Sandboxing Functions for Efficient and Secure Multi-tenant Serverless Deployments

Charalampos Mainas
cmainas@nubis-pc.eu
Nubis PC

Ioannis Plakas
iplakas@nubis-pc.eu
Nubis PC

Georgios Ntoutsos
gntouts@nubis-pc.eu
Nubis PC

Anastassios Nanos
ananos@nubis-pc.eu
Nubis PC

ABSTRACT

Serverless computing has gained significant traction for its ability to streamline development workflows and optimize resource utilization. However, ensuring optimal performance and isolation for workloads in multi-tenant environments remains a critical challenge.

In this work, we identify the need for sandboxing mechanisms to extend the tenancy model of Knative and enhance the security and efficiency of multi-tenant serverless deployments. Existing solutions like gVisor and kata-containers provide a level of isolation but do not meet the requirements for allowing the execution of untrusted workloads in a Knative cluster.

We consider the option of unikernels in serverless environments. We build an end-to-end serverless system based on unikernels and compare its performance and isolation characteristics to existing sandbox solutions. Our initial findings demonstrate that existing sandbox mechanisms exhibit significant overheads. On the contrary, a unikernel-based solution offers a compelling balance between performance and security, achieving identical response times to generic containers.

CCS CONCEPTS

• **Security and privacy** → *Information flow control*; • **Software and its engineering**;

ACM Reference Format:
Charalampos Mainas, Ioannis Plakas, Georgios Ntoutsos, and Anastassios Nanos. 2024. Sandboxing Functions for Efficient and Secure Multi-tenant Serverless Deployments. In *The 2nd Workshop on SErverless Systems, Applications and MEthodologies (SESAME '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3642977.3652096

1 INTRODUCTION

The advent of serverless computing [13] marks a significant departure from traditional approaches to application deployment. Developers are no longer responsible for managing and provisioning the underlying infrastructure of their services and applications. Moreover, serverless computing promotes the transition of software design from traditional monolithic architectures to more modular and distributed architectures. According to the serverless paradigm, an application consists of a set of small independent services, where each performs a specific task and interacts with other services to perform more complex functionalities. These services follow a stateless, event-driven model, waiting for particular events, such as a new connection, to trigger their execution. As a result, serverless architectures can easily scale up or down services, on-demand, optimizing resource utilization [23], leading to cost-efficient and highly scalable solutions.

Serverless services are packaged, distributed, and deployed as containers. Containers streamline the creation and distribution of software components, by packaging the software code, along with all the necessary libraries, dependencies, and configuration. Additionally, they provide a lightweight virtualization solution for the execution of a service, enhancing resource utilization.

Nevertheless, containers share the same operating system (OS) kernel; therefore, compromising one container could impact the security of other containers running on the same host. Numerous CVEs related to privilege escalation, where a malicious user can break out of the isolation boundaries of a container, appear every few weeks in Security Advisory Bulletins [20]. Consequently, the need for stronger isolation guarantees becomes crucial [24] to prevent security breaches and unauthorized access between containers.

Due to the loose isolation of containers, the industry is actively exploring and adopting more robust isolation mechanisms in order to enhance the security and reliability of multi-tenant containerized environments. Technologies such as gVisor [12] or Kata Containers [15], provide enhanced isolation [22] without sacrificing the benefits of containerization. They use additional software barriers and/or more traditional virtualization technologies, such as lightweight Virtual Machines (microVMs) to isolate each container. Such approaches aim to strike a balance between the efficiency of containers and the robust isolation required for secure and compliant multi-tenant environments.

On the other hand, the additional layers for enhancing isolation lead to higher resource overheads [25] than containers. Encapsulating an entire operating system within each microVM contributes to higher memory and storage consumption. Additionally, the startup times of microVMs can no longer compare to the rapid instantiation

SESAME '24, April 22, 2024, Athens, Greece

ACM ISBN 979-8-4007-0545-8/24/04…$15.00
https://doi.org/10.1145/3642977.3652096

of containers. In particular, the slow spawn time of microVMs imposes an essential trade-off for serverless platforms since it affects flexibility and scalability.

