CN4EBH).

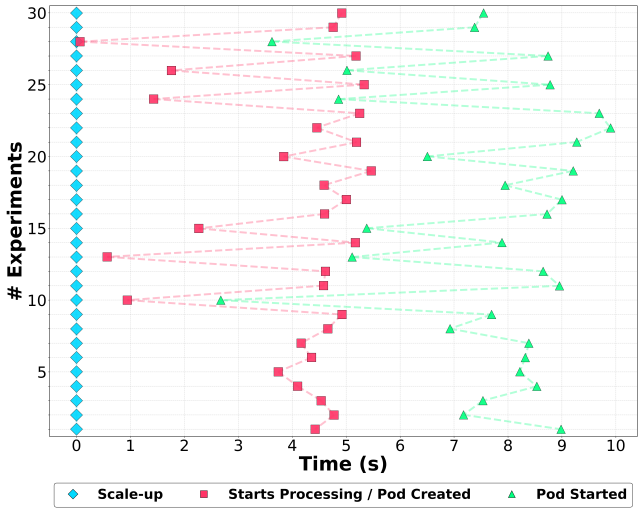


Figure 1 The figure shows the distribution of orchestration events following a scale-up of a Knative service under vanilla K8s. Results are aggregated over 30 experiments, managing 5 identical services, each scaled from 0 to 1 instance. The distributions are obtained under continuous interfering bursts of 45 orchestration requests. A scale-up event triggers: (i) the first controller processing step corresponding to the instance creation, and (ii) the actual instance startup on the worker node.

1 Introduction

In recent years, the orchestration of application instances has become a key issue in cloud computing, also due to the increasing focus on addressing the current demand for AI/ML workloads [43, 39, 19]. Establishing the number of active instances required to serve the expected workload is crucial, since application components exhibit heterogeneous Service Level Objectives (SLOs): some prioritize throughput, while others impose stricter constraints on availability, tail latency, or response time bounds. This is particularly evident in latency-sensitive domains like machine learning inference [41], 5G virtual network functions [29, 27, 36], and robotics [4, 33], in which services are characterized by different criticalities and should be orchestrated accordingly to meet their SLOs and deadlines [17, 47]. Furthermore, estimating the expected load at any given moment is cumbersome, and deviations from the predicted behavior are frequent. Thus, dynamic provisioning, driven by the workload, is a solution for modern cloud environments, leading to the adoption of the *serverless computing* and *Function as a Service (FaaS)* paradigms [53, 28]. Serverless/FaaS platforms enhance resource efficiency through fine-grained, demand-driven scaling, including scale-to-zero during idle periods to save costs. More recently, they have also been explored for latency-sensitive and near real-time applications in edge-cloud scenarios [50, 56].

Serverless and FaaS platforms are commonly built on container technology due to its portability, lightweight footprint, and fast startup characteristics [60, 41, 5, 28]. Container orchestrators, such as Kubernetes [26], handle container lifecycle management, scheduling, monitoring, and resource allocation across distributed nodes. The latency introduced by these orchestration operations (i.e., the time elapsed between an event triggering scaling and the corresponding control-plane action) directly impacts application SLOs and determines how effectively priority-based services are served under contention. This is particularly true in serverless/FaaS environments where improving *time-to-scale* becomes critical [30, 21, 42]. However, the reduction of *cold-start* times, i.e., the time required to activate one or more instances when there are no others available [53], is a significant challenge [50, 56].

Recent studies have focused on reducing sandbox creation time (e.g., container startup latency) to mitigate cold-start overheads [9, 59]. However, these approaches leave the control-plane orchestration path largely unmodified. As a result, state-of-the-art studies overlook the variability introduced by orchestration delays, referred to as a "declarative tax" in [30]. Prior work [30, 6] has shown that such orchestration latency can become a significant performance bottleneck, in some cases comparable to or even exceeding instance startup times (typically on the order of tens to hundreds of milliseconds). This phenomenon is exacerbated under high orchestration loads, where it has a negative impact on the prioritization of critical services over non-critical ones [8, 6]. Figure 1 highlights this aspect precisely, showing the distribution of orchestration events and timing following a scale-up of a Knative service. We used Knative [22] as a popular Kubernetes-based platform for serverless environments. The results show that scale-up actions heavily affect a serverless workload startup due to the high load on the Kubernetes control plane, making it impossible to make timing assumptions on high-priority services.

Contributions. In this paper, we present **PREEMPT-FaaS**, a custom controller of container instances designed to reduce orchestration times of latency-sensitive applications deployed on serverless platforms. We based PREEMPT-FaaS on the Kubernetes container orchestrator [26] (K8s hereinafter), due to its widely adoption: industry surveys report that approximately 93% of organizations use K8s in production, pilot deployments, or active evaluation. Moreover, several orchestrators share the Kubernetes codebase (e.g., OpenShift, K0s, K3s, microK8s) [23] and multiple serverless/FaaS platforms rely on it. Among them, Knative [22] has emerged as a widely adopted open-source serverless platform for Kubernetes [52]. PREEMPT-FaaS is implemented in Rust to fully exploit fixed-priority preemptive scheduling for controller threads. Unlike existing K8s components, which are primarily written in Go and rely on a user-level threading (ULT) model with limited control over OS scheduling semantics, Rust enables direct management of kernel-level threads (KLTs) and their scheduling parameters. This design allows PREEMPT-FaaS to leverage Linux real-time scheduling policies, specifically `SCHED_FIFO`, ensuring deterministic prioritization and reduced scheduling interference. By operating at the kernel scheduling level, PREEMPT-FaaS overcomes the constraints imposed by Go's runtime scheduler and provides tighter control over control-plane execution latency.

We validated PREEMPT-FaaS via a comprehensive experimental evaluation, targeting the assessment of orchestration time latencies as well as of end-to-end throughput and latency, under interfering load. The evaluation has been conducted using a Knative benchmark and AI workloads from the vSwarm benchmark suite (i.e., Recurrent Neural Network and Video Analytics services). We summarized the findings in the following:

- **Orchestration Latency Reduction.** PREEMPT-FaaS reduces control-plane scale-up latency by up to $\sim 6\times$ under high orchestration contention, maintaining bounded and stable processing times even with intense interfering workloads.
- **Impact on Cold-Start Performance.** Faster orchestration directly improves end-to-end behavior during cold starts, yielding lower service-level latencies (up to $\sim 2\,\mathrm{s}$ reduction under stress at 95th percentile) and still throughput improvement compared to Vanilla K8s.

Paper Structure. The remainder of the paper is developed in the following sections. Section 2 provides background on the two platforms we target in this work, i.e., K8s and Knative. Section 3 discusses the PREEMPT-FaaS design and our patched version of Knative. Section 4 presents the experimental evaluation and results obtained to validate our proposal. Section 5 provides related work. Section 6 concludes the paper with insights and future developments.

2 Background

We implemented PREEMPT-FaaS on top of Kubernetes [26], the *de facto* standard among container orchestrators, and Knative [22], one of the most adopted K8s-based serverless platforms. We provide an overview of both technologies in the following.

2.1 Kubernetes

K8s orchestrates containers within a *cluster*, which is a collection of nodes, i.e., a set of computational resources and storage units. Each node is either part of the *control plane* or a *worker node*. The control plane, which can be deployed on a single or multiple nodes, encompasses all Kubernetes internals necessary to properly manage the cluster. Worker nodes host K8s-managed workloads and offer computing and storage resources available to run scheduled containers. An overview of the K8s architecture is given in Fig. 2.

The smallest schedulable unit in K8s is called *Pod*, which is a set of one or more containers. A pod, along with all other main K8s resources, is considered an *Object*. Objects are stored in the K8s database, *etcd*, a key-value store. The vast majority of these resources are deployed in a K8s ***Namespace***, which is an abstraction that isolates workloads and resource instances in the cluster. A K8s object is characterized by its desired and current states; K8s detects a change in the desired states and manages the cluster to match the current and desired states of each object. A *manifest* is a file used to represent an object in the cluster (often encoded in "*.yaml*"), which includes three sections: *metadata*, specification (*spec*, i.e., the desired state), and *status* (i.e., the current state). Users or software interacting with K8s can specify the desired state by editing the spec section and can only inspect its status. K8s main components are:

- The `kube-apiserver` is the central control-plane component. All internal K8s components communicate through it, and every request to or from the cluster is processed via the API server.
- The `kube-controller-manager` runs a collection of controllers, each responsible for a specific K8s resource. Controllers continuously reconcile the current state with the desired state through control loops. While core controllers are compiled into a single binary, K8s supports extensibility via custom controllers, which can be implemented in any programming language.
- The `scheduler` assigns pods to worker nodes based on resource availability and policy constraints. K8s allows deploying custom schedulers or extending the default scheduler through plugins.
- The `kube-proxy`, which configures nodes' virtual networks. Installed on every node of the cluster, the proxy monitors the changes to Service objects and their endpoints, translating them into network rules inside the node.
- The `kubelet` is a node-level agent running on each node. It receives pod specifications from the control plane, instantiates the corresponding containers, and continuously reports their status to maintain an up-to-date cluster state.

K8s offers a high degree of customization, making it a flexible software solution for different use cases. In addition to plugins, custom controllers, and schedulers, K8s also offers the possibility of introducing new resources into the system. A ***Custom Resource Definition*** (**CRD**) is used to specify the structure of a new resource type; each instance of this type is called ***Custom Resource*** (**CR**). The CR is usually associated with a custom controller to introduce new logic in a K8s cluster [34].

