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

Marcello Cinque
Università degli Studi di Napoli Federico II, Italy

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

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

Stefano Toscano
Università degli Studi di Napoli Federico II, Italy

Abstract

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

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

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

Digital Object Identifier 10.4230/DARTS.12.2.1

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

Related Article Marcello Cinque, Luigi De Simone, Raffaele Della Corte, and Stefano Toscano, "PREEMPT-FaaS: Taming Orchestration Times in Latency-Sensitive Serverless Environments", in 38th European Conference on Real-Time Systems (ECRTS 2026), LIPIcs, Vol. 375, pp. 22:1–22:26, 2026.
https://doi.org/10.4230/LIPIcs.ECRTS.2026.22

Related Conference 38th European Conference on Real-Time Systems (ECRTS 2026), July 7–10, 2026, Lund, Sweden

1 Scope

PREEMPT-FaaS is a Rust-based real-time controller of container instances, developed for the Kubernetes (K8s) and designed to reduce orchestration times of latency-sensitive applications deployed on serverless platforms. PREEMPT-FaaS is implemented in Rust to fully exploit fixed-priority preemptive scheduling for controller threads. Unlike existing K8s components, which are primarily written in Go and rely on a user-level threading (ULT) model with limited control over OS scheduling semantics, Rust enables direct management of kernel-level threads (KLTs) and their scheduling parameters. This design allows PREEMPT-FaaS to leverage Linux real-time scheduling policies, specifically SCHED_FIFO, ensuring deterministic prioritization and reduced scheduling interference. By operating at the kernel scheduling level, PREEMPT-FaaS overcomes the constraints imposed by Go's runtime scheduler and provides tighter control over control plane execution latency. The architecture of the PREEMPT-FaaS controller is described in the main paper (see Section 3). PREEMPT-FaaS preserves the "*single queue, multiple servers*" paradigm, but, unlike the Vanilla K8s controller, implements a POSIX multi-priority queue to ensure immediate processing of the high-priority orchestration events. To isolate critical applications from standard workloads, we define a new Custom Resource Definition (CRD), *RTResource*, which serves as a Deployment-like abstraction enriched with a criticality level. The PREEMPT-FaaS controller exclusively monitors these resources and enforces priority-based orchestration through a preemptive scheduling model implemented at the kernel-thread level. We also developed a patch for Knative, a popular serverless computing platform built on top of K8s, for seamless integration with the function-as-a-service (FaaS) model into PREEMPT-FaaS.

In this artifact, we provide a detailed tutorial to setup the PREEMP-FaaS controller in a K8s cluster, along with Knative. Furthermore, for transparency, this document also provides information on our experimental evaluation, found in Section 4 of our paper, along with all the tools and steps required to run experiments.

The experimental campaign provided two major contributions:

- **Orchestration Times**: PREEMPT-FaaS is able to reduce orchestration times under intense orchestration workloads, specifically for critical applications, up to a ∼6× reduction factor compared to the Vanilla K8s controller.
- **E2E improvements**: the reduced orchestration times directly impact end-to-end (E2E) latencies and the application's throughput, especially in cold-start scenarios where the applications are scaled to zero instances when request load is started.

2 Paper Experimental Campaign

Testbed. The testbed consists of a Kubernetes cluster deployed in VMs on two physical machines. The control plane is dispatched on a dedicated host. The host machine is equipped with 8 Intel Xeon CPU E5-2630L CPUs with 16 logical processors, and 32GB RAM. The cluster comprises one master node (4 vCPUs, 4GB RAM) and four worker nodes (8 vCPUs, 16GB RAM each), each deployed on a dedicated VM. All nodes run Ubuntu Linux 22.04.3 LTS with kernel v5.15, Kubernetes v1.29.15, containerd v1.7.11, and runc v1.1.11. The Knative patch has been developed on a fork of the Knative Serving repository. The experimental evaluation is designed to isolate control-plane interference from application-level execution. As shown in Fig. 1, the Kubernetes control plane, including core components and both controllers (Vanilla K8s and PREEMPT-FaaS), is deployed on a dedicated `master node`. The Knative control components are hosted on `worker node 1` to avoid interference with either application workloads or stress generators. `Worker node 2` is dedicated to running the benchmarked serverless applications (KSVCs) (server-side). We

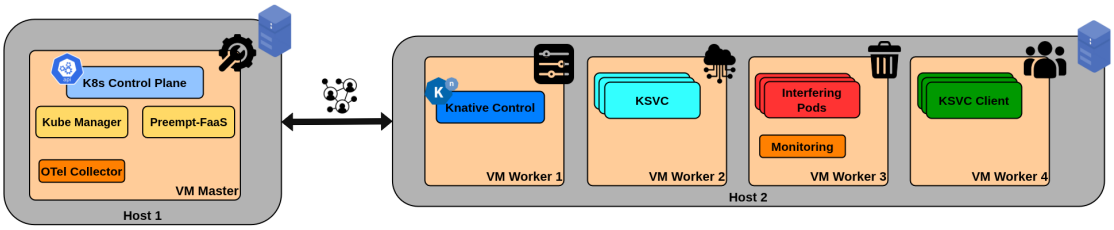

Figure 1 Testbed in the experimental setup.

generate orchestration interference leveraging resource creation and deletion requests. To ensure that control-plane contention does not directly compete with KSVC execution, we employ `worker node 3` to run interfering containers (non-critical services). `Worker node 4` is used as a workload generator for the KSVCs applications. Finally, the monitoring services are deployed in the `master node` and on `worker node 3`. We also remark that the Knative activator is set to be removed from the service request path to ensure that service latencies are not affected by a single bottleneck that buffers and forwards all requests.