In an effort to reduce microVMs' overhead, researchers and engineers are working on slimming down and optimizing the guest OS. Ideally, the guest OS should contain only the necessary components for the proper functionality of the service. Indeed, a fairly new concept in OS, called unikernels [17, 18], could provide a viable solution. Unikernels are lightweight and specialized Operating Systems designed to run a single application, providing a minimal and efficient runtime environment. However, although there have been considerable attempts to integrate unikernels into the cloud-native ecosystem [5, 9, 16], widespread adoption is still underway.

In this work, we examine and evaluate existing virtualization technologies in serverless environments. In addition, we design and implement an end-to-end serverless architecture based on unikernels. Moreover, we introduce a novel sandboxing mechanism for serverless environments, separating the user-provided code from the platform-specific stack. For that purpose, we integrate generic containers and unikernels in our custom-built unikernel container runtime, *urunc* [4]. Our contribution is two-fold:

- First, we provide hardware-enabled isolation (through virtualization) between the user function (user-container) and the Knative stack (queue-proxy container) while letting them co-exist in the Knative Function Pod.
- Second, we reduce the Service Response Latency, by packaging the user function in a unikernel.

To validate our approach, we evaluate the response times of Knative services on a number of scenarios. We customize *kperf*, a tool provided by the Knative software stack to capture results, comparing the deployment of Knative services as generic containers, sandboxed containers and unikernels. We use gVisor, kata-containers, and our custom-built unikernel container runtime (*urunc*) as sandboxing mechanisms. We focus on initial instantiation (cold-boot) and discuss the benefits of each approach.

The rest of the paper is organized as follows: Section 2 introduces the architecture of a typical serverless system. Section 3 outlines the motivation of our work, while Section 4 explains how we integrate unikernels in serverless. In Section 5, we discuss the experiment setup, while in Section 6 we presents the evaluation scenarios and the results we capture. Section 7 concludes, presenting our plan and next steps.

2 SERVERLESS ARCHITECTURE

This section briefly presents the basic blocks of a serverless architecture. In particular, we provide background for Kubernetes (K8s), which acts as the orchestrator, and Knative, a popular serverless framework based on K8s.

2.1 Kubernetes

Kubernetes (K8s) [6] stands as the preeminent open-source container orchestration platform that plays a pivotal role in the seamless deployment and management of containerized applications, including services in serverless architectures. At its core, K8s automates container deployment, scaling, and operation, providing a robust infrastructure for building, running, and orchestrating

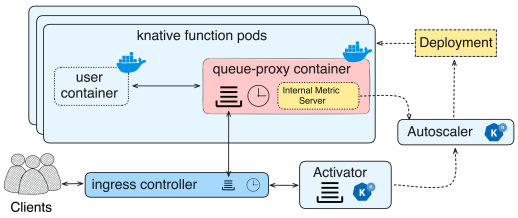


Figure 1: Building blocks of the Knative architecture

distributed systems. In serverless deployments, K8s serves as a foundational layer for container orchestration, efficiently managing the life-cycle of micro-services and serverless functions. Its ability to dynamically scale workloads, handle fail-overs, and optimize resource utilization aligns seamlessly with the ephemeral nature of serverless computing,

The architecture of K8s consists of several vital components that manage containerized applications across a cluster of nodes. At its core is the API server, scheduler, and controller manager. The API server acts as the front end for the K8s control plane, processing API requests and serving the K8s API. The scheduler assigns workloads to nodes based on resource availability, while the controller manager oversees cluster state maintenance. On the worker nodes, the kubelet acts as an agent between the control plane and the container runtime, which executes the containers inside pods. A Pod is the smallest deployable unit in the K8s ecosystem, representing one or more containers that share storage and networking.

2.2 Knative

Knative is an extension to K8s that specifically targets the needs of serverless workloads. It provides an additional abstraction layer and automation for deploying and managing serverless functions or applications. Knative enhances a K8s platform with higher-level abstractions and tooling tailored for serverless computing. Knative abstracts away much of the manual configuration and management tasks, offering a developer-friendly experience.

A key feature of Knative is *scale to zero*, meaning that resource allocation for a service occurs after its invocation. Knative will automatically release the service's resources when a service becomes idle. This feature aligns closely with the serverless computing paradigm, where resources are allocated on demand, contributing to cost efficiency and optimal resource utilization.

