GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Function-as-a-Service in HPC: A Comparative Study of KSI- and K3s-Based Execution

Darius Heere

Supervisor: Dr. Jonathan Decker

Requested ECTS: 9

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2026

Abstract

Function-as-a-Service (FaaS) and serverless computing promise simplified deployment and fine-grained execution of short-lived workloads. While these paradigms are well established in cloud environments, it remains less clear how FaaS-like execution can be realized in an High-Performance Computing (HPC)-style context, which is typically shaped by batch schedulers such as Slurm and unprivileged user spaces. This is especially relevant for short-lived, independent workloads, where orchestration overhead can form a substantial part of the total execution cost.

We investigate FaaS-like execution on an HPC-style platform based on Slurm and Kind Slurm Integration (KSI) and compare it with K3s as a native lightweight Kubernetes baseline. Both execution environments are set up in a comparable way and evaluated using two benchmark classes: a tiny relay latency benchmark with very little per-invocation work and a hybrid Extract, Transform, Load (ETL) micro-batch benchmark for more complex but still short-lived shard-based processing. The study focuses on warm-phase behavior and separates startup/setup overhead from the main cross-platform comparison.

The results show that the relative importance of platform overhead depends strongly on workload shape. For the tiny relay benchmark, the observed behavior is dominated by Kubernetes-level orchestration, and KSI shows lower warm job latency and lower warm makespan than K3s. For the hybrid ETL benchmark, useful in-container work accounts for a substantial share of total invocation time, and both platforms become much closer in warm invocation behavior. Overall, FaaS-like execution on an HPC-style platform is feasible and can be competitive for an important class of short-lived, independent workloads, while the observed results also highlight the influence of workload shape and execution model.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☑ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☑ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Abbreviations

FaaS Function-as-a-Service

HPC High-Performance Computing

KSI Kind Slurm Integration

OCI Open Containers Initiative

NFS Network File System

ETL Extract, Transform, Load

JSON JavaScript Object Notation

JSONL JSON Lines

OS Operating System

NAT Network Address Translation

I/O Input/Output

VNFS Virtual Node File System

VM Virtual Machine

PXE Preboot Execution Environment

DNS Domain Name System

1 Introduction

1.1 Motivation

FaaS and serverless computing have become attractive execution models because they simplify the deployment of short-lived, event-driven workloads and shift much of the operational complexity to the underlying platform [Wen+23]. In cloud environments, these paradigms are already well established, but their transfer to an HPC-related context is far less straightforward. HPC systems are typically organized around batch schedulers such as Slurm, shared file systems and unprivileged user environments, rather than around native FaaS frameworks or long-running cloud control planes.

At the same time, there is growing interest in bringing cloud-native workload models into HPC systems without manually adapting them to classic batch job patterns, for example by enabling Kubernetes-based execution on top of Slurm [Dec+25]. This is especially relevant for short-lived, independent workloads, where orchestration overhead can itself become a substantial part of total execution time, as emphasized in the broader serverless literature [SBW19], [Liu+23]. Our goal is therefore not to reproduce the full functionality of a production FaaS framework, but to evaluate whether key FaaS-related execution semantics can be meaningfully realized in an HPC-oriented setting.

1.2 Research Questions and Scope of Objectives

Our report is guided by two research questions that define its direction and scope.

- RQ1: Which platform-level adaptations are required to deploy and operate KSI on a Slurm/Warewulf testbed with HPC semantics?
- RQ2: How does KSI-based execution compare with native K3s for FaaS-like workloads under a fair benchmark design?

1.3 Contributions and Report Structure

This report contributes an empirical study of FaaS-like execution in an HPC-related context. More specifically, it compares two execution paths: a native lightweight Kubernetes baseline in the form of K3s, and an HPC-style Slurm-based platform extended by KSI. The results show that platform behavior depends strongly on the workload shape, with larger differences for tiny orchestration-sensitive invocations and much closer behavior for the more substantial shard-based ETL workload. In this way, the report provides a systems-oriented perspective on how key FaaS-related execution semantics can be approximated and comparatively evaluated in an HPC-oriented setting.

After this introduction, the report provides the conceptual background for the later analysis in Section 2. The experimental platforms and fairness considerations of the two testbeds are then introduced in Section 3, followed by the technical adaptations required to make KSI operational on the HPC-oriented platform in Section 4. Based on this foundation, the experimental evaluation is presented in Section 5, including the two benchmarks and their comparative summary. Afterwards, the findings are interpreted in the broader FaaS and HPC context in Section 6, before the report closes with the conclusion in Section 7.

2 Background

This section introduces the conceptual foundations for the later analysis, including the restricted FaaS semantics used throughout the report, the role of Kubernetes Jobs as an approximation of FaaS invocations, and the positioning of KSI as an enabling step toward FaaS in HPC.

2.1 FaaS Semantics

This report uses the term Function-as-a-Service (FaaS) in a restricted semantic sense. Rather than referring to the full feature set of commercial serverless platforms, we focus on those characteristics of FaaS that are most relevant for our study.

In the literature, FaaS is commonly described as a model in which short-lived functions are executed on demand in isolated runtime instances, while important parts of provisioning and lifecycle handling are managed by the platform rather than by the user [SBW19], [Wen+23], [Liu+23], [Sha+20]. This includes aspects such as activation, scheduling, execution and the cleanup of runtime instances after completion. Another common characteristic is that functions are typically stateless or nearly stateless across invocations. As a result, durable state and larger forms of communication are usually externalized to storage systems or related services [SBW19], [Wen+23].

Orchestration overhead is a central concern in the FaaS literature. Startup behavior, scheduling effects and other platform-level costs can have a large influence on end-to-end latency, especially for short-running invocations [SBW19], [Liu+23], [Sha+20]. This means that the useful work performed inside the function itself is only one part of what determines performance in this computing model.

This observation directly motivates the restricted definition used in this report. The workloads considered later are intentionally lightweight and short-lived, so platform overhead is not just a secondary effect, but part of the central object of study. For this reason, we narrow the semantic frame of FaaS to:

Lightweight, mostly independent invocations of short-lived functions with externalized input and output, limited internal computation and platform-managed scheduling and lifecycle handling.

This restricted framing is still close enough to be clearly identifiable as FaaS, while keeping the scope of the report manageable. It matches the type of workloads used in our benchmarks and allows us to study how FaaS-like execution conditions can be brought into HPC-oriented platforms.

2.2 Kubernetes Job Invocations as an Approximation of FaaS Execution

We use the Kubernetes Job resource as an approximation of one FaaS invocation unit. In our benchmarks, this means *one invocation = one Job*, which gives us one concrete and measurable unit of execution.

A Kubernetes Job creates one or more Pods[1], which are the smallest deployable compute units that a user can create and manage. Pods have a managed lifecycle such as Pending → Running → Succeeded or Failed[2]. At the Job level, Kubernetes additionally manages the creation of Pods, tracks successful completions, retries failed executions if needed and can clean up Pods after completion[3]. In this way, a Job captures important parts of the platform-managed activation, scheduling, execution and lifecycle handling that we identified as central FaaS semantics in the previous subsection.

This approximation also fits the remaining elements of our restricted semantic framing. The lightweight isolation provided by Pods supports the execution of mostly independent computations. Pods can also access externalized input and output through shared storage or related services, which matches our assumption that durable state and communication are largely externalized. Finally, the overhead associated with Job creation is still small enough to treat a Job as a plausible container for one invocation in the scope of this work, even though it is not identical to a production FaaS runtime.