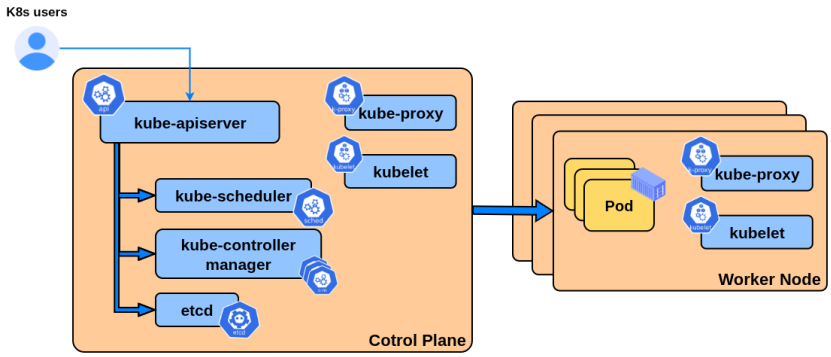


Figure 2 K8s architecture.

K8s leverages two specific resources to allow users to deploy an application with multiple instances. A ***Deployment*** contains the application's pod specification along its required number of instances. A Deployment is inherently associated with a ***ReplicaSet*** object that keeps track of current and desired number of replicas. The respective controllers ensure that the desired number of pods is always met when sufficient hardware resources are available. This also includes failover procedures when a pod or a node hosting it crashes. A labeling structure in the metadata section of the pods allows the controllers to match Deployments to their respective pods. A *Service* offers a common endpoint to reach the application's pods; it can also be seen as a load balancer for a specific application, distributing service requests across all available instances. The same labeling structure is used to redirect traffic to the application's pods [26].

2.2 Knative

Knative [20, 22] is one of the most popular serverless platforms [52] specifically designed to operate on top of Kubernetes. We focus on ***Knative Serving***, an HTTP-triggered autoscaling system that handles the lifecycle and scaling of stateless HTTP services. By using K8s resources, such as Services and Deployments, Knative enables users to deploy applications as a ***Knative Service*** (**KSVC**), which is constantly monitored by Knative components to determine, based on the actual request rate, the number of pods necessary to properly serve the workload. Knative Serving (hereon referred to as Knative for simplicity) is characterized by different resources. ***Knative Services*** are the first resource in the hierarchy for an application and control the lifecycle of the other resources. ***Revisions*** are immutable snapshots of the application's code and configuration, representing specific versions of it; multiple revisions can coexist, and traffic can be explicitly split among them. Each revision is assigned to a K8s Deployment. ***Configurations*** represent the desired state of the applications; each configuration update creates a new revision. ***Routes*** map endpoints to one or more revisions. Knative main components are:

- A **queue proxy**, a container deployed alongside the application in each KSVC pod; it collects and forwards requests and replies to/from the outside. It also collects application metrics and sends them to the Knative autoscaler to determine the necessary number of pods at any given time.
- The **activator** queues incoming requests when the application is scaled to zero. Upon receiving requests, it signals the autoscaler to scale up the application. Once endpoints are reachable, it forwards the queued requests to them. By default, requests are routed directly to active endpoints, but the activator can be configured to always remain in the request path.

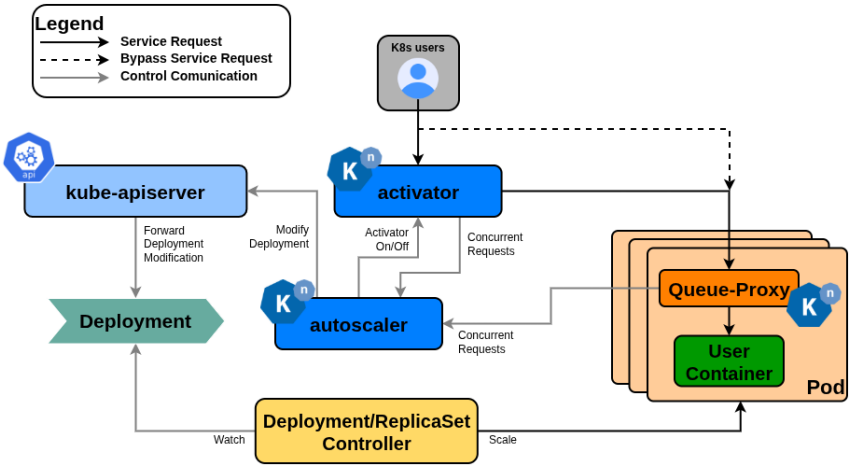


Figure 3 Knative architecture and service request flow.

- The **autoscaler** is responsible for scaling the application up and down, patching revisions' Deployments, based on incoming requests, internal metrics, and application configuration (e.g., the single-instance level of parallelism drives scaling decisions). The "*concurrency*" parameter in the KSVC specification can be set to communicate to the autoscaler the single-instance level of parallelism, a configuration that drives the scaling logic of the service. The autoscaler can be customized, and *scale-to-zero* can be either enabled or disabled. It can leverage the *Horizontal Pod Autoscaler* (HPA) without the scale-to-zero option, or use the default *Knative Pod Autoscaler* (KPA) with scale-to-zero enabled. Custom autoscalers can also be employed.

The service request flow and the main components involved are shown in Fig. 3. The typical Knative operational steps are as follows:

1. The application is deployed as a KSVC; all related resources are created, including a Deployment and a ReplicaSet per revision.
2. An initial application instance is created to verify that the service and its endpoints are healthy; the instance is dropped if no load is detected according to the configured scale-down time window.
3. The activator is enabled to buffer new service requests until the required pods are created.
4. When new requests arrive, a scale-up operation is triggered.
5. Once sufficient pods are ready to respond to requests, the activator forwards the queued requests to the active instances and is disabled.
6. The autoscaler scales the application as long as requests continue to be received.
7. When no requests are detected within the scale-down time window, the application scales down to 0 instances, the activator is enabled, initial conditions are restored, and Knative awaits new traffic.

In this paper, we address the operation between steps 1 and 2 to apply scaling decisions to custom resources monitored by the PREEMPT-FaaS controller in order to reduce orchestration times for critical services.

3 PREEMPT-FaaS

The kube-manager operates as a set of control loops, each dedicated to managing specific K8s objects (e.g., Deployment, etc., see § 2.1 for details). These controllers typically employ a "*single queue, multiple servers*" architecture, where a fixed number of servers, hereon referred to as workers, dequeue events for processing. As already shown in Figure 1, Vanilla K8s is unable to effectively manage critical services over non-critical ones under high load, and orchestration times latency can increase to the order of tens of seconds [51]. Furthermore, built-in K8s tools are inadequate for multi-priority management, and disabling mechanisms like throttling and rate-limiting in K8s components most likely induce system instability [6]. The Vanilla k8s inadequacy for latency-sensitive workloads can be imputed to several design choices that characterize the K8s implementation, specifically, the kube-manager's.

- A worker must complete the handling of an event before acquiring a new one.
- A higher priority event could be enqueued when the fixed set of workers is already busy processing previous, lower priority events.
- K8s shows a Go-based implementation, which relies on $M : N$ user-level threading model managed by the Go runtime scheduler.
- All kube-manager components may run in an environment with higher priority threads preempting them from scheduling, slowing down the orchestration process.

To address these limitations in serverless scenarios, we employ CRDs and the Operator pattern to replicate the Deployment/ReplicaSet control loop with priority handling capabilities. The proposed solution, namely PREEMPT-FaaS, is a custom K8s controller implemented in Rust. Rust allows direct control over operating-system threads and their scheduling attributes. This enables PREEMPT-FaaS to employ real-time Linux scheduling policies (e.g., `SCHED_FIFO`) and fixed-priority preemptive execution at the kernel level. Although recent versions of the Go runtime provide cooperative and limited asynchronous preemption [32], scheduling decisions remain governed by the language runtime, and there is no fine-grained control over thread priorities or direct exposure to kernel-level real-time policies. To the best of our knowledge, Rust is the optimal choice for such a controller, since it simultaneously provides kernel-level thread scheduling and a mature `kube-rs` library to interact with a K8s cluster, including features like: simple JSON parsing, full CR support allowing to manage resources that do not have an pre-established structure, asynchronous interactions with the kube-apiserver, and resource monitoring primitives.

PREEMPT-FaaS provides:

- **A priority-aware CRD (namely *RTResource*)** that embeds service specification and criticality level, leveraging native pod templating to preserve Kubernetes validation and decouple controller logic from application details.
- **A preemptive, priority-driven control-plane architecture** with dynamically scalable worker threads to reduce orchestration latency under bursty and high-load conditions.
- **Cluster-wide, low-intrusive management** of critical resources across namespaces, fully configurable through runtime parameters without recompilation.
- **Seamless Knative integration**, redirecting scaling decisions to RTResources and enabling runtime switching via a simple KSVC annotation, with no redeployment of Knative internals.

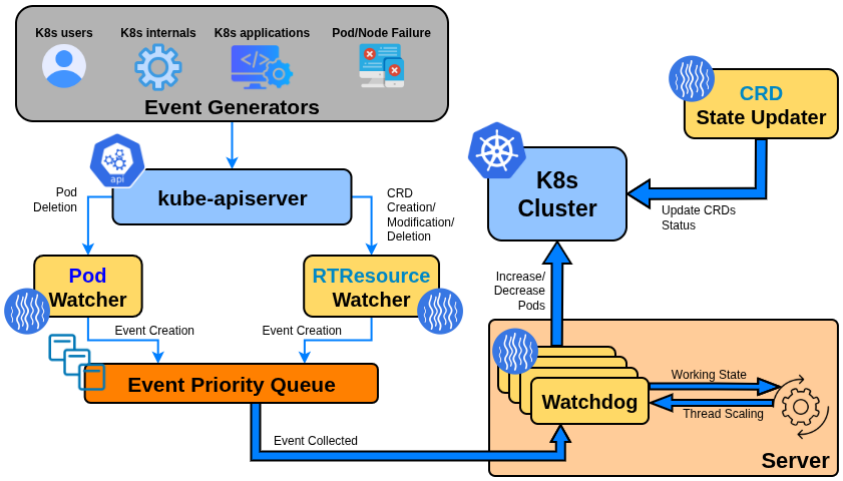


Figure 4 PREEMPT-FaaS Architecture.