Knative introduces the concepts of *serving* and *eventing*; Knative serving facilitates the management and deployment of serverless containers, whereas eventing facilitates event-driven workflows. Figure 1 showcases the high-level architecture of Knative. The basic building blocks, along with their functionality, are: (a) the *Activator*, which starts or stops pods in response to incoming requests to a Knative *Service*; (b) the *Autoscaler*, which manages the dynamic scaling of Knative Services based on demand; (c) the *Queue Proxy*, which coordinates the flow of requests to and from the Knative Service. It also communicates with the Autoscaler and Activator to manage the lifecycle of pods; (d) the *Controller*, which manages the lifecycle of Knative services, including creation, updating, and deletion; (e) the *Metrics Server*, which collects and exposes metrics related to the performance and behavior of Knative services.

Knative's threat model [14] is a work-in-progress document developers consult to determine the severity of a reported exploit. Based on the tenancy model currently in use in the K8s ecosystem [3], Knative assumes a single-tenant per cluster (*Clusters as a service*) as its security model. The project does not consider itself secure for the other tenancy options: *Namespaces as a service*, or *Control planes as a Service*. This option severely limits the applicability of Knative, as, for instance, a serverless provider willing to use Knative should expose a dedicated control plane for each tenant willing to submit arbitrary workloads to the underlying infrastructure.

3 MOTIVATION

In this section, we outline the motivation behind our work. Specifically, we briefly describe the existing container sandboxing mechanisms and highlight how our approach extends single-/multi-tenant assumptions.

As mentioned in Knative's threat model, the mitigation for allowing users to submit untrusted workloads to a Knative cluster is enhanced isolation mechanisms. Container runtimes, such as *Kata-containers* and *gVisor*, provide such mechanisms by using VM-based isolation for containers.

kata-containers. Kata Containers is an open-source container runtime project designed to combine containers' lightweight and fast startup characteristics with the strong isolation of virtual machines (VMs). Kata Containers achieves this by using lightweight VMs to encapsulate each containerized application. These lightweight VMs offer a more secure boundary between containers while preserving the agility and efficiency of traditional container runtimes.

gVisor. gVisor is an open-source container runtime that provides lightweight and secure container isolation. Developed by Google, gVisor offers an additional layer of security by implementing a user-space kernel between the containerized application and the host operating system. This user-space kernel acts as a boundary for containers, ensuring that all system calls made by the container are intercepted and processed within the user space rather than directly interacting with the host kernel. This approach enhances security by isolating containers from the underlying host operating system, making it suitable for running untrusted workloads in multi-tenant environments.

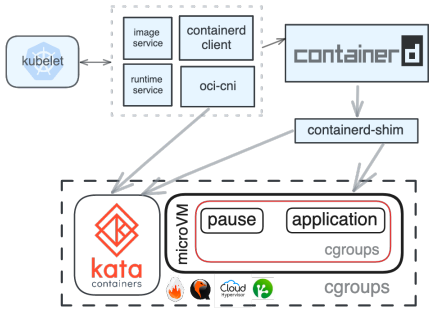


Figure 2: Application execution flow on K8s, using kata

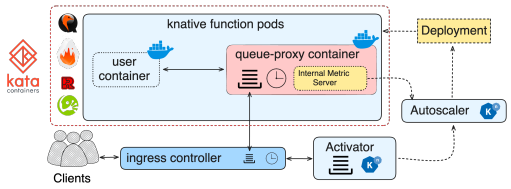


Figure 3: Isolation boundaries for a Knative Pod with kata

The sandboxing mechanisms mentioned above have native integration with CRI and, thus, can provide their functionality to K8s Pods, the smallest unit of reference in the K8s object model. Creating containers inside a K8s pod does not differ between generic and sandboxed container runtimes. In particular, the pause and application containers co-exist inside the sandbox. Figure 2 visualizes the setup of containers in a K8s Pod using Kata-containers. The setup is the same in the case of gVisor, as well.

However, the VM-based sandboxing mechanisms induce significant overhead in the instantiation times of containers. Slow spawn times reduce the responsiveness and the scalability of a serverless system. On the other hand, Therefore, there is a trade-off between isolation and performance in serverless computing. Our proposal, aims to eliminate this trade-off, providing VM-based isolation, while maintaining fast spawn times.

