Paper Presentation:

Presentation Date- 11/26/2024

Sandboxing Functions for Efficient and Secure Multi-tenant Serverless Deployments

Paper Published on 2024-04-22

Soumith Kondubhotla- ksoumith@uw.edu
Siva Srinivasa Aditya Jayanti- sjayan@uw.edu
Sri Sailesh Mylavarapu- srisail@uw.edu

Table of Content

1. Introduction
2. Proposed Solution
3. What is serverless
4. Problem with containers
5. Unikernels
6. Serverless Architecture
7. Motivation
8. Unikernels for serverless
9. Experimental Setup
10. Evaluation
11. The benefits of Sandboxing
12. Conclusion

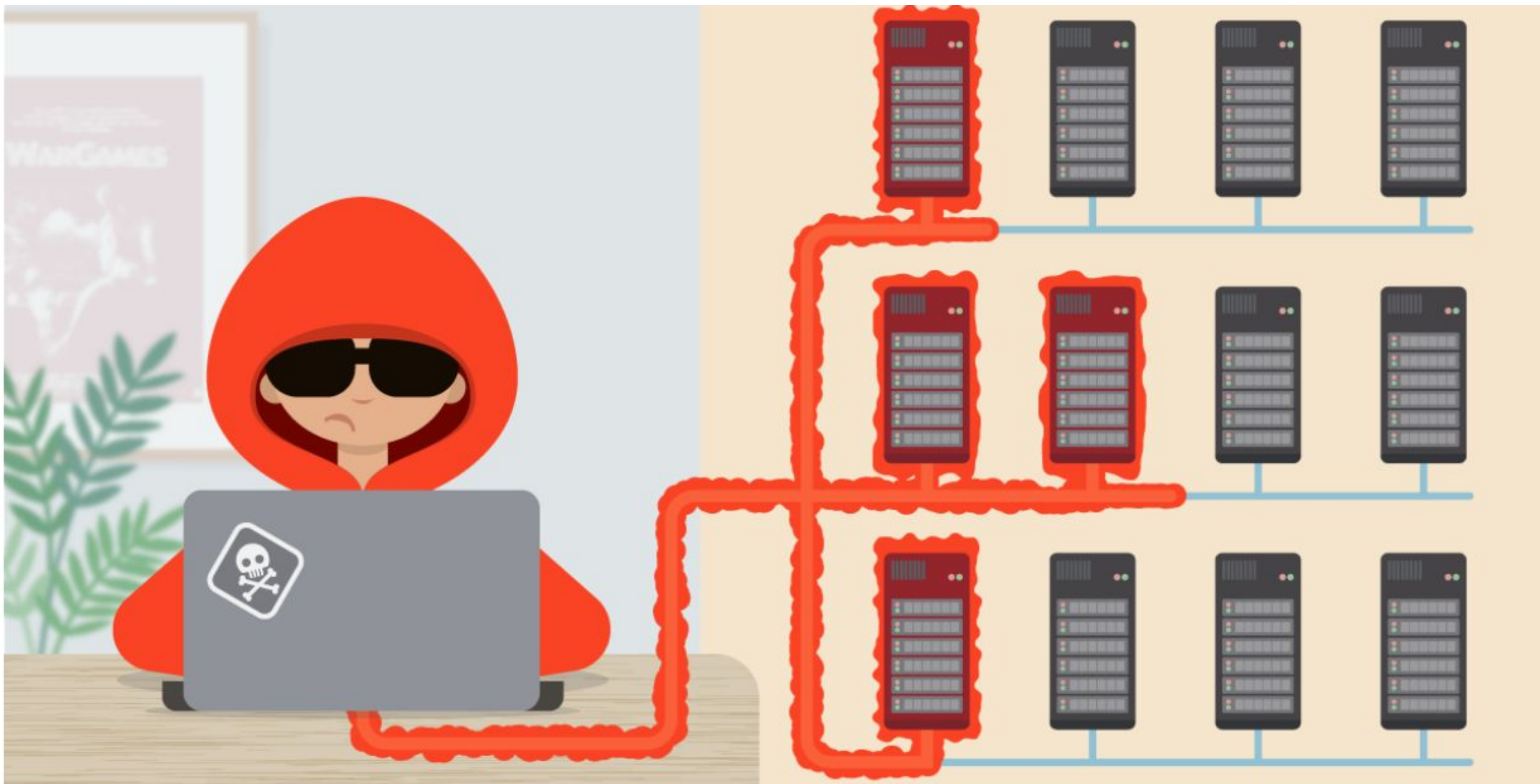
Overview

- Serverless computing optimizes resource usage and simplifies development, but isolation in multi-tenant environments is a key 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.

Why SandBoxes?

Drawbacks of traditional containers

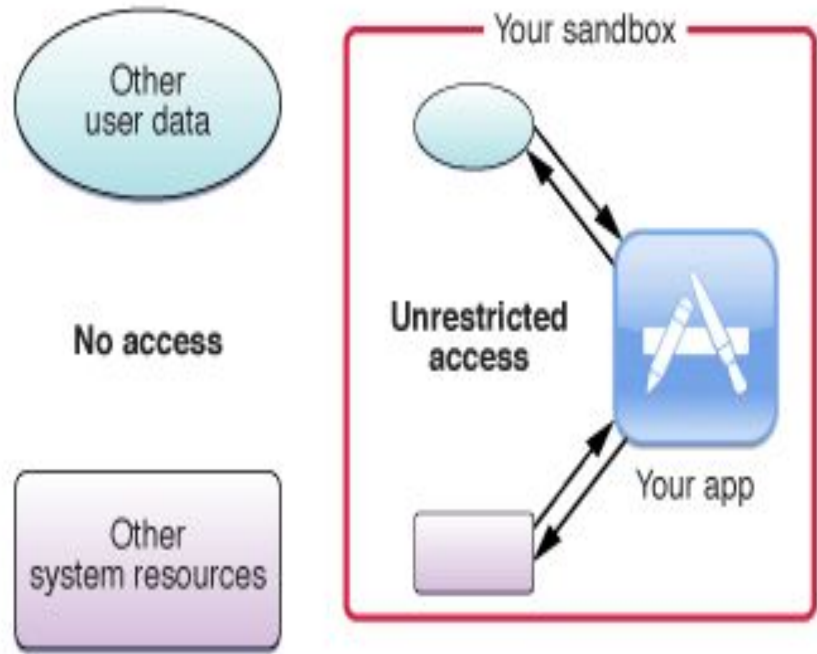
- Containers do provide resource restriction and isolation using technologies like namespaces. This prevents the container from accessing unauthorized parts of the system and ensures that each container has limited access to files, network, CPU, and memory.
- However, containers share the host kernel, which makes their isolation less strict compared to typical sandboxing techniques that focus primarily on security. Sandboxing, on the other hand, is generally stricter, offering deeper isolation



Why SandBoxes?

- Sandboxing is a technique to run code or applications in a restricted environment (sandbox) where they have limited access to system resources and are isolated from the host system. The purpose of sandboxing is to provide security by limiting the potential damage if the application or code behaves maliciously.
- Key Features:

- Resource Restriction: Sandboxing runs code in a way that restricts its access to system resources such as files, network, or hardware devices. This prevents the code from affecting other parts of the system.
- Security Focused: Unlike containers and VMs, sandboxing primarily focuses on security and limiting what the application can do to the host system.
- No Full OS Requirement: Sandboxed environments do not require a full OS but rather use system-level configurations and restrictions to provide isolation.