3.1 PREEMPT-FaaS Architecture

The architecture of the PREEMPT-FaaS controller is depicted in Fig. 4. PREEMPT-FaaS preserves the "*single queue, multiple servers*" paradigm, but, unlike the Vanilla K8s controller, we implement a POSIX multi-priority queue to ensure immediate processing of the high-priority orchestration events. To isolate critical applications from standard workloads, we define a new Custom Resource Definition (CRD), *RTResource*, which serves as a Deployment-like abstraction enriched with a criticality level. The PREEMPT-FaaS controller exclusively monitors these resources and enforces priority-based orchestration through a preemptive scheduling model implemented at the kernel-thread level. The current implementation of PREEMPT-FaaS leverages `SCHED_FIFO` thread scheduling policy since our first choice is to manage applications with fixed priorities. We remark that the controller design can be modified to accommodate different scheduling policies. For instance, `SCHED_DEADLINE` can be employed to introduce a CPU budget control, or if developers want to serve events according to specific deadlines.

In this section, we first describe the RTResource abstraction and its structure, then detail the internal architecture of PREEMPT-FaaS, including its event processing model and scheduling configuration. Finally, we present the minimal modifications required to integrate PREEMPT-FaaS with Knative, enabling transparent switching between default and priority-aware orchestration.

3.1.1 RTResource CRD

The ***RTResource*** CRD is the equivalent of a standard K8s Deployment, specifically tailored for real-time applications. Employing this custom object decouples standard K8s workflows from critical resource management. Consequently, PREEMPT-FaaS controller exclusively monitors critical applications, preventing interference with the native Deployment/ReplicaSet controller behavior and vice versa. The specification (i.e., `spec`) section of RTResource (example in Listing 1) is in the following:

- **Namespace**. The namespace in which to deploy the application; it may differ from the resource namespace.
- **Replicas**. The desired number of application instances.

- **Selector**. A selector that can be leveraged to select these specific application instances.
- **Criticality**. The criticality level of the application (lower levels have higher priority).
- **Template**. The *template* field defines the pod specification for the instances to be created. By leveraging native pod templating, validation is delegated to the kube-apiserver, avoiding custom structural checks within the controller. This preserves standard pod semantics, decouples controller logic from application details, and improves portability.

Listing 1 RTResource spec example.

```yaml
apiVersion: rtgroup.critical.com/v1
kind: RTResource
metadata:
  name: rt-critical-application
  namespace: realtime
spec:
  namespace: realtime
  replicas: 4
  selector:
    matchLabels:
      app-selector: rt-critical-app
  criticality: 1
  template:
    metadata:
      name: rt-critical-pod
      namespace: realtime
      labels:
        app-selector: rt-critical-app
    spec:
      containers:
        - name: rt-critical-container
          image: nginx:latest
          ports:
            - containerPort: 80
              name: http
              protocol: TCP
          resources:
            requests:
              cpu: "500m"
              memory: "512Mi"
            limits:
              cpu: "1000m"
              memory: "1Gi"
```

Listing 2 RTResource status example.

```yaml
status:
  desiredReplicas: 4
  observedGeneration: 1
  replicas: 4
  conditions:
  - lastTransitionTime: "2026-02-10T01
    :12:15.205871718+00:00"
    message: All desired replicas are
      running!
    reason: All desired replicas are
      running!
    status: "False"
    type: Progressing
  - lastTransitionTime: "2026-02-10T01
    :12:15.205871718+00:00"
    message: All desired replicas are
      running!
    reason: All desired replicas are
      running!
    status: "True"
    type: Ready
```

The `status` subresource (example in Listing 2) is structured as follows:

- **Desired Replicas**. The desired number of application instances.
- **Observed Generation**. The last version of the spec that has been managed by PREEMPT-FaaS.
- **Replicas**. The active number of application instances observed.
- **Conditions**. A set of conditions that the resource can assume. Each condition is associated with a type, i.e., `Progressing` or `Ready`, a boolean value reporting the state of the condition, a timestamp of the last state transition, a message, and a reason for the last state transition. The semantic of the status field depends on the condition type:
  1. **Progressing**. Set to `True` when PREEMPT-FaaS processes an event that changes the desired replica count; reset to `False` once the desired number of replicas is reached.
  2. **Ready**. Set to `False` while the resource is converging to the desired state. It is marked `True` once all desired replicas are running and active in the cluster.

3.1.2 PREEMPT-FaaS Components

PREEMPT-FaaS operates with a *priority queue* and four distinct thread classes: *i*) *watcher threads*, which collect events interacting with the kube-apiserver; *ii*) a *server thread*, responsible for dynamic watchdog scaling; *iii*) *watchdog threads*, which dequeue and process

Table 1 PREEMPT-FaaS Scheduling Configuration.

Thread	Base Priority	Scaling	Min	Max
RTResource Watcher	96	static	1	1
Pod Watcher	96	static	1	1
Server	95	static	1	1
Watchdog	94	dynamic	10	20
State Updater	96	static	1	1

orchestration events; *iv*) a *state updater thread*, that synchronizes the CRs state based on active replicas count. Table 1 summarizes the scheduling priority configuration employed in our evaluation.

Watcher Threads. These threads retrieve orchestration events related to critical applications from the kube-apiserver. The *RTResource Watcher* captures RTResource creation, modification, and deletion events, publishing them to the event priority queue. Similarly, the *Pod Watcher* monitors pod deletion events; if a pod associated with a critical resource is deleted, a corresponding message is generated. Critical pods retain essential metadata to map them to their respective CRs. The *priority queue* message payloads contain the RTResource name, unique identifier (UID), and namespace, with the message priority corresponding to the CR criticality level. It should be noted that pod creation events are not monitored, as critical pods managed by the PREEMPT-FaaS controller are instantiated exclusively via RTResources. Furthermore, monitoring these events would cause the controller to react to its own internal control loop operations, leading to redundant reconciliation cycles and an inefficient use of system resources.

Server Thread. This component manages watchdog thread scaling to maintain a pre-warmed pool of threads equal to a configured threshold. By monitoring the working state of each watchdog, the server scales up the thread count when free threads fall below the threshold, up to a specified maximum. Conversely, the server must not be allowed to terminate watchdogs that are serving an event. Therefore, when a watchdog has processed an event, it shuts down if there is a higher watchdog count than the threshold. Nevertheless, the number of active threads never drops below the configured baseline.

Watchdog Threads. These threads retrieve events from the queue and handle them according to the respective application criticality level, implementing the PREEMPT-FaaS core control loop logic. Each iteration of the loop is divided into the following steps. A watchdog starts the iteration at its base priority level. If new events are available in the queue, the thread retrieves the one marked with the highest priority (i.e., lower application criticality value). Once the event is retrieved, the thread modifies its scheduling priority according to the following formula:

- *WBP: watchdog-base-priority*
- *RAP: reversed-application-priority*
- *WSP: watchdog-scheduling-priority*

$$WSP = WBP - RAP$$

Where $AP_i > AP_j \implies RAP_i < RAP_j \quad \forall i, j$*, i.e., the highest application priority (AP) is coded with* $RAP = 1$.

The motivation behind this design choice is to separate the event retrieval phase from its management. All watchdogs handling an event always execute at a lower priority since the highest criticality level allowed is 1, which is subtracted from the thread base priority according to the previous formula. When a new event is enqueued, free threads are able to preempt other watchdogs to retrieve it. This allows a high priority event to preempt lower priority events already being handled by the controller.
Following priority adjustment, the controller verifies the existence of the RTResource associated with the event. If the resource is absent, the event is treated as a deletion, and all associated pods are terminated. If the resource exists, its status is updated from `Ready` to `Progressing`, and the status desired replica value is aligned with the `spec` value. The watchdog then scales the active instances to match the desired state. Finally, the watchdog reverts to its baseline scheduling priority and awaits new events. Note that the watchdog retrieves the pod configuration directly from the RTResource template section. This choice completely decouples the controller from the specific application being managed, ensuring compatibility with K8s pod specification enhancements (e.g., GPU or FPGA resource specification) used in some studies [47].

State Updater. This component synchronizes the status of resources in the `Progressing` state. The updater retrieves these resources from the kube-apiserver and sorts them by priority. For each resource, it verifies whether the running instance count matches the desired count. If the condition is met, the resource state is promoted to `Ready`; otherwise, the evaluation is deferred to the next control loop. During evaluation, any change in the running replica count triggers an update to the respective status field. The State Updater and the two watchers share the same highest priority, to react as soon as possible to changes. Their mutual interference is minimal, considering their limited number (only 3 threads), the simplicity of their code, and the absence of critical sections herein.

Once the `PREEMPT-FaaS` controller is started, the `main` process creates the watchers, the state updater, and the server threads using the `pthread_create` function of the `libc` Rust library [45]. The Server thread then creates the watchdog threads' baseline and scales them up according to the configured threshold and number of events to process. Each thread is coupled with a priority level and the `SCHED_FIFO` policy (through `pthread_attr_setschedpolicy`). Rust allows us to use its *Foreign Function Interface (FFI)* to call C system libraries to manage threads at the kernel level. Rust offers memory-safe management (allows so-called `safe` blocks), preventing common issues such as null pointer de-references, buffer overflows, data races at compile time, and managing the ownership of each variable, allowing a single owner at a time. The `libc` library requires the Rust code that executes it to employ an `unsafe` block, since Rust cannot enforce its memory safety features for such functions, contrary to the other interactions with the K8s cluster, managed through the `kube-rs` library.

