Feather: Lightweight Container Alternatives for Deploying Workloads in the Edge

Tom Goethals[a], Maxim De Clercq[b], Merlijn Sebrechts[c], Filip De Turck[d] and Bruno Volckaert[e]
Ghent University - imec, IDLab, Gent, Belgium

Keywords: Edge Computing, Kubernetes, Containers, Microvm, Unikernels.

Abstract: Recent years have seen the adoption of workload orchestration into the network edge. Cloud orchestrators such as Kubernetes have been extended to edge computing, providing the virtual infrastructure to efficiently manage containerized workloads across the edge-cloud continuum. However, cloud-based orchestrators are resource intensive, sometimes occupying the bulk of resources of an edge device even when idle. While various Kubernetes-based solutions, such as K3s and KubeEdge, have been developed with a specific focus on edge computing, they remain limited to container runtimes. This paper proposes a Kubernetes-compatible solution for edge workload packaging, distribution, and execution, named *Feather*, which extends edge workloads beyond containers. *Feather* is based on Virtual Kubelets, superseding previous work from FLEDGE. It is capable of operating in existing Kubernetes clusters, with minimal, optional additions to the Kubernetes PodSpec to enable multi-runtime images and execution. Both Containerd and OSv unikernel backends are implemented, and evaluations show that unikernel workloads can be executed highly efficiently, with a memory reduction of up to 20% for Java applications at the cost of up to 25% CPU power. Evaluations also show that *Feather* itself is suitable for most modern edge devices, with the x86 version only requiring 58-62 MiB of memory for the agent itself.

1 INTRODUCTION

Modern microservice architectures are shifting from cloud-only to running in a cloud-edge continuum for various reasons. For example, many applications benefit from local processing in terms of response time, improving Quality of Experience (QoE) (Luo et al., 2019). Additionally, processing data on or near the edge devices where it is generated reduces network load and can preserve privacy if implemented correctly (Sadique et al., 2020). Processing workloads on edge devices also has the potential to better utilize local supplies of green energy (Al-Naday et al., 2022). Finally, an important enabler is that modern edge devices have become powerful enough to run various workloads dynamically on-demand.

In the cloud, developers commonly use orchestrators such as Kubernetes [1] to deploy and manage applications on a uniform infrastructure independent of the actual hardware. Edge computing, however, has traditionally used a different set of tools to deploy and manage applications. This adds an additional barrier for cloud application developers to venture into edge computing. Therefore, bringing cloud orchestrators to edge computing has the potential to improve the experience of transitioning from a cloud-only application to a cloud-edge application.

Bringing cloud orchestrators to the edge is challenging, however, because of their resource overhead. For example, Kubernetes agents have a significant resource overhead, around 145MiB for the agent alone and 300+MiB with all necessary components, potentially using most of the memory on an edge device even when idle (Goethals et al., 2020). Although recent developments such as ioFog [2], KubeEdge [3] and *FLEDGE* (Goethals et al., 2020) provide alternative orchestration options, most of them still present a significant overhead, and all of them are exclusively based on containerized workloads. While containers are a highly performant, lightweight virtualization

[a] https://orcid.org/0000-0002-1332-2290
[b] https://orcid.org/0009-0009-1857-8664
[c] https://orcid.org/0000-0002-4093-7338
[d] https://orcid.org/0000-0003-4824-1199
[e] https://orcid.org/0000-0003-0575-5894
[1] Kubernetes: https://kubernetes.io/

[2] Eclipse ioFog: https://iofog.org/
[3] KubeEdge: https://kubeedge.io/

Goethals, T., De Clercq, M., Sebrechts, M., De Turck, F. and Volckaert, B.
Feather: Lightweight Container Alternatives for Deploying Workloads in the Edge.
DOI: 10.5220/0012536300003711
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 27-37
ISBN: 978-989-758-701-6; ISSN: 2184-5042
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

option, they may not always be the best solution for any combination of workload and edge device. Other virtualization options include micro Virtual Machines (microVMs), which boast improved security and resource use compared to containers (Goethals et al., 2022), or WebAssembly (Wasm) (Sebrechts et al., 2022), which aims for microVM security and performance without the necessity for a hypervisor. Example use cases include a home automation edge gateway running privacy sensitive processing tasks in a microVM, while automation rules and dashboards are run in containers. Another use case involves ad-hoc federation of networked resources in emergency situations, running mission critical tasks on possibly untrusted nodes inside microVMs while support services are run in containers.

This paper proposes a Kubernetes-compatible solution for edge workload orchestration named *Feather*. Based on the lightweight *FLEDGE* orchestrator (Goethals et al., 2020), *Feather* extends its capabilities beyond containers to include microVMs. As such, it allows developers to choose the right virtualization option for their workloads, and allows researchers to easily compare different virtualization methods for edge computing. Additionally, Feather uses relevant Open Container Initiative (OCI) standards, which are important for interoperability with Kubernetes and Docker containers.

This paper also presents an end-to-end solution for packaging, distributing and deploying workloads in a runtime-agnostic manner, allowing developers to use an almost identical deployment workflow for both containers and microVMs that seamlessly integrates with the Kubernetes ecosystem.

Concretely, the contributions of this paper are:

- Designing an extensible, Kubernetes-compatible agent which allows for the deployment of non-container workloads on edge devices
- Providing an OCI-compliant method for packaging and distributing non-container workloads through a container repository
- Illustrating the potential of multi-runtime workloads in a Kubernetes (edge) cluster
- Minimizing the resource overhead of edge orchestration, leaving the bulk of device resources for edge computing

The rest of this paper is organized as follows: Section 2 presents existing research related to the various topics in this paper, from which research questions are derived in Section 3. Section 4 introduces all the high level architecture aspects. In Section 5, the evaluation setup, scenarios and methodology are detailed, while the results are presented and discussed in Section 6. Topics for future work are listed in Section 7 and finally, Section 8 draws high level conclusions from the paper.