One could argue that each invocation should be mapped directly to a Pod rather than to a Job. This would reduce some of the pure control overhead between invocation request and code execution. However, a Pod by itself does not encode the higher-level platform management and lifecycle semantics that are relevant in the present context. In particular, completion tracking, retry behavior and the run-to-completion semantics would then have to be handled more explicitly outside the resource itself. Using Jobs therefore keeps more of the invocation handling on the platform side, which is more closely aligned with the FaaS semantics discussed above.

One could also argue for using Kubernetes Indexed Jobs[4], since they support many lightweight and independent Pods within a single Job and may therefore reduce overhead further. However, Indexed Jobs shift the abstraction from request-oriented invocation semantics toward batch-array semantics. This makes them less suitable as an approximation of individual FaaS invocations.

In summary, we use Kubernetes Jobs because they provide a minimal platform-managed invocation model that captures the subset of FaaS semantics relevant to this study. Compared to established FaaS frameworks, this model still lacks higher-level features such as event routing, autoscaling policies and other framework-specific abstractions mentioned in works like [Wen+23] and [Sha+20].

2.3 Kubernetes Enablement as a Step Toward FaaS in HPC

Kubernetes is an important substrate for multiple established open-source serverless platforms [Wen+23], [Liu+23]. For example, Knative is a Kubernetes-based platform for building, deploying and managing serverless workloads[5]. OpenFaaS supports deployment

[1] *Kubernetes Pods.* URL: `https://kubernetes.io/docs/concepts/workloads/pods/` (visited on Mar. 25, 2026).

[2] *Kubernetes Pod Lifecycle.* URL: `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/` (visited on Mar. 25, 2026).

[3] *Kubernetes Jobs.* URL: `https://kubernetes.io/docs/concepts/workloads/controllers/job/` (visited on Mar. 25, 2026).

[4] *Kubernetes Indexed Job.* URL: `https://kubernetes.io/docs/tasks/job/indexed-parallel-processing-static/` (visited on Mar. 25, 2026).

[5] *Knative Technical Overview.* URL: `https://knative.dev/docs/` (visited on Mar. 26, 2026).

on Kubernetes clusters including K3s[6], and Nuclio also supports deployment on Kubernetes[7]. In this sense, Kubernetes provides part of the infrastructure basis on which FaaS-style execution can be built.

In conventional Slurm-managed HPC systems, users operate with restricted permissions [KSB17] and therefore cannot simply deploy and manage Kubernetes-based serverless frameworks on their own [Dec+25]. From a systems perspective, this means that the first step in bringing FaaS into such environments is to make the required orchestration substrate, i.e. a Kubernetes environment, available under these operational constraints.

KSI [Dec+25] addresses exactly this constraint. By allowing a Kubernetes environment to be created inside a Slurm job with limited permissions, KSI provides this substrate layer. It can therefore be understood as an enabling infrastructure step toward FaaS in HPC, because it makes Kubernetes-based execution possible inside a Slurm/Warewulf-based setting.

At the same time, KSI is only a necessary but not sufficient step toward full FaaS support in such systems. Higher-level serverless features like event routing, autoscaling, scale-to-zero behavior and framework-specific function lifecycle management would still have to be provided by an additional FaaS framework on top of Kubernetes.

Because of the scope of this project, we stop at the substrate layer and evaluate approximated FaaS workloads directly on the respective Kubernetes platform. This remains defensible under the semantic framing developed in Sections 2.1 and 2.2, and allows us to analyze how FaaS-like execution behaves in the setting considered here.

[6] *OpenFaaS Deployment*. URL: `https://docs.openfaas.com/deployment/` (visited on Mar. 26, 2026).

[7] *Nuclio on Kubernetes*. URL: `https://docs.nuclio.io/en/latest/setup/k8s/` (visited on Mar. 26, 2026).

3 Experimental Platforms

This section describes the two cluster environments used for the experimental evaluation. Our fair cross-platform comparison requires that they differ primarily in their orchestration model, not in their underlying resources or storage paths.

3.1 Testbed Overview and Fairness Choices

This work evaluates FaaS-like workloads on two small, controlled cluster environments. One cluster represents a cloud-oriented platform and the other an HPC-oriented platform. Both are deployed within the same OpenStack environment[8] and are intentionally kept as comparable as possible in their basic hardware and network setup.

In total, the testbed consists of six Virtual Machine (VM)s, which are split into two independent 3-node clusters, as shown in Figure 1. Across both platforms, all nodes use the same `m1.large` flavor with 4 vCPUs and 8 GB of RAM. Rocky Linux 9[9] is used as the base Operating System (OS) throughout the testbed. On the cloud-oriented side, the nodes are provisioned as standard OpenStack instances from a Rocky Linux 9 image. On the HPC-oriented side, the head node is also a standard instance, whereas the compute nodes are diskless and network-boot into a centrally managed Rocky Linux 9 userspace/root filesystem image.

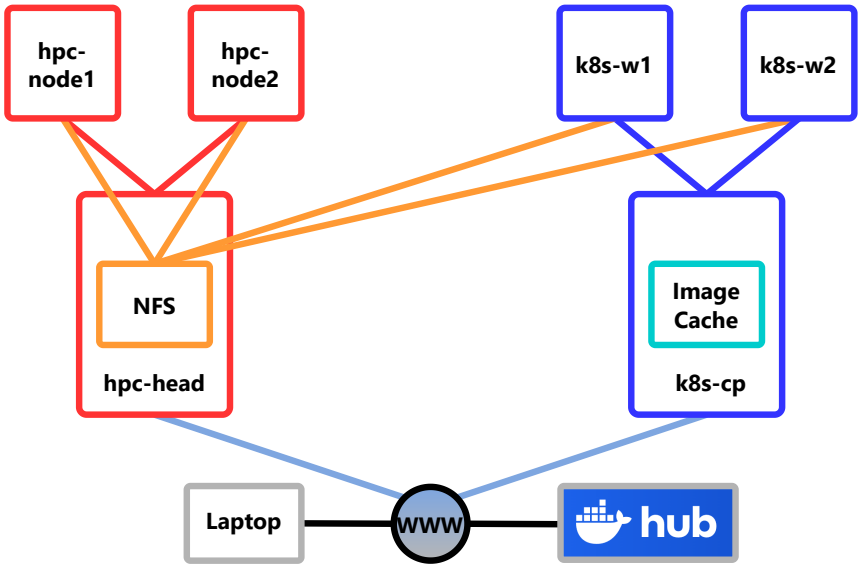


Figure 1: HPC-oriented and cloud-oriented cluster stack overview.

To separate concerns and reduce interference during later measurements, the testbed uses three networks. The first is a routed network with Network Address Translation (NAT), which is used for general node communication and management access. The `hpc-head` node and all three Kubernetes (K3s) nodes (`k8s-cp`, `k8s-w1` and `k8s-w2`) are assigned to this network. Due to the OpenStack configuration used in our project, only

[8] *OpenStack: Open Source Cloud Computing Infrastructure (Project Overview)*. URL: `https://www.openstack.org/` (visited on Mar. 26, 2026).