2.1 Experiment Description

We developed two separate experimental campaigns. The first set of benchmarks is a modified version of the Knative performance benchmarks suite that provides evidence of the improved orchestration times granted by `PREEMPT-FaaS`. The second one is a set of benchmarks provided by the vSwarm test suite, a collection of serverless functions to invoke. These experiments demonstrate how reduced orchestration overhead translates to higher throughput and lower E2E latency.

DISCLAIMER: Priorities are employed only with `PREEMPT-FaaS`.

2.1.1 Relevant Metrics

- **Scale-Up.** The event at which the Knative autoscaler updates the `replicas` field of the Deployment or RTResource. A single iteration may include multiple scale-up events per service.
- **Starts Processing.** The event at which the controller begins handling a scale-up event, identified by its first interaction with the `kube-apiserver`. For the Vanilla K8s controller, this corresponds to the creation of the first pod in the new replica set. For `PREEMPT-FaaS`, it coincides with updating the RTResource status condition to `Progressing`.
- **Pods Created.** The event at which all required pod objects for a scale-up are successfully created in the cluster. A pod is considered "created" once its API object exists, independently of container startup. For single-replica scale-ups under Vanilla K8s, this event coincides with *Starts Processing*.
- **Pods Started.** The event at which all required pods are running and ready to receive traffic. Although this stage depends on kubelet behavior (which is outside the scope of this work), faster orchestration can advance pod scheduling and startup. Nevertheless, kubelet queueing and internal mechanisms may still introduce variability.
- **E2E Latency.** The time interval starts from the moment the client sends a request to the moment it receives the service response.
- **Throughput.** The number of requests per second (RPS) the service was able to serve.

2.1.2 Real Traffic Test Benchmark

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

Test Configuration. The test is executed as follows.

Table 1 Real-Traffic-Test experimental workflow parameters.

Parameter	Value
Number of repetitions per experiment	10
Number of KSVCs	10 (Priorities: 1–10)
Number of interfering resources	{5, 10, 20} One parameter per exp
Time between bursts	10 Seconds

- For each controller (Vanilla K8s, `PREEMPT-FaaS`) we run these three experiments (one per interference load) with the above configuration.
- We repeat the experiment with 20 interfering resources without audit logs generation, for both controllers.
- We repeat the experiment with 20 interfering resources for the `PREEMPT-FaaS` controller, disabling the feature that generates new worker threads to handle bursts. We limit the number of worker threads to 5, following the Vanilla K8s configuration.

Findings. As shown in Section 4.2 of the paper, under high orchestration interference, `PREEMPT-FaaS` substantially reduces the time required for the control plane to react to scaling requests and instantiate new replicas. In the worst-case configuration (20 interfering events), it achieves up to **6.55**× lower control-plane reaction latency and **5.18**× faster replica creation compared to Vanilla K8s. These improvements persist even with fixed worker threads, indicating that the gains primarily derive from priority-aware preemptive scheduling rather than increased parallelism. Residual delays are mainly attributable to `kube-apiserver` saturation under peak load. The monitoring setup did not produce enough overhead to impact the E2E latencies.

2.1.3 vSwarm Benchmarks

Test Description. We employ the widely adopted *vSwarm* serverless benchmark suite (see [3] for further details on the serverless functions we leveraged). Traffic is generated via the default vSwarm *invoker* script, which provides request latencies and throughput when tests complete. We chose *RNN* and *Video Analytics* benchmark services from the suite. We deploy 5 identical

KSVCs, assigning each service the highest priority level under `PREEMPT-FaaS`. The services are executed alongside an orchestration interference workload consisting of continuous bursts of 15, 30, and 45 resource creation/deletion events. The stress load is intentionally increased to drive the system into a regime where E2E metrics are noticeably affected by orchestration delays (higher loads led our system to instability). We evaluate both the Vanilla K8s and `PREEMPT-FaaS` controllers under identical conditions, each subjected to their respective interfering resources (Deployments for Vanilla K8s, RTResources for `PREEMPT-FaaS`), with one pod per resource. The decision to deploy 5 identical services, rather than a single instance, mitigates stochastic variations and ensures statistical significance. Since the `kube-apiserver` operates in a best-effort mode, even when processing and dispatching requests, a single scale-up event could be unrepresentative due to transient timing advantages. We evaluated a scenario where services are scaled from 0 to 1 instances each. For the RNN benchmarks, we also evaluated the scale-up from 1 to 2 instances.

Test Configuration. The test is executed as follows.

Table 2 RNN experimental workflow parameters.

Parameter	Value
KSVCs Images	Pre-Pulled on Worker Node 2
Number of KSVCs	5 (Priorities: 1)
Number of invokers	5 (One per KSVC)
RPS per KSVC	50
Timeout	40 seconds
Duration	30 seconds
Number of iterations	10
Number of interfering resources	{15, 30, 45} One parameter per exp
Time between bursts	None

Table 3 Video Analytics Standalone experimental workflow parameters.