2 RELATED WORK

2.1 Virtualization

The properties and performance of container runtimes have been extensively examined in various studies (Wang et al., 2022; de Velp et al., 2020).

MicroVMs are a lightweight form of VM designed to run individual workloads or processes. There are several technologies that enable the creation of microVMs, among which unikernels are a varied group with excellent security and performance features (Kuenzer et al., 2021; Abeni, 2023). Unikernels are a type of library operating system in which a program, along with only the required system libraries and system calls, is compiled into a single kernel space executable embedded in a VM image, thus minimizing image size and attack surface. Furthermore, they can be roughly classified into two types: POSIX-compatible (Portable Operating System Interface(Walli, 1995)) ones that focus on existing software, and those based on non-POSIX system interfaces which sacrifice compatibility for smaller images and lower resource requirements. OSv(Kivity et al., 2014) in particular is a POSIX-compatible unikernel platform with wide compatibility for existing programs and programming language runtimes. Although microVMs generally support a wide variety of hypervisors for their execution, QEMU(Bellard, 2005) with KVM (Kernel-based Virtual Machine(Habib, 2008)) acceleration is a widely supported option.

Both containers and microVMs are examples of degrees of virtualization, where at least some degree of isolation from the host system is established. Different classes of virtualization technologies, including gVisor and Firecracker, have been compared and benchmarked (Goethals et al., 2022), and their performance examined at the kernel level (Anjali et al., 2020).

The WebAssembly System Interface (WASI) (Ménétrey et al., 2022) is a new and fundamentally different approach to sandboxing, interposing itself between WebAssembly (Wasm) programs and a host system (e.g. Linux kernel). Like gVisor, it implements its own System Interface to intercept program system calls, but it focuses on both security and performance while being entirely device- and system-agnostic. While still under development, WASI covers a wide range of devices up to ESP32 microcontrollers (BytecodeAlliance, 2023), making it suitable for the edge and IoT (Ray, 2023). WASI can also be used to optimize container-oriented architectures, e.g. Kubernetes controllers (Sebrechts et al., 2022).

2.2 Orchestration

Several edge-oriented Kubernetes-based platforms are currently available, for example KubeEdge and Eclipse ioFog (Čilić et al., 2023). KubeEdge has a memory overhead of only 70 MiB for its worker nodes, although it is incompatible with default Kubernetes clusters. ioFog, another popular orchestration platform for the edge, has a memory footprint of only 100 MiB on worker nodes, which is fairly low in comparison with other Kubernetes distributions. However, while these frameworks are aimed at edge computing, they are limited to container workloads.

Some studies (Mavridis and Karatza, 2021) use KubeVirt (KubeVirt, 2023) to deploy and evaluate (micro)VM alternatives on Kubernetes clusters. However, while KubeVirt enables the deployment of virtualized workloads, it also requires extensive intervention in a Kubernetes cluster to work (e.g. custom resources, daemonsets). Unlike KubeVirt, *Feather* is aimed specifically at creating a multi-runtime agent for edge computing, without the need to modify an existing Kubernetes cluster in any way.

FLEDGE (Goethals et al., 2020) is a Kubernetes-compatible edge agent based on Virtual Kubelets [4], and designed for minimal resource overhead, using only around 50MiB of memory. A Virtual Kubelet is essentially a proxy which poses as an actual kubelet to the Kubernetes API, but allows any sort of underlying provider to interpret and execute the received commands. However, despite a low-resource implementation with support for both Docker and containerd, *FLEDGE* is limited to the use of containers. *Feather* aims to extend the State of the Art by allowing the OCI-compliant side-by-side orchestration of various types of workload images (e.g. containers, microVMs) on edge devices, without architectural or operational changes to an existing Kubernetes cluster or its control plane nodes.

[4]Virtual Kubelet: an open source Kubernetes kubelet implementation - https://github.com/virtual-kubelet/virtual-kubelet

3 RESEARCH QUESTIONS

Considering the limitations of the state of the art, the following research questions are stated for *Feather*:

1. **RQ1.** How can Kubernetes-compatible orchestrators be extended to support pluggable backends, including microVM workloads in the edge?
2. **RQ2.** How can edge microVM workloads be seamlessly integrated into the regular cloud native and Kubernetes ecosystem?
3. **RQ3.** What are the *Feather* overhead and performance characteristics of microVM workloads in the edge?

4 ARCHITECTURE

This section presents the solution architecture for both the Kubernetes-side deployment of workloads, and for runtime-agnostic workload packaging and distribution. The code for *Feather* is made available on Github[5].

4.1 Deployments

Feather uses a Virtual Kubelet as a basis, receiving commands from Kubernetes through its interface. Workload platforms (e.g. container, unikernel) are referred to as "backends", which may be supported by different runtimes (e.g. QEMU, VirtualBox), while running workloads are "instances".

A high-level overview of the deployment pipeline from Kubernetes to *Feather* is shown in Fig. 1. On the left side of the figure, deployments are entered into Kubernetes through the API or dashboard. *Feather* nodes are usually given a specific node label and taint to avoid accidentally deploying heavier cloud workloads. While these can be disabled, deployments should target *Feather*-managed devices specifically.