Watchdog threads dynamically adjust their scheduling priority according to the criticality of the event being processed. Although this mechanism strengthens priority enforcement, it may increase the risk of starvation for lower-criticality workloads under sustained high-priority traffic. This behavior is intentional, as PREEMPT-FaaS is designed to favor critical services under contention. To prevent unbounded blocking, priority adjustment is applied only during event processing and reverted immediately afterward. Furthermore, critical sections remain short and free of blocking I/O, limiting the duration of potential priority inversion. In practice, this design ensures strong differentiation while maintaining bounded interference across watchdog threads. As an orthogonal mitigation, PREEMPT-FaaS can further reduce interference by assigning CPU affinity to controller thread classes (e.g., dedicating cores to

watcher/state updater threads and a separate CPU set to watchdogs). This limits cross-class contention, reduces jitter due to thread migration, and improves orchestration responsiveness under load. Since affinity does not eliminate lock-induced priority inversion, we treat it as a complementary optimization (e.g., for highest priority threads) to short critical sections and non-blocking synchronization.

3.2 Knative Patch

Knative manages resources through direct interaction with the K8s, utilizing its Go codebase and K8s native libraries to manipulate both standard K8s objects and Knative resources. Originally designed to cooperate with Kubernetes Deployments, KSVC instantiation involves the following steps to enable dynamic scaling:

1. The user deploys a KSVC.
2. Knative creates a standard K8s Service to provide a unified domain name for the application.
3. A Knative Configuration and the corresponding Revision are generated.
4. Each Revision maps to a Deployment managed by the autoscaler, hosting the required replica count. This Deployment utilizes a pod template propagated from the KSVC.
5. The pod template incorporates container information, including the mandatory queue-proxy container, with Knative-generated configuration to automate connectivity between the proxy and the main service.

The Knative metadata annotation validation procedure is modified to recognize a criticality annotation used by PREEMPT-FaaS:

```
autoscaling.knative.dev/application-criticality-level: <integer-value>
```

This addition enables seamless switching between Knative default behavior and PREEMPT-FaaS one, employing RTResources instead of standard Deployments. Furthermore, the necessary Go structures are introduced to allow Knative to create, patch, delete, and access RTResources.

A Revision is considered healthy once active endpoints are established, hence the default initialization to a single instance. Crucial to this process is monitoring the health state of the underlying Deployment and its ReplicaSet, and propagating their status to the Revision. This behavior is replicated for the RTResource; the Revision is deemed healthy when the corresponding CR reaches the `Ready` state.

During RTResource creation, most specifications are derived from the Revision propagated from the KSVC, with the criticality field extracted from the new annotation. Knative independently defines the pod specification, labels, namespaces, and selectors. Since network routing relies on pod labels, employing pod templating allows delegating this setup to Knative, thereby ensuring that PREEMPT-FaaS remains decoupled from KSVC-specific configurations. Finally, the Knative autoscaler was modified to apply scaling decisions directly to the "*replicas*" field of the CR when the criticality annotation is present.

The Knative patch (including all the mentioned changes) remains compatible with the Knative code generation pipeline, automating standard procedure implementations (e.g., resource finalizers and deep-copy functions). Additionally, Kubernetes authorization options were updated in the Knative installation manifests to grant its components full access to the RTResources API group. The patch required 600 lines of code (LoC), counting both additions and modifications, but excluding Knative auto-generated code and manifest changes.

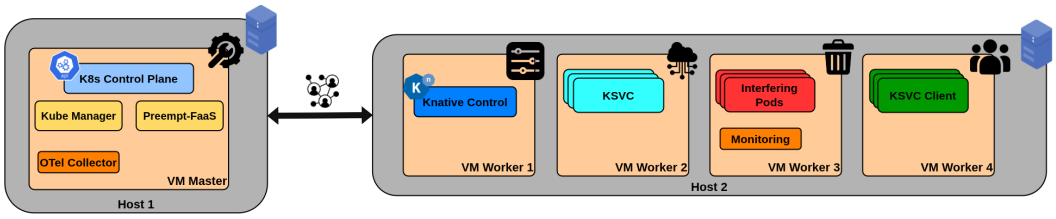


Figure 5 Testbed in the experimental setup.

4 Experimental Evaluation

This section evaluates PREEMPT-FaaS by directly addressing the following research questions:

- **RQ1:** *Can control-plane orchestration latency be reduced under high contention?*
- **RQ2:** *How does reduced orchestration latency affect throughput and end-to-end tail latency?*

We first describe the experimental setup and monitoring methodology (§ 4.1). We then evaluate orchestration time latencies under interfering load using a customized Knative benchmark (RQ1). Next, we measure the impact on end-to-end throughput and latency in both cold-start and pre-warmed scenarios using the vSwarm benchmark suite [55] (RQ2). All experiments are conducted under controlled orchestration interference.

4.1 Experimental Setup

Testbed. The testbed consists of a Kubernetes cluster deployed in VMs on two physical machines. The control plane is dispatched on a dedicated host. The host machine is equipped with 8 Intel Xeon CPU E5-2630L CPUs with 16 logical processors, and 32GB RAM. The cluster comprises one master node (4 vCPUs, 4GB RAM) and four worker nodes (8 vCPUs, 16GB RAM each), each deployed on a dedicated VM. All nodes run Ubuntu Linux 22.04.3 LTS with kernel v5.15, Kubernetes v1.29.15, containerd v1.7.11, and runc v1.1.11. The Knative patch has been developed on a fork of the Knative Serving repository[1]. The experimental evaluation is designed to isolate control-plane interference from application-level execution. As shown in Fig. 5, the Kubernetes control plane, including core components and both controllers (Vanilla K8s and PREEMPT-FaaS), is deployed on a dedicated `master node`. The Knative control components are hosted on `worker node 1` to avoid interference with either application workloads or stress generators. `Worker node 2` is dedicated to running the benchmarked serverless applications (KSVCs) (server-side). We generate orchestration interference leveraging resource creation and deletion requests. To ensure that control-plane contention does not directly compete with KSVC execution, we employ `worker node 3` to run interfering containers (non-critical services). `Worker node 4` is used as a workload generator for the KSVCs applications. Finally, the monitoring services are deployed in the `master node` and on `worker node 3`. We also remark that the Knative activator is set to be removed from the service request path to ensure that service latencies are not affected by a single bottleneck that buffers and forwards all requests.

[1] `https://github.com/knative/serving/commit/9efd9474885028dadb0d8addd1aeaf395d4e80cb`

Monitoring. To collect control-plane data, we enabled the `kube-apiserver` auditing feature, which records all interactions with the API server initiated by users, applications, and internal Kubernetes components. This mechanism provides microsecond-resolution timestamps for both requests and responses, allowing precise reconstruction of orchestration timelines. Unlike standard Kubernetes *events*, audit logs include full request and response payloads, enabling fine-grained tracking of control-plane operations. This approach differs from tracing-based monitoring, which typically captures only high-level request flows to the `kube-apiserver` and `etcd`, without exposing complete object-level state transitions [24, 25, 61]. To collect and process audit logs, we deployed an *OpenTelemetry (OTel)* collector within the cluster [38]. OTel provides a flexible telemetry pipeline for collecting, processing, and exporting logs, metrics, and traces, and integrates natively with Kubernetes. We configured a single-replica collector in the control plane to avoid data duplication while ensuring cluster-wide visibility. The collector ingests `kube-apiserver` audit files through an OTel receiver and exports the processed logs to *Loki*, where they can be queried via *Grafana* dashboards or custom analysis scripts [18]. For the experimental evaluation, we collect audit logs to extract precise timestamps associated with the following key control-plane events:

- **Scale-Up**. The event at which the Knative autoscaler updates the replicas field of the Deployment or RTResource. A single iteration may include multiple scale-up events per service.
- **Starts Processing**. The event at which the controller begins handling a scale-up event, identified by its first interaction with the `kube-apiserver`. For the Vanilla K8s controller, this corresponds to the creation of the first pod in the new replica set. For PREEMPT-FaaS, it coincides with updating the RTResource status condition to `Progressing`.
- **Pods Created**. The event at which all required pod objects for a scale-up are successfully created in the cluster. A pod is considered "created" once its API object exists, independently of container startup. For single-replica scale-ups under Vanilla K8s, this event coincide with *Starts Processing*.
- **Pods Started**. The event at which all required pods are running and ready to receive traffic. Although this stage depends on kubelet behavior (which is outside the scope of this work), faster orchestration can advance pod scheduling and startup. Nevertheless, kubelet queueing and internal mechanisms may still introduce variability [8, 6].

4.2 RQ1: *Can Control-plane Orchestration Latency be Reduced Under High Contention?*

To answer RQ1, we customized the *real-traffic-test* benchmark included in the Knative performance tests suite [14]. We compare Vanilla K8s and PREEMPT-FaaS to handle 10 concurrent KSVCs. We deploy these services, each configured with a maximum parallelism of 5 *requests per instance*. This parameter is set according to a preliminary sensitivity analysis against our testbed to be sure to trigger scaling operations. To evaluate the prioritization effectiveness of PREEMPT-FaaS over non-critical service (stressload), each service is assigned a distinct priority level (1–10) via the corresponding annotation. The stressload is run with repeated bursts of creating/deleting a set of Deployments (with no priority set) or RTResources (each set to lower priorities than KSVCs)-depending on the controller under test, Vanilla K8s and PREEMPT-FaaS, respectively-each associated with a single pod. To prevent excessive load on the `kube-apiserver`, interfering bursts are released at 10-second intervals, as most scale-up events occur during the initial transition from a scaled-to-zero state.

