SLO-aware Prioritization of Orchestration Times for Containerized Services

MARCO BARLETTA, DIETI, Università degli Studi di Napoli Federico II, Naples, Italy
MARCELLO CINQUE, DIETI, Università degli Studi di Napoli Federico II, Naples, Italy
LUIGI DE SIMONE, DIETI, Università degli Studi di Napoli Federico II, Naples, Italy

In this article, we present a timing analysis of orchestration times for containerized services, revealing the inability of current container orchestrators to fully prioritize services under concurrent requests. The analysis identifies the sources of orchestration delays that impact services to be prioritized potentially violating their Service Level Objectives (SLOs). Based on the findings of the timing analysis, we highlight three alternative SLO-aware orchestration system designs aimed at preventing and/or mitigating delays for high-priority services. We provide principles and guidelines that must drive the implementation of these designs. We then introduce *Ulysses*, a *Kubernetes*-based prototype embodying the simplest of the three designs. Ulysses modifies the core Kubernetes control plane components to manage events synchronously and with fixed priority. Through experiments conducted with both synthetic workloads and a containerized cloud-native 5G core network, we demonstrate that Ulysses ensures stable orchestration times for high-priority services, with a reduction of up to 78% under high orchestration load.

CCS Concepts: • **Networks** → **Cloud computing**; • **Computer systems organization** → **Cloud computing**; **Availability**; **Reliability;**

Additional Key Words and Phrases: Orchestration, containers, SLO, failover, services, real-time, cloud, kubernetes

ACM Reference Format:
Marco Barletta, Marcello Cinque, and Luigi De Simone. 2026. SLO-aware Prioritization of Orchestration Times for Containerized Services. *ACM Trans. Internet Technol.* 26, 1, Article 13 (January 2026), 29 pages. https://doi.org/10.1145/3767329

1 Introduction

Container orchestration systems (hereon, *orchestrators*) manage the life cycle of containers, including deployment, monitoring, and failover, across increasingly large-scale clusters of nodes (i.e., hundreds or even thousands [1]) [2–4]. Automatically managing containerized applications and services across the computing infrastructure resources is utterly important when the number of

This study was carried out within the MICS (Made in Italy – Circular and Sustainable) Extended Partnership and received funding from the European Union Next-GenerationEU (**PIANO NAZIONALE DI RIPRESA E RESILIENZA** (**PNRR**) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.3 – D.D. 1551.11-10-2022, PE00000004).
Authors' Contact Information: Marco Barletta, DIETI, Università degli Studi di Napoli Federico II, Naples, Italy; e-mail: marco.barletta@unina.it; Marcello Cinque, DIETI, Università degli Studi di Napoli Federico II, Naples, Italy; e-mail: macinque@unina.it; Luigi De Simone, DIETI, Università degli Studi di Napoli Federico II, Naples, Italy; e-mail: luigi.desimone@unina.it.

ACM 1533-5399/2026/01-ART13
https://doi.org/10.1145/3767329

applications and nodes spanning the edge, fog, and cloud infrastructure is remarkable, for example like in **Internet of Things** (**IoTs**) scenarios, including smart cities [5] and industrial IoT [6]).

In such scenarios, the numerous services composing applications have different requirements defined by their **Service Level Objectives** (**SLOs**): some SLOs define throughput contraints while others encompass increasingly strict constraints regarding availability, tail latency, or worst-case response times (e.g., real-time services like machine learning inference [7], virtual function networks for 5G infrastructures [8–10] or radio access networks [11, 12]).

In this context, the time the orchestrator requires to deploy, scale, or migrate a service/function directly impacts the SLOs of managed applications, especially (but not exclusively) in **function as a service** (**FaaS**) environments [13–15]. We call these *orchestration times*, defined as the time since when an event or request triggers the orchestrator intervention, to the finishing time of said intervention.[1] Examples of orchestration times inlude the deployment time, which directly affects the response time for a service/function without any running instance (e.g., a *cold start* [16]), or the scaling time,[2] which indirectly affects the response times because of request queuing [13]. For example, delays in orchestrating (e.g., scaling, migrating) the 5G User Plane Function (UPF, i.e., the function that routes the traffic of the user equipment) impact the performability, in terms of communication latency and/or availability [17]. During migrations, services may be pre-spawned to ensure availability [18]; however, this approach leads to resource overprovisioning and inefficiency.

Therefore, it is becoming relevant to systematically analyze the orchestration times and the causes of their delays as they affect the system reliability, understood as the capability to respect SLOs. In this regard, popular orchestrators like Kubernetes (hereon, *K8s*) and Docker Swarm are not able to fully prioritize services to safeguard their orchestration times when handling concurrent requests (see Figure 1 and later Section 3.3.1), and provide only partial *orchestration SLOs* [19].

Recent studies aimed at reducing the time to create sandboxes (e.g., containers) [20, 21] to mitigate the response time latencies due to cold starts. However, they overlooked orchestration times related to the control plane and left the core orchestrator components untouched. This neglects the orchestration times' variability due to the complexity of components and their interactions, which was identified as a performance bottleneck in [13] when compared to the instance startup times (which are $\approx$ tens/hundreds of milliseconds). Although orchestration times have a negligible impact on long-lived services with loose timing requirements, they can be utterly detrimental for *functions* scaled on demand and for soft and hard real-time functions/services, which make assumptions on the arrival curve of requests and perform a per-request admission test [22, 23]. When their workload increases, they either fail to meet timing guarantees or refuse requests until they are scaled up. In fact, workload characterization from production clusters showed that orchestration times vary in the order of seconds even in nominal conditions [15, 16], and resiliency studies on orchestrators showed that such times could be even longer and more variable due to overloads, and/or errors [24–26], despite the system capacity planning.

This article argues that orchestration times must be considered a first-class citizen for providing applications' SLOs guarantees in step with startup times, which have been widely investigated by previous literature, and claims that an orchestrator must provide orchestration's SLOs to services regardless of concurrent requests to meet applications' SLOs.

[1] Although the orchestrator is not directly responsible for the time to create containers on a node, its decisions may influence that time and its intervention may require actions after said creation, e.g., container network configuration. Thus, the orchestration times also include the container creation times.

[2] Depending on the orchestrator, the logic to deploy a container of a new service may differ from the service scaling. For example, in Kubernetes, scaling implies modifying an existing Deployment, while deploying requires creating a Deployment. Section 2 presents a breakdown of such times and their difference.

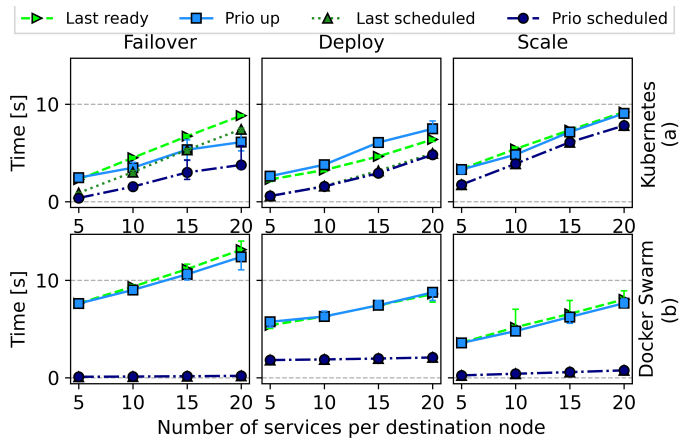


Fig. 1. Orchestration times for "failover", "deploy", and "scale" commands, under increasing orchestration load (each service is one container). A user request triggers simultaneous respawn, deploy, or outscaling (respectively for "failover", "deploy", and "scale" commands) of an increasing number of services (on the *x*-axis), starting the newly created pods on 2 worker nodes chosen as target (i.e., *destination*) by the orchestrator. For example, in the scenario "scale" for 20 services per destination node, the user request simultaneously increases the replicas number for 20 ReplicaSets, creating 40 Pods assigned to 2 destination nodes. A detailed setup is described in Section 3. *"Prio up"* is when a service that required prioritization started to respond to client requests, *"Last ready"* is when the last service was started. Even a single high-priority stateless service can experience fairly large orchestration times.

In this perspective, this article makes the following three contributions: *(i)* a timing analysis to understand the causes of large orchestration times and lack of prioritization of services by orchestrators; *(ii)* the introduction of possible designs and principles that must drive the development of SLO-aware container orchestrators, borrowing concepts used in co-kernels for real-time operating systems; *(iii)* the implementation of a *Ulysses*, i.e., a K8s-based prototype of the simplest architectures between the three proposed, along with experiments that measure the benefits for a high-priority service in terms of orchestration times.

Through the timing analysis, we identify the following sources of delay in orchestration times: *(i)* the asynchronous handling of events; *(ii)* the variable datastore response times; *(iii)* improper queue management; *(iv)* scaling delays; and *(v)* delays dependent on implementation choices, e.g., programming language and optimizations.

We highlight possible algorithmic design principles for SLO-aware orchestration components to mitigate identified delays, including multi-priority and synchronous event management. We propose and discuss three different SLO-aware architectural designs, inspired by real-time operating systems' design, and are: *(i)* a co-orchestrator, *(ii)* an orchestrator with some *ad hoc* SLO-aware components, and *(iii)* a patched orchestrator with components modified to be SLO-aware. Proposed designs build upon an architecture common to several orchestrators.

We implement a K8s-based prototype (named *Ulysses*, code publicly available at [27]) of the patched orchestrator design to show the potential improvements through targeted modifications to K8s. Our modifications turn into synchronous and multi-priority main K8s components (i.e., the *kube-controller-manager* and the *kubelet*). Ulysses is highly configurable, allowing the user to selectively enable SLO-aware features. We performed a comprehensive evaluation on two different clusters, with both increasing synthetic workload and a realistic containerized 5G control plane, i.e., Open5GS [10]. Results show that Ulysses guarantees constant orchestration times for prioritized

services in the most conservative configuration. In our setting, under the heaviest load, prioritized services experienced a 78% reduction of median orchestration times, at the cost of a median increase of 14.5% for low-priority services.

In summary, this article makes the following contributions:

— We found, through timing analysis, that **current orchestrators are not able to prioritize services**, causing delays in orchestration times that can violate SLOs;
— We define possible **designs and principles to SLO-aware orchestrators** able to mitigate the sources of delays;
— We implement a prototype of the simplest design as a patch to K8s that guarantees **constant orchestration times for prioritized services**, under variable load, while not significantly penalizing overall system performance.

2 Background and System Model

Containers allow distributing applications together with their dependencies [28]. Orchestrators are distributed systems providing scheduling, deployment, resource allocation, scaling, health monitoring, availability, load balancing, and networking of containers [3, 29]. A pod is the minimum unit that an orchestrator handles, and it consists of a set of related containers. We use *service* to indicate a set of one or multiple deployed pods responding to clients' requests. Each service is characterized by an SLO, which specifies non-functional requirements (e.g., percentage of availability, percentiles of latency). We characterize each service and its related resource instances with a *criticality level*, i.e., the designation of the level of assurance against a service SLO violation, following the model that we introduced in [30, 31]. Violations of critical SLOs must be avoided even in the presence of errors and/or overloads.

We assume, without lack of generality, a model in which requests to services are scheduled/admitted to guarantee their response time, and scaling is triggered by the number of queued requests. Since both characteristics are common to many environments based on a FaaS paradigm, we interchangeably refer to functions and *services*. Nonetheless, we do not necessarily imply a FaaS model (e.g., with scale-to-zero functions that negatively impact response times): our analyses and assumptions still hold with long-running services that perform per-request scheduling and are scaled based on request arrival.

Services are managed by an orchestrator, which can be generally divided into a *(i) control plane* and a *(ii) compute cluster* composed of *worker nodes*, on which pods run [3]. The control plane receives user requests, monitors the *current state* of the compute cluster (e.g., the number of pods spawned), and compares it with a *desired state* (e.g., the number of running pods). If the two states differ, the orchestrator performs *orchestration commands* to harmonize (*reconcile*) the two states [4]. A *worker node agent* reports to the control plane its state (including the state of the assigned pods), and spawns pods via a *container manager* (e.g., containerd, docker, CRI-O). Each worker node is characterized by an assurance level suited for the criticality of hosted services. We define as *orchestration time* the time required for an orchestration command to be completed. Orchestration times directly impact services' SLOs, thus they can be subject to dedicated SLOs [19].