Next, the deployment is scheduled and sent to *Feather* on an edge device, where it is picked up by the Virtual Kubelet component. In order to separate node and pod logic from atomic workloads (i.e. individual containers), *Feather* implements a Provider which takes care of all pod-level logic, leaving the workload-level (or instance) logic to backends (e.g. containerd, OSv). This approach reduces the complexity of additional backend implementations, and allows mixing different runtimes in a single pod if required, although individual backends may present practical problems as shown in Section 4.2. The

[5]https://github.com/idlab-discover/feather

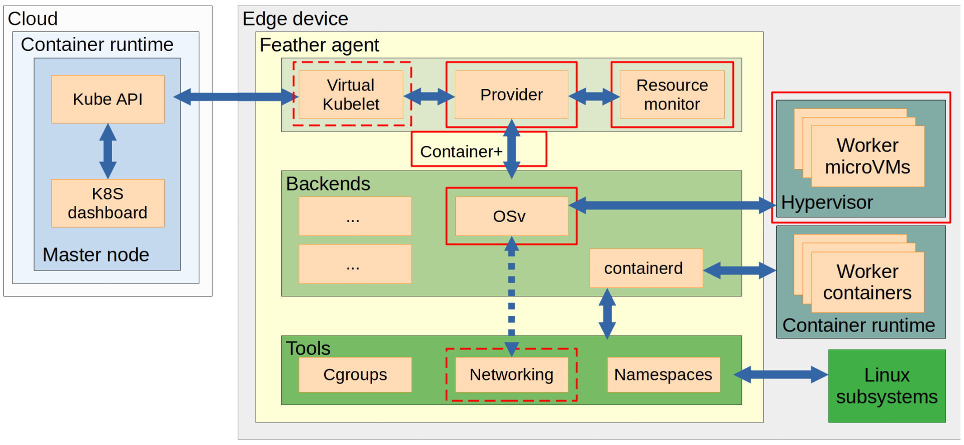

Figure 1: High-level overview of *Feather* components and the separation of virtualization options into backends. Full and dashed red rectangles represent new and heavily modified components compared to FLEDGE, respectively.

Provider also interacts with a basic Resource Monitor to report node status and determine if it has enough resources to execute a deployment. For advanced monitoring outside the default Kubernetes dashboard, a Prometheus Golang exporter may be enabled in addition to default process metrics.

However, some pod-level information may be required to properly set up a workload, for example volumes or pod networking information. To solve this, the Provider extracts the relevant information from the PodSpec (e.g. networking, volumes) on a per-instance basis and embeds it into an extended Container specification indicated by Container+. For example, if the pod is globally configured to use host networking, the backend requires that information to create its workloads under that condition. The extended Container specification is then sent to the correct backend, which is responsible for pulling the correct image and managing its life-cycle.

4.2 Candidate Backends

At this point, the various backends require different tools and kernel features, and manage instances through different backends. This section describes the backends currently implemented in *Feather*. *Feather* is designed to be generic, supporting anything from container runtimes to hypervisors and beyond. In order to function as a sufficient proof-of-concept of this design, both containers and microVMs are supported, through containerd and OSv respectively. For compatibility purposes, the backend design is inspired by the OCI Runtime Spec [6]. This specification describes the configuration, execution environment, and life cycle of a container.

As containerd is a native container runtime supported by Kubernetes by default, and because *FLEDGE* already has a previous implementation for it, creating a backend is self-evident. As shown in Fig. 1, the containerd backend interacts with various *Feather* tools to setup the required cgroups, namespaces and container networking for a single container. These in turn interact with the appropriate Linux subsystems, which for now is limited to cgroups v1, curbing operating system options slightly. The image itself is started using the container runtime (containerd), which may host any number of worker containers.

For microVMs, the unikernel platform OSv is chosen for image creation, as it supports a wide array of backends and devices, and has its own command-line image creation and management tool Capstan. Implementing a backend for OSv is less straightforward for several reasons. Firstly, instead of a container runtime, the backend needs to communicate with a hypervisor, in this case QEMU, which may host any number of worker microVMs. Consequently, the backend needs to extract a VM disk image from the OCI Image Spec compliant image. The backend needs to determine which combination of parameters should be passed to the hypervisor to achieve the same effect as they have on a container deployment. Secondly, besides increased complexity, this can introduce compatibility issues, for example for persistent storage and mounts which have to be mounted through virtio-fs drivers rather than a straightforward shared location in the host filesystem. Due to virtio-fs and OSv limitations, the current version of *Feather* only supports read-only mounts of directories. While Kubernetes ConfigMaps and Secrets may consist of a single file, this is fixed by *Feather* by putting them in a

[6]OCI Runtime Specification: https://github.com/opencontainers/runtime-spec

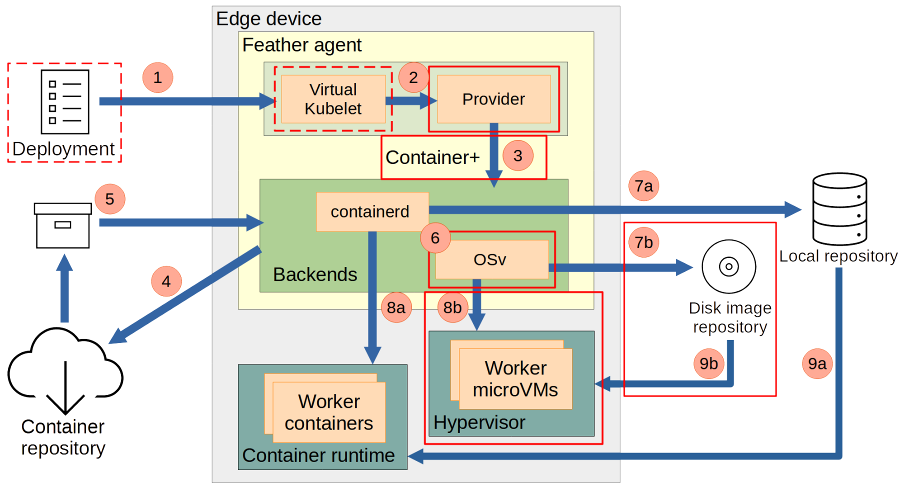

Figure 2: Overview of the deployment sequence of a *Feather* workload, from an incoming deployment yaml to workload image start. Full and dashed red rectangles represent new and heavily modified components compared to FLEDGE, respectively.