Parameter	Value
KSVCs Images	Pre-Pulled on Worker Node 2
KSVC Video Database	Single-Instance for all KSVCs, deployed on Worker Node 1
Number of KSVCs	5 (Priorities: 1)
Number of invokers	5 (One per KSVC)
RPS per KSVC	12
Timeout	40 seconds
Duration	30 seconds
Number of iterations	10
Number of interfering resources	{15, 30, 45} One parameter per exp
Time between bursts	None

- For each controller (Vanilla K8s, `PREEMPT-FaaS`), we run these three experiments (one per interference load) with the above configuration.
- For the RNN benchmark, we follow the same configuration for both the scale 0->1 and 1->2 scenarios. Scale 1->2 does not include the 45 interfering resources scenario.

Findings. As shown in Section 4.3, under high orchestration interference, `PREEMPT-FaaS` improves performance in the orchestration process, leading to substantial gains in terms of service E2E metrics, especially under cold-start conditions where services must be scaled as soon as possible to respond to incoming load. In particular, the sensitivity analysis provided evidence of shorter mean latencies, with a **2-second reduction** at 95th percentile for the highest stressload, and **improved throughput**.

3 Content

In the following, we describe the repositories needed to access the project sources, the experiments, and our experimental dataset (please, refer to AEC – `aec` – branches).

▶ **PREEMPT-FaaS Project Repository [2].** This repository contains the `PREEMPT-FaaS` source code, configuration, and build pipeline. It also provides the monitoring infrastructure, test pods, storage configuration for experimental results, and all the experimental campaigns and tools regarding the vSwarm benchmarks, including the result dataset.
▶ **Knative Patch Repository [1].** The repository contains the Knative patch source code, along with the instructions to set it up in your cluster. This repository also provides the Real Traffic Test benchmark source code and setup configurations, along with the result dataset.

The repositories present detailed `REAMDE.md` files that will guide the user through all the necessary steps to install `PREEMPT-FaaS` and the Knative patch, deploy the monitoring pipeline, and replicate the experiments. In this document, we provide guides to set up the environment (Section 4) and recreate our experimental campaign (Section 5). Note that the exact results from our paper cannnot be reproduced exactly unless the testbed is setup as in Section 2.

Note. The experiment scripts assume that the user has the exact number of workers shown in Section 2. However, we highlight the fact that `PREEMPT-FaaS` and our Knative version can be deployed in any cluster with a control plane and, at least, one worker node. The nodes can be either physical hosts, VMs hosted on type 1 or 2 hypervisors, or even containers. Only experimental results will be affected by these environment configurations.

4 Getting the Artifact

The artifact endorsed by the Artifact Evaluation Committee is available free of charge on the Dagstuhl Research Online Publication Server (DROPS).

This section provides all the steps to set up a Kubernetes environment, install PREEMPT-FaaS, and the provided Knative patch. We also provide instructions to set up the experiments storage unit and monitoring infrastructure.

4.1 Cluster Setup

In this section, we provide instructions to set up the computing nodes in which the Kubernetes cluster will be deployed. We provide a guide to set up 5 VMs with the **VitualBox hypervisor**. We also suggest to create an additional VM to build PREEMPT-FaaS, since its build requires dependencies that might interfere with the cluster nodes. You can also build the images on a physical host.

The provided guide deploys:

- **1 Control Plane VM**. 4 CPUs, 4GB RAM, 100GB Disk.
- **4 Worker Node VMs**. 8 CPUs, 16GB RAM, 100GB Disk.
- **4 Builder (Optional)**. 4 CPUs, 4GB RAM, 25GB Disk.

The guide suggests using VirtualBox, but users can employ any hypervisor they want.

On your host (the machine that will host the VMs), download **VirtualBox**.

```
# Download VirtualBox
sudo apt update
sudo apt install virtualbox
```

On your host, download **TigerVNC**. It is a VNC client that allows to connect to new VMs remotely. In this guide we use a GUI version. Run the following commands on your host, or any machine connected to a monitor that can also access the host via `ssh`.

```
# Download TigerVNC
sudo apt update
sudo apt install tigervnc-viewer
```

On your host, create the Control Plane VM.

```
# Download the ISO
curl -L -O https://old-releases.ubuntu.com/releases/22.04.3/ubuntu-22.04.3-live-server-
    amd64.iso

# Create the Control Plane VM
VBoxManage createvm --name "Control" --ostype Ubuntu_64 --register
VBoxManage modifyvm "Control" --memory 4096 --cpus 4

# The bridge allows to connect the virtual NIC to the physical NIC
BRIDGE_IF=$(ip route | grep default | awk '{print $5}')
VBoxManage modifyvm "Control" --nic1 bridged --bridgeadapter1 $BRIDGE_IF

# Create the disk and the boot configuration
VBoxManage createhd --filename ~/VirtualBox\ VMs/Control/control.vdi --size 102400
VBoxManage storagectl "Control" --name "SATA" --add sata
VBoxManage storageattach "Control" --storagectl "SATA" --port 0 --device 0 --type hdd --
    medium ~/VirtualBox\ VMs/Control/control.vdi
VBoxManage storageattach "Control" --storagectl "SATA" --port 1 --device 0 --type
    dvddrive --medium ./ubuntu-22.04.3-live-server-amd64.iso
VBoxManage modifyvm "Control" --boot1 dvd --boot2 disk

# Expose the VM on port 5900 to install the Ubuntu OS
VBoxManage modifyvm "Control" --vrde on --vrdeport 5900
VBoxManage modifyvm "Control" --vrdeproperty VNCPassword=password

# Start the VM
VBoxManage startvm "Control" --type headless
```