[9] *Rocky Linux Documentation: Rocky Linux 9*. URL: `https://docs.rockylinux.org/9/` (visited on Mar. 26, 2026).

nodes with an attached Floating IP actually have internet access, namely `hpc-head` and `k8s-cp`.

The second is a non-routed Layer-2 network dedicated to the iPXE/Preboot Execution Environment (PXE) boot and provisioning of the diskless HPC compute nodes. This means that only the nodes of the HPC-oriented cluster are assigned to this network (`hpc-head`, `hpc-node1`, `hpc-node2`).

The third is a non-routed Layer-2 storage network used for Network File System (NFS) traffic. The NFS server runs on `hpc-head` and both platforms access the same shared export over this storage network. This ensures that benchmark-related Input/Output (I/O) follows the same storage path on both platforms and is isolated from provisioning and routed management traffic.

A further fairness choice is that benchmark workloads are executed only on the worker or compute nodes of the respective platform. This means that `hpc-head` and `k8s-cp` are excluded from workload placement, since they are responsible for management and cluster services. This helps to ensure that measurements reflect execution on dedicated worker resources that are idle outside the benchmarks.

Overall, the testbed is therefore designed so that the two platforms differ primarily in their orchestration model and system semantics, while basic hardware characteristics and the storage path for benchmark data remain as comparable as possible.

3.2 Cloud-Oriented Platform

The cloud-oriented platform is implemented as a lightweight Kubernetes[10] cluster based on K3s[11]. In contrast to the HPC-oriented platform described later, this environment provides a persistent Kubernetes control plane and a native container orchestration stack for running benchmark workloads.

The node `k8s-cp` hosts the Kubernetes control-plane components and is configured so that it does not execute user workloads. The two worker nodes `k8s-w1` and `k8s-w2` run the K3s agent and are responsible for pulling container images and executing benchmark workloads under Kubernetes scheduling. This separation ensures that workload execution is restricted to dedicated worker resources and that control-plane activity remains isolated from the benchmark runs.

Networking follows the general testbed design introduced in Section 3.1. Cluster control traffic and node-to-node communication use the routed network, while benchmark-related I/O is placed on the dedicated storage network. All three Kubernetes nodes mount the shared NFS export from `hpc-head` on the host OS. As a result, benchmark data reads and writes follow the same storage path and network topology as on the HPC-oriented platform.

Since the worker nodes do not have direct outbound internet access during normal operation, container image retrieval is realized through a local Docker Registry (v2) deployed on `k8s-cp` as an OS-level service. This registry is configured as a pull-through cache for Docker Hub. The worker nodes are set up to fetch images through this internal

[10] *Kubernetes Documentation: Overview.* URL: `https://kubernetes.io/docs/concepts/overview/` (visited on Mar. 26, 2026).

[11] *K3s Documentation: K3s - Lightweight Kubernetes.* URL: `https://docs.k3s.io/` (visited on Mar. 26, 2026).

registry, which allows standard image references to be used in workload manifests (e.g. `docker.io/namespace/image:tag`) while avoiding direct external image pulls from the worker nodes.

In this way, the cloud-oriented platform provides the persistent Kubernetes baseline of the testbed. It serves as the native Kubernetes environment against which the later KSI-enabled execution model on the HPC-oriented side can be compared.

3.3 HPC-Oriented Platform

The HPC-oriented platform is mostly based on the setup used in the High-Performance Computing System Administration course[12]. It follows a conventional cluster design with a central head node and diskless compute nodes.

The node `hpc-head` is the central management node of the platform and provides the cluster-wide services required for operation. This includes the Warewulf[13] provisioning stack with DHCP, TFTP and iPXE support for the diskless compute nodes. The head node also provides the NFS service that is used as the shared storage backend by both platforms, as well as Domain Name System (DNS) resolution[14] and a Squid proxy[15] for the compute nodes. The Squid proxy allows the otherwise restricted compute nodes to access the internet for downloading container images from Docker Hub.

The compute nodes `hpc-node1` and `hpc-node2` are diskless and do not behave like normal long-lived VMs. Instead, they boot into a centrally defined image and can be reprovisioned into a known state via the dedicated iPXE/PXE network. They load a Warewulf-provided Virtual Node File System (VNFS) derived from a Rocky Linux 9 userspace/container filesystem, while node-specific configuration is applied through Warewulf overlays instead of by configuring each node manually. The compute nodes mount `/nfs` during boot, so that benchmark-related input and output follow the same storage path as on the cloud-oriented platform.

Workload scheduling and resource management are handled by Slurm[16]. The Slurm controller (`slurmctld`) accepts job submissions, tracks node states and decides where and when jobs run. For accounting and the storage of job-related metadata, `slurmdbd` is used together with a MariaDB backend[17]. Authentication within the cluster is handled via Munge[18]. A shared Munge key is distributed across the head and compute nodes so that the Slurm components can trust each other.

Overall, this platform represents the HPC-oriented side of the testbed. The KSI enablement described in the following section is built on top of it.

[12] *Practical: High-Performance Computing System Administration*. URL: `https://hps.vi4io.org/teaching/autumn_term_2025/hpcsa` (visited on Mar. 26, 2026).

[13] *Warewulf: Stateless and Diskless Cluster Provisioning*. URL: `https://warewulf.org/` (visited on Mar. 26, 2026).

[14] *BIND 9 Administrator Reference Manual*. URL: `https://bind9.readthedocs.io/` (visited on Mar. 26, 2026).

[15] *Squid Documentation*. URL: `https://www.squid-cache.org/Doc/` (visited on Mar. 26, 2026).

[16] *Slurm Workload Manager: Overview*. URL: `https://slurm.schedmd.com/overview.html` (visited on Mar. 26, 2026).

[17] *Rocky Linux Documentation: MariaDB database server*. URL: `https://docs.rockylinux.org/9/guides/database/database_mariadb-server/` (visited on Mar. 26, 2026).

[18] *MUNGE (MUNGE Uid 'N' Gid Emporium): Overview*. URL: `https://dun.github.io/munge/` (visited on Mar. 26, 2026).

4 Enabling KSI on the HPC Platform

KSI, as proposed by Decker et al. [Dec+25], provides a way to start temporary, rootless Kubernetes environments on Slurm-allocated nodes. The underlying Slurm/Warewulf-based HPC platform was already set up as described in Section 3.3. However, running KSI on top of this platform further requires suitable support for rootless Open Containers Initiative (OCI) execution, writable node-local runtime state and a worker-side service environment compatible with this execution path. These requirements lead to both straightforward installation steps and non-trivial system adaptations.

4.1 KSI Setup on the Existing HPC Platform

Table 1 summarizes the software baseline established on both KSI-enabled worker nodes. While several components could be added straightforwardly to the existing Warewulf node image, others required additional configuration to enable KSI to run correctly on the worker nodes.

Component	Version
OS	Rocky Linux 9.7 (Blue Onyx)
Slurm	25.05.3
containerd	2.2.1
nerdctl	2.2.1
RootlessKit	2.3.5
slirp4netns	1.3.3
bypass4netns	0.4.2
libseccomp	2.5.2
Kind	0.29.0
kubectl (client)	1.33.2
liqoctl (client)	1.0.1

Table 1: Relevant software versions on the KSI-enabled worker nodes (`hpc-node1` and `hpc-node2`).