specific directory which is mounted into the image. A custom C program is provided which can be inserted into a *Feather* unikernel image, which scans for these ConfigMaps and Secrets, and organizes them as required by the workload. Lastly, networking support for unikernels in *Feather* is currently limited; while unikernels are set up with a network interface which is publicly reachable, and are assigned an IP address from the same range as containers, sidecar services are not currently possible for OSv workloads. Considering the current implementation, this section answers **RQ1**.

4.3 Images

Packaging non-container images for backend-agnostic storage is non-trivial. For example, OSv images and other microVMs are packaged as raw disk images with no additional information [7]. In order to distribute these atypical "container" images reliably and correctly, well-established container image standards are used in combination with minimal metadata.

The most straightforward solution is to store a disk image as a layer in an OCI Image, which enables configuration similar to container images. In order to determine the content type of an OCI Image Layer, the OCI ImageConfig is extended, allowing *Feather* to select the correct backend for an image.

1. `feather.backend`: The backend that this image was designed for.
2. `feather.runtime` (optional): The preferred runtime to schedule this instance with. In the case of virtual machines, this refers to the used hypervisor.

Since containerd images are already OCI Images, these fields are only required for the OSv backend. Whenever the `feather.backend` field is not present, *Feather* can safely assume a container image. The general process of pulling an image is shown in Fig. 2. A deployment yaml is sent to the edge device (1), which is forwarded to the provider as a parsed Deployment (2) and split into separate Container+ deployments forwarded to a suitable backend (3). Due to OCI compatibility, all image types can be pulled from a single repository (4), which returns an image of the required type (5). The backend then parses this image and unpacks it, if required (6). Depending on the backend, the image is stored in either the local containerd repository (7a), or the Capstan repository (7b). The backend sends a start command for the image to the required runtime (8a) & (8b), which uses the image from the matching local repository.

Flint is developed alongside *Feather* to provide multi-backend image management between online and local repositories as a command-line tool. For the OSv backend, it allows users to import images from a local Capstan repository into OCI images. The combination of optional extra Image fields and Flint solves **RQ2**.

[7]OSv, a new operating system for the Cloud: https://github.com/cloudius-systems/osv

5 EVALUATION SETUP

This section considers **RQ3** by assessing the performance of *Feather* and the implemented backends through three different evaluation scenarios:

1. Baseline: Measures the resource consumption of *Feather* itself, without any running workloads. While some of the backends may have running processes even without active workloads, they are not yet loaded into *Feather* and no (container) networking is active.
2. Minimal: Measures the overhead of *Feather* per workload container or unikernel by deploying an idle workload to both backends. Specifically, this scenario uses a Hello World application written in C, which sleeps for the duration of the experiment. This ensures that there is no work done, but that it is still possible to verify if the application has started successfully. This approach allows measuring the exact overhead of deploying a basic workload for both *Feather* and the backends.
3. Application Benchmark: Determines the relative resource overheads and performance penalties of both backends for a real-world application, specifically a Minecraft (PaperMC) server running. Due to OSv compatibility issues, the Java runtime is limited to v11.0, which limits PaperMC to v1.16.4. However, for the purposes of this evaluation using an older version is acceptable.

Considering an edge device with 1 GiB of total memory, it is a reasonable choice to allocate a maximum of 800 MiB for the Java application, leaving 200 MiB for the operating system, *Feather*, and its backends. Unlike the minimal application, this application can be configured through a ConfigMap which contains properties such as the world seed and default game mode. A Minecraft game server is initialized with a world seed, where different seeds lead to different worlds to be generated. By keeping the seed consistent between backends, a deterministic testing environment can be created. For this evaluation, the Minecraft worlds are initialized with seed -11970185, although others have been evaluated to rule out outliers. The evaluations in this scenario are performed as follows:

a) Any previous deployment is deleted.
b) The server image is deployed on the worker node using Kubernetes. The logs then state the startup time of the server, and verify the configuration of the seed through the ConfigMap.
c) After 2 minutes, 4 clients join the server simultaneously. This verifies the ability of the server to handle simultaneous events, such as chunk generation, which stresses the CPU.
d) After 4 minutes, villager entities are spawned at a rate of about 3 villagers per 10 in-game ticks. On a fully responsive server running 20 in-game ticks per second, this corresponds to a rate of 6 villagers per second.

For the evaluations, two physical nodes are set up on the IDLab Virtual Wall [8]; a control plane node running the Kubernetes control plane, and a worker node running *Feather* and any required backends. Both nodes are running Ubuntu 20.04.05 LTS, and are equipped with two Intel E5520 CPUs, 12 GB of RAM, and two Gigabit network interfaces.