Now move to the node where you installed the VNC client, connect to the Control Plane VM and install the OS.

```
# Start the VNC client
vncviewer <IP-host>:5900 # Then use 'password' in the popup
```

Install the OS following the instructions. When configuring the network, click on the virtual NIC, choose edit IPv4 and manually set your subnet configuration, including the static IP for the VM. The installation might take a while. It may block when installing some unnecessary dependencies. You can choose to stop the installation and reboot the VM. Then, on your host, run the following commands.

```
# Power Off the VM
VBoxManage controlvm "Control" poweroff

# Remove the boot
VBoxManage storageattach "Control" --storagectl "SATA" --port 1 --device 0 --type
    dvddrive --medium none

# Restart the VM
VBoxManage startvm "Control" --type headless

# Connect via ssh with your credentials and check that the VM is healthy
ssh <user>:<Control-VM-Static-IP>

# If you forcefully stopped the installation, run:
sudo dpkg --configure -a
apt --fix-broken install
```

Now, we clone the control plane VM to create the worker nodes. For each node, use the following procedure. Remember to use a different *vrdeport*.

```
# Power Off the VM
VBoxManage controlvm "Control" poweroff

# Clone the Control Plane VM and change CPU number and RAM size
VBoxManage clonevm "Control" --name "Worker1" --register
VBoxManage modifyvm "Worker1" --memory 16384 --cpus 8

# Modify the vrde port to access the Worker1 VM from the VNC client
VBoxManage modifyvm "Worker1" --vrde on --vrdeport 5901

# Start both VMs
VBoxManage startvm "Control" --type headless
VBoxManage startvm "Worker1" --type headless
```

Now, via VNC client, enter each worker VM.

```
# Start the VNC client
vncviewer <IP-host>:5901 # Then use 'password' in the popup

# Resize the logic disk to use all the allocated space
sudo lvextend -l +100%FREE /dev/ubuntu-vg/ubuntu-lv
sudo resize2fs /dev/ubuntu-vg/ubuntu-lv

# Rename the node
sudo hostnamectl set-hostname <worker-name>

# Change the static IP
$ cat /etc/netplan/00-installer-config.yaml

# Expected output
# This is the network config written by 'subiquity'
network:
  ethernets:
    enp0s3:
      addresses:
```

```
      - 172.28.21.26/24 # This was our control plane static IP
      nameservers:
        addresses:
        - 8.8.8.8
        - 8.8.4.4
        search: []
      routes:
      - to: default
        via: 172.28.21.254 # This was our gateway
  version: 2

sudo nano /etc/netplan/00-installer-config.yaml # Change the Control Plane static IP to
     the Worker1 static IP

sudo netplan apply

# Check the new static IP
ip a show enp0s3
```

Now all the workers can be accessed via `ssh`.

The Builder VM, if needed, can be set up like the control plane VM, providing `25600` as storage size.

4.2 Kubernetes Installation

If users want to reproduce our exact setup, 5 nodes must be available to the user; these nodes can be either Virtual Machines (VMs) or 5 separate hosts. Our testbed is described in Section 2. It shows a single control plane and 4 worker nodes. The control plane VM is hosted on a separate host. Users may choose their favorite configuration, but the experimental results may differ from our dataset. We highlight the fact that we employ a type 1 hypervisor (`ESXI`) on each physical host. A type 2 hypervisor may impact the end results.

Note. The version suggested in this guide refers to the OS and Kernel versions we use in our testbed. If you need to use different versions, please refer to the official docs we report in Section 4.2.11.

Note. For full compatibility with our scripts' default values, use the following names for your nodes (use `sudo hostnamectl set-hostname <node-name>` on each node):

- **Control Plane**. dessertm1;
- **Worker 1**. dessertw1;
- **Worker 2**. dessertw2;
- **Worker 3**. dessertw3;
- **Worker 4**. dessertw4.

4.2.1 Step 1

On each node (here on, we refer to nodes as the actual component running workload, e.g., the VMs of our testbed), run the following commands to prepare the environment.

```
# Update the system
sudo apt update && sudo apt upgrade -y

# Disable the swap (required by Kubernetes)
sudo swapoff -a
sudo sed -i '/ swap / s/^/#/' /etc/fstab

# Check "/etc/fstab" to ensure the SWAP line si commented out
cat /etc/fstab | grep swap

# If the line is not commented, you can either:
# 1. Manually comment it with your shell editor (insert '#' at the beginning of the
    line)
# 2. Run an alternative command (this might still not work, in that case, rely on the
    manual setting)
sudo sed -i '/swap.img/ s/^/#/' /etc/fstab

# Enable kernel modules
cat <<EOF | sudo tee /etc/modules-load.d/k8s.conf
overlay
br_netfilter
EOF

sudo modprobe overlay
sudo modprobe br_netfilter

# Set sysctl params
cat <<EOF | sudo tee /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-iptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.ipv4.ip_forward = 1
EOF

sudo sysctl --system
```

