Fixed-Priority Preemptive Orchestration for Serverless-based Services

Marcello Cinque, Luigi De Simone, Raffaele Della Corte, Stefano Toscano
‡*Università degli Studi di Napoli Federico II*
{macinque, luigi.desimone, raffaele.dellacorte2}@unina.it

***Abstract*—In this paper, we design and implement a fixed-priority and fully preemptable controller for Kubernetes. The aim of this controller is to manage mixed criticality services in a real-time fashion, giving high priority to critical applications to speed-up their container placement operation. Such improvements are necessary in current serverless computing scenarios where there is a need to scale applications dynamically and with the lowest possible creation latency for critical workloads.**

***Index Terms*—Edge-Cloud, Orchestration, Containers, Mixed-Criticality, Kubernetes**

I. Introduction

Recently, containers and orchestration technologies have been increasingly adopted in cloud computing [1], [2] as well as in the Industrial-Internet-of-Things (IIoT) [3] and edge computing infrastructures [4].

In a typical scenario, a cloud service provider would make its application available to end-users through platforms that allow its deployment by adopting container technologies. An important aspect of the deployment phase is the static provisioning, an activity aimed at establishing how many instances of an application are needed to cope with the expected load. If the expected load is lower than the actual one, an under-provisioning problem might occur, with more instances needed to serve all incoming requests. The opposite miscalculation might lead to an over-provisioning situation where too many instances are running, leading to a useless resource consumption and increased resource usage cost.

Serverless platforms automate this process by monitoring applications and their actual workloads to dynamically establish the correct amount of instances, scaling them up or down accordingly [2], [5]. The *time-to-scale* is a major concern in these scenarios since instances must be ready as soon as possible, as they are activated when the load is increasing. This challenge, known in literature as the cold-start problem, has been addressed by several researchers. Some focus on improving the container creation process, trying to reduce the cold-start duration, while others focus on reducing their occurrences by pre-warming instances or keeping them alive to cope with expected requests. These solutions bring with them the overhead of the mathematical models needed to predict the behavior of the incoming load [2], [6].

We believe that these solutions, as good as they can be, will never bring sufficient improvements if the underlying container orchestration technology is not able to keep up with stringent timing requirements.

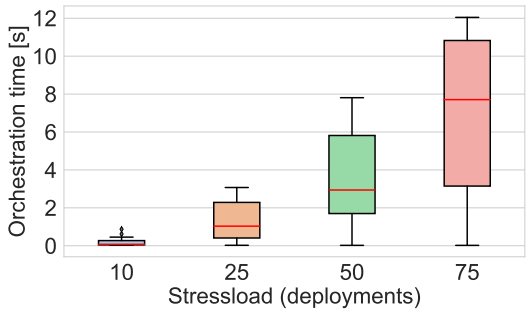

Fig. 1: Time for vanilla Kubernetes to start a high-priority container by creating a ReplicaSet, while applying an increasing number of interfering requests (IR) to the orchestrator.

In industrial scenarios, services may have different criticalities and requirements and must be seamlessly orchestrated across the same computing infrastructure. Therefore, orchestrators must prioritize the management of critical containers over non-critical ones in order to maximize the probability of meeting the service level objectives (SLOs) of critical services [7]–[9].

However, current orchestrators cannot preemptively prioritize some services. For example, Kubernetes, which is the standard *de facto* among orchestrators, is not capable of distinguishing between high- or low-priority containers while handling concurrent events. Fig. 1 concretely shows this problem: the time to place and start a container can increase up to tens of seconds regardless of the container priority when the orchestrator is busy handling an increasing number of interfering (low-priority) requests and related events.

In this work, a novel solution to these problems is presented through the implementation of a fully preemptable custom controller for the Kubernetes platform that allows mixed-criticality orchestration. We implemented our solution relying on Rust to have a fixed-priortiy preemptive scheduling of controller threads. The use of Rust enables full control over the scheduling policy and parameters of the kernel-level threads (KLTs), allowing us to use kernel real-time scheduling policies. In this way, we overcame the limitation of Golang (i.e., the programming language of the other K8s components),

which allows limited control over the scheduling parameters due to its user-level threading model (ULT).

Our experiments show, in our setup, a reduction in the time spent in the control plane when deploying a pod up to 99% ($28ms$ vs. $7140ms$), also benefiting from low variances.

The implemented solution was also integrated in the Knative serverless environment to show which benefits are already addressable from an end-user perspective and which still require further optimization.

The remainder of the paper is organized as follows. Section II provides a background on K8s and Knative. Section III presents the design of the custom solution and the Knative patch required to integrate it with the new workflow. Then, Section IV discusses the experiments conducted to prove the validity of the said solution, while Section V provides a benchmark to test the new design in a serverless environment. Finally, Section VIII concludes the paper.

II. Background

For the purpose of this research, two platforms were adopted: Kubernetes [10] as a container orchestrator and Knative [11] as a serverless platform.

A. Kubernetes

Kubernetes is a container orchestration platform. Its architecture (see Figure 2 revolves around the concept of a cluster, which is a collection of all available computational resources. It can be divided into two main logical units: the control plane and the worker nodes. The control plane consists of several components whose main duty is to manage workloads that will be scheduled on the worker nodes.

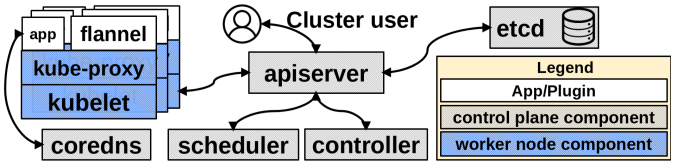

Fig. 2: K8s architecture.

The apiserver is the access point to the cluster. It allows for managing the current and desired state of all Kubernetes objects. It can be used by both external users and the other internal components to communicate with each other.

The kube manger is a collection of controllers, each monitoring a different Kubernetes resource, whose main duty is to match the current and desired state. The scheduler is the unit that assigns pods, the smallest Kubernetes schedulable unit, composed of one or multiple containers, to worker nodes. All cluster data is stored in a key-value database called etcd.