Introduction

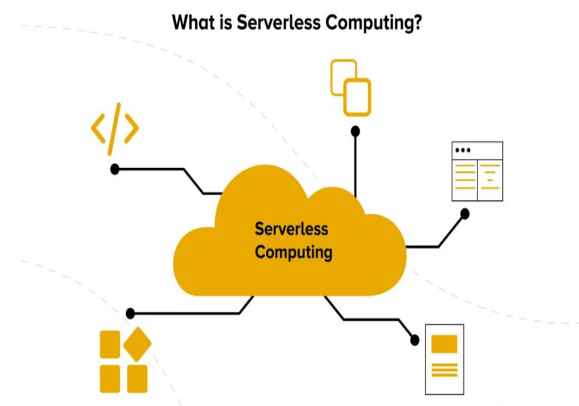
- Serverless computing optimizes resource usage and simplifies development, but isolation in multi-tenant environments is a key challenge.
- Current isolation mechanisms like gVisor and Kata Containers add overhead and are insufficient for untrusted workloads in serverless environments.

Proposed Solution

- Use **unikernels** to balance performance and security.
- Introduce **urunc**, a unikernel-based container runtime for better isolation and reduced service response latency.

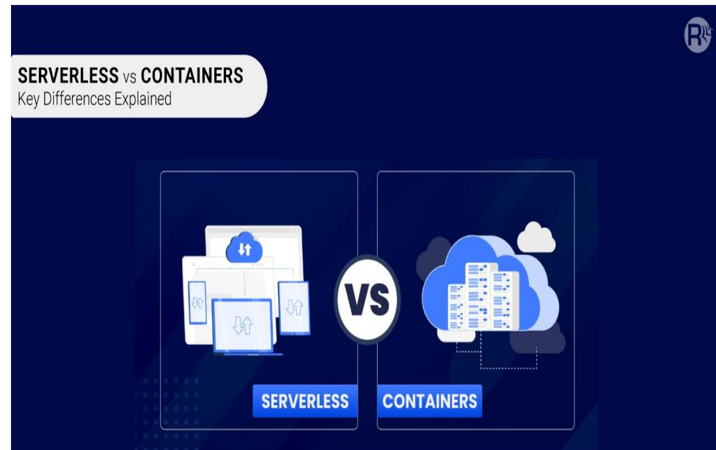
What is Serverless?

- Serverless architectures allow developers to focus on code while the infrastructure is abstracted.
- Applications are modular, stateless, and event-driven, scaling dynamically with demand.



Problem with containers

- Containers share the same OS kernel, leading to security risks (e.g., privilege escalation attacks).
- Enhanced isolation mechanisms like gVisor and Kata Containers exist but add overhead:
  - Higher memory/storage usage.
  - Slower startup times compared to lightweight containers.



Unikernels

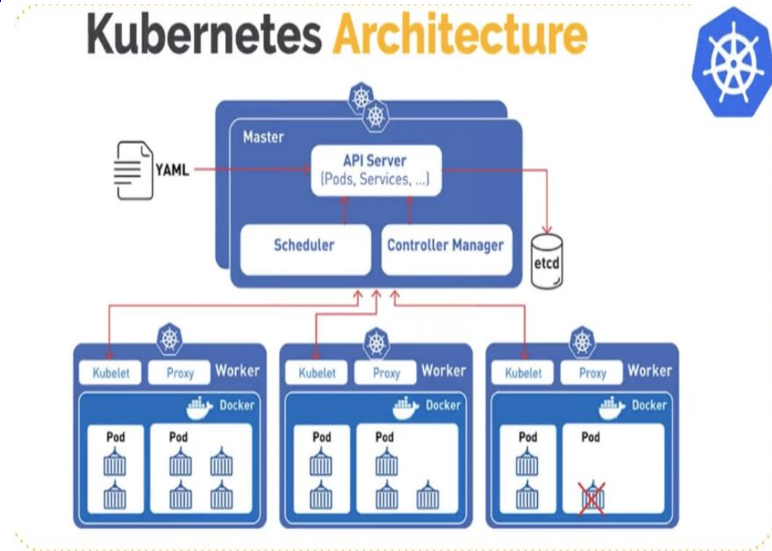
- Lightweight, application-specific OS designed for a single task.
- Advantages:
  - Faster startup.
  - Reduced memory/storage usage.
  - Improved security by minimizing the attack surface.