4.2.2 Step 2

On each node, install the container runtime `containerd`. We report the latest version available compatible with `Kubernetes 1.29.0` (`containerd 1.17.18`). We highlight that we employed (`containerd 1.17.11` instead, which is no longer available. You can still find it here: `https://github.com/containerd/containerd/releases/tag/v1.7.11`. Keep in mind that you might also need to install `runc` separately if you decide to install `containerd 1.17.11` from the provided source.

```
# Install prerequisites
sudo apt install -y apt-transport-https ca-certificates curl gnupg lsb-release

# Add Docker repo (for containerd)
sudo mkdir -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/
    apt/keyrings/docker.gpg
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
```

```
  https://download.docker.com/linux/ubuntu \
  $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

# Install containerd
sudo apt update
sudo apt install -y containerd.io=1.7.18-1

# Configure containerd
sudo mkdir -p /etc/containerd
containerd config default | sudo tee /etc/containerd/config.toml

# Change cgroup driver to systemd
sudo sed -i 's/SystemdCgroup = false/SystemdCgroup = true/' /etc/containerd/config.toml

# Restart containerd
sudo systemctl restart containerd
sudo systemctl enable containerd
```

4.2.3 Step 3

On each node install Kubernetes components and tools: `Kubelet`, `Kubeadm`, and `Kubectl`.

```
# Download and set public key
sudo curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.29/deb/Release.key | \
  sudo gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg

# Add Kubernetes repository
echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] \
https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /" | \
  sudo tee /etc/apt/sources.list.d/kubernetes.list

# Install Kubelet, Kubeadm and Kubectl
sudo apt update
sudo apt install -y kubelet kubeadm kubectl
sudo apt-mark hold kubelet kubeadm kubectl
```

4.2.4 Step 4

Only on the control plane node, run the following command to initialize the cluster.

```
# Init Kubernetes control plane
sudo kubeadm init --pod-network-cidr=10.244.0.0/16

# Expected output
...

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

```
Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 129.192.68.126:6443 --token smid91.8r9ploghc6az0lxz \
    --discovery-token-ca-cert-hash sha256:<sha-produced>
```

The expected output listed above ends with a `kubeadm join` command. You will need to run it on each worker node. More on that later.

4.2.5 Step 5

Now, configure the `Kubectl` cmd tool on the control plane.

```
# Configure Kubectl
mkdir -p $HOME/.kube
sudo cp /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

4.2.6 Step 6

From the control plane, start Pod Network `Flannel`.

```
# Set up Flannel
kubectl apply -f https://raw.githubusercontent.com/flannel-io/flannel/master/
    Documentation/kube-flannel.yml

# Expected output
namespace/kube-flannel created
clusterrole.rbac.authorization.k8s.io/flannel created
clusterrolebinding.rbac.authorization.k8s.io/flannel created
serviceaccount/flannel created
configmap/kube-flannel-cfg created
daemonset.apps/kube-flannel-ds created
```

4.2.7 Step 7

Now run the previously discussed `kubeadm join` command on all worker nodes. Paste the exact command provided at the end of Step 4 of your run. We report our example. Note that it should be run with `sudo`.

```
# Worker nodes Join command
sudo kubeadm join 129.192.68.126:6443 --token smid91.8r9ploghc6az0lxz \
    --discovery-token-ca-cert-hash sha256:<sha-produced>

# Expected output
...
```

```
This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.
```

4.2.8 Step 8

Verify that your nodes have joined the cluster by running the following command in the control plane (here on, you can work only on this node, unless explicit exceptions).

```
# Check the worker nodes state
# Wait a couple of minutes to let the
# control plane see the workers as ready
kubectl get nodes

# Expected output
NAME STATUS ROLES AGE VERSION
master Ready control-plane 666d v1.29.0
worker1 Ready <none> 431d v1.29.0
worker2 Ready <none> 575d v1.29.0
worker3 Ready <none> 431d v1.29.0
worker4 Ready <none> 94d v1.29.0
```

4.2.9 Reset

If something went wrong or you simply want to restart the procedure, run the following commands on the control plane.

```
# Reset the cluster
sudo kubeadm reset -f
sudo rm -rf ~/.kube
sudo systemctl restart containerd
```

4.2.10 Cleanup Flannel

If you need to uninstall `Flannel`, run the following command on the control plane.

```
# Cleanup Flunnel
kubectl delete -f https://raw.githubusercontent.com/flannel-io/flannel/master/
    Documentation/kube-flannel.yml
namespace "kube-flannel" deleted
```

4.2.11 Official Docs

This installation is outside the scope of the project and is only included to simplify the setup pipeline. Each hardware, OS, and kernel configuration may lead to different problems during the installation. Please refer to the official docs for troubleshooting.

- **Docker**. `https://docs.docker.com/manuals/`
- **Kubernetes**. `https://kubernetes.io/docs/setup/production-environment/tools/`

4.3 PREEMPT-FaaS

This section guides the user through the build and installation process of the `PREEMPT-FaaS` controller.