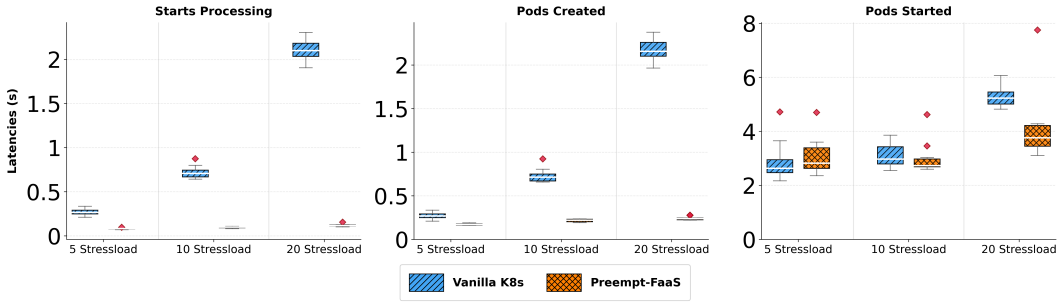


Figure 6 Key orchestration times latencies for Vanilla K8s and PREEMPT-FaaS controllers after a scale-up operation.

The benchmark execution is repeated 10 times for each controller under test for statistical significance purposes. We vary the stressload intensity at 5, 10, and 20 requests of creation/deletion per burst. Each iteration lasts 60s, during which the workload generator issues 50 RPS (requests per second) towards the 10 deployed KSVCs, with 5 RPS for each service (in a round-robin fashion) to ensure triggering scaling. We ensure that KSVCs are scaled to zero before starting the stress load.

Figure 6 reports the latency distributions for the key orchestration events, measured from the scale-up trigger to each corresponding control-plane stage. The "Starts Processing" distributions show that PREEMPT-FaaS achieves more stable and bounded latencies under all interference levels with a median reduction percentage of 94.4% in the worst case. In contrast, the Vanilla K8s controller exhibits increasing variability as contention grows. Across all scenarios, PREEMPT-FaaS consistently maintains mean latencies below 0.2 seconds. The "Pods Created" results follow a similar trend. While Vanilla K8s performs comparably under minimal interference, its latency and variability increase significantly as the orchestration load rises, with a median reduction percentage of 88.9% in the worst case. Since pod creation requires repeated interactions with the `kube-apiserver`, sustained control-plane stress amplifies these delays. The slightly higher latency observed for PREEMPT-FaaS in some high-load cases can be attributed to `kube-apiserver` saturation, acting as a shared bottleneck, consistent with prior observations [8]. Finally, the "Pods Started" panel indicates that faster orchestration translates into earlier pod readiness when the kubelet is not overloaded. Although pod startup also depends on node-level mechanisms beyond our control, improved control-plane responsiveness accelerates the overall scale-up pipeline.

Furthermore, we repeated the analysis in the worst case (20 creation/deletion as stressload) for the PREEMPT-FaaS, limiting to 5 watchdog threads min/max to match the Vanilla K8s configuration. Table 2 shows that PREEMPT-FaaS outperforms Vanilla K8s in both mean

Table 2 Orchestration latency metrics (seconds) under 20 interfering events per burst, with 5 fixed workers/watchdogs.

Metric	Vanilla K8s		PREEMPT-FaaS		Speedup
	Mean (s)	Var (s^2)	Mean (s)	Var (s^2)	(Mean)
Starts Processing	2.105	0.013	0.321	0.004	6.55×
Pods Created	2.172	0.014	0.420	0.003	5.18×
Pods Started	5.280	0.136	3.679	0.853	1.43×

orchestration latency and variance, even with thread scaling disabled. We do not increase the number of control loop threads in K8s (default is 5) since they work in a "single queue, multiple server scheme", i.e., they pick requests from the same queue (one for each resource type). In [6], the authors demonstrated that increasing the parallelism for K8s control loop threads can lead to high instability.

RQ1: Control-plane Orchestration under Contention

Under high orchestration interference, PREEMPT-FaaS substantially reduces the time required for the control plane to react to scaling requests and instantiate new replicas. In the worst-case configuration (20 interfering events), it achieves up to **6.55**× lower control-plane reaction latency and **5.18**× faster replica creation compared to Vanilla K8s. These improvements persist even with fixed worker threads, indicating that the gains primarily derive from priority-aware preemptive scheduling rather than increased parallelism. Residual delays are mainly attributable to `kube-apiserver` saturation under peak load.

To quantify the overhead introduced by the monitoring infrastructure, we measured service-level mean latencies with auditing enabled and disabled and 20 interfering resources for both PREEMPT-FaaS and Vanilla K8s controllers. For the Vanilla K8s controller, latency measurements followed a normal distribution with homogeneous variance; therefore, we applied a Student's t-test ($\alpha = 0.05$), which revealed no statistically significant difference attributable to the monitoring pipeline ($p = 0.8874$). In contrast, latency samples for PREEMPT-FaaS did not satisfy normality assumptions. We thus employed a Mann–Whitney U test, which likewise showed no statistically significant degradation due to monitoring ($p = 0.6232$).

4.3 RQ2: *How does reduced orchestration latency affect throughput and end-to-end tail latency?*

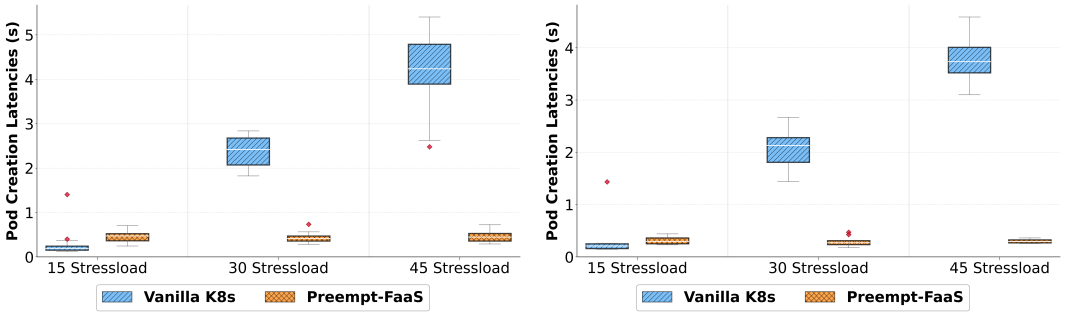
In this section, we evaluate whether reducing orchestration latency translates into measurable improvements in end-to-end service behavior. To this end, we employ the widely adopted *vSwarm* serverless benchmark suite [55]. Unlike the Knative *real-traffic-test* benchmark, primarily designed to stress the control plane and measure orchestration dynamics, vSwarm provides realistic application-level workloads, ranging from simple functions to multi-component services, across multiple programming languages and runtimes. This diversity enables us to assess the impact of orchestration optimizations on throughput and latency under representative serverless scenarios. Moreover, vSwarm has been used in several recent studies [40, 46, 48], making it a suitable and reproducible baseline for evaluating application-level performance improvements. Traffic is generated via the default vSwarm *invoker* script, which provides request latencies and throughput when tests complete.

We chose *RNN* and *Video Analytics* benchmark services from the suite. RNN benchmark generates a meaningful text given a language, employing a PyTorch character-level vanilla Recurrent Neural Network (RNN) model. Instead, the video analytics benchmark preprocesses an input video and runs an object detection model (i.e., squeezenet, a lightweight 18-layer convolutional neural network designed for high efficiency in image classification) on a video stored on a MongoDB database. We deploy 5 identical KSVCs on the designated worker node (see § 4.1), assigning each service the highest priority level under PREEMPT-FaaS. The services are executed alongside an orchestration interference workload consisting of continuous bursts of 15, 30, and 45 resource creation/deletion events. Unlike the previous experiments, which employed a 10-seconds interval between bursts, the continuous interfering

loads allows us to further stress the platform, even at lower interfering loads. The stress load is intentionally increased to drive the system into a regime where E2E metrics are noticeably affected by orchestration delays. A burst size of 45 represents the maximum sustainable load for our cluster; beyond this point, we observe `kube-apiserver` instability that must be avoided, ensuring that observed performance differences reflect controller behavior rather than control-plane collapse. As such, these parameters should be calibrated per deployment environment to establish a stable interference baseline. We evaluate both the Vanilla K8s and PREEMPT-FaaS controllers under identical conditions, each subjected to their respective interfering resources (Deployments for Vanilla K8s, RTResources for PREEMPT-FaaS), with one pod per resource. The decision to deploy 5 identical services, rather than a single instance, mitigates stochastic variations and ensures statistical significance. Since the `kube-apiserver` operates in a best-effort mode, even when processing and dispatching requests, a single scale-up event could be unrepresentative due to transient timing advantages.

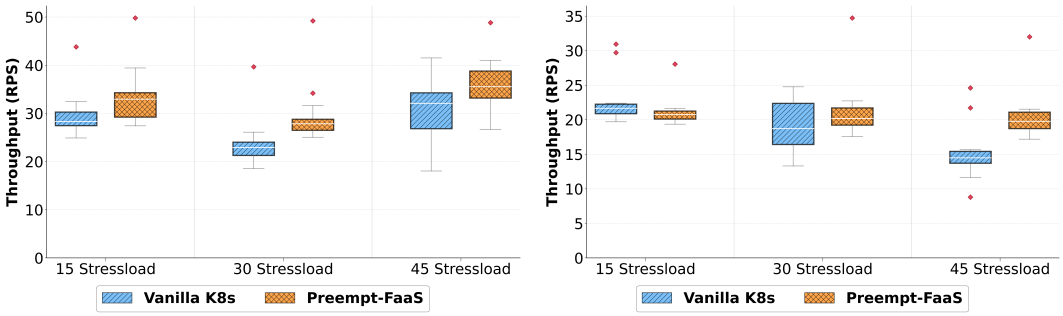
We executed 10 independent runs to ensure statistical significance. In each run, 5 vSwarm invokers target their respective KSVCs for 30 seconds at a constant rate of 50 RPS for the RNN service and 12 RPS for the video analytics service. A request timeout of 40 seconds is configured to capture delayed responses without premature termination. Preliminary sensitivity experiments identified 50 RPS for the RNN services as the scale-up threshold that consistently trigger Knative autoscaling, even when a pre-warmed instance is available. We emphasize that the goal of this evaluation is not to measure the maximum throughput capacity of the services themselves, but rather to assess how orchestration efficiency affects the achieved throughput and latency under controlled scaling conditions when comparing Vanilla K8s and PREEMPT-FaaS in cold-start conditions.