A substantial part of the required software environment can be established by installing user-space components on top of the existing Warewulf node image. To keep this process controlled, the existing image is cloned into a separate KSI-oriented variant, which is first tested on a single worker node before being applied to the second one. The installed components include the rootless container runtime stack (`containerd`, `nerdctl`, `rootlesskit`), rootless networking components (`slirp4netns`, `bypass4netns`) and the Kubernetes-related tools (`kind`, `kubectl`, `liqoctl`). Together, these components form the user-space foundation of the KSI runtime stack on the worker nodes.

Not all elements of the KSI environment are established by package installation alone. A dedicated non-root user called `ksiuser` is used as the execution user for the rootless runtime path. This includes the required subordinate UID/GID mappings and the per-user runtime directories and service environment needed by the rootless container stack. These configuration steps are necessary because the rootless execution model depends on a correctly prepared user-level environment on each worker node.

Correct installation and basic functionality of each component were verified through version checks and minimal runtime tests, followed by early smoke tests of the rootless container and `kind`-based Kubernetes path. This bottom-up verification is important because it isolates failures at individual layers before they compound in a full KSI run. These early checks also show that KSI is not a plug-and-play extension of the existing HPC setup. Although the software stack can be installed successfully, the first runtime experiments reveal additional constraints in the worker environment. These constraints and the corresponding system adaptations are discussed in the following subsection.

4.2 Identified Runtime Constraints and Required Adaptations

Rootless container startup on the Warewulf workers does not work in the original configuration. Even a minimal `nerdctl` test fails with `pivot_root .: invalid argument`. The cause is that the original Warewulf root mode is not compatible with the rootless OCI startup path. The first required adaptation is therefore the switch to `root=tmpfs` for the Warewulf worker nodes[19], which makes rootless OCI container startup possible on the workers.

The second constraint is the lack of sufficient local writable space for the rootless KSI path. During image pulls and `kind` startup, temporary image layers, runtime metadata, CNI state and temporary Kubernetes node state have to be written locally on the worker. In the original setup, the available local temporary area is too small for this. We therefore increased the `tmpfs`-backed `/tmp` from 2 GB to 4 GB on the worker nodes to provide the required local writable capacity.

Beyond capacity, the placement of runtime state is also a decisive constraint. The default rootless paths are unsuitable, because relevant parts of the rootless runtime state would otherwise end up on NFS-backed paths, which is not appropriate for local container and networking state. The corresponding adaptation is the relocation of the relevant runtime directories (rootless runtime data, CNI state and local temporary container/image state) from NFS to local `tmpfs`-backed storage. This ensures that runtime state remains local to the worker nodes instead of being placed on shared storage.

Rootless networking also required the presence of the kernel modules `ip6_tables`, `ip6table_nat`, `ip_tables` and `iptable_nat`. These modules were not loaded in the initial worker image, so a persistent `/etc/modules-load.d/iptables.conf` entry was added to the KSI-oriented image. This ensures that the networking prerequisites for rootless `nerdctl`/`kind` execution are present after boot.

The final constraint is that the existing `slurmd.service` on the worker nodes is incompatible with the KSI execution path. The older service style causes startup and scope-related problems for the rootless container environment. We updated the worker-side `slurmd` unit to a modern systemd-native form and adjusted the related Slurm settings accordingly. This completes the system adaptations required for operational KSI runs.

[19] *Fix as proposed in Ephemeral Kubernetes GitHub.* URL: `https://github.com/gwdg/pub-2025-ephemeral-kubernetes` (visited on Mar. 28, 2026).

4.3 Resulting Execution Model and Benchmark Implications

The resulting KSI setup on the HPC-oriented cluster is not a native multi-node Kubernetes cluster in the classical sense. Instead, each Slurm-allocated worker node provides its own temporary KSI execution domain, while Slurm remains the outer resource manager. This distinction is important for the benchmark interpretation, because it defines what is actually being compared to the cloud-like platform.

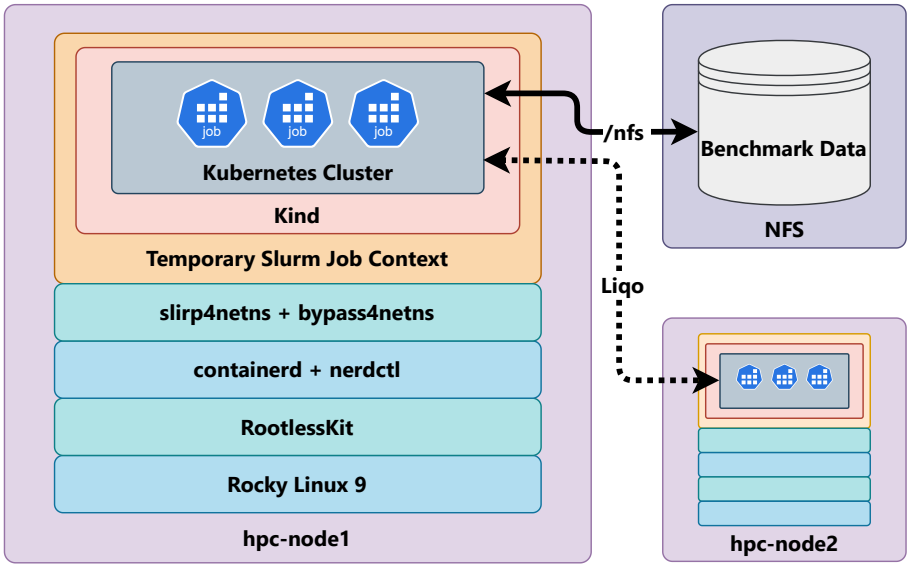


Figure 2: Layered software stack and execution model of a KSI-enabled HPC worker node.

The resulting execution model of the KSI-enabled HPC platform is shown in Figure 2. On each Slurm-allocated worker node, KSI uses the prepared rootless container stack together with `kind` to start a temporary Kubernetes environment. The Kubernetes layer is created on demand inside the Slurm job and removed again after the workload finishes. In this sense, the setup follows the intended "Over-model" [Dec+25] idea of KSI: Kubernetes is placed on top of the existing HPC scheduling environment instead of replacing it.

Liqo provides the networking layer that connects the temporary per-node Kubernetes environments into one logical cluster [Dec+25]. In Figure 2, it appears only as a dashed connection because it was installed and tested but is not used in the benchmark design. For the benchmarks, the workload is split into two independent halves, with one half assigned to each worker node. This design ensures balanced NFS access across nodes and avoids dependence on cross-node communication. Both platforms use this two-partition structure, but they realize it differently: K3s schedules both halves through one persistent control plane, whereas KSI runs each half in its own temporary Kubernetes environment.

5 Experimental Evaluation

This chapter presents the experimental evaluation of FaaS-like workloads on the two platforms. It begins with the shared methodology and measurement approach, followed by the two benchmarks and their comparative summary.

5.1 Common Methodology and Measurement Approach

The common benchmark structure follows directly from the platform setup in Section 3 and the KSI execution model discussed in Section 4.3. Both platforms are therefore evaluated as two independent execution partitions rather than as one shared multi-node cluster. In both benchmarks, invocations are modeled as short-lived and independent units of work, without cross-invocation communication or long-running worker processes. Following the approach discussed in Section 2.2, one Kubernetes Job represents one logical invocation. This keeps the experiments close to the function-oriented execution model of FaaS while remaining practical on both platforms.