We define the "deploy", "scale", and "failover" (also ascribable to *migration*) orchestration commands, which are widely used by orchestrators [2] and in the rest of this article. The "deploy" and "scale" commands create new pods or increase the pod replicas, respectively, including scheduling and starting such pods. The "failover" command restores the desired number of pod replicas after their number has decreased (due to a worker node failure for failover, or a stateful migration).

An orchestrator handles several *resource types* (e.g., pods, services, nodes), which have multiple *resource instances*. Each resource instance has a *priority level* that mirrors the criticality level of

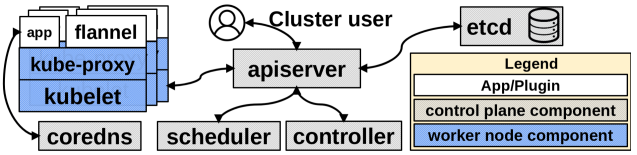


Fig. 2. K8s architecture.

the related service. Each resource instance has a desired and an observed state, reconciled by orchestration commands that account for priority.

An orchestration command includes a chain of actions. An action modifies the state of a resource instance, possibly triggering other actions. Each action is implemented by a control loop, which performs necessary reconciliations.

2.1 Kuberentes

Hereafter, we focus on K8s as a representative case study, extending the analysis to other systems when possible. K8s is currently the most widely used orchestrator with 97% of companies surveyed in [32] that use or evaluate to use it. Multiple solutions share its codebase (e.g., OpenShift, K0s, K3s, microK8s) [33] and multiple FaaS platforms rely on it.

K8s main components (depicted in Figure 2) are: *(i) Etcd*, a key-value store that keeps the cluster state guaranteeing sequential consistency; *(ii) Kube-apiserver* (hereafter, *Apiserver*), which interconnects control plane components, exposing the APIs to modify the state; *(iii) Kube-scheduler* (hereafter *Scheduler*), which places pods on nodes based on resource requests, availability, and constraints; *(iv) Kube-controller-manager* (hereafter, *Controller*), which implements the reconciliation control loops (see Section 2); *(v) Kubelet*, which is an agent on each worker node that spawns pods and reports the pods and node state to the control plane; *(vi) Kube-proxy*, which configures nodes' virtual networks.

The K8s resource types of interest include the *Pod*; the *ReplicaSet*, which ensures that a desired number of *Pod* replicas is running at any given time for availability and load balancing purposes; the *Deployment*, which makes it easy to change the number of replicas in *ReplicaSets*, and perform rolling updates; the *Service*, which provides a single network endpoint for accessing a set of load-balanced *Pods* that are characterized by different IP addresses; the *Node*, characterized by a state, available resources, and metadata, e.g., *Taints*, which can influence the *Pod* scheduling.

2.2 Orchestration Time Breakdown

This section describes a typical flow of actions (represented in Figure 3) that happen in the K8s control plane in order to exemplify what can be included in the duration of the *orchestration time* introduced in Section 1. Specifically, this subsection focuses on the K8s failover workflow because it involves multiple K8s subsystems. Subsequent sections will break down other orchestration commands based on the failover one.

It is worth noting that, although we here strictly refer to K8s, similar architectural concepts can be found in other platforms: for example, OpenFaaS [34] and Knative [35] are FaaS platforms on top of K8s, OpenWhisk [36] has controllers that rely on CouchDB as a consistent database to store function code and activation records, Docker relies on controllers acting upon a Raft-based distributed store.

Let us assume a cluster that has reached the steady state desired by the user, with the Pods are running on the Nodes, which may fail. After multiple missing heartbeats from a worker node, the Controller marks a node as failed and issues a state change to the Apiserver ⓪, which dispatches it

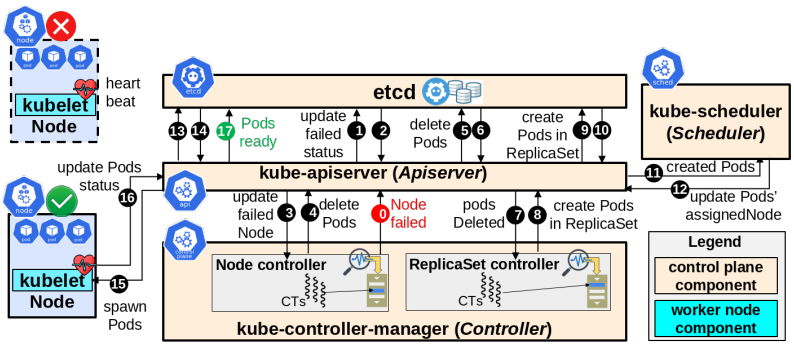


Fig. 3. Kubernetes' operations workflow during a failover command.

to Etcd to modify the persisted cluster state ①. The Controller is hence notified of the failed worker node (② and ③). At this point, the Controller handles the update regarding the failed worker node and orders to delete the Pods running on it (④,⑤). Later, the Controller handles the Pod deletion event (⑥, ⑦), e.g., the ReplicaSet controller reacts by creating the necessary number of Pods for the ReplicaSet involved (⑧, ⑨, and ⑩). The creation response ⑩ is forwarded to the Scheduler ⑪ that assigns the Pod to a worker node (⑫, ⑬, and ⑭). The required information is forwarded to the Kubelet to spawn the Pod ⑮, and update its status (⑯,⑰). Once the Pod is ready, the endpoint controller will configure the network to include the Pod in the Service load balancing (not shown in figure).

In this sense, the *orchestration time* as defined in Section 1 starts with ⓪ and ends with the network configuration triggered by ⑰, including ⑮ that is not a responsibility solely of the orchestrator. Similarly to the failover workflow, when a user requests a deployment, a new Deployment is created and stored on Etcd. The Controller consequently creates a ReplicaSet for the Deployment. Afterward, the ReplicaSet controller notices the newly created ReplicaSet and creates the requested number of Pods. Next, the Scheduler notices the new Pods and schedules them. The sequence is similar when the orchestrator decides a scaling, but in that case, the Deployment and/or ReplicaSet are only updated, because they already exist, as mentioned in Section 1.

3 Timing Analysis

This section presents an experimental analysis of orchestration times and their variability under increasing orchestration load conditions. The experiments are performed to answer the following **research questions** (**RQs**):

RQ1: *Are commonly used orchestrators able to fully prioritize services by their criticality?*
RQ2: *What are key factors that can affect the orchestration times?*
RQ3: *What are the sources of delay of orchestration times?*

3.1 Experimental Setup

Two setups are used: (i) a *cloud cluster*, (ii) an *edge cluster*. The cloud cluster includes 5 VMs (8 cores, 8 GB RAM each) running on multiple servers (Intel Xeon E5-2630L, SCSI storage). Each VM is a node of the cluster. One of the 5 VMs is the control plane VM, and runs on a dedicated server. The edge cluster is composed of a control plane workstation (Intel i7-4790, 16 GB RAM, SATA HDD) and 5 worker nodes, including two embedded devices (a Raspberry Pi 4, Cortex-A72 1.5 GHz, 4 GB RAM, and a Xilinx Kria KV260, Cortex-A53 1.5 GHz, 4 GB RAM, FPGA) and 3 VMs running across 2 servers with the same characteristics of the cloud servers. The Xilinx Kria KV260 and

Raspberry Pi 4 compose a representative heterogeneous edge cluster for cloud-edge systems. The KV260 provides real-time AI acceleration and reconfigurable logic for high-performance tasks, while the Raspberry Pi 4 offers a lightweight, general-purpose compute node. This combination is aligned with real-world cloud-edge deployment scenarios [37, 38]. All nodes run Linux v5.15 and are all connected through CAT5e Ethernet cables.[3] Both clusters run K8s v1.29 (`kubeadm` default configuration) with `Flannel` Container Network Interface, and `containerd` v1.7 as container manager.

3.2 Experiment Methods

The aim of the experiment is to measure how the orchestration times of a high-priority service vary when the orchestration system is busy handling an increasing number of concurrent events triggered by other low-priority services. We focus on the "failover" command since it involves more orchestrator subsystems than other workloads: health monitoring, replication management, pod scheduling, creation, and networking. In K8s, a failover subsumes both deployments and scaling, and takeaways similar to what we show for the "failover" command can be obtained with the other orchestration commands (recall Figure 1).

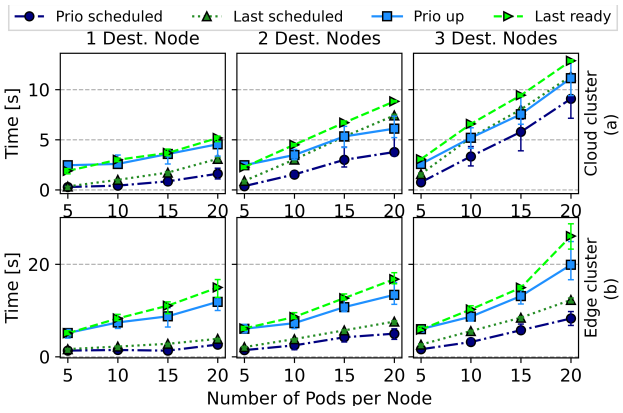
To perform a failover experiment, we trigger the respawn of one or more pods, which were initially deployed on a set of failed **Source worker Nodes** (*SN*), to a set of **Destination worker Nodes** (*DN*). The pods to be respawned include 1 pod of a *test service*, which is our high-priority service, while the others are *nginx* web servers, chosen because of the minimal startup overhead. The test service is a stateless UDP echo server that responds to a client. The client sends a UDP packet each 50 *ms*. The client waits for the response to each packet sent and logs the timestamps once received. The client is deployed on a node not involved in the failover.

We first place the test service on one of the *SN*. After the system stabilization, the failure of all *SN* is simulated through a *NoExecution taint* in K8s [39], and a custom label in Docker Swarm [40]. Through placement constraints, we force the test service to be respawned onto a VM (in the cloud cluster) or the Raspberry Pi (in the edge cluster). The other pods are free to be respawned on any *DN*. While control plane effects are independent of the choice of the worker node where the service is respawned, the worker node effects depend on it. Thus, we further repeated some experiments to compare the effects of the Raspberry Pi and the Kria KV260.

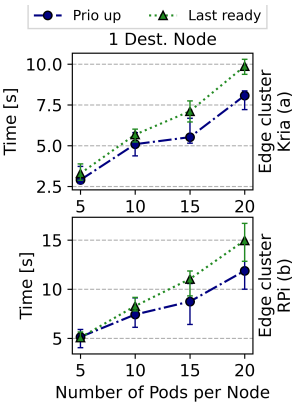
We configure the test service to have the highest *scheduling priority* in K8s. The container images are already downloaded (i.e., *pulled*) on the *DN* to exclude the high variability of pulling times [41]. During each experiment, the logs of K8s control plane components are collected to obtain the event timings. The key collected metrics are time differences between the observed event (e.g., the high-priority test service is up, or the last pod is recognized ready by the control plane) and the time of the *taint* request.

Experiment Parameters. Recalling that the default maximum number of pods that can run on a node is 110, we set 60 as a reasonably high number of pods to be orchestrated simultaneously for our testbed with 4 worker nodes. Previous works [29, 33] used similar numbers and production workload characterization backs these experiments [16]. To this aim, we run the experiment with 5, 10, 15, and 20 pods to be respawned for each destination worker node. We run the experiment for a *DN* size of 1, 2, and 3. Hence, with 3 *DN*, the total number of pods is respectively 15, 30, 45, 60. Said total number of pods aims at analyzing trends in the timing of the control plane, while the

[3]Despite not being included in this article in the interest of space, we also performed experiments with the Raspberry Pi connected through Wi-Fi, showing no substantial difference in results. This result was expected since we chose experimental parameters on purpose to reduce the interference due to network delays as much as possible, including, for example, having the images already downloaded on the nodes. Further details in Section 3.2.



(a) Results for cloud and edge clusters for an increasing number of Destination Nodes (DN).



(b) Comparison between Raspberry Pi (RPi) and Kria as DN for edge cluster.