Figure 7 reports the sensitivity analysis of pod creation delays under increasing interference. For single-replica scale-ups, this metric corresponds to the initial control-plane reaction time in Vanilla K8s, and we report the equivalent stage for PREEMPT-FaaS for consistency. Results are aggregated across all KSVCs for each experimental run. Results confirm (as demonstrated in RQ1) that Vanilla K8s exhibits increasing latency and variability as interference grows, failing to maintain bounded orchestration times. In contrast, PREEMPT-FaaS preserves stable and predictable behavior across all stress levels. Although the data might initially appear inconsistent, specifically at lower stress loads, with the results presented in Figure 6, these discrepancies derive from three different experimental configurations. Firstly, the continuous interfering bursts allow us to efficiently stress the kube-apiserver, even when employing a low number of interfering resources. Secondly, the controllers for



(a) RNN benchmark. **(b)** Video analytics benchmark.

Figure 7 Pod creation latencies under varying stressload for Vanilla K8s and PREEMPT-FaaS during cold-start (i.e., 0-to-1 scale-up).

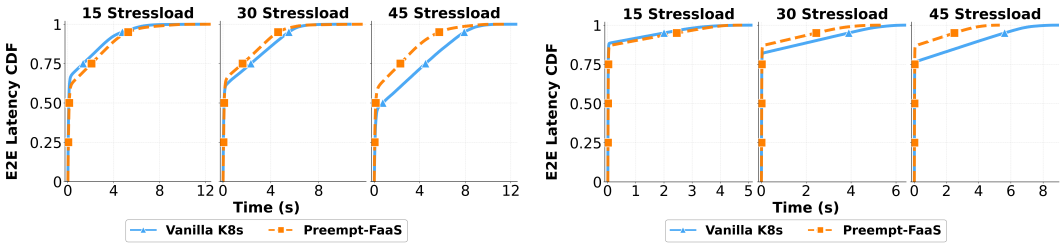


(a) RNN benchmark. **(b)** Video analytics benchmark.

Figure 8 Throughput distributions for benchmarks across Vanilla K8s and PREEMPT-FaaS controllers by varying stressloads. The data refers to a scale-up from 0 to 1 (cold-start) instance per KSVC.

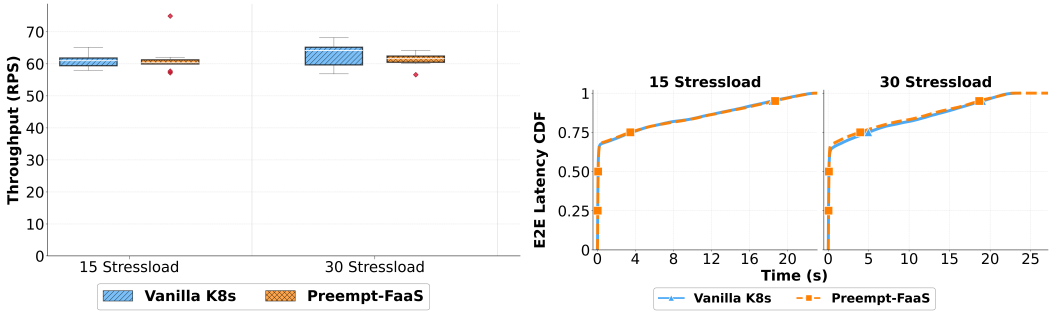
standard Deployments and RTResources, despite sharing the same overall logic, differ in the implementation of the respective control loops. For instance, whenever PREEMPT-FaaS processes a new event, it triggers an immediate interaction with the Kube-apiserver to update the state of the RTResource, ensuring low-latency synchronization with serverless platforms, e.g., Knative. The last critical factor involves service prioritization. Finally, we remark that while Figure 6 provides aggregated results across multiple scale-up events, Figure 7 focuses on individual scale-up operations per Knative Service (KSVC) within each iteration.

Figure 8 and Figure 9 show the improvements in E2E metrics achieved by a faster orchestration process performed by the PREEMPT-FaaS controller. Figure 8 illustrates the aggregate throughput achieved by the KSVCs, calculated as the sum of the actual RPS served by all deployed applications per experimental run. The results demonstrate a consistent performance gain for each level of interference. Specifically, the PREEMPT-FaaS controller brings a marginal but steady improvement of approximately 5 RPS. When integrated over the full test duration, this leads to a non-negligible increase in QoS. Unlike video analytics, the RNN benchmark does not show a consistent throughput decrease as the interfering load increases. This behavior is primarily attributed to the inherent instability of the invokers, which provide RPS that fluctuate around the nominal configured value. Furthermore, a load of 50 RPS per KSVC represents a saturation point for these services. In contrast, the video analytics benchmark consistently handles the maximum achievable RPS per KSVC. For this reason, we selected 12 RPS per KSVC to avoid unnecessary network congestion. Consequently, under lower interfering loads, Vanilla K8s is still able to outperform PREEMPT-FaaS.



(a) RNN benchmark. **(b)** Video analytics benchmark.

Figure 9 CDF of E2E latencies for benchmarks across Vanilla K8s and PREEMPT-FaaS controllers by varying stressloads. The data refers to a scale-up from 0-to-1 (cold-start) instance per KSVC.



(a) RNN pre-warmed (scale 1->2) throughput. **(b)** RNN pre-warmed (scale 1->2) E2E latencies.

Figure 10 Throughput distribution and CDF of E2E latencies for both the Vanilla K8s and PREEMPT-FaaS controllers, across stressloads, when scaling up instances in a pre-warmed scenario.

However, the most significant contribution is observed in the service-level performance. Figure 9 reports the mean latency, averaged across all active services. The Cumulative Distribution Function (CDF) highlights the following improvements granted by our proposal: while the Vanilla K8s controller exhibits a distribution that shifts significantly toward higher latencies as the interfering load increases, the PREEMPT-FaaS controller maintains a nearly constant profile. At the highest stressload, the RNN benchmark shows for the Vanilla K8s controller a mean latency of 761.45ms at the 50th percentile and 7913.29ms at the 95th percentile. The respective values for PREEMPT-FaaS are 145.21ms and 5716.75ms, showing an improvement of 81% at 50th percentile and 28% at the 95th percentile. For the video analytics benchmark, 50th and 95th percentiles are 20.58ms and 5616.86ms respectively for Vanilla K8s, while the same values for PREEMPT-FaaS are 23.37ms and 2500.57ms. PREEMPT-FaaS shows an improvement of 55% at the 95th percentile. This stability confirms that by minimizing orchestration overhead and ensuring service prioritization, the PREEMPT-FaaS controller directly enhances the end-to-end responsiveness of the system, regardless of control-plane congestion.

We evaluate the RNN benchmark in a pre-warmed configuration, with one active instance per KSVC and 10 runs per interference level. By exceeding the default autoscaling threshold, we trigger a scale-up from 1 to 2 replicas to assess the impact of orchestration efficiency on end-to-end (E2E) metrics. Figure 10 reports results for interfering loads of 15 and 30 events. Although experiments were also conducted at a stressload of 45 creation/deletion events, the maximum sustainable load of our cluster, the resulting control-plane and communication overhead led to unstable behavior; therefore, we focus on the two stable interference levels. The results indicate that a single replica can sustain the nominal request rate under moderate load. However, by accelerating the activation of the second replica, PREEMPT-FaaS reduces the time during which requests are queued on a single instance. This earlier load distribution mitigates transient congestion effects, leading to more stable throughput. While average latencies remain comparable, PREEMPT-FaaS exhibits lower variability in end-to-end throughput under contention.

RQ2: E2E Latency and Throughput Improvements due to Faster Scaling

Under high orchestration interference, PREEMPT-FaaS improves performance in the orchestration process, leading to substantial gains in terms of service E2E metrics, especially under cold-start conditions where services must be scaled as soon as possible to respond to incoming load. In particular, the sensitivity analysis provided evidence of shorter mean latencies, with a **2-second reduction** at 95th percentile for the highest stressload, and **improved throughput**.

5 Related Work

The optimization of application instance management in *Function-as-a-Service* (FaaS) environments is a widely discussed topic, addressed primarily through two distinct approaches: cold-start avoidance and cold-start latencies reduction [16].

Avoiding Cold Starts. A primary strategy to mitigate cold-start impact is to extend the lifecycle of instances beyond their execution time, allowing them to handle requests that may arrive shortly before or after the main burst that would trigger the scaling. Popular frameworks such as Knative [20] and OpenFaaS [37] implement this through *keep-alive* policies, which maintain instances in an active state for a predefined temporal window. For instance, Knative defaults to a 60-second keep-alive period [22], whereas OpenFaaS configurations can extend this window up to 30 minutes. Similarly, *pre-warming* mechanisms attempt to scale up resources in advance, before predicted traffic reaches the platform. Although effective, these methods introduce a significant drawback: idle instances consume resources without performing work. This behavior partially undermines the core value of the serverless computing approach. These strategies also shift the burden towards load prediction [58, 44, 35, 10]; inaccurate forecasting can lead to the same over- and under-provisioning miscalculations that serverless computing aims to solve. Moreover, implementation choices, e.g., the programming language adopted to develop the service, play a crucial role. Research [57] indicates that certain runtimes significantly influence the instance spawning process; therefore, selecting a language aligned with the specific performance requirements of the application is a fundamental step in avoiding cold-start overhead.