In addition, while sandboxing mechanisms like gVisor or Kata-containers reduce the exposed attack surface of the rest of the system, they do not address a potential vulnerability in the Knative stack. In particular, kantive's *queue-proxy* co-exists with the untrusted workload: the user-submitted workload, packaged in the *user-container* inside a sandbox Figure 3 illustrates the coexistence of *queue-proxy* with *user-container*. In an effort to address the aforementioned potential vulnerability, we change the design of Knative, by separating the *queue-proxy* container from the *user-container*

4 UNIKERNELS FOR SERVERLESS

This section presents our approach on using unikernels for serverless architectures and, in particular, the mechanism to separate user-submitted code (the *user-container*) from the platform's software stack (the *queue-proxy* container).

4.1 Unikernels

Unikernels are specialized single-address space operating systems designed to run a single application, providing a minimal and efficient runtime environment. Unlike traditional operating systems that support a wide range of applications and services, unikernels include only the essential components necessary for a specific workload. As a result, unikernels achieve a small memory and storage footprint. Moreover, by design, unikernels significantly reduce the attack surface too. The small attack surface and strong VM isolation minimizes the impact of security vulnerabilities and reduces the risk of exploitation. Unikernels represent a paradigm shift towards lightweight and specialized runtime environments, offering security, efficiency, and performance benefits.

Due to their near-instant spawn times, unikernels support rapid scaling and elasticity, aligning well with dynamic and on-demand

computing paradigms. Moreover, efficient provisioning and deprovisioning resources make unikernels suitable for scenarios where workloads fluctuate. All these characteristics make unikernels an attractive solution in scenarios where fast application deployment and responsiveness are critical, such as serverless computing.

4.2 urunc

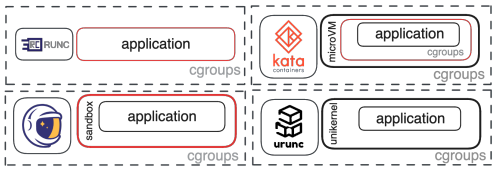


Figure 4: Application execution modes for generic containers (top left), gVisor sandbox (bottom left), kata-containers (top right) and urunc (bottom right)

urunc is a custom-built, low-level container runtime that can spawn unikernels. urunc complies with OCI[1] and expects OCI-compatible images that contain a unikernel binary. urunc leverages a unique feature of unikernels: Unikernels are self-contained; an application runs along with its library and OS dependencies as a single address-space VM image, combining the systems and application software stack. Therefore, urunc maps each application/unikernel to a single container, thus, enabling direct management of applications from the container runtime itself. Figure 4 visualizes the sandboxing properties of urunc, along with existing container runtimes (generic and sandboxed).

4.3 Unikernels and Knative

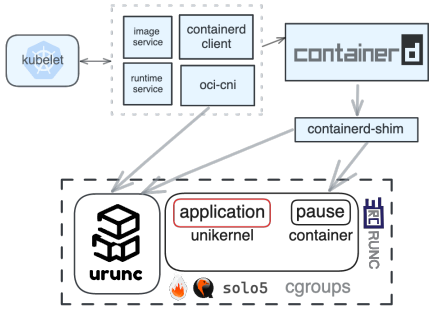


Figure 5: Unikernel execution flow on K8s, using urunc

A key aspect of urunc is the separation of containers from unikernels. In particular, urunc determines the type of the container using OCI annotations. Depending on the type of the container, urunc either handles the spawning by itself or forwards the request to a generic container runtime (e.g., runc). In that way, urunc can seamlessly integrate with K8s, allowing generic container runtimes to handle platform-specific containers. We visualize the execution flow of spawning containers with urunc over K8s in Figure 5.

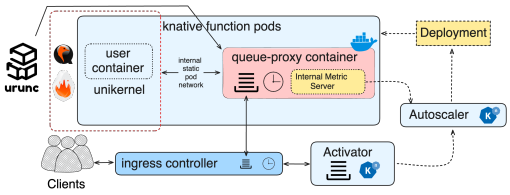


Figure 6: Isolation boundaries for a Knative Pod with urunc

We exploit the inherent separation of containers within the same pod that *urunc* offers by spawning the *queue-proxy* container as a generic container. In contrast, the *user-container* is a unikernel. Figures 3 and 6 visualize the isolation boundaries for a Knative Function Pod, deployed on a bare-metal K8s, over kata-containers and urunc.