Fig. 4. Timing of the events during the failover experiment with increasing activity. Error bars represent the interquartile ranges. *"Prio up"* is when the high-priority pod reacts to client requests, *"Last ready"* is when the last pod was recognized as ready by the control plane. The *"scheduled"* event is when the control plane decides the worker node to spawn the pod, and is the last event regarding the control plane.

number of pods per destination node aims at analyzing the effects on the worker node. Although scheduling 20 pods to the same node simultaneously is a rather unfortunate case, it has been proven in the real world that it can happen due to outstanding load conditions or errors [42]. The pods are grouped into *Deployments* of two pods each. For each combination $(DN, pod\_number, setup)$, we perform 30 experiment repetitions for statistical purposes.

3.3 Experimental Results

The answers to the RQs are summarized as follows:

Answers to RQs

- **A1**: Commonly used orchestrators are not able to prioritize critical services;
- **A2**: The key factors affecting orchestration times are: queue scheduling configuration (e.g., rate-limiting thresholds, controller parallelism level), datastore response time, and the adopted container manager;
- **A3**: The sources of delay are asynchronous event management, datastore unpredictable operations, request queuing, scaling effects, and implementation choices

The following subsections provide the details of experiment results answering the RQs.

3.3.1 A1 - Unability to Prioritize Services. The experiments on K8s and Docker swarm anticipated in Figure 1 clearly show that both platforms are not able to prioritize a service. More in details, in Figure 4(a) we show the timing of K8s events of interest in the two target clusters. The times of the events of interest in the figure are:

—"Prio up": time when the high-priority pod reacts to client requests since the taint request;
—"Last ready": time when the last pod is recognized as ready by the control plane since the taint request;
—"Prio scheduled": time when the control plane selects the WN where to spawn the high-priority pod, since the taint request;

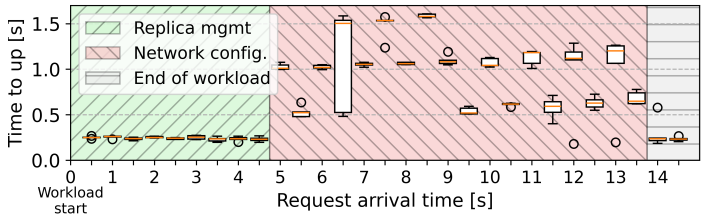


Fig. 5. In K8s, the time to reconfigure the network of a high-priority service to forward the traffic to a running pod, in the presence of a concurrent low-priority orchestration workload (i.e., failover of 60 pods). The y-axis represents the time interval from the reconfiguration request to the first response by the high-priority service; the x-axis represents the time between the start of the concurrent workload and the arrival of the network reconfiguration request for the high-priority service. When the configuration request for the high-priority service arrives $[0s, 5s[$ after the start of the concurrent workload, it is served relatively fast ($\approx$ 0.25 s). Indeed, during that interval, the low-priority workload stresses the replica management subsystem of K8s, which creates the 60 pods required to restore the desired state. However, when the request is sent in $[5s, 14s[$, the low-priority workload stresses the networking management subsystem of K8s, which must configure the network for all the pods created in the previous phase (replicas created). Thus, the network configuration for the high-priority service becomes slower and more variable.

— "Last scheduled": time of the last pod scheduling decision made by the control plane, since the taint request.

The results show an approximately linear increase of times as the number of pods increases. In the cloud cluster, the contributions in terms of time spent on the control plane and worker nodes look roughly balanced. The *"Last ready"* event is $\approx$2 seconds after the *"Last scheduled"* event, which suffers from a steep slope when the pod number increases up to 60 (i.e., 3 *DN*, 20 pods per node). Despite the similar trends in the edge cluster, spawning the pods on the Raspberry Pi becomes the main bottleneck under heavy load, as expected.

Although a higher scheduling priority is assigned to the high-priority service, K8s does not prioritize it because the priority only affects the scheduling phase. As shown in Figure 4(a) (see *"Prio up"* event), the high-priority service pod can take as long as 20 seconds to respond in the worst case. Even worse, it is not the first ready pod: in the answer to RQ2 in Section 3.3.2, we analyze the distribution of pod ready times for 20 pods on one node, showing that the first pod can be ready in $\approx$ 2 seconds (see Figure 9).

One of the reasons for such behavior is the queuing delay that can affect event handling when the system load increases. Most of the queues in K8s are managed through rate-limited FIFO (e.g., in the Controller) or coarse-grained Fair Queuing (e.g., in the Apiserver) policies, preventing a full prioritization of a request more urgent than others.

Similarly, Docker Swarm is not able to prioritize services over the others during orchestration. Indeed, it also uses FIFO queues and has no feature similar to scheduling priority.

One can argue that those times are just a matter of creating pods. However, we show that even assuming an already running replica, the time to bring up the service can be up to 6 times longer under orchestration load. Figure 5 shows the results of an experiment where the time required to configure the network of a high-priority (already running) service is plotted against its requests' arrival time. When K8s networking components are under pressure (red area in Figure 5) because of concurrent low-priority workload (i.e., failover of 60 pods), the time to bring up the service increases from $\approx$0.25s to $\approx$1.5s. Hence, any worker node optimization to reduce start times, like container caching, cannot reduce this delay.

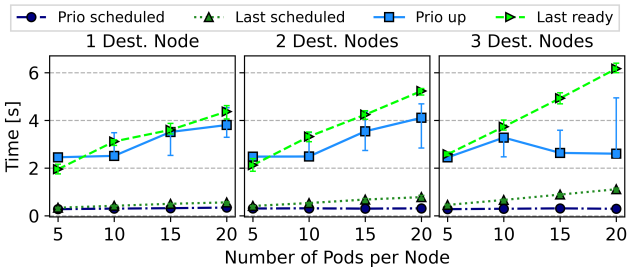


Fig. 6. Failover experiment in cloud cluster, with controller configured without rate limiting and high-priority service exempted from fair queuing in the Apiserver.

3.3.2 A2 - Main Factors Affecting Orchestration Times. Through our experiments, we identified three main factors affecting orchestration times: the configuration of queue scheduling, the response times of the datastore, and the adopted container manager. In the following, we examine the factors in detail.

Queue Scheduling Configuration. We focus on three key parameters of the K8s queue scheduling configuration: *(i) rate limit thresholds* for requests directed/coming to/from the Apiserver, *(ii) weighted fair queuing request scheduling* configuration in Apiserver, and *(iii) the number of concurrent control loop threads*. Note that we focus on K8s since it is the most mature orchestrator today; other orchestrators might not support such a fine-grained queuing control and configuration, presenting even more unpredictability.

To investigate the effect of rate limiting and fair queuing on orchestration times, we performed additional experiments on the cloud cluster with request rate limits between the Controller and the Apiserver disabled and the request belonging to the high-priority service configured to be exempt from weighted fair queuing.

Figure 6 shows that, when not rate-limited, the control plane scales better (see *"Prio scheduled"* and *"Last scheduled"* events) than the worker nodes (see *"Prio up"* and *"Last ready"* events), where most of the time is spent to spawn pods. The delays on the worker nodes are amplified as the pod number increases, becoming a bottleneck. Hence, the Kubelet must be designed to prioritize high-priority pods as well. We observed a similar behavior in Docker Swarm (not reported for brevity), which has no rate-limiting parameter.

In particular, Figure 7(a) and (b) show the histograms of runtimes of the control loops of K8s Controller, by enabling/disabling rate-limiting. The runtimes show that the default K8s Controller used for the experiment in Section 3.3.1 handled requests as soon as possible until it started throttling them to respect the rate limit (i.e., 20 requests per second by default). The *"Prio scheduled"* event in Figure 4(a) was affected by throttling, which does not account for priority.

Similar considerations hold for rate-limiting parameters and synchronization periods of Scheduler and Kube-proxy. The Scheduler never hits the rate limit and contributes for a few milliseconds to orchestration times in our experiments. Conversely, the Kube-proxy determines the orchestration times under heavy load in Figure 5 because of the synchronization period of the routing tables (i.e., 1s by default).

Although results show that disabling request rate limits can be beneficial on orchestration times, this configuration is highly discouraged by K8s developers, as it threatens production environments with random failures of orchestration components [43]. Similarly, exempting a set of requests from the weighted fair queuing can cause overloads and starve other requests.

Another crucial parameter of the K8s controller is the number of control loop threads. The K8s Controller has a control loop for each resource type, which spawns multiple control loop threads (5,

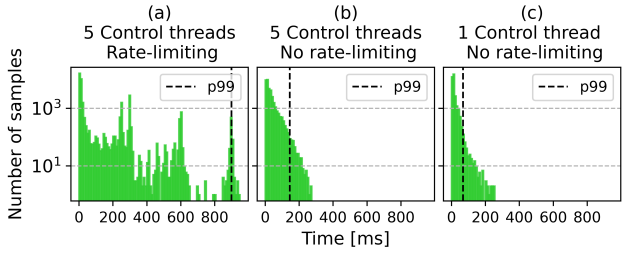


Fig. 7. Control loop runtimes for cloud cluster. 5 control loop threads and rate-limiting is the default configuration.

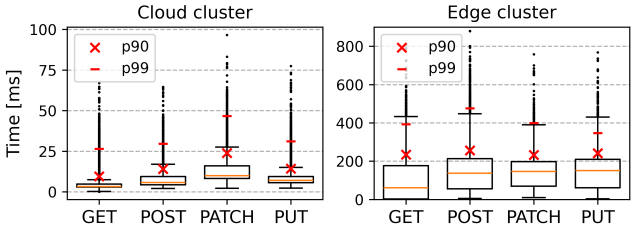


Fig. 8. Latencies for the Apiserver replies in the two clusters divided by HTTP verb. The different timescales are caused by Etcd, which is much slower in our edge cluster. $p90$ and $p99$ are, respectively, the 90^{th} and 99^{th} distribution percentiles.

by default). Such threads work in a "single queue, multiple server scheme", i.e., they pick requests from the same queue (one for each resource type). In the following, we investigate the effect of the number of control loop threads with rate-limit disabled. When the controller is configured to work with only one control loop thread (see Figure 7c), the results show that the 99^{th} percentile of loop times is drastically lower than Controller with increased parallelism. The root cause of the long-tail latencies is the asynchronous management of events. Indeed, the higher parallelism causes greater interferences caused by the pressure on the orchestrator components and resource contention due to the uncontrolled rate of handled requests, which suffer partial sequentialization (e.g., due to mutexes). The experiment has pure ablation purposes, and the reader should notice that reducing the number of control threads, although reducing interferences, drastically reduces the achievable throughput and scalability of the control plane, and it is not a viable solution in production.

Finally, it is worth noting that the asynchronous event management can cause priority inversions [44]: when a control loop thread immediately starts processing an incoming request as soon as it is free, it becomes unable to serve other requests until completion of the current. Hence, if a high-priority request arrives soon after, it must be queued.

Datastore Response Times. During previous experiments, the orchestration times related to the control plane significantly differed between the two clusters. Figure 8 shows the response times distributions of the Apiserver, which interacts with Etcd.