The kubelet is a daemon present in each node whose main function is to retrieve pods to schedule from the apiserver and start them on the respective node. It also monitors those pods to keep the apiserver informed on their state and to trigger the appropriate control loops.

Kubernetes offers a high degree of customization. New plugins can be registered to be used in various scheduling phases, and new schedulers and controllers can be introduced into the system in addition to the already available ones, or even replacing them. There is no restriction on the programming language adopted. Custom resources can be used to define new resources to extend the standard object schema. Operators can be developed to monitor these new resources and introduce custom logic into the system [12].

B. Knative

Knative is a serverless platform based on Kubernetes as a container orchestrator. It configures itself as a Kubernetes service.

Knative handles applications as services, monitoring the current load for each one of them and establishing how many pods are necessary for each of them according to the application's configuration and current load. Each Knative application pod is composed of one or multiple user containers and a queue-proxy that forwards traffic to the said containers and monitors their state.

To better understand Knative's internal components, we must understand the request flow of each application.

- The application is deployed as a Knative service; all related resources are created, including a deployment and a replicaset (Kubernetes resources that host the desired number of pods).
- An initial instance is created to establish endpoints and service health status and is dropped immediately after, since no load is detected.
- The activator is inserted into the request flow to buffer new requests until pods are created.
- When new requests arrive, a scale-up operation is triggered.
- When sufficient pods are ready to answer requests, the activator is removed from the requests' flow since they are sent to active instances directly.
- The autoscaler is the component in charge of scaling up and down the application according to current load, service configuration, and containers' monitored parameters.
- When no requests are detected for a certain amount of time, the application scales down to 0 instances and the activator is reinserted in the flow; initial conditions are restored and Knative awaits for new traffic.

III. Preempt-K8S: design and implementation

The implemented controller replicates the deployment/replicaset controllers' control loops in charge of assuring that the specified number of pods for a certain application is always active in the system. The deployment resource was replaced by a custom resource that shows a field to specify the number of instances of an application (a field already present in a deployment object) and a field representing the criticality level of the application.

The "single queue, multiple servers" scheme of the kube manager can cause delays and interferences to the orchestration of critical services. The events contained in the single queue are handled as soon as they are free without taking

into account the services' priority level. Once a control loop thread starts to serve an event, the thread must complete it before serving another. In other words, queued requests are non-preemptable.

The designed controller watches over CRs that work as custom replicasets, without acting (and thus not interfering) on the standard K8s replicasets. In fact, our controller only manages services with timing constraints. The advantages brought by this solution are as follows:

- clear separation between the management of real-time and non-real-time services;
- mixed-criticality workloads can be brought into the picture thanks to the priority level of each service handled by the new controller;
- an event related to a real-time resource will be handled by a thread whose priority is determined by the priority of the event and related resource;
- by limiting the number of concurrently running threads, their interference is bounded and event handling becomes fully preemptable.

Conversely, the new controller scheme with a variable number of servers is based on the following intuition: every event, characterized by the respective application priority, must always be immediately assigned to a server thread that handles it, but let the kernel schedule threads based on their priority. In this way, if a high-priority event arrives, it can be served immediately.

The data flow diagram of the fixed-priority and fully preemptive controller, hereon PREEMPT-K8S, is shown in Fig. 3.

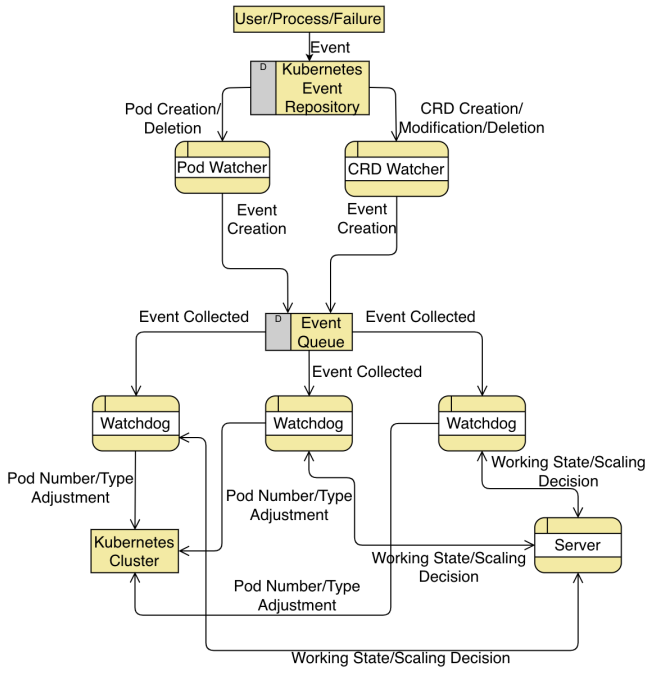

Fig. 3: Preempt-K8s Data Flow Diagram

A. Architecture

PREEMPT-K8S is composed of several threads, each implementing a step of the control loop. They are classified as follows:

- watcher threads are in charge of event collecting;
- watchdog threads in charge of event handling;
- server in charge of thread management.

Application-related events are published by Kubernetes, and watcher threads can connect to them to collect them. The CRD Watcher thread collects creation, modification, and deletion events related to custom resources that represent the applications themselves. The Pod Watcher thread collects deletion events connected to the pod related to the monitored real-time applications. These threads execute at the highest priority among the PREEMPT-K8S threads (i.e., 96 FIFO in our case). This design choice is due to the fact that these threads revolve around HTTP interactions with the apiserver that could take an indefinite amount of time, and we want to collect events as soon as possible in order to handle them in the smallest amount of time possible.

All collected events are posted in a priority queue where every event is posted with the relative application priority level. This queue is shared among all the involved threads.

Watchdog threads are the actual event handlers. There is a configurable base number of threads always active that can then grow up to a configurable maximum number if they should be necessary to handle several concurrent events. They all start with the same scheduling priority (a lower level than the one used for the watchers) to retrieve new events as soon as possible, since their priority level is not known in advance.