Such a design facilitates the complete separation between the platform stack and the user's function. Moreover, the user's function executes inside a unikernel, minimizing the attack surface and inheriting strong VM-based isolation from the rest of the stack. As a result, we can significantly reduce the exploiting capabilities of malicious parties.

Although the merits of unikernels have already been quantified in serverless frameworks [7, 8, 19, 21], to the best of our knowledge, there has been no cloud-native integration with popular, state-of-practice tools such as k8s and Knative.

5 EXPERIMENT SETUP

In this section, we describe the experiment scenario, elaborate on the aspects of the software stack we benchmark, and present the experimental tests, as well as the tools we used for capturing the results.

5.1 Knative setup

We focus our setup around Knative Serving, the Knative component designed for deploying and managing serverless applications. Knative Serving introduces concepts like Services, which represent serverless applications, and Revisions, which represent different versions of those applications. Knative Serving's autoscaling capabilities ensure dynamic resource allocation based on incoming traffic, optimizing efficiency and cost. We chose to focus on Knative Serving for simplicity. The benefits of function isolation through sandboxing and fast spawn times through unikernels would also apply equally to Knative Eventing.

Our setup includes a generic Knative installation on a bare-metal K8s cluster. We pin services to a single node to avoid network noise. Essentially, we create Knative Services and we evaluate: (a) service response times, (b) the degree of scaling, and (c) the maximum number of containers (generic or sandboxed) the host can sustain. To generate the requests needed for function spawning and to capture the measurements that refer to the above evaluation parameters, we customize the default benchmark tool of Knative, *kperf*.

[1]OCI stands for Open Container Initiative, an open governance structure to create open industry standards around container formats and runtimes.

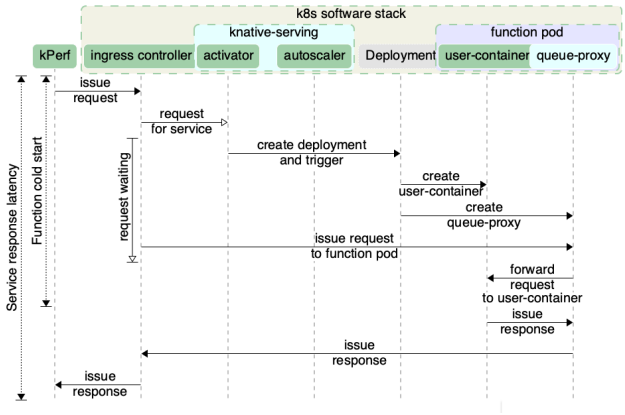


Figure 7: Request servicing on Knative (cold instantiation)

5.2 kperf

kperf is a tool designed for Knative benchmarking. It helps generate workloads for Knative services and gives measurement results about the duration of resource creation based on server-side timestamps. As kperf was initially designed for scale and load tests, we had to customize the tool to account for our specific experiment parameters: *1:1 mapping* Knative does not distinguish multiple instances of the same function. As a result, when scaling to more than one instance of a specific function, we could not verify that separate functions are handling requests. *custom HTTP headers* kperf points to a specific function hostname (e.g., `service.svc.local`) that, via DNS (local or remote), is resolved to the ingress controller of the k8s cluster, which, in turn, forwards the request to the specific endpoint of the service function. To reduce the networking/DNS resolving noise from the measurements, we bypass this by adding the particular service function hostname as an HTTP header in the request. *timeouts addition* kperf sends HTTP requests to endpoints/services to spawn the desired serverless workloads. Despite that, it does not provide a timeout option. So, when pushing the limits of the nodes, responses can be quite expensive in terms of latency, slowing down the testing workflow.

5.3 Knative Service Function

To reduce compute and network noise, we consider a simple HTTP reply function [1, 2] as the application.

Figure 7 visualizes the steps that comprise the service latency measured in a sequence diagram, presenting the interaction of individual components, as well as the time spent at each part of the flow: (1) kperf issues the request that reaches the ingress controller, (2) the request traverses the networking stack of Kubernetes and reaches the activator, (3) the activator triggers the deployment of a function pod, (4) upon the creation of the function pod, the request reaches queue-proxy, (5) queue-proxy forwards the request to the user-container, (6) the user-container replies to kperf.