We observed that datastore response times affect the control plane timing behavior when not rate-limited, since Etcd is accessed by each action of an orchestration command chain (recall Section 2). For example, a failover in K8s requires at least 5 writes. Indeed, the control loop times (Figure 7) include one or more requests to the Apiserver. Response times can present orders of magnitude of variability due to load, underlying physical disk technology (e.g., HDD or SSD), and consensus protocols when replicated. Furthermore, datastore response times have a skewed distribution, characterized by long tails exceeding the $p99$. Repeating the experiments with an

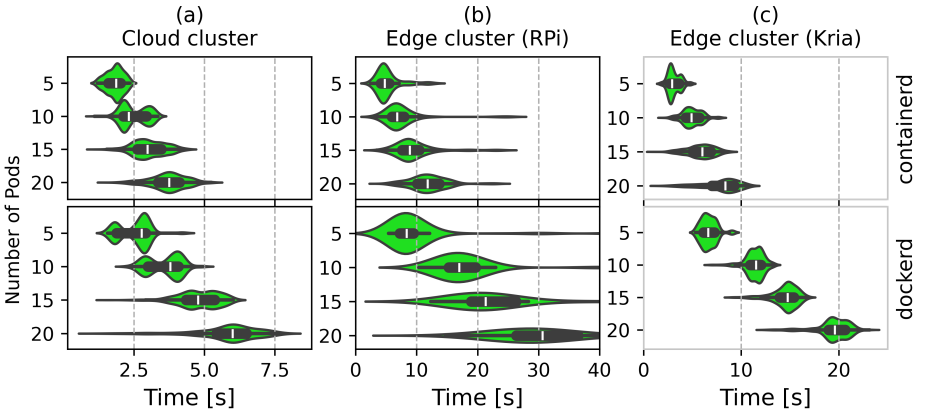


Fig. 9. Pod ready times distribution (1 *DN*). RPi is the Raspberry Pi, Kria is the Kria KV260.

in-memory Etcd in the cloud cluster, we obtained median latency reductions of ≈50% (≈6*ms*) for POST, PATCH, and PUT requests with a p-value≈ 10^{-7}. However, while in-memory datastores might be used to reduce latencies, consistent datastores preserving the status of the system usually require a distributed consensus among a majority of replicas for every write operation (e.g., creating a pod, updating a ReplicaSet).

Adoped Container Manager. We assess the impact of the container manager choice when spawning multiple pods on a single worker node. We compare `containerd` (K8s default choice) and `dockerd` (v24.0.7) in the cloud and edge clusters. The *DN* is a VM in the cloud cluster, while we repeat the experiment with both the Raspberry Pi and the Kria KV260 as *DN* in the edge cluster.

Figure 9 shows all "pod ready" times distributions. As the pod number increases, the time distributions present an increasing mode, and the ready time is clearly delayed for each pod. The pod spawn requests are served as soon as they arrive at the worker node, causing resource contention for container creation and configuration. This is particularly evident for `dockerd` on the embedded devices (Figure 9(b)–(c)). Pods including multiple containers add further delays.

3.3.3 A3 - Sources of Delay. The performed timing analysis allowed us to identify several sources of delay affecting orchestration times, here summarized. They are not specific to K8s, since other popular container orchestrators share the same architectural concepts as K8s, including a common datastore preserving the state, queues in the components, and asynchronous event management.

— ***Indirect delays* in asynchronous event management.** The state change events are often handled asynchronously, i.e., as soon as possible, in both worker nodes and the control plane. Asynchronous event handling creates interferences and resource contention that delay high-priority requests. Moreover, the requests can be throttled to respect possible rate limits. We define them *indirect delays* in both the control plane and the worker nodes as the delays caused by asynchronous management, including resource contention, pressure on shared components, throttling, and priority inversion.

— **Datastore delays.** Orchestrators generally store the current and desired cluster state in a datastore [4] (Etcd for K8s), which is frequently accessed during orchestration commands. Commonly used datastores (like Etcd) do not distinguish the incoming requests to accommodate services' SLOs and prioritize high-priority services. Replicated datastores can increase the delays because of the network delays that can affect the consensus protocol messages. Although read requests can be served through caching to avoid accessing the datastore [45], several write

requests are necessary to modify the cluster state and perform an orchestration command. Thus, the datastore remains a determining component for the overall orchestration latency.
- **Queuing delay.** An event is generally handled through a chain of cascading actions. Events can suffer from queuing delay for each action of the cascading sequence. Indirect delays and unpredictable datastore response times can cause order inversions in the components' queues, amplifying the orchestration times for high-priority services.
- **Scaling delays.** Algorithms implemented in orchestrators may have runtimes that depend on the number of resource instances in the system, increasing the orchestration times as the system size increases. For example, the K8s Scheduler runtime increases with the number of worker nodes to evaluate (up to a configurable maximum) to place a pod.
- **Implementation-related delays.** Delays due to the underlying software stack must be considered. Examples are *(i)* code optimizations baked in the code to improve the scalability at the cost of response times, e.g., request batching; *(ii)* communication over stacks without timing guarantees like HTTP over TCP; *(iii)* language-runtime-related delays, e.g., the unpredictable routine scheduling of Golang for K8s.

3.4 Summary of Answers to Research Questions

Orchestration Timing Analysis Findings

A1 – Inability to Prioritize Services

Although K8s allows to specify priorities, high-priority services are not prioritized end-to-end:
- Priority only affects scheduling. Other control plane operations, network setup, and pod readiness cannot be prioritized.
- Under load, a high-priority pod may take up to 20s to become ready while a regular pod may take only ~2s.
- Delays stem from queuing in K8s components (e.g., Controller, Apiserver) using FIFO or coarse-grained fair queuing.
- Despite pre-warming, high-priority services can suffer 6× longer network reconfiguration times (e.g., kube-proxy load).

A2 – Main Factors Affecting Orchestration Times

Queue Scheduling Configuration
- K8s rate limits and fair queuing may throttle even high-priority tasks.
- Disabling/increasing limits improves performance, but reduces stability and reliability.
- Increasing control loop threads can cause priority inversion due to asynchronous interference.

Datastore (Etcd) Response Times
- Accessing Etcd is essential to orchestration; latency varies by load, hardware, and network.
- Cloud vs. edge and persistent vs. in-memory Etcd show substantial timing differences.

Container Manager on Worker Nodes
- `containerd` generally outperforms `dockerd`, especially under load.
- All pod startup times increase as the pod number increases, more so on constrained devices.
- Multi-container pods and worker congestion worsen delays.

A3 – Sources of Delay
- *Indirect Delays:* Asynchronous processing leads to interference and priority inversion.
- *Datastore Delays:* Etcd latency is unavoidable and sensitive to hardware, network, and replication.
- *Queuing Delays:* Sequential orchestration stages accumulate queuing latency.
- *Scaling Delays:* The runtimes of the algorithm used for orchestration can increase with the number of nodes and/or pods.
- *Implementation Delays:* Framework-level choices (e.g., batching, TCP overhead, Go runtime) introduce additional variability.

4 SLO-aware Orchestration

This section uses the key findings from Section 3 to drive the design of SLO-aware orchestrators. Section 4.1 introduces **Design Principles** (**DPs**) to implement SLO-aware orchestrator components that mitigate the sources of delay identified in A3. Section 4.2 introduces three alternative architectures for SLO-aware orchestration systems, which leverage SLO-aware components.

4.1 Design Principles for SLO-aware Components

DP1 - Resource reservation and synchronous queue management. Resource reservation can be used, on both the control plane and worker nodes, to mitigate *indirect delays* due to asynchronous processing and priority inversion. The reservation can be achieved by using: *(i)* dedicated computational resources, or (ii) real-time schedulers (e.g., real-time servers [46]) to synchronously handle requests.

Synchronous queue management can guarantee enough resources to reduce the interferences, e.g., when spawning a pod on a worker node, and fix the problem of priority inversion (see Section 3.3.2). Indeed, similarly to non-work-conserving and anticipatory schedulers [47], synchronous queue management does not immediately serve low-priority requests.

DP2 - Low-latency datastores. Real-time datastores [48] can reduce *datastores delays*. In specific scenarios, an in-memory datastore could be considered to further reduce the response times, prioritizing low latencies over persistency.

DP3 - Multi-priority queues. *Queuing delays* can be bounded by allowing a finite population (i.e., number of resource instances) and using multi-priority management, to let high-priority requests preempt lower-priority ones. For example, dispatching messages among control plane components (i.e., Apiserver in K8s) can use algorithms typical of ***Time Sensitive Networking (TSN)***, including periodic, fixed-priority, and best-effort messages.

DP4 - Bounded-time algorithms. *Scaling delays* can be mitigated by employing algorithms whose runtimes do not depend on the number of resource instances to be served. The extreme solution is to define the behavior of the components ahead of time for each possible case. For example, the scheduler and controller can leverage pre-generated rules that make the reaction to an event, like deciding *a priori* where to re-schedule a service upon a node failure.

DP5 - Implementation choices oriented to real-time scheduling and low latency. *Implementation delays* can be addressed via a proper choice of the programming language to be used in an SLO-aware component, which can differ from vanilla components. This may help to leverage real-time scheduling and avoid language-runtime–related delays. Moreover, all the optimizations baked into the implementation details of a component can be designed to prefer low latencies over high throughput.

4.2 Architectural Designs

The design principles and mitigations listed in Section 4.1 can be implemented (all or a subset of them) in what we call orchestrator *SLO-aware components*, such as: the *controller*, the *scheduler*, the *datastore*, and the *node agent*.

We propose three architectural designs that use SLO-aware components (illustrated in Figure 10): a **co-orchestrator** (①) in Figure 10), an orchestrator with **additional components** (②), and a **patched orchestrator** (③).

We define *critical node* as a worker node running an SLO-aware node agent and providing a higher assurance level (recall Section 2) compared to other worker nodes. Critical services should be scheduled only on critical nodes. We define the other worker nodes as *standard nodes*. Proposed designs do not depend on a specific orchestrator: several products, like Docker Swarm, Apache

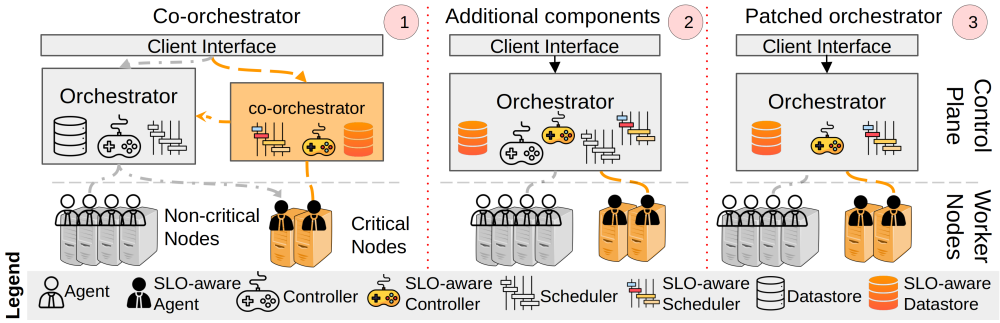


Fig. 10. Three proposed SLO-aware architectures for orchestrators.

Mesos, Openwhisk [36], Oakestra [49], share similar architectures [3], with a main datastore and controllers and schedulers acting on it.

4.2.1 Co-orchestrator. A *co-orchestrator* managing critical services operates alongside the *vanilla orchestrator* (hereon shortened as *orchestrator*). The system interface discerns requests and dispatches them to the orchestrator or co-orchestrator according to the criticality level. The idea is similar to real-time co-kernels for operating systems: co-kernels intercept interrupts to serve them with low latency through *ad hoc* kernel subsystems implementations.

When the co-orchestrator manages resources shared with the orchestrator, like critical nodes, the view of the node status must be coherent with the orchestrator. Hence, events regarding those nodes must be sent to the co-orchestrator, which propagates them to the orchestrator when possible.

The co-orchestrator includes an SLO-aware scheduler, controller, and datastore that can run on a dedicated node or use resources isolated from the vanilla control plane.

Co-orchestrator components are *dedicated*, written from scratch, and adopt all the design principles DP1 - DP5 listed in Section 4.1. Hence, co-orchestrator components can be written in a programming language that is different from the one of orchestrator components (i.e., usually Golang) and supports sound real-time scheduling of threads and, consequently, requests. Despite the high implementation effort, handling the critical services' requests through an end-to-end path of actions involving components with an *ad hoc* design achieves the best performances and reliability guarantees. Indeed, an issue (e.g., overloads, errors, failures) in the vanilla orchestrator (which remains unmodified) will not affect the co-orchestrator.

4.2.2 Additional Components. Some duplicated *ad hoc* (written from scratch) SLO-aware components are integrated into the vanilla orchestrator architecture (e.g., scheduler and controller designed according to DP1, DP3, DP4, DP5). Those components handle a set of resource instances that are disjoint from the set handled by vanilla components.

Conversely, other components (e.g., the datastore and the interconnecting component like the Apiserver in K8s), are shared between the action paths handling critical and non-critical requests. Those components must be configured or patched to differentiate the management of critical and non-critical requests, to reduce delay for critical requests (e.g., by adopting an in-memory datastore only for critical requests, according to DP2), while not overly affecting non-critical services.