Reducing Cold-Start Latencies. Recent studies on cold-start time reduction provide worker node optimizations through application-based [15, 31], checkpoint-based [54], and cache-based [59] solutions. These studies reduce the overhead of downloading images and configuring execution environments, and optimize container layer management and runtime initialization. [1] proposes a study that compares different virtualization technologies and guest OS configurations to evaluate their impact on real-time applications startup times. However, to the best of our knowledge, they all neglect the orchestration latency, also referred to as the "declarative tax" in [30]. This tax represents the time required by the orchestrator to translate declarative configurations into imperative cluster reconciliation commands. As container startup times shrink due to technical optimizations, this orchestration overhead becomes a dominant bottleneck in end-to-end responsiveness, as we have shown in our experiments. In addition, a critical gap remains in how current platforms manage mixed-criticality workloads. Under heavy loads, concurrent scaling requests for multiple applications can lead to unpredictable growth in event-handling times. Our approach addresses this gap by introducing SLO-aware preemptive orchestration at the Kubernetes control-plane level following a priority driven architecture that leverages OS-level real-time scheduling, similarly to some studies in database environments [3, 2].

Real-Time Orchestration. Other research projects focus on the real-time behavior of critical services, once deployed. A notable example is **KubeDeadline** [47], a framework designed to enhance Kubernetes' real-time capabilities by focusing on container runtime behavior. The core of their architecture leverages the ***Dynamic Resource Allocation*** (**DRA**) feature, which extends the standard resource claim mechanism, typically used to mount persistent

volumes, to any kind of CR. This feature differs from traditional CRs, since CRs are employed to introduce custom data and logic in the cluster, while DRA allows to improve the Pod specification. Authors utilize it to allow pods to claim specialized resources – such as FPGAs, GPUs, or specific CPU scheduling policies – by referencing a "ResourceClaim" in the pod manifest. Specifically, they allow pods to request the `SCHED_DEADLINE` policy by defining the required period (which also represents the implicit deadline), CPU count, and runtime fraction. Other real-time orchestration studies typically concentrate on task execution times on nodes or orchestrator throughput [7, 17, 49, 13, 33], however, neglecting the impact of the management layer on the critical path of scaling operations.

Our approach is complementary, and not alternative, to the above proposals. For instance, considering the aforementioned KubeDeadline, since our RTResource employs a templating mechanism to embed the pod manifest, it preserves DRA claim fields in a completely transparent manner. This allows seamless integration: PREEMPT-FaaS optimizes the early orchestration stages, while KubeDeadline can handle subsequent resource allocation and runtime enforcement.

6 Conclusions and Future Work

In this paper, we addressed the problem of slow and unpredictable orchestration latencies within Kubernetes in serverless computing scenarios. We proposed and evaluated PREEMPT-FaaS, a framework based on preemptive service handling, designed to maintain Quality of Service in latency-sensitive environments. Our experimental results demonstrate that optimizing the orchestration times significantly reduces cold-start end-to-end latencies and throughput of services to be prioritized over non-critical ones. We found that vanilla Kubernetes' best-effort behavior is inadequate to manage services with strict SLOs and different priorities under heavy load.

Our analysis of the state-of-the-art and the results provided by our experimental campaign also identified two critical components that remain the main bottlenecks in the service startup pipeline. The kube-apiserver struggles to process and dispatch high volumes of concurrent requests efficiently. The worker nodes' kubelet performance degrades significantly when overwhelmed by simultaneous requests, directly impacting end-to-end responsiveness.

Future work will address the above issues by extending the approach we employed for the Deployments/ReplicaSet control loop also to other resources' lifecycle and K8s internals, i.e., the kubelet and the kube-apiserver. In addition, in the current implementation, the single-threaded State Updater is a performance bottleneck, since it has to update the status of every RTResource in the system, interacting with a kube-apiserver that could already be under pressure. We plan to re-design the State Updater as a multi-threaded task, with each thread having a specific set of priority levels to monitor. Finally, the experimental evaluation can be repeated by enabling real-time OS features (e.g., the `PREEMPT_RT` patch) to remove as many latency killers as possible.

References

1 Luca Abeni. Virtualized real-time workloads in containers and virtual machines. *J. Syst. Archit.*, 154:103238, 2024. doi:10.1016/j.sysarc.2024.103238.

2 Remo Andreoli, Tommaso Cucinotta, and Daniel Bristot de Oliveira. Priority-driven differentiated performance for nosql database-as-a-service. *IEEE Trans. Cloud Comput.*, 11(4):3469–3482, 2023. doi:10.1109/TCC.2023.3292031.

3 Remo Andreoli, Tommaso Cucinotta, and Dino Pedreschi. Rt-mongodb: A nosql database with differentiated performance. In Markus Helfert, Donald Ferguson, and Claus Pahl, editors, *Proceedings of the 11th International Conference on Cloud Computing and Services Science, CLOSER 2021, Online Streaming, April 28-30, 2021*, pages 77–86. Science and Technology Publications (SciTePress), SCITEPRESS, 2021. doi:10.5220/0010452400770086.

4 Ali Balador, Johan Eker, Raihan Ul Islam, Raquel Mini, Klas Nilsson, Mohammad Ashjaei, Saad Mubeen, Hans Hansson, and Karl-Erik Årzén. Aorta: Advanced offloading for real-time applications. In *Real-Time Cloud (RT-CLOUD), 2023*, 2023.

5 Rabindra K. Barik, Rakesh K. Lenka, K. Rahul Rao, and Devam Ghose. Performance analysis of virtual machines and containers in cloud computing. In *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pages 1204–1210, 2016. doi:10.1109/CCAA.2016.7813925.

6 Marco Barletta, Marcello Cinque, and Luigi De Simone. Slo-aware prioritization of orchestration times for containerized services. *ACM Trans. Internet Techn.*, 26(1):13:1–13:29, January 2026. doi:10.1145/3767329.

7 Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Criticality-aware monitoring and orchestration for containerized industry 4.0 environments. *ACM Trans. Embed. Comput. Syst.*, 23(1):5:1–5:28, 2024. doi:10.1145/3604567.

8 Marco Barletta, Luigi De Simone, Raffaele Della Corte, and Catello Di Martino. Failover timing analysis in orchestrating container-based critical applications. In *19th European Dependable Computing Conference, EDCC 2024, Leuven, Belgium, April 8-11, 2024*, pages 81–84. IEEE, 2024. doi:10.1109/EDCC61798.2024.00026.

9 Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen, and Jordy Dieltjens. Reducing cold starts during elastic scaling of containers in kubernetes. In Chih-Cheng Hung, Jiman Hong, Alessio Bechini, and Eunjee Song, editors, *SAC '21: The 36th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, Republic of Korea, March 22-26, 2021*, pages 60–68. ACM, 2021. doi:10.1145/3412841.3441887.

10 David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. Using application knowledge to reduce cold starts in faas services. In Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini, editors, *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, pages 134–143. ACM, 2020. doi:10.1145/3341105.3373909.

11 Marcello Cinque, Luigi De Simone, Raffaele Della Corte, and Stefano Toscano. dessertlab/knative-crd-scaled. Software, version 1.0., swhId: swh:1:dir:248b9febbae1cd09735cc4c896f544e72cbed87b (visited on 2026-06-15). URL: https://github.com/dessertlab/knative-crd-scaled, doi:10.4230/artifacts.26590.

12 Marcello Cinque, Luigi De Simone, Raffaele Della Corte, and Stefano Toscano. dessertlab/preempt-k8s. Software, version 1.0., swhId: swh:1:dir:aa067aa2469f1a7a0ef5dad20b909e8c1453a6eb (visited on 2026-06-15). URL: https://github.com/dessertlab/preempt-k8s, doi:10.4230/artifacts.26589.

13 Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. Strong temporal isolation among containers in openstack for NFV services. *IEEE Trans. Cloud Comput.*, 11(1):763–778, 2023. doi:10.1109/TCC.2021.3116183.

14 Knative developers. Performance tests. https://github.com/knative/serving/tree/main/test/performance, 2025. Accessed 15th June 2026.

15 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 *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020. doi:10.1145/3373376.3378512.

16 Ana Ebrahimi, Mostafa Ghobaei-Arani, and Hadi Saboohi. Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. *J. Syst. Archit.*, 151:103115, 2024. doi:10.1016/j.sysarc.2024.103115.

17 Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. Rt-kubernetes: containerized real-time cloud computing. In Jiman Hong, Miroslav Bures, Juw Won Park, and Tomás Cerný, editors, *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*, pages 36–39. ACM, 2022. doi:10.1145/3477314.3507216.

18 Grafana. Grafana/loki documentation, 2026. Accessed: 15th June 2026. URL: https://grafana.com/docs/.

19 Byeonghui Jeong and Young-Sik Jeong. Autoscaling techniques in cloud-native computing: A comprehensive survey. *Comput. Sci. Rev.*, 58:100791, 2025. doi:10.1016/j.cosrev.2025.100791.

20 Nima Kaviani, Dmitriy Kalinin, and E. Michael Maximilien. Towards serverless as commodity: a case of knative. In *Proceedings of the 5th International Workshop on Serverless Computing, WOSC@Middleware 2019, Davis, CA, USA, December 09-13, 2019*, pages 13–18. ACM, 2019. doi:10.1145/3366623.3368135.

21 Vojdan Kjorveziroski and Sonja Filiposka. Kubernetes distributions for the edge: serverless performance evaluation. *J. Supercomput.*, 78(11):13728–13755, 2022. doi:10.1007/s11227-022-04430-6.

22 Knative. Knative documentation, 2026. Accessed: 15th June 2026. URL: https://knative.dev/docs/.

23 Heiko Koziolek and Nafise Eskandani. Lightweight kubernetes distributions: A performance comparison of microk8s, k3s, k0s, and microshift. In Marco Vieira, Valeria Cardellini, Antinisca Di Marco, and Petr Tuma, editors, *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, ICPE 2023, Coimbra, Portugal, April 15-19, 2023*, pages 17–29. ACM, 2023. doi:10.1145/3578244.3583737.