4.3.1 Build from Source

To build the controller, clone the repository on a new VM, or on your host, so that the `Docker Engine` required for building does not interfere with the cluster environment. Any host with Linux should be fine.

```
# Clone PREEMPT-FaaS repo (old name: PREEMPT-K8s)
git clone --branch aec --single-branch https://github.com/dessertlab/preempt-k8s.git
cd ./preempt-k8s
```

Once you cloned the repository, take a look at the `README.md` file in `~/preempt-k8s/` for a quick overview of the repository.
The build process requires `Docker` and `Docker Buildx`. Use the following commands to install them before building the controller. We also suggest a `Docker Hub` account to store the built image.
Start by installing Docker.

```
# Install prerequisites
sudo apt-get update
sudo apt-get install apt-transport-https ca-certificates curl gnupg-agent software-
    properties-common -y
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

# Add Docker repository
sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(
    lsb_release -cs) stable"

# Install Docker engine
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli -y

# Manage Docker as non-root user
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker

# Start Docker
sudo systemctl start docker

# Run a test Docker container
docker run hello-world

# List your Docker containers
docker ps -a

# Expected output
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
ba7dc8341bda hello-world "/hello" 8 seconds ago Exited (0) 3 seconds ago zen_dewdney
```

Then, install Docker Buildx.

```
# Install the Dockerx Plugin
sudo apt-get install docker-buildx-plugin -y

# Verify the install
docker buildx version
```

Build. Now build the controller following the guide in `~/preempt-k8s/controller/README.md`.

4.3.2 Deploy in a K8s cluster

On your control plane, install the Kubernetes Packet Manager `Helm`, create the `realtime` namespace, and clone the `PREEMPT-FaaS` repository wherever you want.

```
# Install Helm, the Kubernetes Package Manager
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-
    helm-4
chmod 700 get_helm.sh
./get_helm.sh
rm ./get_helm.sh

# Create realtime namespace
kubectl create namespace realtime

# Clone PREEMPT-FaaS repo (old name: PREEMPT-K8s)
git clone --branch aec --single-branch https://github.com/dessertlab/preempt-k8s.git
cd ./preempt-k8s
```

`~/preempt-k8s/helm/README.md` contains detailed instructions on how to properly configure and deploy, via Helm, the controller in your cluster. This is the suggested alternative.

`~/preempt-k8s/resources/README.md` guide you for manual configuration and installation without Helm.

For clarity, we also report here a quick installation example. First, configure the controller in `~/preempt-k8s/helm/values.yaml` by setting the reference to your Docker Hub `PREEMPT-FaaS` image. Replace the placeholder `<your-image-ref>`.

```
preempt_k8s:
  general:
    name: preempt-k8s
    namespace: realtime
  pod:
    restartPolicy: Always
    resources:
      limits:
        cpu: "2"
        memory: "3Gi"
      requests:
        cpu: "2"
        memory: "2Gi"
    securityContext:
      runAsUser: 0
      runAsGroup: 0
      fsGroup: 0
```

```
      allowPrivilegeEscalation: true
      privileged: true
      readOnlyRootFilesystem: false
      capabilities:
        add:
          - SYS_NICE
          - IPC_OWNER
          - SYS_ADMIN
          - MKNOD
          - SYS_RESOURCE
    container:
      name: preempt-k8s-controller
      image:
        repository: <your-image-ref>
        tag: 1.0.0
        pullPolicy: Always
      port: 80
  configMap:
    MIN_WATCHDOGS: "10"
    MAX_WATCHDOGS: "20"
    THRESHOLD: "3"
    EVENT_QUEUE: "/eventqueue"
    THREAD_CPU_PINNING: "false"
    RESOURCE_WATCHER_CPU_LIST: "0"
    POD_WATCHER_CPU_LIST: "1"
    SERVER_CPU_LIST: "2"
    STATE_UPDATER_CPU_LIST: "3"
    WATCHDOGS_CPU_LIST: "4,5"
```

Now deploy the configured controller via Helm.

```
# Deploy PREEMPT-FaaS through Helm
helm install preempt-k8s ./helm -n realtime

# Verify the deploy
kubectl get pods -n realtime
kubectl logs -f -n realtime preempt-k8s-0
```

4.4 Knative Patch

To deploy our patched version of Knative, first clone the repository.

```
# Install Brew Packet Manager
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
    install.sh)"
echo >> /home/test/.bashrc
echo 'eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv bash)"' >> /home/test/.
    bashrc
eval "$(/home/linuxbrew/.linuxbrew/bin/brew shellenv bash)"

# Install Ko tool
brew install ko

# Install Go
brew install go
```

```
# Install Docker
brew install docker

# Ensure containerd and runc are still the versions we installed (1.7.18)
containerd --version
runc --version

# Login to yout Docker Hub repository
docker login -u <your-username>

# Clone Knative patched version
git clone --branch aec --single-branch https://github.com/dessertlab/knative-crd-scaled.
    git
cd ./knative-crd-scaled
```

`~/knative-crd-scaled/README.md` contains a quick overview of this forked Knative repository. `~/knative-crd-scaled/DEVELOPMENT.md` guides you through the installation steps, including the `Knative Ingress` configuration (we employed `Kourier`). Pay attention to the guide, as it also explains how to deploy Knative with an older version of Kubernetes, in case you have installed one.

For extensive documentation on **Brew**, **Ko**, and **Go**, see:

- `https://brew.sh/`
- `https://ko.build/install/`
- `https://formulae.brew.sh/formula/go`

4.5 Storage and Monitoring Infrastructure

Follow the guide in `~/preempt-k8s/experiments/k8s-results-store/LONGHORN.md` to setup the **Longhorn Storage Manager** on all your worker nodes. The guide also provides manifest files to deploy the `Storage Class` and the `storage instance`. The files are located in the same directory: `~/preempt-k8s/experiments/k8s-results-store/`.