6 EVALUATION

For our measurements, we use a bare metal server with an AMD EPYC 7502P (Rome, 32 cores) and 128GB RAM. The software stack consists of: a K8s cluster (v1.28.2) with Knative (v1.12), along with the sandbox container runtimes (kata-containers v3.2.0, gVisor release-20231113.0) and urunc v0.2. We use Ubuntu 20.04 as the host OS. To ensure we capture reproducible metrics and eliminate fluctuations in our measurements, we disable the CPU frequency scaling and the turbo feature.

Table 1: Kperf Parameters

Param	Value	Metric	Description
timeout	3	sec	wait for service to be ready
time-interval	90	sec	Duration for each scale-up interval
iterations	30	-	nr of consecutive runs
scale-client-timeout	100	sec	non-responsive service timeout
stable-window	25	sec	metrics average time window

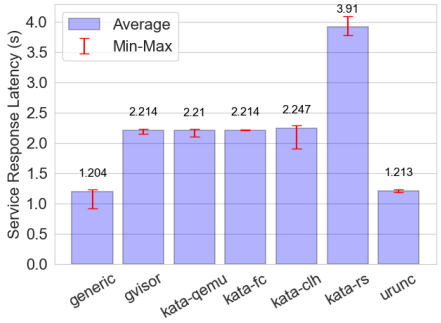


Figure 8: Service Response Latency (single instance)

6.1 Service Response Latency (single instance)

For the first scenario, we capture the total time needed for a single request to reach a non-provisioned Knative service. This metric represents how long the user will wait for a service to respond to an HTTP request when they first access it. The dominating part of this metric is the duration it takes for a serverless function to become operational from the moment it is triggered (cold-boot[10]).

kperf defines Knative Services and triggers them through HTTP requests iteratively. The trigger produces an HTTP response from the user-container each time it is spawned. The time required to receive the response is reported as *Service Response Latency*. To prevent inconsistency in measurements, requests are sent when the Knative service endpoints become ready, while re-triggering occurs after the service has scaled to zero (time-interval parameter). We summarize kperf's parameters for this test in Table 1.

Figure 8 represents the average response latency. An interesting observation is that the sandboxed container runtimes serve the request in 2-2.5s. Moreover, the generic container runtime (runc) serves the request in approximately 1.20 seconds. Additionally, the behavior of urunc is on par with the generic container runtime (runc). Finally, the maximum service request latency for runc and urunc does not exceed 5% of the total latency.

However, when capturing latency results, an important consideration is tail latency: this metric is crucial because it represents the user experience in real-world scenarios. Figure 9 quantifies tail

latency, by plotting the 99th percentile of the Service Response Latency.

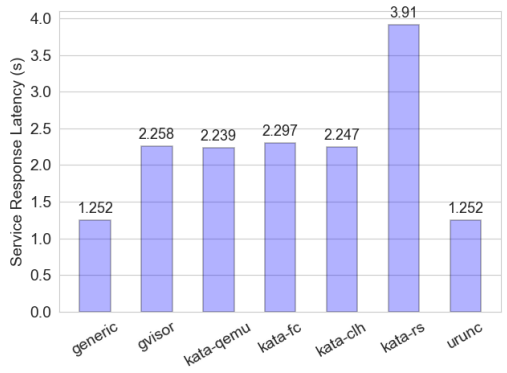


Figure 9: Service Response Latency for the 99th percentile

Figure 9 shows that runc and runc are identical (1.25s), whereas all other sandbox mechanisms exhibit almost twice the latency (2.23s to 3.91s). The fact that runc sustains equivalent performance to generic containers (such as runc) is significant, as it proves that using unikernels can enable multi-tenancy, without sacrificing performance.

6.2 Concurrent servicing (multiple instances)

In this scenario, we assess the footprint and responsiveness of a Knative Service by scaling to many instances for each container runtime.

We use kperf to initiate multiple HTTP requests concurrently. We aim to gather average latency data for spawning multiple Knative Services. To ensure sufficient time for response collection, we extend the stable-window parameter to 300.

To mitigate measurement inconsistencies, we set the time interval to 95 seconds, ensuring all services are scaled to zero before the next iteration begins.