24 Kubernetes. Auditing, 2026. Accessed: 15th June 2026. URL: https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/.

25 Kubernetes. Event, 2026. Accessed: 15th June 2026. URL: https://kubernetes.io/docs/reference/kubernetes-api/cluster-resources/event-v1/.

26 Kubernetes. Kubernetes documentation, 2026. Accessed: 15th June 2026. URL: https://kubernetes.io/.

27 Jon Larrea, Andrew E. Ferguson, and Mahesh K. Marina. Corekube: An efficient, autoscaling and resilient mobile core system. In Xavier Costa-Pérez, Joerg Widmer, Diego Perino, Domenico Giustiniano, Haitham Al-Hassanieh, Arash Asadi, and Landon P. Cox, editors, *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2023, Madrid, Spain, October 2-6, 2023*, pages 25:1–25:15. Association for Computing Machinery, ACM, 2023. doi:10.1145/3570361.3592522.

28 Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Cheng-Zhong Xu. Serverless computing: State-of-the-art, challenges and opportunities. *IEEE Trans. Serv. Comput.*, 16(2):1522–1539, 2023. doi:10.1109/TSC.2022.3166553.

29 Linux Foundation. Nephio: Cloud Native Network Automation. https://nephio.org/about/. Accessed 15th June 2026.

30 Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. The gap between serverless research and real-world systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing, SoCC 2023, Santa Cruz, CA, USA, 30 October 2023 - 1 November 2023*, pages 475–485. ACM, 2023. doi:10.1145/3620678.3624785.

31 Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. *FaaSLight*: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Trans. Softw. Eng. Methodol.*, 32(5):119:1–119:29, 2023. doi:10.1145/3585007.

32 Google LLC. Go 1.15 release notes, 2026. Accessed: 15th June 2026. URL: https://go.dev/doc/go1.15.

33 Francesco Lumpp, Franco Fummi, Hiren D. Patel, and Nicola Bombieri. Enabling kubernetes orchestration of mixed-criticality software for autonomous mobile robots. *IEEE Trans. Robotics*, 40:540–553, 2024. doi:10.1109/TRO.2023.3334642.

34 Philippe Martin. Extending kubernetes api with custom resources definitions. In *Kubernetes Programming with Go: Programming Kubernetes Clients and Operators Using Go and the Kubernetes API*, pages 193–207. Springer, 2022.

35 Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In Christina Delimitrou and Dan R. K. Ports, editors, *11th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2019, Renton, WA, USA, July 8, 2019*. USENIX Association, 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/mohan.

36 Open5Gs developers. Open5Gs. https://github.com/open5gs/open5gs. Accessed 15th June 2026.

37 OpenFaas developers. OpenFaas homepage. https://www.openfaas.com/. Accessed on 15th June 2026.

38 OpenTelemetry. Opentelemetry documentation, 2026. Accessed: 15th June 2026. URL: https://opentelemetry.io/docs/platforms/kubernetes/getting-started/.

39 Istvan Pintye, József Kovács, and Róbert Lovas. Enhancing machine learning-based auto-scaling for cloud resource orchestration. *J. Grid Comput.*, 22(4):68, 2024. doi:10.1007/s10723-024-09783-1.

40 Georgios Pournaras, Vasileios Karakostas, George Papadimitriou, and Dimitris Gizopoulos. Benchmarking support for RISC-V cpus in serverless computing. In *IEEE International Symposium on Workload Characterization, IISWC 2025, Irvine, CA, USA, October 12-14, 2025*, pages 520–523. IEEE, 2025. doi:10.1109/IISWC66894.2025.00050.

41 Bin Qian, Jie Su, Zhenyu Wen, Devki Nandan Jha, Yinhao Li, Yu Guan, Deepak Puthal, Philip James, Renyu Yang, Albert Y. Zomaya, Omer F. Rana, Lizhe Wang, Maciej Koutny, and Rajiv Ranjan. Orchestrating the development lifecycle of machine learning-based iot applications: A taxonomy and survey. *ACM Comput. Surv.*, 53(4):82:1–82:47, 2021. doi:10.1145/3398020.

42 Haoran Qiu, Saurabh Jha, Subho S. Banerjee, Archit Patke, Chen Wang, Hubertus Franke, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Is function-as-a-service a good fit for latency-critical services? In *WoSC '21: Proceedings of the Seventh International Workshop on Serverless Computing (WoSC7) 2021, Virtual Event, Québec City, Canada, 6 December 2021*, pages 1–8. ACM, 2021. doi:10.1145/3493651.3493666.

43 Ajay Rathee and Sandeep Dalal. A systematic literature review of machine learning-based resource allocation techniques in cloud computing. *Computing*, 107(9):179, August 2025. doi:10.1007/s00607-025-01526-8.

44 Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In Babak Falsafi, Michael Ferdman, Shan Lu, and Thomas F. Wenisch, editors, *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*, pages 753–767. ACM, 2022. doi:10.1145/3503222.3507750.

45 Rust. Libc Rust Library Documentation. https://docs.rs/libc/latest/libc/, 2026. Accessed 15th June 2026.

46 Farid Samandi, Natheesan Ratnasegar, and Michael Ferdman. A case for hardware memoization in server cpus. *IEEE Comput. Archit. Lett.*, 23(2):231–234, 2024. doi:10.1109/LCA.2024.3505075.

47 Nasim Samimi, Luca Abeni, Daniel Casini, Mauro Marinoni, Twan Basten, Mitra Nasri, Marc Geilen, and Alessandro Biondi. Enabling containerisation of distributed applications with real-time constraints. In Renato Mancuso, editor, *37th Euromicro Conference on Real-Time Systems, ECRTS 2025, Brussels, Belgium, July 8-11, 2025*, volume 335 of *LIPIcs*, pages 3:1–3:29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPIcs.ECRTS.2025.3.

48 Larissa Schmid, Marcin Copik, Alexandru Calotoiu, Laurin Brandner, Anne Koziolek, and Torsten Hoefler. Sebs-flow: Benchmarking serverless cloud function workflows. In *Proceedings of the Twentieth European Conference on Computer Systems, EuroSys 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, pages 902–920. ACM, 2025. doi:10.1145/3689031.3717465.

49 Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. Hierarchical resource orchestration framework for real-time containers. *ACM Trans. Embed. Comput. Syst.*, 23(1):4:1–4:24, 2024. doi:10.1145/3592856.

50 Márk Szalay, Péter Mátray, and László Toka. Real-time faas: Towards a latency bounded serverless cloud. *IEEE Trans. Cloud Comput.*, 11(2):1636–1650, 2023. doi:10.1109/TCC.2022.3151469.

51 Stefano Toscano, Luigi De Simone, Marco Barletta, and Marcello Cinque. PREEMPT-K8S: pod prioritization for mixed-criticality edge-cloud services. In *28th Euromicro Conference on Digital System Design, DSD 2025, Salerno, Italy, September 10-12, 2025*, pages 206–213. IEEE, IEEE, 2025. doi:10.1109/DSD67783.2025.00039.

52 Minh-Ngoc Tran and Young-Han Kim. Optimized resource usage with hybrid auto-scaling system for knative serverless edge computing. *Future Gener. Comput. Syst.*, 152:304–316, 2024. doi:10.1016/j.future.2023.11.010.

53 Vincent Uchenna Ugwueze. Serverless computing: Redefining scalability and cost optimization in cloud services. *International Research Journal of Modernization in Engineering Technology and Science*, 2024.

54 Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In Tim Sherwood, Emery D. Berger, and Christos Kozyrakis, editors, *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021*, pages 559–572. ACM, 2021. doi:10.1145/3445814.3446714.

55 vhive serverless. vswarm, 2025. Accessed: 15th June 2026. URL: https://github.com/vhive-serverless/vSwarm.git.

56 Bin Wang, Ahmed Ali-Eldin, and Prashant J. Shenoy. Lass: Running latency sensitive serverless computations at the edge. In Erwin Laure, Stefano Markidis, Ana Lucia Verbanescu, and Jay F. Lofstead, editors, *HPDC '21: The 30th International Symposium on High-Performance Parallel and Distributed Computing, Virtual Event, Sweden, June 21-25, 2021*, HPDC '21, pages 239–251. ACM, 2021. doi:10.1145/3431379.3460646.

57 Dong Xie, Yang Hu, and Li Qin. An evaluation of serverless computing on X86 and ARM platforms: Performance and design implications. In Claudio Agostino Ardagna, Carl K. Chang, Ernesto Daminai, Rajiv Ranjan, Zhongjie Wang, Robert Ward, Jia Zhang, and Wensheng Zhang, editors, *14th IEEE International Conference on Cloud Computing, CLOUD 2021, Chicago, IL, USA, September 5-10, 2021*, pages 313–321. IEEE, IEEE, 2021. doi:10.1109/CLOUD53861.2021.00045.

58 Zhengjun Xu, Haitao Zhang, Xin Geng, Qiong Wu, and Huadong Ma. Adaptive function launching acceleration in serverless computing platforms. In *25th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2019, Tianjin, China, December 4-6, 2019*, pages 9–16. IEEE, IEEE, 2019. doi:10.1109/ICPADS47876.2019.00011.

59 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 Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir, editors, *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024 – 1 May 2024*, pages 335–350. ACM, 2024. doi:10.1145/3617232.3624871.

60 Wenzhao Zhang, Yi Gao, and Wei Dong. Providing realtime support for containerized edge services. *ACM Trans. Internet Techn.*, 23(4):56:1–56:25, 2023. doi:10.1145/3617123.

61 Zipkin. Zipkin documentation, 2026. Accessed: 15th June 2026. URL: https://zipkin.io/.