The measurement approach differentiates between multiple timing layers. The warm per-invocation Job latency measures the time from Kubernetes Job submission to Job completion via wall-clock timestamps, once the execution environment is already available. The inner function time is measured inside the worker container using a local elapsed-time clock and captures only the actual application work. A worker-local warm makespan is determined for each partition as the time from the first Job submission on that partition until the last assigned Job has completed. The platform warm makespan is then defined as the maximum of the two partition-local makespans.

KSI startup and setup overhead is measured separately via wall-clock timestamps, from the start of the KSI launch path until the temporary execution environments are ready for benchmark Job submission. This cost is analyzed separately because it captures environment bootstrap on the KSI side, while K3s is benchmarked as an already running persistent cluster. The main cross-platform analysis therefore follows a warm-versus-warm interpretation, while KSI startup overhead is discussed as a distinct platform characteristic.

Both platforms use the same benchmark-specific OCI worker images and the same shared `/nfs` path for benchmark inputs and outputs. We generate the datasets deterministically so that both platforms process exactly the same invocation units. All experiments were repeated several times under controlled conditions. Between repetitions, we cleared the page caches on the K3s workers. On the KSI side, the same cache clearing was combined with cleanup of temporary container runtime state, since repeated temporary-cluster execution could otherwise lead to memory pressure on the local `tmpfs`-backed writable storage. These measures ensure comparable starting conditions across all repetitions.

5.2 Benchmark 1: Tiny Relay Latency Benchmark

Benchmark 1 represents the minimal realization of FaaS-like execution considered in this report. Its purpose is to isolate the behavior of very short-lived invocations whose actual application work is intentionally negligible, so that orchestration overhead becomes the dominant component and can be clearly observed.

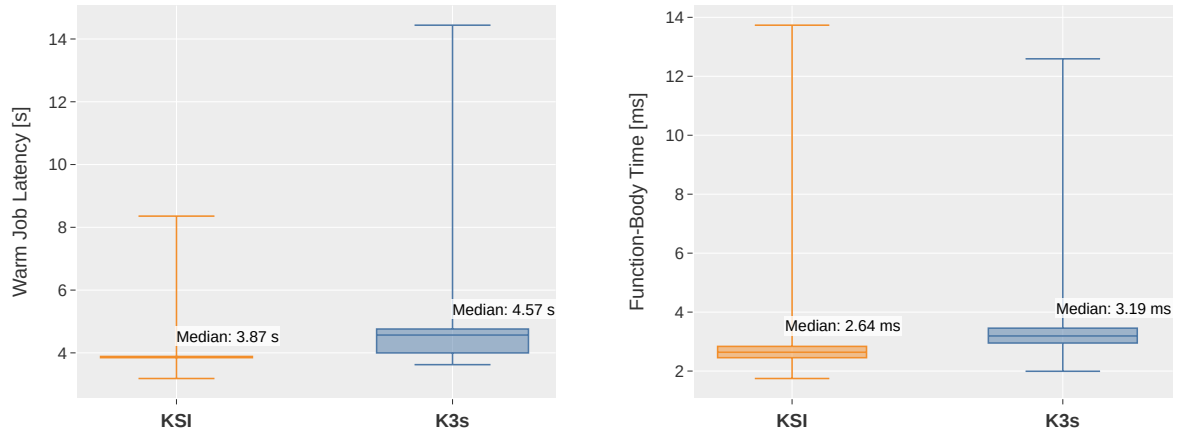
5.2.1 Workload Definition

Each invocation in Benchmark 1 performs a deliberately minimal unit of work. The worker container reads one small JavaScript Object Notation (JSON) input object from shared storage, parses it and computes a `sha256` hash over the payload field together with lightweight runtime metadata. It then writes one compact JSON result and terminates. The benchmark therefore follows a strict one-to-one mapping between logical invocation and execution instance: one input file corresponds to one Kubernetes Job, one container execution and one output file.

Both platforms use the same dataset, which is generated once on the shared `/nfs` path. It consists of deterministic JSON input files with deterministic naming and content generation. Each file represents one logical invocation and is only a few kilobytes in size, so that the benchmark remains dominated by short-lived I/O and orchestration effects rather than by substantial application work.

In the benchmarked two-node runs, a subset of 512 invocations is processed per repetition. Assignment to the two execution partitions is deterministic, so that partition 0 handles the even invocation IDs and partition 1 the odd ones. On the K3s side, these two partitions are assigned to the two worker nodes. On the KSI side, they are realized within one Slurm allocation spanning the same two worker nodes, where each node starts its own temporary Kubernetes environment and executes the Jobs assigned to its partition. Because the benchmark contains no cross-invocation communication and no shared mutable state beyond the common read/write path on `/nfs`, this two-partition structure is sufficient for the workload and aligns with the execution model established in Section 4.3.

5.2.2 Benchmark Results



(a) Warm end-to-end latency of Kubernetes Job invocations measured from Job submission to observed completion.

(b) Function-body time inside the container of each Kubernetes Job invocation.

Per stack *(KSI/K3s)*: $n = 2560$ successful invocations (5 runs × 512 Jobs) in the two-node benchmark setup. Only warm-phase execution is shown, i.e. after the execution environments were ready for benchmark Job submission; KSI startup/setup overhead is reported separately.

Figure 3: Benchmark 1 results for the Tiny Relay Latency Benchmark workload.

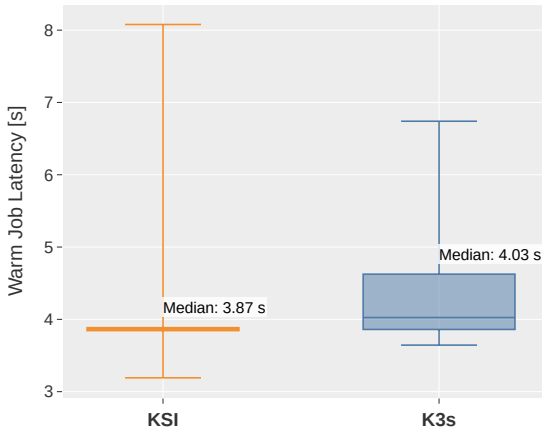
The warm-phase results in Figure 3 confirm that the benchmark behaves as intended: on both platforms, the end-to-end latency of one Kubernetes Job invocation is substantially larger than the time spent inside the worker container itself. The total cost of one invocation is therefore dominated by Kubernetes-level orchestration rather than by the actual function body. Within this warm execution phase, KSI shows consistently lower Job latencies than K3s.

The function-body measurements remain in the low millisecond range on both platforms, whereas the warm end-to-end Job latencies are in the range of several seconds. The two timing layers therefore differ by roughly three orders of magnitude, which confirms that Benchmark 1 is primarily an orchestration-sensitive benchmark. Figure 3 also shows that the warm Job-latency distribution of KSI is shifted downward relative to K3s, while the function-body times differ only slightly in absolute terms.