Once installed, you can deploy any Pod that mounts the storage unit. You can find Pod examples in `~/preempt-k8s/experiments/test-pods/`. This directory also provides a `README.md` to help you set them up.

Now, move to the monitoring directory: `~/preempt-k8s/monitoring/`. The `MONITORING.md` file provides a quick overview of this infrastructure and prerequisite installation.

Enable the **auditing logs generation** in your cluster following `~/preempt-k8s/monitoring/audit/AUDITING.md`.

Start the **OpenTelemetry** (**OTel**) **collector** following the guide in `~/preempt-k8s/monitoring/open-telemetry-collector/OTEL.md`.

Enable **trace generation** in your cluster using `~/preempt-k8s/monitoring/tracing/TRACING.md`.

You can also deploy the **Zipkin** trace collector and visualizer. We only employed tracing for initial studies, thus, it is not required to run the experiments. However, when we ran the experiments, we had this feature and Zipkin active.

Now, set up **Loki** to store audit logs and **Grafana** to visualize them. Grafana is not mandatory, but we left it running during our evaluation. Follow the instructions in `~/preempt-k8s/monitoring/grafana-loki/LOKI.md` and `~/preempt-k8s/monitoring/grafana-loki/GRAFANA.md`.

After all the installations, some components might need to be restarted.

```
# Check Monitoring components health
kubectl get pods -n observability

# For each pod provided by the above command run:
kubectl logs -n observability <pod-name>

# If any pod is not in "Running" state, or shows any relevant errors in the logs,
# kill the pod with the following command.
kubectl delete pod -n observability <pod-name>

# Since all Monitoring components are either Deployments or StatefulSet,
# Pods will be automatically restarted.
# Run again:
kubectl get pods -n observability
kubectl logs -n observability <pod-name> # For each restarted Pod
```

5 Run the Experiments

In this section, we provide a quick guide to reproduce our experimental campaign or simply run each experiment.

Note. We performed a preliminary analysis to find out the maximum interfering load we can use against our testbed. Using different hardware for the testbed should require re-running the sensitivity analysis.

5.1 Real Traffic Test Experiments

To reproduce the test, enter the `~/knative-crd-scaled/test/performance/benchmarks/real-traffic-test/` folder. The `README.md` file will describe each step to run the scripts.

In this document, we provide the steps to reproduce our experimental campaign. Check Section 2.1.2 for the tests reference parameters and the `README.md` in the folder to configure every single test.

Note. We restarted each VM in-between tests to avoid aging overhead.

1. Run the experiment for Vanilla K8s, at 5, 10, and 20 interfering resources.
2. Disable the auditing feature, see `~/preempt-k8s/monitoring/audit/AUDITING.md`.
3. Run the experiment for Vanilla K8s, at 20 interfering resources without monitoring.
4. Enable the auditing feature, see `~/preempt-k8s/monitoring/audit/AUDITING.md`.
5. Run the experiment for `PREEMPT-FaaS`, at 5, 10, and 20 interfering resources.
6. Uninstall `PREEMPT-FaaS`, set the `MIN_WATCHDOGS` and `MAX_WATCHDOGS` envs to 5, and re-install `PREEMPT-FaaS`, see `~/preempt-k8s/helm/README.md`.
7. Run the experiment for `PREEMPT-FaaS`, at 20 interfering resources with limited worker threads.
8. Uninstall `PREEMPT-FaaS`, set the `MIN_WATCHDOGS` env to 10 and the `MAX_WATCHDOGS` env to 20 (the original values), and re-install `PREEMPT-FaaS`, see `~/preempt-k8s/helm/README.md`.

9. Disable the auditing feature, see `~/preempt-k8s/monitoring/audit/AUDITING.md`.
10. Run the experiment for `PREEMPT-FaaS`, at 20 interfering resources without monitoring.
11. Enable the auditing feature, see `~/preempt-k8s/monitoring/audit/AUDITING.md`.
12. Run `analyze_kube_manager_results.py` on each Vanilla K8s experiment.
13. Run `analyze_preempt_k8s_results.py` on each `PREEMPT-FaaS` experiment.
14. Run `aggregate_results.py` on each experiment of both controllers.
15. Run `monitoring-overhead.py` fo the Vanilla K8s experiments at 20 interfering resources (providing the results with and without monitoring, respectively). IT LOGS RESULTS TO THE CONSOLE.
16. Run `monitoring-overhead.py` for the `PREEMPT-FaaS` experiments at 20 interfering resources (providing the results with and without monitoring, respectively). IT LOGS RESULTS TO THE CONSOLE.
17. Run `sensitivity-analysis.py` on all experiments of both controllers (5, 10, 20 interfering resources, nominal number of threads for `PREEMPT-FaaS`, and monitoring enabled).
18. Run `sensitivity-analysis-5-threads.py` on all experiments of both controllers (20 interfering resources, 5 threads for `PREEMPT-FaaS`, and monitoring enabled).

Note. Each experiment with 10 iterations should last 20/30 minutes.

Note. When testing Vanilla K8s, we uninstalled, via Helm, the `PREEMPT-FaaS` controller. Since the patched Knative needs the RTResource manifest to run, we manually deployed `~/preempt-k8s/resources/managed/RTResource.yaml`. It must be uninstalled to re-deploy `PREEMPT-FaaS` via Helm.