Each watchdog retrieves an event from the queue, dequeuing the one with the highest priority. Once the event is collected, the thread changes its scheduling priority to match the one of the retrieved event (higher priority events are handled by higher priority threads). This allows watchdogs to preempt threads handling less critical events in order to speed up the orchestration for higher priority applications. The actions performed are as follows:

1) it retrieves the event with the highest priority;
2) it changes its scheduling priority to adapt to the specific event it is handling;
3) it checks for the existence of the related resource;
4) if the resource is found, it retrieves the desired number of application instances, otherwise, it is considered to be '0' to delete all associated pods;
5) it checks for the number of currently running pods associated with said resource;
6) it calculates the number of pods to create or delete to match the desired state;
7) it creates/deletes the computed number of pods.

Once the event has been handled, the thread reverts to its original scheduling priority to retrieve new events from the queue.

When a new event arrives in the queue, it can have any priority level. With a fixed number of watchdogs, a higher

priority even could end up waiting for less critical ones if all threads are already busy handling them.

The server thread scales the watchdogs during the control loop to ensure there are always sufficient threads to handle new incoming events. A configurable threshold can be set. The server thread keeps track of the number of busy and free threads. When the number of free ones goes below the threshold, it spawns new ones. This design choice also has the advantage of pre-warming threads before new events actually arrive in a saturated system.

Watchdogs are in charge of deleting themselves when too many threads are free in order to reduce resource consumption. The server does not take care of this scaling aspect because making it able to distinguish between free threads and threads that are currently handling an event would have made the design uselessly complex.

The server thread has a management role, thus, its scheduling priority was set higher than the watchdogs' base one, but lower than the watchers' one.

We wrote the PREEMPT-K8S in 526 lines of Rust code. Rust allows creating threads scheduled by the fixed-priority preemptive real-time Linux scheduler and allows interacting with a K8s cluster through the K8s API library. It is worth noting that K8s and the majority of custom controllers are written in Golang, but the use of user-level threads (i.e., the *goroutines*) does not allow full control over the priority of kernel-level threads.

B. Knative Patch

To interact with PREEMPT-K8S Knative had to be patched. In the usual workflow, when a Knative service is deployed, several Kubernetes resources are created. Among these resources, there are the application deployment and the respective replicaset. Knative autoscaler directly interacts with these resources to store scaling decisions that will trigger the standard Kubernetes controller that will actually apply them in the cluster.

There were three challenges that were addressed: i) PREEMPT-K8S only watches over custom resources, thus needing Knative to create them for the managed application in order to interact with our custom controller; ii) the Kubernetes standard resources associated to a Knative services needed to be removed since autoscaler decisions must be applied only to custom resources to avoid duplicated pods created by the vanilla Kubernetes controllers; iii) the incoming traffic needs to be redirected to PREEMPT-K8S managed pods in the same manner Knative routed it in the standard workflow.

The initialization code was modified to create a deployment with no associated pods. The patch gives for granted that a user deploys a custom resource first with one associated pod to establish endpoints, followed by the deployment of the actual Knative services.

A new scaler was introduced in the system to apply scaling decisions to custom resources, thus never updating the default deployment that remains silenced.

TABLE I: Traces from CRD and CRD's pods.

#	Request	Content	User Agent
R1	GET	the CRD namespace	kubectl
R2	GET	the CRD (rtresource that triggered the event)	kubectl
R3	POST	the CRD (rtresource that triggered the event)	kubectl
R4	LIST	directed to CRD namespace	kube-apiserver
R5	GET	the CRD (rtresource that triggered the event)	PREEMPT-K8s Controller
C1	LIST	the Pods related to the CRD being handled	PREEMPT-K8s Controller
C2	POST	the Pod being created	PREEMPT-K8s Controller
C3	GET	the Pod created	kubelet
C4	PATCH	the Pod created	kubelet
C5	GET	the Pod created	kubelet
C6	PATCH	the Pod created	kubelet

The PREEMPT-K8S stub function used to create pods description files (i.e., manifests) was modified to include labels used to route traffic to the managed pods.

C. Monitoring

The proposal makes use of a monitoring strategy to observe the Kubernetes behavior during runtime, with a focus on the pod creation operation during scale-up. The collected data are also used to evaluate whether the proposal is suitable for real-time environments with critical and time-sensitive workloads (as presented in Section IV). For this reason, the monitoring strategy takes advantage of distributed tracing, which allows obtaining fine-grained monitoring data describing the interactions between components with millisecond resolution[1]. Notice that tracing could introduce overhead, but it is limited to specific control-plane events involved in pod creation (e.g., CRD operations and pod creation calls) and is not continuous for all cluster operations.

Kubernetes has been configured to enable the generation of tracing data for both the `apiserver` and `etcd` components, to monitor the pod creation operation, and measure the orchestration time. The tracing data has been collected through two OpenTelemetry [13] pipelines, for cluster-wide and node-specific data, respectively, which send the traces to the observability back-end, i.e., Zipkin [14] in our setup.

In order to determine how to use the tracing data to measure the orchestration time, we have analyzed the data generated during the scaling scenario, for both the proposed PREEMPT-K8S and the vanilla controllers. In this scenario, the PREEMPT-K8S controller is expected to perform the following operations:

- CRD Watcher: `GET` request to retrieve CRD event;
- Watchdog: `LIST` operation to retrieve a list of current pods associated with the resource;
- Create Pod Function called by the Watchdog: `POST` request to create a pod (repeated for each pod to create).

TABLE I provides an overview of the traces collected from both the CRD and the CRD's pods when using the PREEMPT-K8S controller.

Kubernetes first gets the CRD namespace and then the resource (R1-R2). If the resource does not exist, it is created

[1]Tracing is preferred instead of monitoring tools like Prometheus, since they usually collect monitoring data with second granularity, which is not enough in the real-time context.

by using a POST request (R3); otherwise, a PATCH request is generated to change the state of the resource. Then, Kubernetes makes a LIST request (R4) to get resource quotas. The last trace (R5) is the first controller operation. We used R3 as the start event to calculate the orchestration time, since it is the request used for creating a resource.