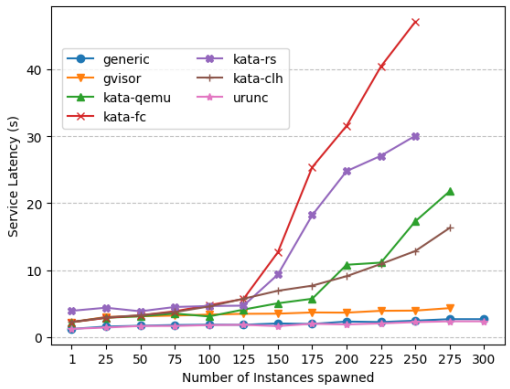


Figure 10: Service Response Latency (multiple instances)

Figure 10 presents the average service response latency (sec) for each container runtime as a function of the number of concurrent instances. As we can see in the Figure, (a) generic (runc), gvisor, and runc exhibit similar behavior when increasing the number of instances; (b) runc and generic (runc) show indistinguishable response latency, even when scaled up to 300 services; (c) gvisor introduces an average latency increase of approximately +1.5s compared to runc, accounting for twice the latency for instances up to 125; (d) kata-containers display an overhead ranging from 2x to 3x latency compared to runc for up to 125 instances, and more than 10x for more than 200 instances.

Despite the defined target number of services, we observe that some instances fail to respond when we reach 125 instances. Figure 11 summarizes the number of actual instances that responded in our test for various runtimes. Almost all container runtimes fail to produce equivalent service instances to the respective target. Several factors contribute to the above result: system load, K8s settings, activator congestion, and interaction between the system components are not always efficient [11]. We further investigate this issue, focusing on runc and runc in the next section.

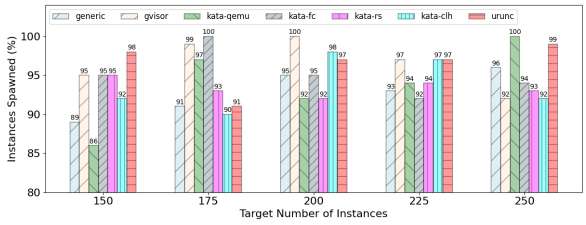
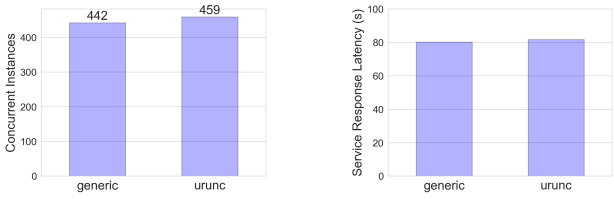


Figure 11: Percentage of Actual Instances spawned vs Target Instances

6.3 Pushing the scaling limits

To determine the maximum amount of services supported on our testbed hardware, we increase the number of services to 500 for runc and capture: the number of actual services spawned and the service response latency in this context. We compare these results with the generic container runtime.



(a) Instances spawned **(b) Service Response Latency**

Figure 12: Instances (500) and Service Response Latency

Figure 12a plots the number of spawned instances for the generic container runtime(runc) and runc with a target of 500. We can see that runc can spawn as many instances as runc, enabling the sandboxing of user code without imposing any additional overhead regarding memory / CPU footprint.

To validate that the services are responsive and assess the latency imposed by so many services running simultaneously, we plot the average response latency for these services. Figure 12b shows that urunc can sustain low response latency compared to the generic container runtime (runc), even with 450 instances running.

7 CONCLUSIONS

Our work addresses the challenges of optimizing performance and isolation for workloads in multi-tenant serverless environments, focusing on Knative. We introduce a novel approach to sandboxing functions that separates the user-provided code from the platform-specific stack, achieving hardware-enabled isolation between the user function and the Knative stack while reducing service response latency by packaging the user function in a unikernel.

Our approach extends the state of practice by introducing a custom-built unikernel container runtime (urunc). We provide benchmarks comparing the performance and isolation characteristics of different sandbox solutions in a serverless system. We highlight the trade-offs and limitations of different sandboxing approaches for multi-tenant serverless deployments. Our evaluation shows that Unikernels are a great fit in serverless computing, combining the isolation principles of VMs without the overhead and management burden of generic, full virtualization stacks. Our work contributes to the ongoing efforts to optimize performance and enhance isolation and efficiency for workloads in multi-tenant serverless environments.

Building on this work, we plan to further analyse the service response latency of Knative and alternative serverless frameworks and provide insights into where time is spent, and how to optimize cold function invocation. Furthermore, an important aspect of the evolving paradigm of cloud-native workloads involves exploring compute-intensive tasks. Specifically, we aim to examine how hardware acceleration can be seamlessly incorporated into a FaaS programming model.