Compared to the co-orchestrator design, this design is simpler since it does not require implementing all components from scratch. On the other hand, the disadvantage is that shared components, like the datastore if not properly patched, still handle the load of the entire system, and in non-nominal conditions may not be able to provide guarantees for critical services.

4.2.3 Patched Orchestrator. All the orchestrator components are shared between critical and non-critical action paths. The components are patched to be SLO-aware, adopting design principles DP1 - DP3, and provide differentiated guarantees according to the criticality of requests. DP4 instead requires extensive modifications that are only possible for the other architectural designs. And, for DP5, language choices are not applicable as the modification must be implemented by using the same programming language as for the orchestrator. Hence, the components keep providing all the features required by non-critical services while preserving critical services. This design is the simplest of the three, and the idea is similar to a Linux patched with *PREEMPT_RT* [50].

This design does not require implementing components from scratch, but existing ones must be patched, reducing the implementation effort. On the other hand, fully-featured components can be bloated and unable to provide the same guarantees as *ad hoc* components. For example, it is challenging to fully address indirect, implementation-related, and scaling delays by patching only a fully-featured component, as the sources of possible delays are embedded and scattered throughout the component's source code.

5 Ulysses

This section describes the implementation of a K8s-based *Patched Orchestrator* prototype, named *Ulysses*. We selected this design to show the benefits of our design principles with the lowest implementation effort, compared to ad hoc components written from scratch. Moreover, the patch keeps all K8S functionalities out of the box, fostering practitioners' quick uptake of the solution.

Our modification consists of only ≈800 LoC and involves the *kube-controller-manager* (i.e., Controller) and the *kubelet*. The former prioritizes requests concerning critical services to get to the worker node sooner, and the latter reduces the interference of concurrent pod spawning. The current *patched orchestrator* implementation adheres to design principles DP1 and DP3. We did not consider DP2 in the implemented solution to avoid modifications to the etcd datastore. Likewise, DP4 and DP5 require extensive modifications that are only possible for the other architectural designs. For example, since we patched native components, those are still written in Go, preventing the use of sound real-time scheduling. However, we will show in the evaluation section that DP1 and DP3 are enough to have *soft* real-time characteristics.

5.1 Multi-priority Queues

The queues in the Controller and the Kubelet are turned into multi-priority queues, according to DP3. The queues have 3 priority levels. Requests related to resource instances that do not specify any priority level have the lowest level. The priority level of a resource instance can either be defined in a *manifest* file or inherited from related resource instances. For example, we define the *ReplicaSet* (which ensures a desired number of *Pod* replicas running) priority as the highest priority of its *Pods*. Furthermore, the Apiserver is configured not to delay high-priority requests due to weighted fair queuing through an *exempt flow-schema* (see Apiserver configuration in Section 3.3.2) on a dedicated *namespace*.

5.2 Synchronous Queue Management

The queues in both Controller and Kubelet are modified to be managed synchronously, following DP1.

— **Controller.** We turn the control loop threads into periodic tasks, resulting in a *periodic Controller* with synchronous handling of low-priority requests. Conversely, additional asynchronous control loop threads are dedicated to handling high-priority events. Although Golang (the K8s programming language) does not support sound real-time scheduling, we implemented a solution similar to polling servers [51] through *channels* and *tickers*.

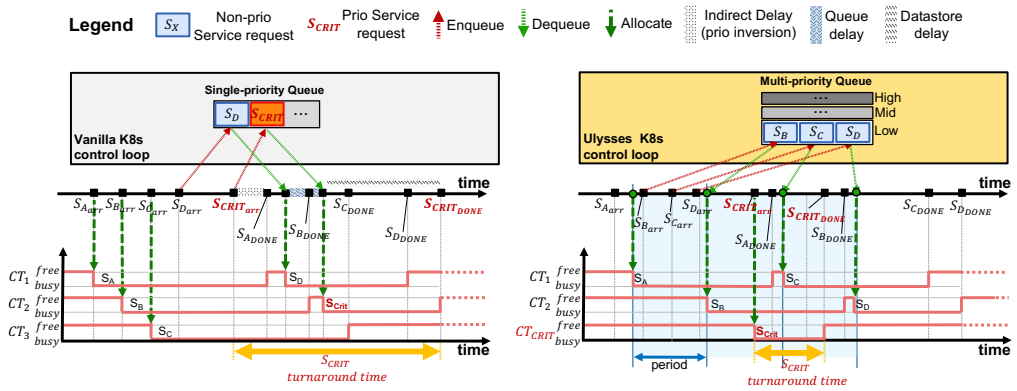


Fig. 11. Ulysses-based K8s (on the right) compared to K8s Vanilla (on the left). In K8s vanilla, S_A, S_B, and S_C are immediately taken in charge by the control threads (respectively $CT1$, $CT2$, and $CT3$). When S_D arrives, it is enqueued because there are no free control threads, and the same happens to S_{Crit}. Assuming that $CT1$, which processes S_A, is the first CT to finish, it immediately starts processing S_D. When $CT2$ is freed, it starts processing S_{Crit}. Hence, S_{Crit} is delayed by S_A, S_B, and S_C because they are served as soon as they arrive and there is no free CT left. Moreover, S_{Crit} is subject to a queueing delay due to S_D, and a datastore delay due to orchestration actions. In Ulysses-based K8s, allocation to control threads is a synchronous action taken periodically, i.e., every time the periodic timer fires. When S_B, S_C, and S_D arrive, they are enqueued (according to their priority, in the example, all of them have a low priority), waiting for the next period to be served. When S_{Crit} arrives, it is immediately served by a dedicated CT. Since the service of S_A and S_B is controlled in time, S_{Crit} undergoes minimal delay. Therefore, the overall turnaround time of S_{Crit} is shorter.

Figure 11 shows the behavior of the implemented solution. The control loop threads (CTs) have a phase $\phi = k * T/N$, where k is the thread index, T is the period, and N is the number of active threads. Hence, a request is processed every T/N. In this perspective, the Controller periodicity can be seen as a constant, fine-grained, and controllable throttling to respect the rate-limiting threshold. Indeed, periods can be controlled at resource type granularity, improving the control over the system compared to a unique rate-limiting for the entire Controller. This also helps to mitigate scaling delays, as critical requests are served within a fixed number of periods, are are not starved by low priority requests (DP4). Clearly, periods can be tuned to have a throughput similar to the vanilla component, trading off between interference due to short periods and performance degradation due to long periods.

— **Kubelet.** We modify the asynchronous request handling (i.e., as soon as received) of the Kubelet for spawning, deletion, and update requests. Our SLO-aware Kubelet waits for a given amount of time between the handling of two low-priority requests, delaying the handling of the latter. This limits the interference (and delays) that affect high-priority requests, which are never delayed.

We implement two different policies to determine the waiting time between low-priority requests: *(i)* a *fixed*, in which the waiting time between requests is constant; *(ii)* an *exponential decay*, in which the waiting time between requests varies accordingly to a decreasing geometric progression, and it is reset after a period of inactivity (see Figure 12). The fixed policy guarantees lower interference at the cost of lower throughput. The exponential decay guarantees low interference only to the first requests of a burst, allegedly more critical than the later ones because of control plane prioritization.

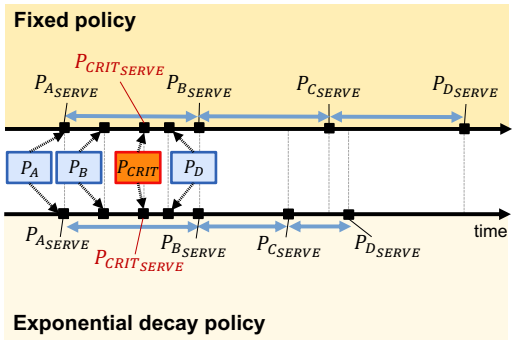


Fig. 12. The proposed timing policies (i.e., fixed vs. exponential decay) for the Kubelet. P_X is a pod request arriving at the Kubelet. $P_{X_{SERVE}}$ is a pod request being served.

Table 1. Experimental Configurations

Config. No.	Kubelet	Apiserver	Controller
1	Vanilla	Vanilla	Vanilla
2	Patched	Vanilla	Vanilla
3	Patched	Multi-priority	Vanilla
4	Patched	Multi-priority	Multi-priority Asynchronous
5	Patched	Multi-priority	Multi-priority Periodic

5.3 Configurability

The listed modifications reduce delays in high-priority services but can be detrimental to orchestration delays and throughput in low-priority ones. Since K8s allows dynamic redeployment of single control plane components, we implemented a set of components so that the Ulysses modifications can be selectively enabled. For example, the vanilla Controller is used until a critical service (requiring low orchestration times) is admitted to the system; then, the multi-priority (and optionally periodic) Controller is enabled.

6 Ulysses Evaluation

We evaluate the improvement given by Ulysses, by analyzing the contribution of each proposed modification (Section 6.1) and using Ulysses for a cloud-native 5G scenario (Section 6.2).

6.1 Ablation Study

We repeated the experiments performed in Section 3.3 to analyze the impact of each implemented modification.

6.1.1 Experiment Parameters. The parameters are the same as described in Section 3.3. Rate-limiting is enabled, as usual in production environments. We repeat and compare the experiment on both cloud and edge clusters for the configurations reported in Table 1. *"Kubelet patched"* means both multi-priority and synchronous. *"Multi-priority Apiserver"* means Apiserver configured according to Section 5.2. *"Multi-priority asynchronous Controller"* means a vanilla Controller with modifications to make it multi-priority.

We configured the Kubelet exponential decay policy in the following way: *(i)* we set 700*ms* as initial waiting time, reduced by 50% at each request, for worker nodes in the cloud cluster; (ii) we set 1.5*s* as initial waiting time, reduced by 40% at each request, for worker node of the edge cluster.

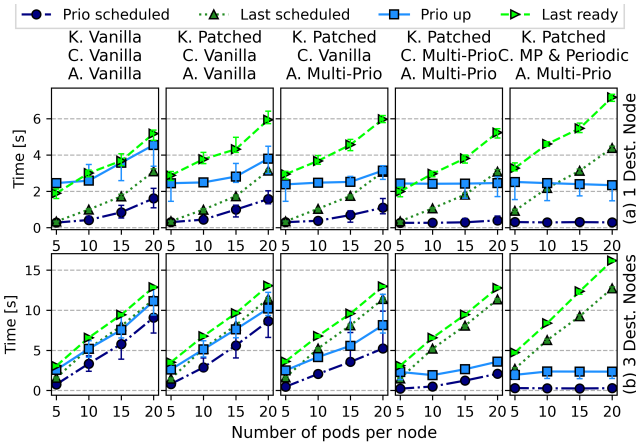


Fig. 13. Comparison of the configurations in the cloud cluster with throttling enabled. *"K."* is Kubelet, *"C."* is Controller, *"A."* is Apiserver, *"MP"* is multi-priority.

The initial waiting is reset after 5*s*. All these parameters are set according to the times measured in Section 3.3. The control loop threads were configured with the following parameters: 5 threads (K8s default) for each control loop (i.e., for each resource type), one of which is asynchronous and dedicated to high-priority requests. We configured *Nodes*, *Endpoints*, and *EndpointSlices* controller to serve a request each 50*ms*; and *ReplicaSet*, *Deployment*, and *Services* controllers to serve a request each 100*ms*. These period parameters aimed at differentiating the number of resource instances managed by different controllers and having a Controller maximum request limit similar to K8s default one (i.e., 20 per second).

6.1.2 *Results.*

Orchestration Times. Figure 13 shows the results for the cloud cluster for 1 (a) and 3 *DN* (b). The modified Kubelet improves the time to spawn the high-priority pod on the worker node: the pod starts serving the client $\approx$ 2.1*s* after the *"Prio scheduled"* event, despite the increasing number of pods. However, the Kubelet cannot avoid the control plane delays (see 3 *DN*).