Regarding the CRD's pods traces, the first trace (C1) is the LIST request made by PREEMPT-K8S to get all pods related to the CRD being handled, followed by the pod creation operation (C2). This is the last event in the control plane and it can be used as the end event for the orchestration time measurement. After orchestration, Kubernetes has to start the pod on the selected node. This stage involves the kubelet running on that node, which exhibits a precise communication pattern during the pod creation (C3-C6). The pod is created, going into a "Container Creating" state (the first GET−PATCH couple, C3-C4). Then, the pod goes into a "Running" state (the second GET−PATCH couple, C5-C6). These traces can be used as other candidates for the end event of the orchestration time measurement. Specifically, the first kubelet GET request (C3) can be used as the end event to make a fair comparison with the vanilla controller, being the first common event since the orchestration starts.

For the vanilla controller, there is a slight variation concerning our proposal. The POST/PATCH trace representing the start time is found in the same namespace where the pod will reside, but the user agent is always kubectl. The orchestration time can be taken at two different points. If the "nodeName" field is already set, the scheduler will not be involved in the flow. This is also the case for the custom controller scenario, where a scheduling operation is simulated by assigning pods the name of the node they will execute on. In this situation, the kube manager POST operation trace can be taken as the end event for orchestration measurement. Otherwise, the scheduler's one is used. The kubelet operations remain the same.

Based on this analysis, we use two ways to calculate the orchestration time. The first one defined as the time between the request for creating a resource/deployment and the last action before sending the request to the kubelet, e.g., $C2 - R3$ for PREEMPT-K8S, and the second one defined as the time between the request for creating a resource/deployment and the first kubelet operation relative to the created pod, e.g., $C3 - R3$ for PREEMPT-K8S. It is important to note that the first way allows focusing only on the scheduling action of K8s, neglecting the kubelet actions. Instead, the second one allows for a fairer comparison with the vanilla controller, as indicated earlier. Both ways are used in our evaluation.

IV. PREEMPT-K8S EVALUATION

We compare the proposed PREEMPT-K8S against the vanilla Kubernetes deployment controller in various configurations to understand its capability in handling mixed-criticality services. We used Kubernetes v1.29.6 and containerd v1.7.11. We also leveraged the OpenTelemetry framework along with Zipkin to collect Kubernetes traces (millisecond resolution), as described in Section III. We deployed the Kubernetes cluster via 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 VMs is the control plane VM, and runs on a dedicated server, while the others are used as worker nodes, depending on the experiments detailed in the following. Critical resources/deployments include an nginx server pod (v1.27.5). For statistical significance purposes, we repeated each experiment 30 times, reporting the average (Avg) and standard deviation (SD) as metrics of interest.

A. RQ1 - Does PREEMPT-K8S perform better than vanilla Kubernetes?

We first evaluate how PREEMPT-K8S behaves compared to the most basic Kubernetes setup in terms of built-in priority handling features. About this latter, K8s implements:

- *Flowschema* [15], which is a feature that creates a sort of "channel" for requests to separate them from the others. In this case, it has been used to isolate the most critical deployment in the namespace where it is located. A level of precedence can be given to these "channels" [10];
- *Priority classes*, which is a feature that gives a certain level of priority to pods in the scheduling and its following phases. It was enabled alongside the previous one to see if it impacted in some way the orchestration time in the control plane [10].

We evaluated the following scenarios:

- vanilla Kubernetes deployment/replicaset controller;
- vanilla Kubernetes deployment/replicaset controller with a basic FlowSchema (FS) and priority class (PC) enabled;
- PREEMPT-K8S deployed within Kubernetes with no priority feature enabled.

We set an experiment that creates one critical resource/deployment within a separate namespace and assigned it the highest priority (i.e., priority set equal to 1). We measured orchestration times, as the time between the request for creating a resource/deployment and the time within the last action before sending the request to the kubelet, while an interfering load (stressload) is applied. We neglect the kubelet time in this experiment to quantify orchestration times only due to the scheduling stuff of K8s. The stressload is implemented as several single-pod resources/deployments (i.e., 10, 25, 50, and 75) running in a separate namespace compared to the one used for the critical resource/deployment.

Please note that PREEMPT-K8S does not involve the K8s scheduler for creating a resource/deployment. So, in the scenario with vanilla Kubernetes, the scheduler was bypassed by assigning the node name in the deployment manifest to make a fair comparison. Instead, in the scenario with FlowSchema and priority classes enabled, the scheduling time is considered since the priority class feature leverages it.

Table II and Figure 4 show results across all interfering resources/deployments.

PREEMPT-K8S shows better performance with an orchestration time on average, for the most critical workload,

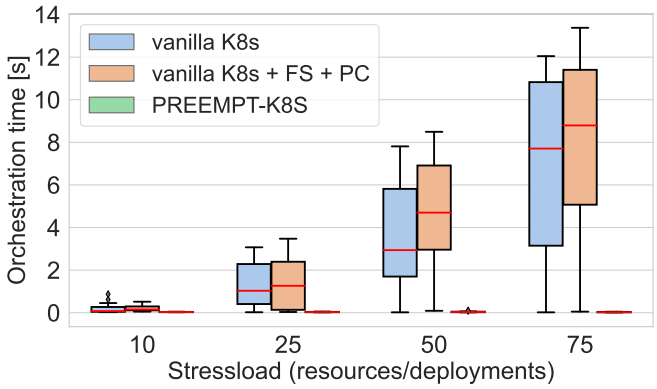

Fig. 4: Orchestration times for critical pod with different interfering loads, comparing basic built-in priority handling mechanism in K8s against PREEMPT-K8S.

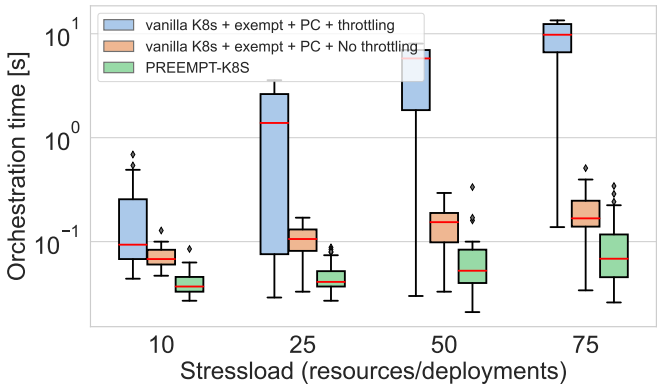

Fig. 5: Orchestration times for critical pod with different interfering loads, comparing vanilla K8s with improved mechanism for priority handling (exempt FlowSchema and throttling disabled) against PREEMPT-K8S.