Specific software versions used for the evaluations are:

- containerd v1.6.18 to run container images
- qemu-system-x86 v4.2.1 to run unikernel images
- virtiofsd v1.6.0 for unikernel file system access
- Virtual Kubelet v1.9.0 as a basis for *Feather*
- Azul Zulu OpenJDK v11.0.19 for the application benchmark
- PaperMC v1.10.2 as Minecraft server in the application benchmark

In order to emulate an edge device with only 1 CPU and 1 GiB of RAM, *Feather* imposes resource limits on containerd and OSv using cgroups and QEMU options respectively.

In all scenarios, Prometheus[9] node and process exporters are used to gather relevant node and workload data. Note that while all relevant processes are examined in order to gauge the full impact of *Feather* operation, processes on the control plane node are not included as there are no additional requirements apart from normal Kubernetes operation. The node exporter is run as a separate process outside *Feather* to avoid influencing either *Feather* or workloads, while the process exporter gathers workload, QEMU, and containerd metrics. For the application benchmark, further application metrics are gathered using the PaperMC Minecraft exporter. For the *Feather* process itself, the Prometheus Golang exporter is disabled and metrics are gathered using ‘top‘ to avoid interference.

6 RESULTS

This section presents the evaluation results for all scenarios, presented as timeseries charts over 5 to 10

[8]Virtual Wall — imec iLab.t documentation: https://doc.ilabt.imec.be/ilabt/virtualwall/

[9]https://prometheus.io/

minute ranges of the gathered metrics. Series are color coded green for *Feather* and purple for backend processes, with other colors reserved for scenario-specific metrics. Importantly, all CPU percentages are for a single core, not relative to the entire server.

6.1 Baseline

Fig. 3 shows the storage requirements of various components and alternatives. The *Feather* binary used for the experiment is a stripped, statically linked x86_64 binary. While *Feather* is around 13MiB larger than *FLEDGE*, it is still 15MiB smaller than KubeEdge (v1.13.0). The main size increase over *FLEDGE* is caused by supporting more recent Kubernetes versions, and OSv images. As such, *Feather* offers high flexibility at low storage overhead. Workload backends represent the bulk of storage requirements; containerd requires twice as much storage as *Feather* itself, while a default installation of QEMU requires seven times as much. However, QEMU can be built from source to reduce this storage overhead, excluding all features that are non-essential for the OSv backend.

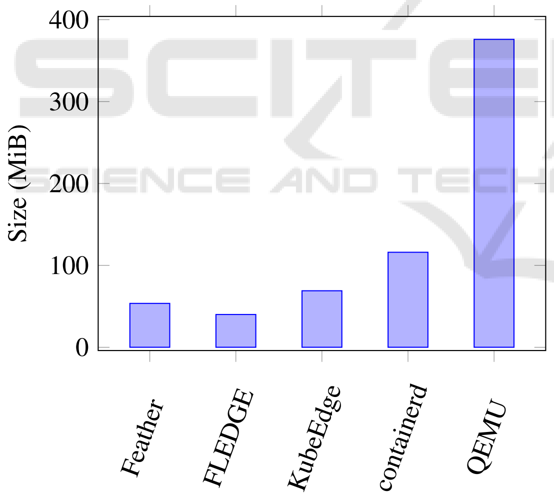

Figure 3: Installed size of various components and alternatives for edge workload orchestration.

Fig. 4 shows the CPU use of relevant processes for the baseline scenario. *Feather* uses only around 0.3% of a single core when idle, mostly due to backend polling and Kubernetes status updates. As shown in Fig. 5, *Feather* only uses around 58 MiB of memory after initialization. While this is significantly more than *FLEDGE*, which only requires around 46 MiB (Goethals et al., 2020), it is an acceptable increase for offering multiple backends, and for newer Kubernetes feature support due to using a higher version Virtual Kubelet. The containerd process has no significant CPU usage after initialization, and uses 55 MiB of memory on average.

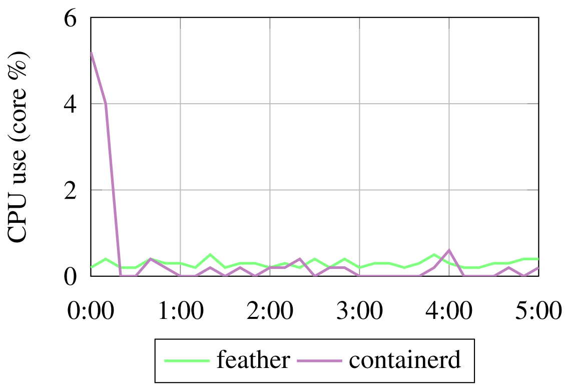

Figure 4: CPU usage of relevant processes in the baseline scenario.

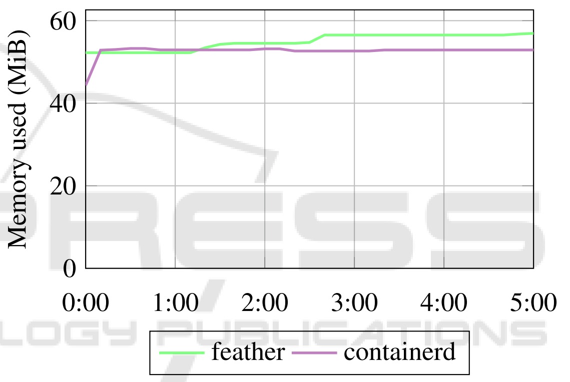

Figure 5: Resident memory usage of relevant processes in the baseline scenario.

6.2 Minimal

Fig. 6 shows that *Feather* CPU use peaks for a few seconds for both OSv and containerd workload creation. Although containers have a limited impact at 2.8%, creating a unikernel needs significantly more CPU at 6.8%. A higher setup cost is a natural consequence of the superior isolation of unikernels compared to containers, which requires setting up a much more extensive virtual infrastructure. After container creation CPU use drops back to around 0.4%. As such, idle CPU use is no higher than in the baseline scenario.