ACKNOWLEDGMENTS

This work has been funded in part by the Horizon Europe research and innovation programme of the European Union, under grant agreement no 101092912, project MLSysOps.

REFERENCES

[1] 2023. HTTP reply function in C. https://github.com/nubificus/app-httpreply/blob/nbfc-knative/main.c
[2] 2023. HTTP reply function in go. https://github.com/nubificus/helloworld-knative/blob/main/hello.go
[3] 2023. K8s tenancy model. https://kubernetes.io/blog/2021/04/15/three-tenancy-models-for-kubernetes/
[4] 2023. urunc: A unikernel container runtime. https://github.com/nubificus/urunc
[5] Alexander Jung, Unikraft. 2022. Beyond Orchestration: The Cloud Native Runtimes Ecosystem for Performance and Security. https://kccncna2022.sched.com/event/182OM
[6] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* 59, 5 (apr 2016), 50–57. https://doi.org/10.1145/2890784
[7] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages. https://doi.org/10.1145/3342195.3392698
[8] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. 2019. USETL: Unikernels for Serverless Extract Transform and Load Why should you settle for less?. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hangzhou, China) *(APSys '19)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/3343737.3343750
[9] Gaulthier Gain, Cyril Soldani, Felipe Huici, and Laurent Mathy. 2022. Want more unikernels? inflate them!. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 510–525. https://doi.org/10.1145/3542929.3563473
[10] Muhammed Golec, Guneet Kaur Walia, Mohit Kumar, Felix Cuadrado, Sukhpal Singh Gill, and Steve Uhlig. 2023. Cold start latency in serverless computing: A systematic review, taxonomy, and future directions. *arXiv preprint arXiv:2310.08437* (2023).
[11] Tim Goodwin, Andrew Quinn, and Lindsey Kuper. 2023. What goes wrong in serverless runtimes? A survey of bugs in Knative Serving. In *Proceedings of the 1st Workshop on SErverless Systems, Applications and MEthodologies* (Rome, Italy) *(SESAME '23)*. Association for Computing Machinery, New York, NY, USA, 12–18. https://doi.org/10.1145/3592533.3592806
[12] Google. 2018. gVisor. Documentation website.. https://gvisor.dev/docs/
[13] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).
[14] Julian Friedman. 2020. Knative Threat Model. https://github.com/knative/community/blob/main/working-groups/security/threat-model.md
[15] Kata Containers Community. 2019. kata-containers. Splash page. https://katacontainers.io
[16] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu, and Felipe Huici. 2021. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 376–394. https://doi.org/10.1145/3447786.3456248
[17] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: library operating systems for the cloud. *SIGARCH Comput. Archit. News* 41, 1 (mar 2013), 461–472. https://doi.org/10.1145/2490301.2451167
[18] Anil Madhavapeddy and David J. Scott. 2014. Unikernels: the rise of the virtual library operating system. *Commun. ACM* 57, 1 (jan 2014), 61–69. https://doi.org/10.1145/2541883.2541895
[19] Chetankumar Mistry, Bogdan Stelea, Vijay Kumar, and Thomas Pasquier. 2020. Demonstrating the Practicality of Unikernels to Build a Serverless Platform at the Edge. In *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 25–32. https://doi.org/10.1109/CloudCom49646.2020.00001
[20] MITRE. 2024. CVE-list related to containers. CVE list of vulnerabilities related to the term 'containers'.. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=containers
[21] Felix Moebius, Tobias Pfandzelter, and David Bermbach. 2024. Are Unikernels Ready for Serverless on the Edge? arXiv:cs.DC/2403.00515
[22] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2019. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* 4, POPL, Article 32 (dec 2019), 32 pages. https://doi.org/10.1145/3371100
[23] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Comput. Surv.* 54, 11s, Article 239 (nov 2022), 32 pages. https://doi.org/10.1145/3510611
[24] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 121–135. https://doi.org/10.1145/3297858.3304016
[25] Vincent van Rijn and Jan S. Rellermeyer. 2021. A fresh look at the architecture and performance of contemporary isolation platforms. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 323–335. https://doi.org/10.1145/3464298.3493404