In the 3 *DN* scenario, while the multi-priority Apiserver (column 3 in the figure) gives some improvement, adding a multi-priority Controller (column 4) significantly improves the time to schedule the high-priority pod. Nonetheless, the *"Prio scheduled"* event still shows an increasing slope when the control plane hits the rate limits and triggers throttling. The multi-priority periodic Controller (column 5) solves this issue by accurately controlling request rates. It assures a constant orchestration time of the high-priority pod in all tested conditions, with a reduction up to 78% in the scenario with 3 *DN*, 20 pods per node. The *"Last ready"* event timing is roughly the same in all cases, except for the periodic Controller, which presents a slowdown of up to $\approx$3*s* (median 14.5%) in the worst case (3 *DN*, 20 pods per node).

For this latter configuration, we repeat the experiment with multiple critical pods, with 1 and 3 *DN* and 20 pods per node. The results in Figure 14 show only a slight increase, while the slowdown for other pods is unchanged.

Control Loop Runtimes. We compared the control loop runtimes of the periodic multi-priority Controller and the vanilla Controller in the cloud cluster. Figure 15 shows that the modified Controller reduces the interferences among control loop threads and prevents throttling, resulting in a 96% reduction of the *p99*. The standard deviation is lower than 1 control loop thread (in Figure 7).

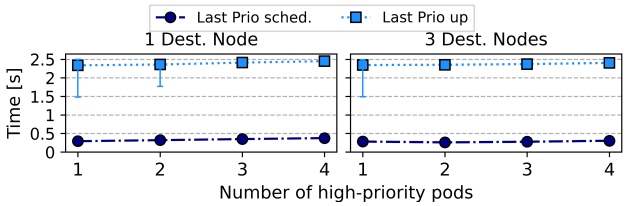


Fig. 14. Orchestration times of multiple critical pods, under a load of 20 low-priority pods per *DN*, for 1 and 3 *DN*.

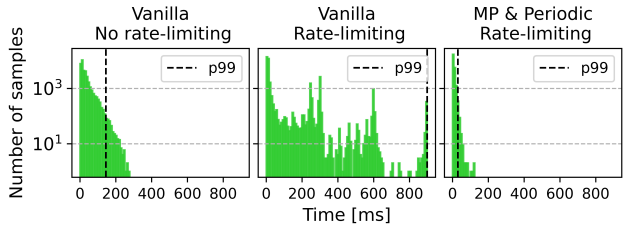


Fig. 15. Control loop runtimes (5 control loop threads).

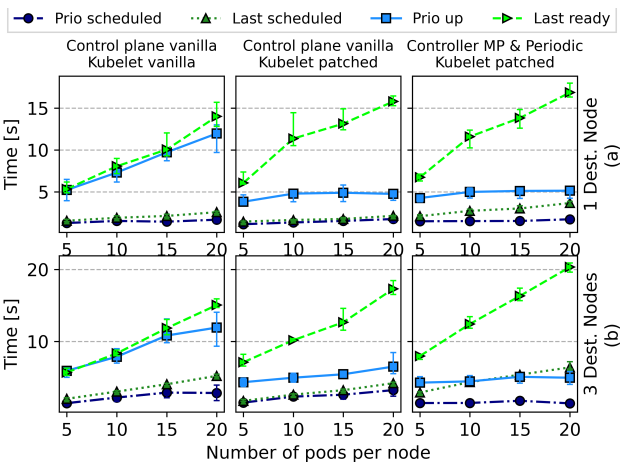


Fig. 16. Comparison of the configurations on the edge cluster without control plane rate-limiting (throttling). The Raspberry Pi is the destination node for the priority service.

This witnesses the negligible temporal overhead of our modifications, while the median value of memory usage percentage increase due to Ulysses is +3.2%.

Effect of Patched Kubelet. We disabled the rate-limiting on the control plane to quantify only the effects of the modifications on the Kubelet. We consider both vanilla and multi-priority periodic Controllers. Figure 16 shows the results for the edge cluster. Only patching the Kubelet (column 2) provides noticeable improvements. The *"Prio up"* time is almost flat and ≈40% less than the vanilla case. A multi-priority periodic Controller introduces further benefits at the cost of slight performance degradation.

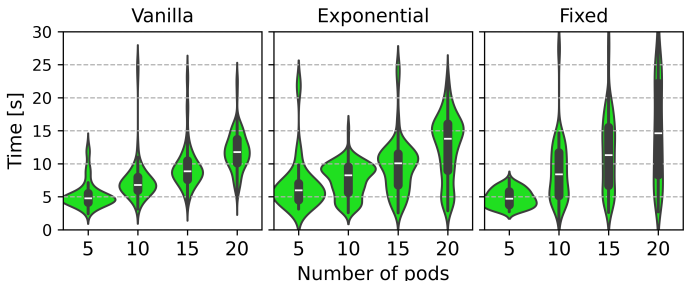


Fig. 17. Pod ready times on the Raspberry Pi with the different Kubelet waiting time configurations.

Kubelet Waiting Time Policies. We measured the pod ready times when setting vanilla, exponential decay, and fixed (configured with constant 1.2*s* waiting time) waiting time policies on the Raspberry Pi in the edge cluster.

Figure 17 shows the results. The distributions for *Exponential* and *Fixed* policies show that more pods become ready earlier than *Vanilla*. The *Fixed* configuration is more conservative, presenting an almost uniform distribution and higher maxima (i.e., the last pod ready). The mode of the *Exponential* policy is similar to *Vanilla* in the upper part of plots because of the increasingly short waiting times used for the last pods that arrived at the worker node. The exponential decay policy is a good tradeoff between throughput and interference.

6.2 Cloud Native 5G

5G is fundamental in connecting IoT devices providing support for ultrareliable and low-latency communications [52]. We aim at proving the benefits of Ulysses in a realistic scenario with multiple services and differentiated SLOs. We join the edge and cloud clusters, obtaining a cluster with 8 VMs, 1 workstation, and the Raspberry Pi. The K8s control plane node is a cloud VM as before. We used *Open5GS* as a softwarized 5G core network [10]. The 5G control plane is deployed over three cloud VMs, while the ***User Plane Function (UPF)*** of the data plane (i.e., the function that forwards the traffic of the users) has high priority and is deployed on the Raspberry Pi on the edge. We used UERANSIM [53] (deployed in an edge VM) as a commonly used 5G simulator for user equipment and radio access networks.

Train-ticket [54] is used as a realistic microservice-based application, with less stringent SLOs than the 5G network functions. Train-ticket microservices are deployed across 5 cluster nodes (nodes used for 5G control and data plane are excluded), with K8s being free to decide the scheduling.

Two edge nodes, including the one where the UPF is deployed, are tainted to simulate node failures (similarly to what is performed in Section 3.2). The UPF pod must be prioritized during the failover because it may provide connectivity to some **User Equipment** (**UE**) requiring low latency and high availability. Similar considerations would hold if the experiment analyzed the scaling time of the UPF of other 5G functions. We repeated the experiment 30 times for statistical purposes for each configuration, i.e., K8s vanilla and Ulysses periodic and multi-priority.

Figure 18 shows the timing of the main events, including the 5G connection recovery. The time to schedule the UPF pod is drastically reduced (about −92%), as well as the UPF ready time (about −68%). Although the connection recovery times ("Connection Restored" in Figure 18) are variable due to the configuration of a brand-new connection, the median time is approximately the same as the median of UPF ready times. This matters in scenarios where the mean time to recovery for UE connections must be as low as possible [55].

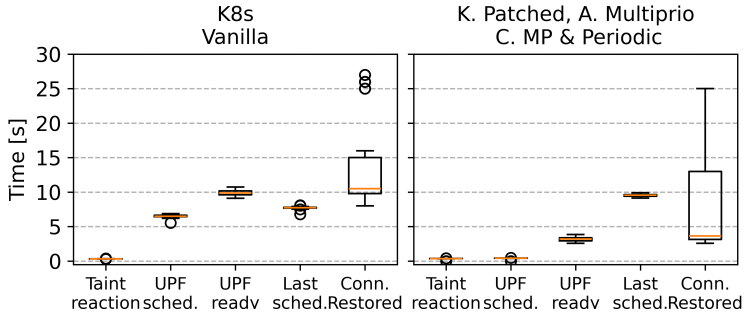


Fig. 18. Main events for the 5G data plane failover example.

7 Discussion

Targeted Modifications and Performance Improvements. Although Ulysses modifications consist of a limited number of lines of code, targeting core timing aspects of K8s led to significant improvements in different configurations. The choice of configuration depends on application scenarios and tolerated latencies: our experiments showed that using SLO-aware components keeps the orchestration time of a high-priority service stable (≈2 seconds in our settings, including pod creation time), despite the increasing system load.

The Critical Role of Orchestration SLOs. Such results, although useful and appreciable in themselves, shed a more general light on the importance of orchestration times: defining precise orchestration SLOs that cover all system areas is compelling as real-time workloads are being migrated to edge/cloud environments. Although K8s developers introduced some orchestration SLOs, those are still limited, highlighting how current cloud systems are more focused on throughput than latency (as also backed by [15]). This issue is not peculiar to K8s: as mentioned in Section 2.2, current platforms either rely on K8s or adopt a very similar architecture, based on a desired-state model, chained controllers, and consistent datastores.

Implications for Real-Time and FaaS Workloads. We stressed the importance of orchestration SLOs in all system areas and provided a fine-grained analysis of *why* and *when* such orchestration SLOs can be violated. This becomes utterly important when per-request scheduling and admission are in place for real-time cloud services/functions [22, 23]: an orchestrator with a known timing behavior would add another dimension to job/request scheduling models. Similarly, soft real-time services with no request admission policy would benefit from orchestration SLOs, taming latency spikes under transient conditions without pessimistic system capacity planning, hence saving costs. Orchestration SLOs are most relevant with the spread of the FaaS paradigm, which presents on-demand scaling based on queued requests rather than resource utilization, and possibly *scale-to-zero* to save costs. In this perspective, we showed that neglecting orchestration times possibly induces delays that overshadow downstream optimizations, e.g., boot times optimizations of hundreds of milliseconds.

Design Choices and Practical Tradeoffs. Finally, further improvements are possible through the other architectural designs. The prototype in this article is based on the "Patched orchestrator" design. While "Co-orchestrator" and "Additional Components" designs could guarantee better performances, their implementation effort is drastically higher. In fact, the core codebase of K8s Controller and Apiserver is respectively ≈113 and 153 thousand lines of Golang. Similarly, although more complex timing policies in both the control plane and worker nodes can lead to further improvements, our goal is not to achieve minimum orchestration times through careful parameter tuning, because timing figures strictly depend on the setup. Conversely, we wanted to show i) the

importance of orchestration times; ii) the need for SLO-aware orchestration; and iii) the benefits of applying the delineated SLO-aware design principles.

8 Threats to Validity

Internal Validity. The measure of absolute times can be affected by the setup (recall scaling delays in Section 3.3.3), however, experiments have been repeated multiple times and replicated on two different clusters, showing how the highlighted trends on orchestration times hold on both of them. Experiments have been conceived after thorough inspections of the source code and event logs. To support replicability, the code of the proposal and experiments is publicly available [27].

The representativeness of the workload and experimental parameters may be another concern regarding the internal validity of the results. In our case, choosing the application run within pods is not crucial, as our study only focuses on orchestration times. We preferred light services (e.g., Nginx web server) to exclude application startup time from experiments. However, we also analyzed orchestration times and the benefits of our solution in the context of a realistic 5G scenario.

During our experiments, we left several default parameters of the orchestrator unmodified. Although some of them definitely impact the orchestration times, like the number of worker nodes evaluated by the Scheduler, they would have little impact on our specific setup, which has a limited number of nodes. Hence, we focused only on parameters that we found to affect the results drastically.

The number of worker nodes considered in the experimentation can also threaten the validity of our results. However, in K8s, the sequence of events triggered in the control plane does not depend on the number of worker nodes but only on the number of pods to be created, which was one of the variable parameters examined in our experiments. Due to our setup limitation, we did not analyze the variability of the time required by the scheduler to schedule a pod, which is affected by the number of worker nodes. However, the scheduler can be tuned to limit the number of nodes to consider,[4] and the scheduling time has already been analyzed by several articles in the literature [56].