TABLE II: Orchestration times for deployment of critical service varying interference across vanilla K8s, FlowSchema- and priority class-enabled K8s, and PREEMPT-K8S.

Interference	Vanilla K8s		FlowSchema+PC K8s		PREEMPT-K8S	
	Avg [ms]	SD	Avg [ms]	SD	Avg [ms]	SD
10	183	196	208	136	34	7
25	1260	1014	1369	1149	33	11
50	3555	2436	4697	2505	40	21
75	7140	4081	7680	4352	28	9

always under 100 milliseconds independently of the number of interfering resources. The standard deviation is also smaller than the vanilla version. It can also be noted that the FlowSchema+PC scenario is the worst case, leading to the conclusion that the built-in mechanisms for handling priority are ineffective compared to vanilla K8s.

The vanilla Kubernetes version analyzed in the previous experiments was not the most effective in terms of performance and priority handling. Therefore, we conduct new experiments to reveal whether a better flow control schema and a built-in throttling mechanism can improve the orchestration times of a critical deployment. In K8s, throttling is a mechanism that limits the number of requests per second from a certain component. In that case, the kube manager traffic will be shaped towards the apiserver to not overload it.

The experiments performed are very similar to the first ones described before. This time, a more powerful FlowSchema was used, replacing the previous one, in which all requests coming from that namespace are *exempted* from queuing and handled as soon as possible. The priority class remains the same. In addition, by disabling throttling, it is possible to accelerate the handling times of deployments under test [10]. Further, the end time for orchestration is taken considering the first common event in all the scenarios, i.e., the first kubelet operation relative to the created pod. In that way, we have a fairer comparison. Thus, we considered for comparison also the following scenarios:

- vanilla Kubernetes deployment/replicaset controller with exempt FlowSchema and priority class enabled, and throttling disabled;
- vanilla Kubernetes deployment/replicaset controller with exempt FlowSchema and priority class enabled, and throttling enabled;

Table III and Figure 5 show the results, comparing the proposal with vanilla K8s with an improved mechanism for priority handling (exempt FlowSchema and throttling enabled/disabled). We can notice how similar the performance of the controllers is to the case with throttling disabled. PREEMPT-K8S still shows slightly better mean handling times and variances. It can also be noted that throttling is the most limiting feature. We obtain a p-value < 0.0003 with the Mann–Whitney U test (since the normality test failed) with a 0.05 confidence interval, resulting in statistical significance of the analyzed distributions.

We could point out that the slightly better performance are not a sufficient justification for the development effort. This is not true because all these features cannot be used in production. Indeed, disabling throttling leads to good enough results compared to PREEMPT-K8S. However, Kubernetes developers strongly discourage disabling that mechanism because it could lead to several problems that are unacceptable in a real-time mixed-criticality environment [16]. The exempt FlowSchema is only useful in a situation like the one shown in these experiments, where there is only one deployment in the associated namespace. If more namespaces were exempted from queuing or if a single exempted namespace was hosting

TABLE III: Orchestration times for critical deployment varying interference across K8s with exempt FlowSchema- and priority class- enabled K8s, enabling and disabling throttling, compared to PREEMPT-K8S.

Interfence	K8s exempt+PC+no throttling		K8s exempt+PC+throttling		PREEMPT-K8S	
	Avg [ms]	SD	Avg [ms]	SD	Avg [ms]	SD
10	72	17	191	184	41	12
25	106	37	1499	1318	47	17
50	145	70	4675	2874	73	61
75	196	110	9094	3886	100	81

multiple deployments, there would be no prioritization at all. Further experiments will investigate this aspect. For the priority classes mechanisms, it is a complex mechanism that requires users to create a new Kubernetes resource for each priority level. Our PREEMPT-K8S only needs the priority level number given through the custom resource of an application, making it a more user-friendly mechanism.

TABLE IV: Kubelet effect comparing vanilla K8s with exempt FlowSchema, priority classes, throttling disabled against PREEMPT-K8S with or without a dedicated worker node to the most critical application.

Scenario	Avg [ms]	SD
K8s exempt+PC+no throttling (dedicated)	144	61
PREEMPT-K8S (dedicated)	39	19
K8s exempt+PC+no throttling	196	110
PREEMPT-K8S	100	81

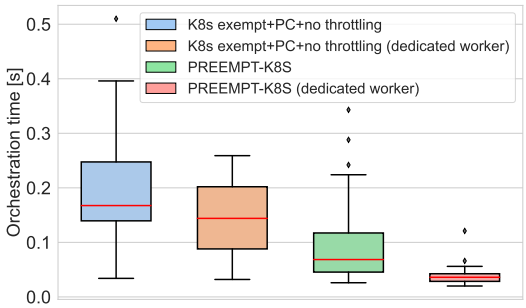

Fig. 6: Boxplots of Kubelet effect comparing vanilla K8s with exempt FlowSchema, priority classes, throttling disabled against PREEMPT-K8S with or without a dedicated worker node to the most critical application.

B. RQ2 - How much does the kubelet affect the orchestration times?

The Kubelet component handles the creation of containers on the worker node. In this experiment, the objective is to evaluate the impact of Kubelet on orchestration times. We set up the K8s cluster with 4 worker nodes. We set PREEMPT-K8S and the vanilla K8s scheduler to deploy pods uniformly on all available worker nodes. In this way, each Kubelet service has to manage all incoming requests for the assigned worker node, unaware of the priority level of the application. We consider 75 interfering resource/deployment, and assign to the most critical application a dedicated worker node. The results were compared with the previous ones and they can be seen in Table IV, while Figure 6 shows the respective box plots.

We can notice that the Kubelet represents an actual bottleneck that needs to be patched to be aware of the choices made by schedulers for priority handling. PREEMPT-K8S would benefit from a Kubelet that works only for the most critical workload. The Mann–Whitney U test showed a clear statistical difference, with a 0.05 confidence interval, between distributions for PREEMPT-K8S experiments (p-values are < 0.0001), while there is no statistical evidence for vanilla K8s experiments (p-value 0.0484), although the results show slightly better performance when using an exclusive node.