Both containerd and containerd-shim-runc-v2 use an insignificant amount of processing resources, except at container creation, when containerd spikes at 4.2% CPU. The QEMU hypervisor, however, uses a significant amount of processing power, idling at

11.5% after a 21% spike at creation.

Memory use for this scenario is shown in Fig. 7. Note that all processes are relatively stable in their memory use; containerd itself seems to use around 5 MiB extra memory during container creation which is later released, while in both cases *Feather* allocates an extra 3 MiB at some point which is not directly tied to workload creation. Comparing this to the baseline scenario, the *Feather* process requires 3-4 MiB of memory to instantiate and manage backends, putting *Feather* itself at 60-62 MiB depending on the active backend.

Comparing both backends, containerd uses 56.5 MiB of memory on average, while containerd-shim-runc-v2 uses around 9.5 MiB, resulting in a total memory overhead of 66 MiB on average. With a memory consumption of 63.3 MiB on average, QEMU performs slightly better than containerd and containerd-shim-runc-v2 combined, although this advantage quickly disappears for multiple workloads as containerd memory use is amortized over all containers.

Note that there are no metrics for virtiofsd included in Fig. 6, as the minimal scenario does not make use of any volume mounts.

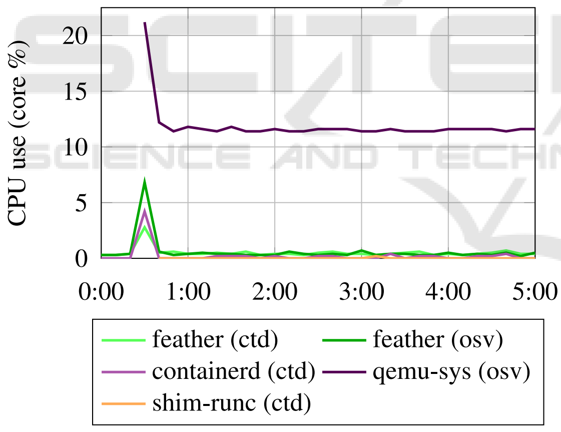

Figure 6: CPU usage of relevant processes in the minimal scenario.

6.3 Application

Fig. 8 shows the performance of PaperMC in in-game terms. The responsiveness of the server (expressed in ticks per second, or TPS) drops significantly after 2 minutes as clients join the server, leveling back out as the server stops generating chunks. Then, it drops steadily after 4 minutes as villagers are spawned. The rate at which villagers are spawned also decreases in time, which notes its relation to TPS as mentioned in Section 5. Throughout the chunk generation phase,

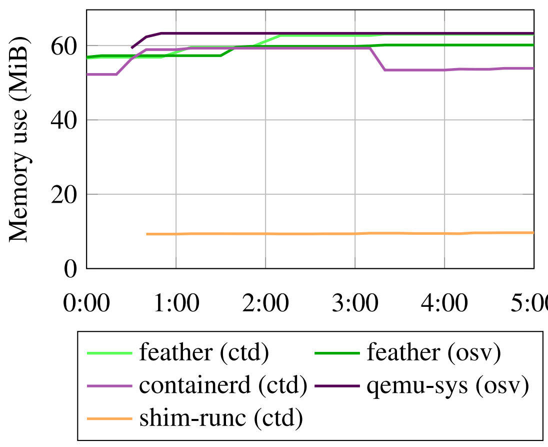

Figure 7: Memory usage of relevant processes in the minimal scenario.

OSv performance consistently changes incrementally faster than containerd is affected, although in magnitude both are equally affected. This may indicate that unikernels are significantly more effective at chunk generation.

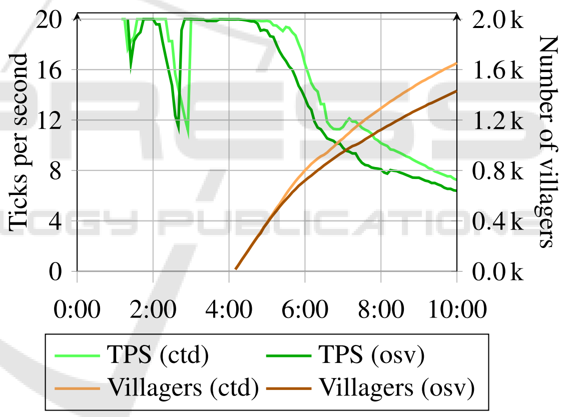

Figure 8: Ticks per second and number of villagers in the application scenario.

Additionally, TPS starts to drop slightly faster for the OSv instance than for the containerd instance, indicating that the OSv may be less efficient at the type of calculation involving villagers. Finally, the logs show that the containerd instance starts in 38.713s, while the OSv instance only takes 37.199s to start, or around 4% faster. Fig. 9 shows that QEMU uses more than the 1 CPU limit set by the deployment despite producing a lower TPS. Taking into account the higher "idle" use at 2:00 and 3:00, and the CPU results from Fig. 6, QEMU appears to have a significant, structural CPU overhead, but the application itself is limited to 1 CPU. Comparing Fig. 8 at constant 1200 villagers, OSv on QEMU runs at a 28% lower TPS compared to the container instance, despite a 30% CPU overhead.