Regarding the high number of pods considered, several real-world stories showed how pessimistic cases, such as the ones assumed in the experiments, actually happen in production [24]. We do not consider time and accuracy of failure detection, by emulating a worker node failure through a *taint*. However, our focus is on the behavior of the orchestrator upon worker node failure detection, which is complementary to monitoring-related issues. Moreover, monitoring techniques available in the literature can shorten the detection time.

External Validity. While most experiments have been conducted only on K8s, it is a *de facto* standard for container orchestration, multiple orchestrators reuse its code. Moreover, we showed that Docker Swarm presents common issues while offering fewer features. Regarding the hardware setup, we adopted two heterogeneous clusters (cloud and edge scenarios) including different CPU architectures and technologies.

9 Related Work

A wide literature concerning orchestration aims at improving SLOs. A lot of efforts have especially been devoted since the spread of the serverless paradigm. This related work section divides the work related to control plane enhancements and those related to worker node optimizations.

9.1 Control Plane

Failover and migration. Reducing repair times through replication [57, 58], state migration [59–61], and placement policies [62–64] are the main solutions to increase service availability.

[4]More information at https://kubernetes.io/docs/concepts/scheduling-eviction/scheduler-perf-tuning/

In [57], the authors propose a custom K8s controller to manage services standby replicas, switching the routing of stateful services upon failure. In [58], K8s is modified to offer automatic and transparent state machine replication to replicated services. In [59], the authors extend K8s to perform state migration of industrial controllers with timing requirements ($\sim 10ms$) from a pod to its updated hot standby replica. In [60], the failure-repair of container replication configurations is analyzed via a state-space and a non-state-space model.

Those works always assume having a stand-by replica. Our SLO-aware design complements them: their solution could benefit from ours. Indeed, replicating is not a silver bullet: control plane management can be delayed (see Figure 5). The listed works extend K8s but leave untouched the core of the control plane. Compared to them, we directly act on the control plane internals to further reduce delays.

In [63] failover times are compared between different K8s-based industrial automation architectures. In [62], spare replicas are allocated accounting for required response times.

Creation and Scaling. *Bhardwaj et al.* [65] proposed a performance-aware resource allocation framework for K8s, which uses online feedback and lightweight learning techniques to meet custom SLOs like latency or throughput, adapting resource assignments dynamically based on observed performance. *Autopilot* [66] is Google's production-grade workload autoscaler, which automates both vertical and horizontal scaling using historical workload data and predictive modeling. Autopilot reduces resource slack and improves reliability by preventing over- and under-provisioning, particularly in large-scale, homogeneous cloud settings. Our proposal complements these studies since neither of these systems tackles the timing and prioritization of the orchestration process, which can become a bottleneck under load. The work in [20] shows that pre-creating containers' network and directly contacting the container manager on the worker nodes in K8s (i.e., not using orchestration functionalities) enables meeting deadline-driven SLOs. Indeed, they show that "non-deterministic delays appear when multiple containers are created in parallel". Our work solves this precise issue.

Scheduling. Scheduling policies influence both dependability (e.g., recovery times) and SLOs. We refer to [56, 67] for a complete portrait of the works that tackle the scheduling problem in orchestration, while we only focus here on works close to our article. In [64], the authors used an integer linear programming model to decide the VM placement to minimize the Mean Time To Repair. The work in [68] claims that orchestrators must place a pod on the node with the maximum number of the task's dependencies stored locally to reduce pulling times. Our effort is orthogonal to those works since we deal with the variability of the orchestration times in the control plane in general, while they deal with pod creation times.

Other works accounted network latencies in the scheduling problem [63, 69, 70]. The work in [69] presented an analytical model for real-time FaaS that considers the end-to-end latency of real-time request/response pairs. The authors of [63] proposed a greedy border allocation algorithm to minimize communication costs in fog/edge environments for industrial automation contexts. The work in [70] extended the K8s scheduler to be aware of the network latencies, which are periodically probed. However, these works focus on choosing the best node to meet SLOs while the services are running, but neglect orchestration times.

Benchmarking Orchestrators. The works in [29, 33] analyze and compare the orchestration times of several Kubernetes distributions. The authors of [29] also propose a systematic benchmarking method. They analyze the pod startup times with an increasing number of pods. Nonetheless, in their studies, those metrics are considered to measure the throughput of the orchestrator rather than focusing on the latency itself. Remarkably, orchestration times are neglected even in works

dealing with orchestration for real-time environments, like the ones in [31, 71–74], which only focus on timeliness of the applications on the worker nodes.

9.2 Worker Nodes

Startup times. Provisioning times for scaling and deployment can be improved by borrowing the solutions investigated in several works concerning **Function as a Service** (**FaaS**) cold starts. Those solutions can be divided [75] into application-based [76, 77], checkpoint-based [78], prediction-based [79–81], and cache-based [21]. Those techniques involve pre-warming containers, scheduling and reusing containers, using the pool of warm containers, and keeping containers alive. They aim at reducing as much as possible the activities to be performed to start a new container, including downloading it, setting up a sandbox, initializing a language runtime and the application code. Nevertheless, they only focus on activities related to worker nodes, neglecting variable orchestration times. Our solution would benefit those, which still need an SLO-aware orchestration to prioritize critical functions under heavy load conditions.

The authors of [13] showed that, although the time to spawn a new pod is not in the critical path, it determines the end-to-end service response times during the scaling process. The authors also identified (as reported in the introduction), the "declarative tax" (i.e., the time the orchestration system takes to translate the declarative steady state in imperative commands to reconcile the cluster) as a possible bottleneck in the era of serverless, which deals with orchestration units with increasingly small startup times.

10 Conclusion

In this article, we performed a timing analysis of orchestration times, which revealed that there is currently no effective means of prioritizing applications in the presence of concurrent events. The analysis also revealed that asynchronous management of events can lead to interference and consequently delay.

We proposed possible architectures to reduce orchestration times of critical services to meet their SLOs, and modified Kubernetes to implement a prototype. The capability of meeting strict SLOs is becoming relevant in IoT scenarios encompassing real-time (or low-latency) services and a wide number of scattered and heterogeneous nodes.

We showed that a synchronous and multi-priority design of orchestration components can guarantee reduced and constant orchestration times for critical services when several multiple events have to be handled. Respecting the orchestration SLOs (i.e., SLOs on orchestration times) helps to meet the SLOs of deployed applications, increasing the overall system reliability without needing overly pessimistic system capacity planning.

11 Acknowledgments

This manuscript reflects only the authors' views and opinions, neither the European Union nor the European Commission can be considered responsible for them.

References

[1] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the 10th European Conference on Computer Systems*. Association for Computing Machinery, 17 pages. DOI: http://dx.doi.org/10.1145/2741948.2741964

[2] Asif Khan. 2017. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing* 4, 5 (2017), 42–48. DOI: http://dx.doi.org/10.1109/MCC.2017.4250933

[3] Maria A. Rodriguez and Rajkumar Buyya. 2019. Container-based cluster orchestration systems: A taxonomy and future directions. *Wiley Software: Practice and Experience* 49, 5 (2019), 698–719. DOI: http://dx.doi.org/10.1002/spe.2660

[4] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *ACM Queue* 14, 1 (2016), 70–93. DOI: http://dx.doi.org/10.1145/2898442.2898444

[5] Thiago Pereira Da Silva, Thais Batista, Frederico Lopes, Aluizio Rocha Neto, Flávia C. Delicato, Paulo F. Pires, and Atslands R. Da Rocha. 2022. Fog computing platforms for smart city applications: A survey. *ACM Transactions on Internet Technology* 22, 4 (2022), 1–32.

[6] Wenzhao Zhang, Yi Gao, and Wei Dong. 2023. Providing realtime support for containerized edge services. *ACM Transactions on Internet Technology* 23, 4 (2023), 1–25.

[7] Bin Qian, Jie Su, Zhenyu Wen, Devki Nandan Jha, Yinhao Li, Yu Guan, Deepak Puthal, Philip James, Renyu Yang, Albert Y. Zomaya, et al. 2020. Orchestrating the development lifecycle of machine learning-based IoT applications: A taxonomy and survey. *ACM Computing Surveys* 53, 4 (2020), 1–47.

[8] Linux Foundation. Nephio: Cloud Native Network Automation. Retrieved January 15, 2026 from https://nephio.org/about/

[9] Andrew E. Ferguson, Jon Larrea, and Mahesh K. Marina. 2023. CoreKube: An Efficient, Autoscaling and Resilient Mobile Core System. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*. Association for Computing Machinery. DOI: http://dx.doi.org/10.1145/3570361.3592522

[10] Open5Gs developers. Open5Gs. Retrieved January 15, 2026 from https://github.com/open5gs/open5gs

[11] Francesco Linsalata, Eugenio Moro, Maurizio Magarini, Umberto Spagnolini, and Antonio Capone. 2024. Open RAN-empowered V2X architecture: Challenges, opportunities, and research directions. In *Proceedings of the 2024 IEEE Vehicular Networking Conference*. IEEE, 113–116. DOI: http://dx.doi.org/10.1109/VNC61989.2024.10576004

[12] Andrea Lacava, Leonardo Bonati, Niloofar Mohamadi, Rajeev Gangula, Florian Kaltenberger, Pedram Johari, Salvatore D'Oro, Francesca Cuomo, Michele Polese, and Tommaso Melodia. 2025. dApps: Enabling real-time AI-based open RAN control. *Elsevier Computer Networks* 269 (2025), 111342.

[13] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. 2023. The gap between serverless research and real-world systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 475–485.

[14] Vojdan Kjorveziroski and Sonja Filiposka. 2022. Kubernetes distributions for the edge: Serverless performance evaluation. *Springer The Journal of Supercomputing* 78, 11 (2022), 13728–13755.

[15] Haoran Qiu, Saurabh Jha, Subho S. Banerjee, Archit Patke, Chen Wang, Franke Hubertus, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2021. Is function-as-a-service a good fit for latency-critical services?. In *Proceedings of the 17th International Workshop on Serverless Computing*. Association for Computing Machinery, 1–8. DOI: http://dx.doi.org/10.1145/3493651.3493666

[16] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Darlow, Jianfeng Wang, Qiwen Deng, and Adam Barker. 2025. Serverless cold starts and where to find them. In *Proceedings of the 12th European Conference on Computer Systems*. Association for Computing Machinery, 938–953.

[17] Intel Corportation. 2020. Low Latency 5G UPF Using Priority Based 5G Packet Classification. Retrieved from https://networkbuilders.intel.com/docs/networkbuilders/low-latency-5g-upf-using-priority-based-5g-packet-classification.pdf. Access Date: 2025.

[18] Tobias Pfandzelter and David Bermbach. 2024. Komet: A serverless platform for low-earth orbit edge services. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. Association for Computing Machinery, 866–882.

[19] Kubernetes. Kubernetes scalability and performance SLOs. Retrieved January 15, 2026 from https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md

[20] Emad Heydari Beni, Eddy Truyen, Bert Lagaisse, Wouter Joosen, and Jordy Dieltjens. 2021. Reducing cold starts during elastic scaling of containers in kubernetes. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. Association for Computing Machinery, 60–68. DOI: http://dx.doi.org/10.1145/3412841.3441887

[21] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 335–350.

[22] Nasim Samimi, Mitra Nasri, Twan Basten, and Marc Geilen. 2024. Guaranteeing weakly-hard timing constraints of real-time server-based systems. In *Proceedings of the 2024 IEEE 29th International Conference on Emerging Technologies and Factory Automation*. IEEE, 1–8.

[23] Nasim Samimi, Mitra Nasri, Twan Basten, and Marc Geilen. 2025. Online admission test for real-time tasks with arrival curves for server platforms. In *Proceedings of the 32nd International Conference on Real-Time Networks and Systems*. Association for Computing Machinery, New York, NY, USA, 266–277. DOI: http://dx.doi.org/10.1145/3696355.3696359

[24] Marco Barletta, Marcello Cinque, Catello Di Martino, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2024. Mutiny! How does Kubernetes fail, and what can we do about it?. In *Proceedings of the 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 1–14.