Metric	Stack	Median	Mean	Std	Min	Max
Warm makespan [s]	KSI	1038.10	1038.33	1.56	1036.15	1040.14
Warm makespan [s]	K3s	1202.80	1198.86	11.15	1179.63	1206.93
Warm p50 job latency [s]	KSI	3.87	3.87	0.00	3.87	3.87
Warm p50 job latency [s]	K3s	4.58	4.56	0.03	4.51	4.59
Warm p95 job latency [s]	KSI	3.92	3.93	0.02	3.91	3.96
Warm p95 job latency [s]	K3s	4.95	4.97	0.05	4.91	5.03
Startup/setup overhead [s]	KSI	49.98	49.96	1.95	48.05	53.05
Startup/setup overhead [s]	K3s	N/A	N/A	N/A	N/A	N/A

Table 2: Warm run-level metrics across the five 512-invocation benchmark runs. Startup/setup overhead is not reported for K3s because the cluster is persistent.

The run-level summary in Table 2 confirms the same pattern across all five two-node repetitions. The median warm makespan is lower for KSI than for K3s, and the same holds for both warm p50 and warm p95 Job latency. In addition, the run-to-run variation is comparatively small, so that the observed difference cannot be attributed to a single outlier repetition.



Per stack *(KSI/K3s)*: $n = 512$ successful invocations (1 run $\times$ 512 Jobs) in the warm single-node control setup. Job latency was measured from Kubernetes Job submission to observed completion; KSI startup/setup overhead is not included.

Figure 4: Warm end-to-end latency of Kubernetes Job invocations for the control run.

The single-node control run restricts each platform to one worker node and 512 invocations, as shown in Figure 4. KSI again shows the lower warm Job-latency median, yet the difference is less pronounced than in the two-node setup. This result is consistent with the main benchmark while indicating that the stronger two-node difference should be interpreted in light of the differing execution structures discussed in the following subsection.

5.2.3 Benchmark Discussion

The results of Benchmark 1 show that the workload behaves as intended: It makes the cost of short-lived, function-style activations visible at the orchestration layer and is therefore well aligned with the FaaS context of this report. The measured function-body times remain in the low millisecond range on both platforms, whereas the warm end-to-end latency of Kubernetes Job invocations lies in the range of multiple seconds. The useful application work is therefore negligible compared to the total invocation cost.

Given that both clusters are warm before benchmark start, KSI shows consistently lower Job latency and lower makespan than K3s. This pattern is visible in the aggregated boxplots (Figure 3) and in the run-level summary across all five benchmark repetitions (Table 2). Both platforms execute the same worker logic, so the small difference in function-body time reflects minor environmental effects such as I/O behavior and runtime overhead.

This interpretation should be limited to the present benchmark setting. The observed warm-phase advantage of KSI in Benchmark 1 does not mean that KSI generally provides "faster Kubernetes". In the current setup, the two execution partitions on the KSI side are handled by two independent temporary Kubernetes environments, whereas K3s realizes the same logical split within one persistent cluster. A plausible interpretation is therefore that the current KSI realization benefits from distributing orchestration work across two separate execution domains, whereas K3s routes both partitions through a single shared control plane.

The single-node control run supports this interpretation: KSI again shows the lower warm Job latency, but the gap is smaller than in the two-node benchmark shape. This suggests that the stronger difference in the main experiment is related to the two-partition execution structure rather than to the worker code or dataset itself. At the same time, KSI startup and setup overhead remains a substantial platform cost, with a median duration of 49.98 s, and must therefore be considered separately. Benchmark 1 thus supports the conclusion that KSI-based execution can be competitive for tiny, short-lived, FaaS-like invocations in the warm phase, but that this finding is specific to the current benchmark design and execution model.

5.3 Benchmark 2: Hybrid ETL Micro-Batch Benchmark

Benchmark 2 complements Benchmark 1 by representing a more realistic approximation of FaaS-like execution in this report. Its purpose is to evaluate short-lived invocations whose function body performs meaningful ETL-style work on one independent data shard, so that orchestration overhead remains relevant but no longer dominates the workload as strongly as in Benchmark 1.

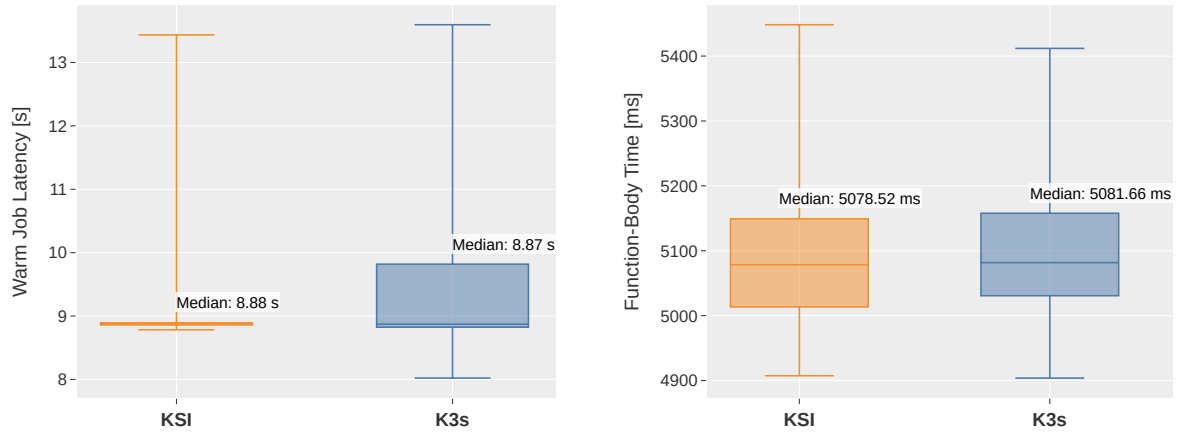
5.3.1 Workload Definition

Each invocation in Benchmark 2 performs a short-lived ETL-style unit of work on one independent input shard. The worker container reads exactly one shard file from shared storage and deserializes the contained JSON Lines (JSONL) records. It then applies lightweight transformation and filtering logic, computes aggregate statistics and writes one compact JSON result before terminating. One shard file therefore corresponds to one Kubernetes Job, one container execution and one output file.

Both platforms use the same dataset, which is generated once on the shared `/nfs` path. It consists of deterministic shard files in JSONL format with deterministic naming and content generation. Each shard represents one logical invocation and contains a substantially larger amount of data than in Benchmark 1, so that the workload is no longer almost entirely reduced to orchestration effects. In the benchmarked configuration, each repetition processes 64 shard invocations, with each shard having an approximate target size of 64 MiB. The resulting workload remains short-lived and independent at invocation level, but introduces a more realistic amount of work inside the container.

In the benchmarked two-node runs, assignment to the two execution partitions is again deterministic, so that partition 0 handles the even shard identifiers and partition 1 the odd ones. On the K3s side, these two partitions are assigned to the two worker nodes. On the KSI side, they are again realized within one Slurm allocation spanning the same two worker nodes, where each node starts its own temporary Kubernetes environment and executes the Jobs assigned to its partition. Because the benchmark contains no cross-invocation communication and no shared mutable state beyond the common read/write path on `/nfs`, this two-partition structure is sufficient for the workload and remains consistent with the execution model established for the KSI platform in Section 4.3.

5.3.2 Benchmark Results



(a) Warm end-to-end latency of Kubernetes Job invocations.

(b) Function-body time inside the container of each Kubernetes Job invocation.

Per stack *(KSI/K3s)*: $n = 320$ successful invocations (5 runs $\times$ 64 Jobs) in the two-node benchmark setup. Only warm-phase execution is shown, i.e. after the execution environments were ready for benchmark Job submission

Figure 5: Benchmark 2 results for the hybrid ETL micro-batch FaaS workload.