Fig. 9 shows that OSv on QEMU uses significantly less memory than its container counterpart throughout the evaluation. After spawning all villagers, QEMU uses only 661 MiB of RAM, significantly less than the container Java instance at 820 MiB. Note that spawning villagers, and thus the slightly different population of both instances, does not seem to affect memory use significantly. Looking at JVM statistics, Fig. 11 shows that OSv garbage collection is more efficient and stable than containerd, averaging 0,83% ± 1,11% vs 1,68% ± 2,16% CPU.

Similarly, Fig. 12 shows that the JVM on OSv allocates 20-40% less memory than when running in a container, and uses the allocated memory more efficiently. This result is expected, as OSv claims that it is able to expose specific hardware features to the JVM because it is running in the same address space as the kernel. This allows the JVM to manage memory more efficiently, resulting in fewer allocations and time spent on garbage collection.

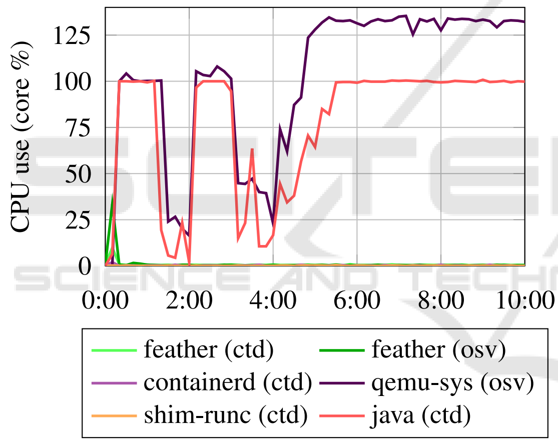

Figure 9: CPU usage of relevant processes in the application scenario.

6.4 Analysis

The evaluation is performed for a simulated edge device with 1 CPU and 1 GiB of memory. Both the minimal and application scenarios indicate that QEMU has some CPU overhead independent of the actual workload, greatly impacting its potential for edge devices. While the overhead may be managed by QEMU itself through cgroups, overall workload performance would suffer as a result. However, some types of workloads (e.g. chunk generation, or REST services (Goethals et al., 2022)) are shown to be more efficient with OSv on QEMU despite this overhead. Additionally, different hypervisors may produce better results.

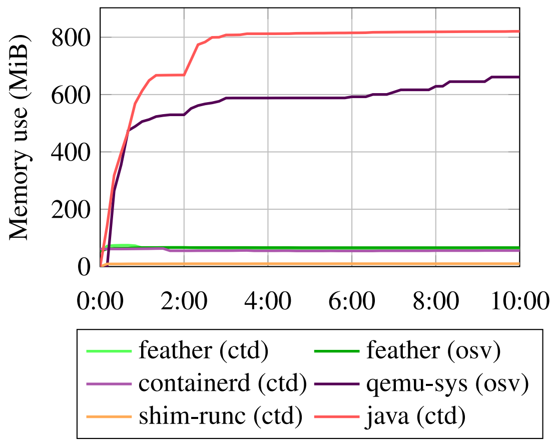

Figure 10: Memory usage of relevant processes in the application scenario.

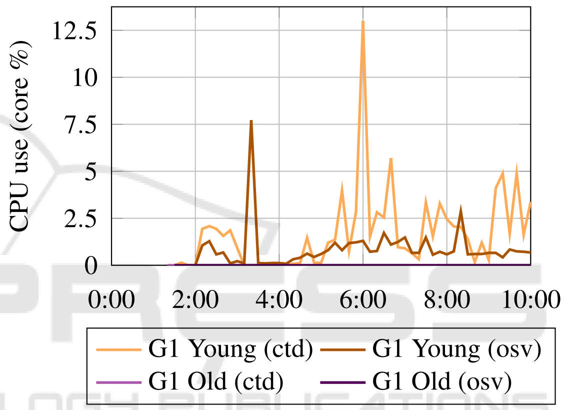

Figure 11: CPU time spent on garbage collection by the JVM in the application scenario.

In terms of memory, QEMU far outperforms the containerized Java application; the OSv application instance requires 20% less memory than the containerd instance. Generally superior memory management is also clearly visible in the increased stability of garbage collection and memory management of the JVM. As unikernels have a base memory overhead of around 50 MiB compared to container instances, the advantage of using a unikernel increases with the memory requirements of its workload.

Finally, the evaluation does not cover ARM-based edge devices. However, OSv is shown to perform similarly on a Raspberry Pi 4 (Goethals et al., 2022), although slower in absolute terms.

7 FUTURE WORK

In Kubernetes environments, data is generally stored in PersistentVolumes, however, *Feather* has no sup-

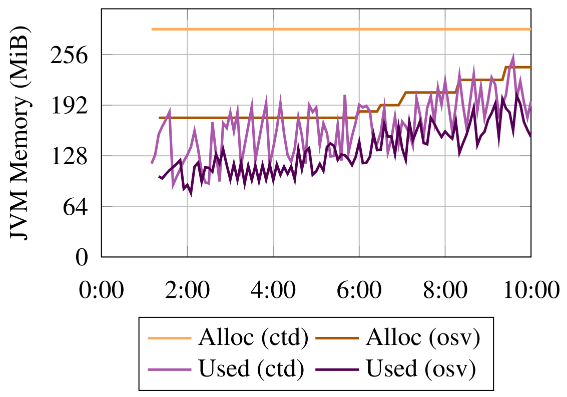

Figure 12: Used and allocated JVM memory statistics in the application scenario.

port for PersistentVolumes at this point, which is a limitation of using a Virtual Kubelet.