V. PREEMPT-K8S IN A SERVERLESS ENVIRONMENT

The experiments in Section IV show the relevant improvements provided by PREEMPT-K8S during the orchestration phase against vanilla K8s, by testing various configurations to properly handle services at differentiated priorities. In this section, instead, we analyze to what extent PREEMPT-K8S adds benefits to a *serverless-based service* with high-priority constraints. As a critical service, we set up an HTTP server pod[2] that receives POST requests with a time parameter that specifies the time to sleep within the serving (this wait is used for simulating connection delays). We measure the service latency, i.e., the end-to-end response time for 10 Knative services with priorities ranging from 1 to 10 for each. Each service is set with 10 threads as the concurrency level. As workload we trigger 40 requests-per-second for each service for a total of 400 requests. For the critical service, we dedicated 2 worker nodes among the 4 available. For statistical significance purposes, we repeated each experiment 30 times, duration of 1 minute each, and we report the average (Avg) and standard deviation (SD) as metrics of interest. As the stressload, we based on "real-traffic-test" benchmark in [17], which simulates realistic traffic with random request latency, service startup latency, and payload sizes. The benchmark was adapted to stress the scaling feature of Knative to show the improvements provided by PREEMPT-K8S during scaling operations. We compare PREEMPT-K8S against Knative scaling achieved through the deployment/replicaset vanilla controllers.

The duration of a single experiment (1 minute) allows us to properly test the scaling mechanism before reaching a stable number of active instances for each service. The scaling logic is driven by the set number of concurrent requests, i.e., scale up when too many requests are detected for a given service, and scale down in the opposite situation.

Fig. 7 and Table V show the results. There is a clear indication that the services are properly prioritized when scaled with the PREEMPT-K8S controller compared with vanilla K8s in both configurations. However, PREEMPT-K8S exhibits a slightly higher end-to-end service latency average than vanilla cases, especially when the priority decreases. By recalling RQ2 (see §IV-B), the kubelet operations affect PREEMPT-K8S (as well as vanilla configurations) since there is no awareness of priorities, which are properly handled in the scheduling phase. More effort is needed to properly integrate PREEMPT-K8S into Knative, by also patching the kubelet and performing a detailed timing analysis to spot bottlenecks. However, even for our preliminary integration in a serverless setting, the orchestration times obtained by PREEMPT-K8S show a smaller standard deviation than the vanilla settings in the majority of cases, especially for the highest priority

[2]A simple Go webserver used as a test image, https://github.com/knative/serving/tree/main/test/test_images/autoscale

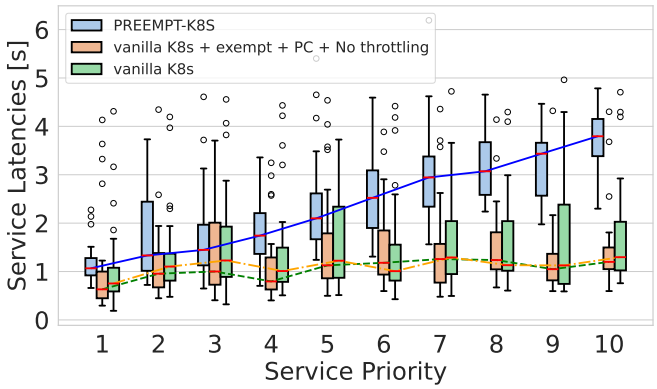

Fig. 7: Boxplots of service latencies for 10 critical services at 10 different priority levels, scaled by the PREEMPT-K8S controller, vanilla K8s, and vanilla K8s with best built-in priority handling, via Knative.

TABLE V: Service latencies (Avg in ms and SD) for 10 critical services at 10 different priority levels, scaled by the PREEMPT-K8S controller, vanilla K8s, and vanilla K8s with best built-in priority handling, via Knative.

Service Priority	Vanilla K8s		Vanilla K8s + Exempt + PC + No Throttle		PREEMPT-K8S	
	Avg [ms]	SD	Avg [ms]	SD	Avg [ms]	SD
1	1029	906	1019	1029	1169	395
2	1312	904	1158	789	1866	1035
3	1517	1089	1382	880	1733	924
4	1434	1077	1101	770	1816	683
5	1648	1001	1547	1087	2331	982
6	1479	1104	1480	855	2535	798
7	1660	1001	1438	977	3079	973
8	1606	978	1518	782	3160	637
9	1693	1244	1254	719	3208	688
10	1726	1088	1403	803	3733	624

service, as already demonstrated in §IV. More importantly, they show the ability of PREEMPT-K8S to correctly prioritize the services during the scale-up operations.

VI. Discussion and Limitations

While the proposed PREEMPT-K8S controller demonstrates significant improvements in orchestration time for mixed-criticality serverless workloads, potential threats to the validity and generalizability of our results must be acknowledged.

Thread Scheduling Limitations. Our design relies on dynamically adjusting the scheduling priority of kernel-level threads after retrieving events from the queue. Although this approach enables preemption of low-criticality tasks, there remains a brief interval between event retrieval and priority escalation. This design choice may introduce minimal delays, which could be avoided with alternative multi-queue or static-priority thread-pool designs. In practice, our empirical results suggest this overhead is negligible; however, different system loads or hardware could amplify this effect.

Runtime and Implementation Constraints. We implemented PREEMPT-K8S in Rust to enable fine-grained control over kernel-level threads and their real-time scheduling attributes. In contrast, common Kubernetes extensions are typically developed in Go, where goroutines do not map directly to individual kernel threads. This architectural mismatch limits direct portability of our solution to controllers written in Go, unless a similar native-thread mechanism is integrated. This could constrain adoption by the broader Kubernetes ecosystem.

Monitoring and Instrumentation Overhead. To measure orchestration performance with high temporal resolution, we employed distributed tracing with millisecond granularity. Although we carefully scoped tracing to only control-plane interactions (e.g., CRD and pod lifecycle operations), extensive tracing could add non-negligible overhead under heavy load if misconfigured. To mitigate this, we used sampling and verified that trace collection did not significantly distort orchestration latencies in our setup.