Note. Ensure the control plane and worker nodes are configured on the same time zone.

Expected Outcome. The PREEMPT-FaaS *orchestration times* should be noticibly lower than Vanilla K8s's, specifically under higher interfering loads. This is shown in Figure 6 in the paper.

- `~/knative-crd-scaled/test/performance/benchmarks/real-traffic-test/results/` contains the raw data of our experimental campaign (`preempt-k8s` and `kube-manager`).
- `sensitivity-analysis` contains the processed results of the paper. The raw data folders contain the output of all the data processing scripts employed.

5.2 vSwarm Experiments

To reproduce the test, enter the following folder:
`~/preempt-k8s/experiments/vSwarm-benchmarks/<benchmark-name>/<scale-type>/`.
The `README.md` file will describe each step to run the scripts.

In this document, we provide the steps to reproduce our experimental campaign. Check Section 2.1.3 for the tests reference parameters and the `README.md` in the folder to set every single test.

Note. We restarted each VM in-between tests to avoid aging overhead.

1. Set up the Database, if required.
2. Set up the invoker pods, KSVCs manifests, and endpoint configurations through the `setup-invoker-pods.sh` script. Set the flag to generate Deployments interfering with resources.

3. Run the experiment for Vanilla K8s, at 5, 10, and 20 interfering resources through the `experiment.sh` script. A VM restart might require re-creating the invoker Pods.
4. Remove the invoker Pods, KSVCs manifests, and endpoint configurations through the `setup-invoker-pods.sh` script.
5. Set up the invoker pods, KSVCs manifests, and endpoint configurations through the `setup-invoker-pods.sh` script. Set the flag to generate RTResources interfering resources.
6. Run the experiment for `PREEMPT-FaaS`, at 5, 10, and 20 interfering resources through the `experiment.sh` script. A VM restart might require re-creating the invoker Pods.
7. Remove the invoker Pods, KSVCs manifests, and endpoint configurations through the `setup-invoker-pods.sh` script.
8. Run `results.py` one each experiment.
9. Run `all-mean-latency.py` on all experiments to compute a CDF for each interfering value. IT LOGS SOME RESULTS, LIKE PERCENTILES VALUES, TO THE CONSOLE.
10. Run `sensitivity-analysis.py` on all experiments to compute a CDF for each interfering value.

Note. Each experiment with 10 iterations should last 15/20 minutes.

Note. When testing Vanilla K8s, we uninstalled, via Helm, the `PREEMPT-FaaS` controller. Since the patched Knative needs the RTResource manifest to run, we manually deployed `~/preempt-k8s/resources/managed/RTResource.yaml`. It must be uninstalled to re-deploy `PREEMPT-FaaS` via Helm.

Expected Outcome. The PREEMPT-FaaS *orchestration times* should be noticibly lower than Vanilla K8s's, specifically under higher interfering loads. The *throughput* of the applications should be higher for PREEMPT-FaaS in the RNN benchmarks, since the service struggles to serve the maximum achievable load. For Video Analytics, the effect is noticible at the highest stressload. The *E2E latency* is improved under PREEMPT-FaaS.

- `~/experiments/vSwarm-benchmarks/<benchmark-name>/<scale-type>/results/` contains the raw data of our experimental campaign (`preempt-k8s` and `kube-manager`).
- `sensitivity-analysis` contains the processed results of the paper. The other folders, including the raw data ones, contain the output of all the data processing scripts.

The plots reported in the paper in Figure 7, Figure 8, and Figure 9, respectively, can be seen in the following paths:

- `~/preempt-k8s/experiments/vSwarm-benchmarks/<benchmark-name>/<scale-type>/results/sensitivity-analysis/<your-current-results>/sensitivity_boxplot_pod_creation_delays.png`
- `~/preempt-k8s/experiments/vSwarm-benchmarks/<benchmark-name>/<scale-type>/results/sensitivity-analysis/<your-current-results>/sensitivity_boxplot_real_rps.png`
- `~/preempt-k8s/experiments/vSwarm-benchmarks/<benchmark-name>/<scale-type>/results/sensitivity-analysis/<your-current-results>/sensitivity_cdf_all_mean_latencies.png`

6 License

PREEMPT-FaaS repository is licensed under GNU General Public License v3.0, while the Knative patch is licensed under Apache License 2.0. The LICENSE files can be found in the repositories' default branches, "main" for PREEMPT-FaaS, and "knative-patch" for the Knative forked repository.

7 MD5 sum of the artifact

ccebaefeabb4c2885da1f723557e4dec

8 Size of the artifact

200MiB

Note. The MD5 sum and size of the artifact refers to a folder containing the PREEMPT-FaaS and Knative Patch repositories (the "aec" branch) without the ".git" folder.

References

1 Stefano Toscano. Knative patch, 2026. Accessed: 17th June 2026. URL: https://github.com/dessertlab/knative-crd-scaled/tree/aec.

2 Stefano Toscano. Preempt-faas, 2026. Accessed: 17th June 2026. URL: https://github.com/dessertlab/preempt-k8s/tree/aec.

3 vhive serverless. vswarm, 2025. Accessed: 17th June 2026. URL: https://github.com/vhive-serverless/vSwarm.