[25] Henning Jacobs. 2023. Kubernetes Failure Stories. Retrieved January 15, 2026 from https://k8s.af/. (2023).
[26] Rémy-Christophe Schermesser. 2019. How to kill the Algolia dashboard during Black Friday. Retrieved January 15, 2026 from https://www.youtube.com/watch?v=Fjyg7cxRZQs&list=PLuHdbqhRgWHJg9eOFCl5dgLvVjd_DFz8O&index=6. (2019).
[27] Marco Barletta and Luigi De Simone. 2024. Ulysses Repository. Retrieved from https://github.com/dessertlab/preempt-kubernetes. Accessed Date: 2025.
[28] 2020. The ideal versus the real: Revisiting the history of virtual machines and containers. *Computing Surveys* 53, 1, (2020), 31 pages.
[29] Martin Straesser, Jonas Mathiasch, André Bauer, and Samuel Kounev. 2023. A systematic approach for benchmarking of container orchestration frameworks. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 187–198. DOI: http://dx.doi.org/10.1145/3578244.358372
[30] Daniele Ottaviano, Marco Barletta, and Francesco Boccola. 2025. Zero-interference containers: A framework to orchestrate mixed-criticality applications. In *Proceedings of the 2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 627–636.
[31] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. 2024. Criticality-aware monitoring and orchestration for containerized industry 4.0 environments. *ACM Transactions on Embedded Computing Systems* 23, 1 (2024), 1–28 DOI: http://dx.doi.org/10.1145/3604567
[32] Cloud Native Computing Foundation. 2022. CNCF Annual Survey 2021. Retrieved January 15, 2026 from https://www.cncf.io/reports/cncf-annual-survey-2021/. (2022).
[33] Heiko Koziolek and Nafise Eskandani. 2023. Lightweight Kubernetes distributions: A performance comparison of MicroK8s, k3s, k0s, and Microshift. In *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*. 17–29. DOI: http://dx.doi.org/10.1145/3578244.3583737
[34] OpenFaas developers. OpenFaas homepage. Retrieved January 15, 2026 from https://www.openfaas.com/
[35] Nima Kaviani, Dmitriy Kalinin, and Michael Maximilien. 2019. Towards serverless as commodity: A case of Knative. In *Proceedings of the 5th International Workshop on Serverless Computing*. Association for Computing Machinery, 13–18.
[36] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. 2016. Cloud-native, event-based programming for mobile applications. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*. ACM, 287–288. DOI: http://dx.doi.org/10.1145/2897073.2897713
[37] Gabrielle Evan Farrel, Widhi Yahya, Achmad Basuki, Kasyful Amron, and Reza Andria Siregar. 2023. Scalable edge computing cluster using a set of Raspberry Pi: A framework. In *Proceedings of the 8th International Conference on Sustainable Information Engineering and Technology*. 287–296.
[38] Antônio José Alves Neto, José Aprígio Carneiro Neto, and Edward David Moreno. 2022. The development of a low-cost big data cluster using Apache Hadoop and Raspberry Pi. A complete guide. *Elsevier Computers and Electrical Engineering* 104 (2022), 108403.
[39] K8s developers. 2023. Taints and tolerations. Retrieved January 15, 2026 from https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
[40] Docker developers. 2023. Docker object labels. Retrieved January 15, 2026 from https://docs.docker.com/config/labels-custom-metadata/
[41] Martin Straesser, André Bauer, Robert Leppich, Nikolas Herbst, Kyle Chard, Ian Foster, and Samuel Kounev. 2023. An empirical study of container image configurations and their impact on start times. In *Proceedings of the 2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing*. IEEE, 94–105. DOI: http://dx.doi.org/10.1109/CCGrid57682.2023.00019
[42] Moonlight Work Enterprises, LLC. Outage post-mortem. Retrieved January 15, 2026 from https://updates.moonlightwork.com/outage-post-mortem-87370
[43] Kubernetes developers. Issue #18266. Retrieved January 15, 2026 from "https://github.com/kubernetes/kubernetes/issues/18266"
[44] Özalp Babaoğlu, Keith Marzullo, and Fred B Schneider. 1993. A formalization of priority inversion. *Springer Real-Time Systems* 5, 4 (1993), 285–303. DOI: http://dx.doi.org/10.1007/BF01088832
[45] Xudong Sun, Lalith Suresh, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lilia Tang, and Tianyin Xu. 2021. Reasoning about modern datacenter infrastructures using partial histories. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 213–220.
[46] Robert I. Davis and Alan Burns. 2005. Hierarchical fixed priority pre-emptive scheduling. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*. IEEE, 10 pp.–398. DOI: http://dx.doi.org/10.1109/RTSS.2005.25
[47] Sitaram Iyer and Peter Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*. Association for Computing Machinery, 117–130. DOI: http://dx.doi.org/10.1145/502059.502046

[48] Remo Andreoli, Tommaso Cucinotta, and Dino Pedreschi. 2021. RT-MongoDB: A NoSQL database with differentiated performance. In *Proceedings of the 11th International Conference on Cloud Computing and Services Science-CLOSER*. Science and Technology Publications (SciTePress), 77–86. DOI: http://dx.doi.org/10.5220/0010452400770086

[49] Giovanni Bartolomeo, Mehdi Yosofie, Simon Bäurle, Oliver Haluszczynski, Nitinder Mohan, and Jörg Ott. 2023. Oakestra: A lightweight hierarchical orchestration framework for edge computing. In *Proceedings of the 2023 USENIX Annual Technical Conference*. 215–231.

[50] The Linux Foundation. Technical details of the real-time preemption. Retrieved from https://wiki.linuxfoundation.org/realtime/documentation/technical_details/start. Access Date: 2025.

[51] Giorgio C. Buttazzo. 2011. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Science & Business Media. DOI: http://dx.doi.org/10.1007/978-1-4614-0676-1

[52] GSM Association. 2019. Internet of Things in the 5G era: Opportunities and Benefits for Enterprises and Consumers. Retrieved from https://www.gsma.com/solutions-and-impact/technologies/internet-of-things/wp-content/uploads/2019/11/201911-GSMA-IoT-Report-IoT-in-the-5G-Era.pdf. Access Date: 2025.

[53] UERANSIM developers. UERANSIM. Retrieved January 15, 2026 from https://github.com/aligungr/UERANSIM

[54] FudanSELab. 2024. Train Ticket. Retrieved January 15, 2026 from https://github.com/ovkulkarni/train-ticket

[55] ETSI. 2021. *5G; Management and Orchestration; 5G End to end Key Performance Indicators (KPI) (3GPP TS 28.554 version 16.7.0 Release 16)*. Technical Report ETSI TS 128 554. ETSI.

[56] Carmen Carrión. 2022. Kubernetes scheduling: Taxonomy, ongoing issues and challenges. *ACM Comput. Surv.* 55, 7, Article 138 (July 2023), 37 pages. https://doi.org/10.1145/3539606

[57] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. 2021. A Kubernetes controller for managing the availability of elastic microservice based stateful applications. *Journal of Systems and Software* 175 (2021), 110924. DOI: http://dx.doi.org/10.1016/j.jss.2021.110924

[58] Hylson Vescovi Netto, Aldelir Fernando Luiz, Miguel Correia, Luciana de Oliveira Rech, and Caio Pereira Oliveira. 2018. Koordinator: A service approach for replicating docker containers in kubernetes. In *Proceedings of the 2018 IEEE Symposium on Computers and Communications*. IEEE, 00058–00063. DOI: http://dx.doi.org/10.1109/ISCC.2018.8538452

[59] Heiko Koziolek, Andreas Burger, and Abdulla Puthan Peedikayil. 2024. Fast state transfer for updates and live migration of industrial controller runtimes in container orchestration systems. *Journal of Systems and Software* 211 (2024), 112004. DOI: http://dx.doi.org/10.1016/j.jss.2024.112004

[60] Stefano Sebastio, Rahul Ghosh, and Tridib Mukherjee. 2018. An availability analysis approach for deployment configurations of containers. *IEEE Transactions on Services Computing* 14, 1 (2018), 16–29. DOI: http://dx.doi.org/10.1109/TSC.2017.2788442

[61] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. 2018. Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing* 18, 9 (2018), 2020–2033. DOI: http://dx.doi.org/10.1109/TMC.2018.2871842

[62] László Toka. 2021. Ultra-reliable and low-latency computing in the edge with Kubernetes. *Springer Journal of Grid Computing* 19, 3 (2021), 31. DOI: http://dx.doi.org/10.1007/s10723-021-09573-z

[63] Raphael Eidenbenz, Yvonne-Anne Pignolet, and Alain Ryser. 2020. Latency-aware industrial fog application orchestration with Kubernetes. In *Proceedings of the 2020 5th International Conference on Fog and Mobile Edge Computing* . IEEE, 164–171. DOI: http://dx.doi.org/10.1109/FMEC49853.2020.9144934

[64] Manar Jammal, Ali Kanso, and Abdallah Shami. 2015. CHASE: Component high availability-aware scheduler in cloud computing environment. In *Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing*. IEEE, 477–484. DOI: http://dx.doi.org/10.1109/CLOUD.2015.70

[65] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro: Performance-Aware resource allocation for general objectives via online feedback. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation*. 623–643.

[66] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. 2020. Autopilot: workload autoscaling at Google. In *Proceedings of the 15th European Conference on Computer Systems*. Association for Computing Machinery, 1–16.

[67] Zeineb Rejiba and Javad Chamanara. 2022. Custom scheduling in Kubernetes: A survey on common problems and solution approaches. *ACM Computing Surveys* 55, 7 (2022), 1–37.

[68] Silvery Fu, Radhika Mittal, Lei Zhang, and Sylvia Ratnasamy. 2020. Fast and efficient container startup at the edge via dependency scheduling. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Edge Computing*.

[69] M. Szalay, P. Mátray, and L. Toka. 2022. Real-time FaaS: Towards a latency bounded serverless cloud. In *IEEE Transactions on Cloud Computing* 11, 2 (2022), 1636–1650. DOI: 10.1109/TCC.2022.3151469

[70] Angelo Marchese and Orazio Tomarchio. 2024. Network SLO-Aware container orchestration on Kubernetes clusters. In *Proceedings of the International Conference on Service-Oriented Computing*. Springer, 96–104.

[71] Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. 2022. Rt-kubernetes: Containerized real-time cloud computing. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. 36–39. DOI: http://dx.doi.org/10.1145/3477314.3507216

[72] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. 2024. Hierarchical resource orchestration framework for real-time containers. *ACM Transactions on Embedded Computing Systems* 23, 1 (2024), 1–24. DOI: http://dx.doi.org/10.1145/359285

[73] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. 2021. Strong temporal isolation among containers in OpenStack for NFV services. *IEEE Transactions on Cloud Computing* 11, 1 (2021), 763–778. DOI: http://dx.doi.org/10.1109/TCC.2021.3116183

[74] Francesco Lumpp, Franco Fummi, Hiren D. Patel, and Nicola Bombieri. 2024. Enabling Kubernetes orchestration of mixed-criticality software for autonomous mobile robots. *IEEE Transactions on Robotics* 40 (2024), 540–553. DOI: http://dx.doi.org/10.1109/TRO.2023.3334642

[75] Ana Ebrahimi, Mostafa Ghobaei-Arani, and Hadi Saboohi. 2024. Cold start latency mitigation mechanisms in serverless computing: Taxonomy, review, and future directions. *Elsevier Journal of Systems Architecture* 151 (2024), 103115. DOI: http://dx.doi.org/10.1016/j.sysarc.2024.103115

[76] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 467–481. DOI: http://dx.doi.org/10.1145/3373376.337851

[77] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, Ding Li, Junkai Chen, Yi Liu, Haoyu Wang, and Xin Jin. 2023. Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–29. DOI: http://dx.doi.org/10.1145/3585007

[78] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572. DOI: http://dx.doi.org/10.1145/3445814.3446714

[79] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. Icebreaker: Warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, 753–767. DOI: http://dx.doi.org/10.1145/3503222.3507750

[80] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. 2019. Agile cold starts for scalable serverless. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Cloud Computing* .

[81] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. 2020. Using application knowledge to reduce cold starts in FaaS services. In *Proceedings of the 35th annual ACM Symposium on Applied Computing*. Association for Computing Machinery, 134–143. DOI: http://dx.doi.org/10.1145/3341105.3373909

Received 13 February 2025; revised 15 July 2025; accepted 13 August 2025