System Configuration Dependencies. Our experiments assume explicit control of system-level settings, including real-time scheduling (SCHED_FIFO), disabled real-time throttling, fixed CPU frequencies (performance governor), and limited idle states. These configurations ensure minimal jitter and maximum predictability, but they may not be feasible or desirable in multi-tenant or production Kubernetes clusters where strict CPU isolation is impractical. Additionally, improper configuration of real-time scheduling parameters could lead to resource starvation for other workloads.

Comparative Baselines. We compared PREEMPT-K8S with the default Kubernetes controller as well as with Kubernetes configured with FlowSchemas, PriorityClasses, and throttling adjustments. While these baselines represent standard priority-handling techniques in Kubernetes, they do not exhaustively cover all possible fine-tuning options or third-party schedulers. Moreover, the specific workload, cluster size, and stress loads in our experiments might differ from production deployments, which could influence observed orchestration times.

Kubelet Scheduling Effects. Our results highlight that the kubelet remains unaware of the orchestration priorities set by PREEMPT-K8S. Consequently, while our controller can prioritize control-plane scheduling, the final container startup may still experience delays at the node level due to the kubelet's lack of real-time awareness. Future work is needed to align kubelet operations with orchestrator-level priorities to close this gap.

VII. Related Work

Provisioning times for scaling and deployment can be optimized by adopting techniques explored in various studies addressing Function-as-a-Service (FaaS) cold starts.

Many serverless platforms already employ mechanisms such as keep-alive windows to reduce cold-start occurrences, like Knative [18] and OpenFaaS [19]. Even though these mechanisms are already functioning, many optimizations should be considered since these keep-alive windows are still too wide to reduce resource consumption effectively (e.g., OpenFaaS sets a keep-alive window of 30 minutes, by default, for a pod running the last active function). In [20], the authors pointed out how the chosen programming language of an application affects cold-start times, and this technical aspect should be considered as an important requirement in the development phase.

Other techniques, as categorized in [21], include application-based [22], [23], checkpoint-based [24], prediction-based [25]–[28], and cache-based [29]. Their core strategies involve pre-warming containers, reusing or pooling warm containers, scheduling ahead of time, and keeping containers alive. All these aim to minimize the overhead associated with launching a new container—such as downloading the image, configuring the sandbox, and initializing the language runtime and application code.

However, these solutions primarily target optimizations at the worker node level and often overlook the orchestration overhead, particularly under variable or heavy-load conditions. Our approach complements these techniques by incorporating SLO-aware orchestration to ensure that critical functions receive priority when system demand is high.

Liu et al. [30] demonstrated that although the time to start a new pod may not lie directly on the critical execution path, it still significantly affects the overall service response time during scaling operations. As noted in their introduction, they also highlight the ”declarative tax”—the latency introduced by the orchestration system as it converts declarative configuration into imperative commands for cluster reconciliation—as a growing bottleneck in serverless environments, where startup latencies are rapidly shrinking.

The studies in [31], [32] investigate and compare orchestration times across various Kubernetes distributions. In particular, [32] introduces a systematic benchmarking methodology, examining pod startup times under increasing pod counts. However, these measurements are primarily used to evaluate the orchestrator's throughput, rather than explicitly addressing orchestration latency. Notably, even research focused on real-time orchestration—such as [33]–[37]—tends to overlook orchestration latency. These works concentrate on ensuring the timeliness of application execution on worker nodes, without accounting for the orchestration overhead that may impact end-to-end responsiveness.

Each of these studies targets a different part of the scaling process, never focusing on the orchestration process. As we highlighted in this paper, current orchestration platforms are not equipped with appropriate priority handling features to manage mixed-criticality workloads. This leads to an unpredictable growth in the events handling times, since serverless platforms might request the scaling of multiple applications at the same time, which should be addressed by the underlying orchestrator in the shortest amount of time possible to reduce service response times and keep a stable quality of service. It is not surprising that the vast majority of works focused on the orchestration process can be found in the IIoT field, since this problem requires a real-time approach.

Finally, some studies applied to databases (e.g., MongoDB in [38], [39]) share the core idea of introducing priority-driven differentiated performance by leveraging OS-level priorities for threads handling requests. Our work is complementary to these since it targets orchestration-level scheduling in Kubernetes, focusing on the control plane and container lifecycle management, whereas [38], [39] focus on I/O operations of a NoSQL database.

VIII. Conclusions

In this research, we highlighted the problem of slow orchestration times with specific reference to the Kubernetes container orchestration platform. Facing these kinds of problems becomes imperative if the objective is to develop a real-time environment with mixed-criticality applications.

We proposed a custom controller that replicates one of the most important control loops of the platform to make it compliant with scenarios where slow start-up times are not allowed for critical workloads and where there is a need to take into account the criticality level of each application managed by the system. The principal characteristic is not only to have a handling software that handles events according to their priority, but also to make itself a real-time software with high-priority scheduling and preemption mechanisms. These activities are vital in serverless computing environments where instances must be active as soon as they are required, especially those with high criticality and stringent timing requirements.

Beyond the deployment loop, other control loops could be re-imagined for real-time operation in PREEMPT-K8S. Ultimately, the goal is to enable a co-orchestrator by integrating a standard Kubernetes instance with a lightweight, patched version for critical tasks, ensuring effective shared resources management. This work marks an initial step toward that vision.

Acknowledgment

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). This work has been partially supported by the GENIO project (CUP B69J23005770005) funded by MIMIT, ”Accordi per l'Innovazione” program, and by the “IDA—Information Disorder Awareness” Project funded by the European Union-Next Generation EU within the SERICS Program through the MUR National Recovery and Resilience Plan under Grant PE00000014 and by the SERENA-IIoT project, which has been funded by EU - NGEU, Mission 4 Component 1, CUP

J53D23007090006, under the PRIN 2022 (MUR - *Ministero dell'Università e della Ricerca*) program (project code 2022CN4EBH).

REFERENCES

[1] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, 2016, pp. 1204–1210.