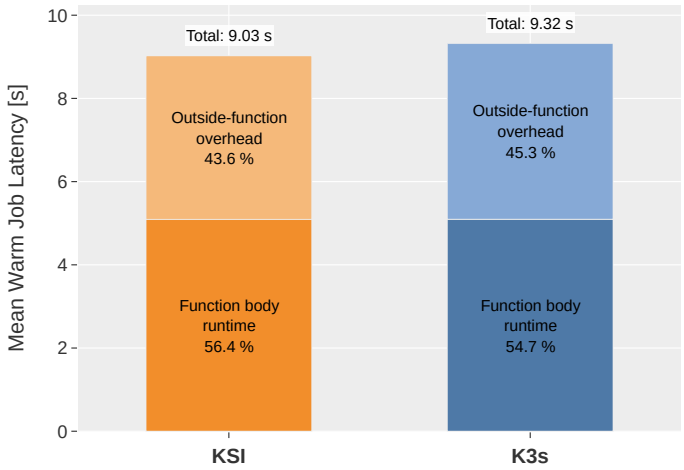
In contrast to Benchmark 1, the warm-phase results of Benchmark 2 (Figure 5) show that the end-to-end latency of one Kubernetes Job invocation is no longer orders of magnitude larger than the time spent inside the worker container itself. The warm Job-latency medians of KSI and K3s are almost identical, but their distributions have a different shape: KSI is concentrated in a very narrow range, whereas K3s is right-skewed with a spread above the median much larger than the spread below it. The two platforms are therefore much closer in warm Job latency than in Benchmark 1.

The function-body times are also very similar on both platforms (Figure 5b). For KSI, the quartiles are spaced relatively evenly around the median, while K3s has a mild upper skew. The absolute differences at function-body level therefore remain small, but Figure 5 still shows that Benchmark 2 shifts the evaluation toward a more balanced FaaS-like workload, in which useful in-container work and Kubernetes-level overhead both contribute meaningfully to total invocation time.

Metric	Stack	Median	Mean	Std	Min	Max
Warm makespan [s]	KSI	294.40	294.29	0.89	292.92	295.30
Warm makespan [s]	K3s	307.92	306.84	5.24	299.16	312.17
Warm p50 job latency [s]	KSI	8.88	8.88	0.01	8.87	8.88
Warm p50 job latency [s]	K3s	8.86	9.18	0.46	8.84	9.78
Warm p95 job latency [s]	KSI	9.72	9.43	0.48	8.91	9.88
Warm p95 job latency [s]	K3s	9.97	9.97	0.11	9.85	10.10

Table 3: Summary of warm run-level metrics across the five 64-Job Benchmark 2 runs.

The run-level summary in Table 3 refines this picture. The warm p50 Job latency is nearly identical on both platforms, but KSI shows a lower warm makespan and a slightly lower warm p95 Job latency across the five repetitions. In particular, the median warm makespan is 294.40 s for KSI and 307.92 s for K3s, which suggests that the overall batch completes somewhat faster on the KSI side under this workload shape.



Per stack *(KSI/K3s)*: mean over $n = 320$ successful invocations (5 runs × 64 Jobs) in the warm two-node benchmark setup. The mean is used so that the stacked bars decompose total warm Job latency exactly and the percentage shares sum to 100 %.

Figure 6: Mean warm Job latency decomposition.

The latency decomposition in Figure 6 reveals how warm Job latency in Benchmark 2 is distributed between function body and platform overhead. On both platforms, slightly more than half of mean warm Job latency is spent inside the function body itself, namely 56.4 % for KSI and 54.7 % for K3s. The remaining 43.6 % and 45.3 %, respectively, arise outside the function body. The figure therefore shows that warm end-to-end Job latency in Benchmark 2 is shaped by in-container processing and additional overhead outside the worker container.

5.3.3 Benchmark Discussion

The results of Benchmark 2 show that the workload behaves as intended in the FaaS context of this report. In contrast to Benchmark 1, the function body accounts for a substantial share of total invocation time. Benchmark 2 therefore represents a more balanced case of short-lived, independent function-style execution. In this setting, useful in-container work and Kubernetes-level overhead contribute meaningfully to total latency. This makes it the more representative benchmark for FaaS-like data processing in the HPC context, since each invocation remains isolated and short-lived while already performing a non-trivial ETL-style task on one shard.

Within this warm-phase view (as described in Section 5.1), Benchmark 2 suggests that once a FaaS-like invocation performs a meaningful amount of ETL work, the strong latency gap observed in Benchmark 1 largely disappears. This indicates that platform differences matter much less once useful in-container work becomes a substantial part of the total invocation time, instead of the invocation being dominated almost entirely by Kubernetes-level orchestration. At the same time, KSI still appears slightly more stable in this setting, whereas K3s shows more variability toward slower invocations. It is also important to point out that the pure containerized workload (the function-body time inside the container) yields nearly identical results on both platforms.

From the FaaS perspective of this report, this is an important result, because it suggests that short-lived, shard-based data functions can be realized on both platforms with very similar warm invocation behavior. This is a notable result for KSI, since it relies on a more indirect execution path. At the same time, the interpretation should remain limited to the present benchmark setting. Benchmark 2 does not show that orchestration overhead disappears, nor does it justify a general claim that one platform is categorically superior for FaaS. In addition, KSI startup and setup overhead remains a separate platform cost and must still be considered outside the warm comparison.

5.4 Comparative Evaluation Summary

Taken together, the two benchmarks show that the relative importance of orchestration overhead depends strongly on how much useful work is performed inside each invocation. In Benchmark 1, the function body remains negligible compared to warm end-to-end Job latency, so the observed platform behavior is dominated by Kubernetes-level orchestration effects. In Benchmark 2, the function body accounts for a substantial share of total invocation time, and the warm invocation behavior of KSI and K3s becomes much closer overall. Across both benchmarks, KSI most often shows the tighter latency distributions. KSI also tends to show lower or comparable warm makespans than K3s. However, KSI startup and setup overhead remains a separate platform cost and is therefore not reflected in the warm-phase comparison itself.

6 Discussion

This chapter interprets the experimental findings in the context of the research questions and the broader FaaS and HPC literature. It also addresses limitations and directions for future work.

6.1 Answers to the Research Questions

In this report, we addressed two main research questions: (RQ1) whether FaaS-like execution can be enabled on an HPC-style platform based on Slurm and KSI, and (RQ2) how such an execution path compares with a native K3s baseline for short-lived, independent FaaS-like workloads as defined in Sections 2.1 and 2.2.

For RQ1, the results in Chapters 3 and 4 show that this is possible, but only after a number of non-trivial platform adaptations. In the basic Slurm/Warewulf environment, KSI required changes to the worker-node root mode, larger local temporary storage and a dedicated rootless container runtime stack for `ksiuser`. The writable runtime state also needed to be relocated from `/nfs` to local `tmpfs`-backed storage. Together with the networking-related adjustments, these steps demonstrate that KSI-based FaaS-like execution is feasible on an HPC-style platform, but not as a drop-in deployment.