Serverless Architecture

Kubernetes

What is Kubernetes (K8s)?

- A container orchestration platform managing deployment, scaling, and failovers.
- **Core components:**
- **API SERVER:**Processes API requests
- **Scheduler:**Allocates workloads to nodes
- **Kubelet:**Executes containers within pods.

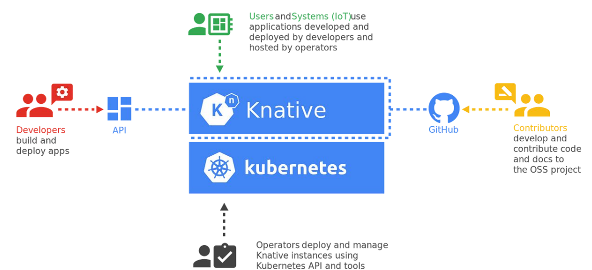


Serverless Architecture

Knative

What is Knative?

- Knative is tailored for serverless, enabling features like "scale-to-zero" and event-driven workflows.
- **Core components:**
- **Activator:**Manages pod lifecycle
- **Autoscaler:**Scales resources dynamically
- **Queue Proxy:**Handles request flow and communicates with other Knative components.



Motivation

Existing Isolation Mechanisms:

Kata Containers

- Encapsulate containers in lightweight VMs for better security
- Trade-off: Slower startup times and higher resource usage.

gVisor

- Intercepts container system calls using a user-space kernel.
- Good for untrusted workloads but slower than native containers.

Knative Challenges:

- Knative’s stack (e.g., queue-proxy) coexists with untrusted workloads, risking security breaches.
- The need for isolation without compromising performance.

Unikernels for Serverless

Unikernels

Specialized OS containing only the components necessary for a single application.

Advantages:

- Near-instant startup times.
- Reduced resource consumption.
- Strong isolation through minimal attack surface.

Unikernels for Serverless

What is urunc?

•Uses Open Container Initiative (OCI) standards for compatibility with Kubernetes.
•A unikernel runtime for serverless environments.
•Handles unikernel and generic containers, assigning tasks based on container type.

Integration with Knative:
•Separates user workloads (unikernels) from Knative’s stack (generic containers like queue-proxy).
•Provides VM-level isolation while maintaining performance.

The Benefits Of Sandboxing

Create and deploy environments

Gain access to advanced networking and support

Prepare for future attacks

Enhance collaboration

Save your company money

Experiment Setup and Evaluation of Knative with Unikernels

Experiment Setup (Overview)

Knative Setup

- Experiment focused on Knative Serving for managing serverless applications.
- Bare-metal Kubernetes cluster (v1.28.2) with Knative Serving (v1.12).
- Services pinned to a single node to reduce network noise.

Metrics Evaluated

- Service Response Latency.
- Scaling Efficiency.
- Maximum Number of Supported Instances.

Sandboxing Functions for Efficient and Secure Multi-tenant Serverless Deployments

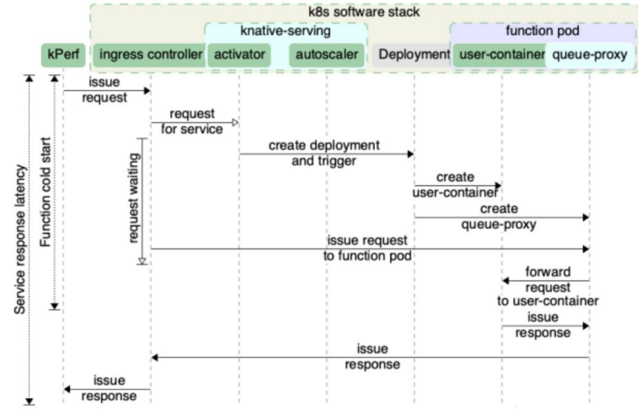


Figure 7: Request servicing on Knative (cold instantiation)

Customizations and Tools

kperf Modifications

- Added custom HTTP headers to reduce DNS resolution delays.
- Introduced timeout options for handling high-latency.
- Ensured unique service instance mapping during scalability tests.