Additionally, individual files cannot be mounted, and the virtio-fs driver provided by the OSv kernel only supports read-only mounts. As such, the OSv backend can not currently bind mounts for persistent storage. Other options supported by OSv include NFS and ZFS, but as Kubernetes only supports NFS filesystems as a VolumeSource, future work will likely focus on this option to store files persistently.

Another area of improvement is container networking, as OSv unikernels are not currently integrated into the container network. However, they do have flexible networking capabilities, and *Feather* already integrates other initialization programs for Kubernetes features into unikernels (e.g. ConfigMaps, storage, etc), making full network integration a logical next step. The two most significant steps for future work are assigning a suitable container network IP address, which is trivial with initialization programs, and ensuring localhost availability between workloads in the same Pod, likely through eBPF programs.

Feather can be further extended by developing backends for other technologies, such as Wasm, Kata Containers, Firecracker, Unikraft. Some of these are image creation tools, while others are runtimes or may be designed with both image creation and runtime tools. As *Feather* allows deployments to distinguish between type of workload (i.e. container, OSv) and actual runtime (e.g. containerd, QEMU), future work will focus on both providing additional workload image formats and potential runtimes.

8 CONCLUSION

Feather provides a solution for extending the Kubernetes ecosystem in the edge with non-container alternatives such as unikernels, which can reduce resource requirements for certain use cases.

Building on previous work from *FLEDGE*, *Feather* leverages existing standards such as the OCI Specifications and the Kubernetes ecosystem, ensuring that the proposed solution is seamlessly interoperable with existing infrastructure, requiring minimal effort by developers to leverage the capabilities of *Feather*.

Two backends, containerd and OSv, are implemented and evaluated in three different scenarios, showing the relative strengths and weaknesses of each backend for specific computational tasks. While unikernel workloads on QEMU tend to have a significant CPU overhead compared to containers, even when idle, their memory use is up to 20% lower in high load scenarios.

This work opens up possibilities for further research and development, including research towards additional lightweight alternatives to containerized workloads in the edge.

In conclusion, extending Kubernetes workloads to cover multiple backends shows that containers are not always the ideal choice, and other backends can significantly improve performance in several cases. However, automatically determining the ideal choice of runtime, when implemented, remains future work.

REFERENCES

Abeni, L. (2023). Real-time unikernels: A first look. In *Lecture Notes in Computer Science*, pages 121–133. Springer Nature Switzerland.

Al-Naday, M., Goethals, T., and Volckaert, B. (2022). Intent-based decentralized orchestration for green energy-aware provisioning of fog-native workflows. In *2022 18th International Conference on Network and Service Management (CNSM)*. IEEE.

Anjali, Caraza-Harter, T., and Swift, M. M. (2020). Blending containers and virtual machines. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM.

Bellard, F. (2005). Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, page 46. Califor-nia, USA.

BytecodeAlliance (2023). Wasm micro runtime - build iwasm.

Čilić, I., Krivić, P., Žarko, I. P., and Kušek, M. (2023). Performance evaluation of container orchestration tools in edge computing environments. *Sensors*, 23(8):4008.

de Velp, G. E., Rivière, E., and Sadre, R. (2020). Understanding the performance of container execution environments. In *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*. ACM.

Goethals, T., Sebrechts, M., Al-Naday, M., Volckaert, B., and Turck, F. D. (2022). A functional and performance benchmark of lightweight virtualization platforms for edge computing. In *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*. IEEE.

Goethals, T., Turck, F. D., and Volckaert, B. (2020). Extending kubernetes clusters to low-resource edge devices using virtual kubelets. *IEEE Transactions on Cloud Computing*.

Habib, I. (2008). Virtualization with kvm. *Linux Journal*, 2008(166):8.

Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., and Zolotarov, V. (2014). {OSv—Optimizing} the operating system for virtual machines. In *2014 usenix annual technical conference (usenix atc 14)*, pages 61–72.

KubeVirt (2023). Building a virtualization api for kubernetes.

Kuenzer, S., Bădoiu, V.-A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, Ş., Răducanu, C., Banu, C., Mathy, L., Deaconescu, R., Raiciu, C., and Huici, F. (2021). Unikraft. In *Proceedings of the Sixteenth European Conference on Computer Systems*. ACM.

Luo, J., Deng, X., Zhang, H., and Qi, H. (2019). Qoe-driven computation offloading for edge computing. *Journal of Systems Architecture*, 97:34–39.

Mavridis, I. and Karatza, H. (2021). Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. *Concurrency and Computation: Practice and Experience*, 35(11).

Ménétrey, J., Pasin, M., Felber, P., and Schiavoni, V. (2022). WebAssembly as a common layer for the cloud-edge continuum. In *Proceedings of the 2nd Workshop on Flexible Resource and Application Management on the Edge*. ACM.

Ray, P. P. (2023). An overview of webassembly for iot: Background, tools, state-of-the-art, challenges, and future directions. *Future Internet*, 15(8).

Sadique, K. M., Rahmani, R., and Johannesson, P. (2020). Enhancing data privacy in the internet of things (IoT) using edge computing. In *Communications in Computer and Information Science*, pages 231–243. Springer International Publishing.

Sebrechts, M., Ramlot, T., Borny, S., Goethals, T., Volckaert, B., and Turck, F. D. (2022). Adapting kubernetes controllers to the edge: on-demand control planes using wasm and WASI. In *2022 IEEE 11th International Conference on Cloud Networking (CloudNet)*. IEEE.

Walli, S. R. (1995). The posix family of standards. *StandardView*, 3(1):11–17.

Wang, X., Du, J., and Liu, H. (2022). Performance and isolation analysis of RunC, gVisor and kata containers runtimes. *Cluster Computing*, 25(2):1497–1513.