For RQ2, the evaluation in Sections 5.2–5.4 shows that the answer depends strongly on the workload shape. For the tiny relay benchmark, where the function body is negligible, KSI shows lower warm Job latency and lower warm makespan than K3s. For the hybrid ETL micro-batch benchmark, where useful in-container work becomes substantial, both platforms are much closer in warm invocation behavior, with KSI still tending toward tighter latency distributions and comparable or slightly lower warm makespans. However, these findings must be interpreted within the realized execution model of the present setup, in which KSI currently acts as two temporary execution domains rather than as one native multi-node Kubernetes cluster. The startup/setup overhead also remains a separate platform cost outside the warm comparison.

6.2 Comparative Synthesis

Taken together, the two benchmarks position our work at the intersection of two lines of research. On the one hand, the KSI paper frames KSI as an “Over-model” integration that brings Kubernetes-style workload execution into a Slurm-managed HPC context [Dec+25]. On the other hand, the serverless literature emphasizes that short-lived FaaS invocations are particularly sensitive to platform overhead, cold starts and execution-path inefficiencies [SBW19], [Sha+20], [Liu+23]. The present report connects these perspectives by examining how a KSI-based HPC execution path behaves for short-lived FaaS-like workloads under a fair comparative benchmark design.

In this regard, Benchmark 1 and Benchmark 2 form a small but meaningful workload spectrum. Benchmark 1 shows that, when the useful work per invocation is almost negligible (Section 5.2.3, Figure 3), the observed behavior is dominated by Kubernetes-level orchestration. This is consistent with the broader FaaS literature, which reports that very short functions are especially exposed to startup, scheduling and execution overheads [SBW19], [Liu+23]. Benchmark 2 then shifts the focus toward a more substantial shard-based ETL invocation (Section 5.3.3, Figures 5 and 6), where useful in-container

work accounts for a substantial share of total warm latency. In this setting, the stronger warm-phase gap from Benchmark 1 largely disappears, which suggests that the relative importance of the surrounding execution framework decreases once the FaaS-like function body itself becomes more meaningful.

At the same time, the synthesis clarifies what kind of FaaS this report actually addresses. The evaluated workloads remain short-lived, independent and stateless, consistent with the semantic framing in Section 2. They therefore correspond most closely to the mainstream serverless model summarized in recent surveys [Wen+23]. Thus, our report does not claim to evaluate the full design space of FaaS. Instead, the results show that the currently implemented KSI execution model enables a Slurm-based HPC platform to support an important class of short-lived FaaS-like workloads. In the warm phase, its behavior is competitive with that of a native lightweight Kubernetes baseline. This is notable because KSI achieves this result through a more indirect execution path than K3s.

6.3 Limitations

This study has several limitations that bound the scope of its conclusions. First, it does not evaluate a native FaaS platform, but approximates FaaS-like execution through Kubernetes Jobs, as motivated in Chapter 2. The reported results therefore concern short-lived, independent invocation semantics under Kubernetes-style execution, not the full behavior of production serverless platforms.

The studied workloads also only capture a small subset of FaaS-style functions. Both benchmarks were designed around short-lived, independent and stateless invocations without inter-node communication. The report therefore does not assess whether the observed results would carry over to FaaS-like applications that require direct communication across execution nodes.

Finally, the experiments were conducted on a small OpenStack-based virtualized testbed rather than on a larger physical HPC system. The findings should therefore be seen as a controlled comparative study of the two concrete execution paths, not as a universal statement about HPC environments or FaaS realizations.

6.4 Future Work

The most direct next step is to deploy a full FaaS framework on top of both platforms, especially the KSI-based one. This would show whether the findings of this report remain stable under a more explicit FaaS layer. It would also be valuable to repeat the study on larger physical cluster environments to assess how well the present observations transfer beyond the current small virtualized testbed.

A second line of future work concerns the workloads themselves. Follow-up experiments should include larger and more diverse workload classes, for example different shard sizes or workloads with stronger sensitivity to storage behavior.

Finally, future work should revisit the multi-node and networking aspects of KSI more directly. In the present study, both benchmarks were intentionally designed without inter-node communication, so that Liqo-based node-to-node interaction was not part of the measured workload behavior. A useful extension would therefore be to evaluate communication-aware FaaS-like workloads and to determine how direct node-to-node interaction influences the comparative behavior of the two platforms.

7 Conclusion

This report examined how FaaS-like execution can be realized in an HPC-related context and how such an execution path compares with a native lightweight Kubernetes baseline. To investigate this, two comparable platforms were established: a cloud-like K3s cluster and an HPC-style Slurm-based platform extended by KSI. A central part of the work was to make the KSI-based platform operational under the given conditions and to clarify what execution model it actually provides in the present setup. The experimental evaluation then compared both platforms under two deliberately different FaaS-like workload shapes.

Our evaluation showed that the relative importance of platform overhead depends strongly on the workload shape. For the tiny relay benchmark, the useful work per invocation was negligible, so the observed behavior was dominated by Kubernetes-level orchestration. In this case, KSI showed lower warm Job latency and lower warm makespan than K3s. For the hybrid ETL micro-batch benchmark, by contrast, the function body accounted for a substantial share of total invocation time, and the warm behavior of both platforms became much closer overall. Across both benchmarks, KSI often showed tighter warm Job-latency distributions and competitive or slightly lower warm makespans, although startup and setup overhead still had to be considered separately.

Overall, we conclude that FaaS-like execution on an HPC-style platform is feasible and can be competitive for an important class of short-lived, independent workloads. At the same time, the results depend strongly on the workload shape and on the way KSI is realized in the present setup. The report therefore does not claim that one platform is generally superior. Instead, it shows more specifically that, under the current KSI execution model, a Slurm-based HPC platform can support short-lived FaaS-like workloads with warm-phase behavior that is close to, and in some cases better than, a native lightweight Kubernetes baseline. Future work should extend this basis toward fuller FaaS frameworks, broader workload classes and larger cluster environments.

References

[Dec+25] Jonathan Decker, Mojtaba Akbari, Ali Doosthosseini, et al. "Enabling Kubernetes Workload Execution on Rootless HPC Systems with KSI: A Slurm Integration Framework". In: *International Journal on Advances in Intelligent Systems* 18 (Dec. 2025). URL: https://www.iariajournals.org/intelligent_systems/tocv18n34.html.

[KSB17] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. "Singularity: Scientific containers for mobility of compute". In: *PLOS ONE* 12.5 (May 2017), pp. 1–20. DOI: 10.1371/journal.pone.0177459. URL: https://doi.org/10.1371/journal.pone.0177459.

[Liu+23] Xuanzhe Liu, Jinfeng Wen, Zhenpeng Chen, et al. "Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing". In: *ACM Transactions on Software Engineering and Methodology* 32.5 (2023), pp. 1–29.

[SBW19] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. "Architectural Implications of Function-as-a-Service Computing". en. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. Columbus OH USA: ACM, Oct. 2019, pp. 1063–1075. ISBN: 978-1-4503-6938-1. DOI: 10.1145/3352460.3358296. URL: https://dl.acm.org/doi/10.1145/3352460.3358296 (visited on Mar. 25, 2026).

[Sha+20] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 205–218. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[Wen+23] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. "Rise of the Planet of Serverless Computing: A Systematic Review". In: *ACM Trans. Softw. Eng. Methodol.* 32.5 (July 2023). ISSN: 1049-331X. DOI: 10.1145/3579643. URL: https://doi.org/10.1145/3579643.