Knative Service Function

- Simple HTTP reply application to minimize compute noise.
- Latency measured from request to response.

Criticisms of Experiment Setup

1. Limited Scope of Benchmarking

- Issue: Single server doesn't reflect real-world environments.
- Suggestion: Use a multi-node cluster or public cloud.

2. Unrealistic Workloads

- Issue: Simple HTTP function isn't representative of real-world tasks.
- Suggestion: Use diverse, real-world workloads.

3. Lack of Documentation

- Issue: Custom kperf modifications are not fully documented.
- Suggestion: Provide open-source access or detailed descriptions.

4. Missing Security Metrics

- Issue: No security tests performed.
- Suggestion: Include security benchmarks like container breakout simulations.

Evaluation Overview

1. Service Response Latency (Single Instance)

- Unikernel runtime (urunc): ~1.25s, similar to generic containers (runc).
- gVisor & Kata Containers: Higher latency (2-2.5s).

2. Concurrent Servicing (Multiple Instances)

- urunc scaled efficiently with low latency.
- gVisor & Kata Containers had delays under high load.

3. Scaling Limits

- urunc and runc supported up to 500 instances effectively.
- Kata Containers struggled with high latencies and reduced responsiveness.

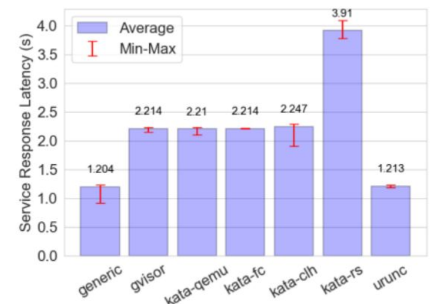
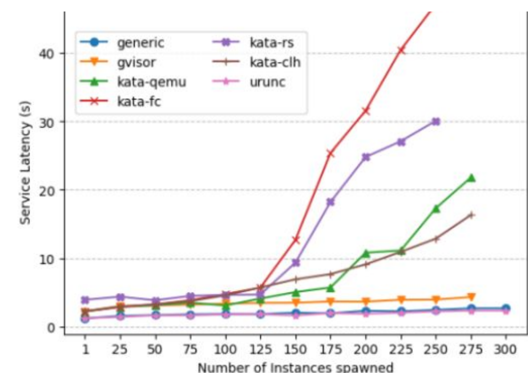


Figure 8: Service Response Latency (single instance)



Key Criticisms of Evaluation

1. Cold-Start Latency Analysis Lacks Detail

- Suggestion: Provide a breakdown of delay stages.

2. Scaling Failures Not Explored

- Suggestion: Conduct root cause analysis for failures at high instance counts.

3. Limited Comparison with Alternatives

- Suggestion: Compare with technologies like Firecracker VMs or Unikraft.

4. Inconsistent Latency Metrics

- Suggestion: Analyze resource utilization to explain discrepancies.

5. Lack of Resource Efficiency Assessment

- Suggestion: Include metrics like CPU, memory, and I/O for each mechanism.

Conclusion and Recommendations

Strengths

- Innovative integration of unikernels into serverless environments.
- Demonstrates potential for improved latency without sacrificing isolation.

Areas for Improvement

- Use realistic workloads and environments.
- Better documentation of methodology.
- Broader analysis of scalability, security, and resource efficiency.

Final Note

- Addressing these gaps will make this work a strong benchmark for unikernel adoption in serverless platforms.

Conclusion

Key Takeaways:

- **Unikernels combine VM-like isolation with container-like performance.**
- **urunc provides a practical solution for secure multi-tenant serverless deployments.**

Future Work:

- **Further optimize cold start latency.**
- **Explore hardware acceleration for computationally intensive workloads.**

Questions?

Thank you!