[2] Y. Li, Y. Lin, Y. Wang, K. Ye, and C. Xu, "Serverless computing: state-of-the-art, challenges and opportunities," *IEEE Transactions on Services Computing*, vol. 16, no. 2, pp. 1522–1539, 2022.

[3] B. Qian, J. Su, Z. Wen, D. N. Jha, Y. Li, Y. Guan, D. Puthal, P. James, R. Yang, A. Y. Zomaya *et al.*, "Orchestrating the development lifecycle of machine learning-based iot applications: A taxonomy and survey," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, pp. 1–47, 2020.

[4] W. Zhang, Y. Gao, and W. Dong, "Providing realtime support for containerized edge services," *ACM Transactions on Internet Technology*, vol. 23, no. 4, pp. 1–25, 2023.

[5] V. U. Ugwueze, "Serverless computing: Redefining scalability and cost optimization in cloud services," *International Research Journal of Modernization in Engineering Technology and Science*, 2024.

[6] M. Golec, G. K. Walia, M. Kumar, F. Cuadrado, S. S. Gill, and S. Uhlig, "Cold start latency in serverless computing: A systematic review, taxonomy, and future directions," *ACM Computing Surveys*, vol. 57, no. 3, pp. 1–36, 2024.

[7] A. Balador, J. Eker, R. Ul Islam, R. Mini, K. Nilsson, M. Ashjaei, S. Mubeen, H. Hansson, and K.-E. Årzén, "Aorta: Advanced offloading for real-time applications," in *Real-Time Cloud (RT-CLOUD), 2023*, 2023.

[8] S. Fiori, L. Abeni, and T. Cucinotta, "Rt-kubernetes: containerized real-time cloud computing," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 36–39.

[9] F. Lumpp, F. Fummi, H. D. Patel, and N. Bombieri, "Enabling kubernetes orchestration of mixed-criticality software for autonomous mobile robots," *IEEE Transactions on Robotics*, 2023.

[10] Kubernetes, "Kubernetes documentation," https://kubernetes.io/ [Accessed: 2025-04-06].

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

[12] P. Martin, "Extending kubernetes api with custom resources definitions," in *Kubernetes Programming with Go: Programming Kubernetes Clients and Operators Using Go and the Kubernetes API*. Springer, 2022, pp. 193–207.

[13] OpenTelemetry, "Opentelemetry documentation," https://opentelemetry.io/docs/platforms/kubernetes/getting-started/ [Accessed: 2025-04-06].

[14] Zipkin, "Zipkin documentation," https://zipkin.io/ [Accessed: 2025-04-06].

[15] Kubernetes, "Api priority and fairness," https://kubernetes.io/docs/concepts/cluster-administration/flow-control/ [Accessed: Today].

[16] ——, "Kubernetes cluster api documentation," https://cluster-api.sigs.k8s.io/developer/core/tuning?utm_source=chatgpt.com [Accessed: 2025-04-06].

[17] Knative developers. Performance tests. https://github.com/knative/serving/tree/main/test/performance. Accessed on June 29, 2025.

[18] N. Kaviani, D. Kalinin, and M. Maximilien, "Towards serverless as commodity: A case of knative," in *Proceedings of the 5th International Workshop on Serverless Computing*, 2019, pp. 13–18.

[19] OpenFaas developers. OpenFaas homepage. https://www.openfaas.com/. Accessed on June 29, 2025.

[20] D. Xie, Y. Hu, and L. Qin, "An evaluation of serverless computing on x86 and arm platforms: Performance and design implications," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 313–321.

[21] A. Ebrahimi, M. Ghobaei-Arani, and H. Saboohi, "Cold start latency mitigation mechanisms in serverless computing: taxonomy, review, and future directions," *Journal of Systems Architecture*, p. 103115, 2024.

[22] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, "Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 467–481.

[23] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, and X. Jin, "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 5, pp. 1–29, 2023.

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

[25] Z. Xu, H. Zhang, X. Geng, Q. Wu, and H. Ma, "Adaptive function launching acceleration in serverless computing platforms," in *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 2019, pp. 9–16.

[26] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: Warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 753–767.

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

[28] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in faas services," in *Proceedings of the 35th annual ACM symposium on applied computing*, 2020, pp. 134–143.

[29] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, "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, Volume 1*, 2024, pp. 335–350.

[30] Q. Liu, D. Du, Y. Xia, P. Zhang, and H. Chen, "The gap between serverless research and real-world systems," in *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*. ACM, 2023, p. 475–485.

[31] H. Koziolek and N. Eskandani, "Lightweight kubernetes distributions: a performance comparison of microk8s, k3s, k0s, and microshift," in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, 2023, pp. 17–29.

[32] M. Straesser, J. Mathiasch, A. Bauer, and S. Kounev, "A systematic approach for benchmarking of container orchestration frameworks," in *Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering*, 2023, pp. 187–198.

[33] M. Barletta, M. Cinque, L. De Simone, and R. D. Corte, "Criticality-aware monitoring and orchestration for containerized industry 4.0 environments," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, pp. 1–28, 2024.

[34] S. Fiori, L. Abeni, and T. Cucinotta, "Rt-kubernetes: Containerized real-time cloud computing," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 36–39.

[35] V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, "Hierarchical resource orchestration framework for real-time containers," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 1, pp. 1–24, 2024.

[36] T. Cucinotta, L. Abeni, M. Marinoni, R. Mancini, and C. Vitucci, "Strong temporal isolation among containers in openstack for nfv services," *IEEE Transactions on Cloud Computing*, vol. 11, no. 1, pp. 763–778, 2021.

[37] F. Lumpp, F. Fummi, H. D. Patel, and N. Bombieri, "Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots," *IEEE Transactions on Robotics*, vol. 40, pp. 540–553, 2024.

[38] R. Andreoli, T. Cucinotta, D. Pedreschi *et al.*, "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), 2021, pp. 77–86.

[39] R. Andreoli, T. Cucinotta, and D. B. De Oliveira, "Priority-driven differentiated performance for nosql database-as-a-service," *IEEE Transactions on Cloud Computing*, vol. 11, no. 4, pp. 3469–3482, 2023.