From Containers to Unikernels:

Navigating Integration Challenges in Cloud-Native Environments

urunc

NUBIS

nubificus

Georgios Ntoutsos, Charalampos Mainas, **Ioannis Plakas**, Anastassios Nanos

{gntouts,cmainas,iplakas,ananos}@nubificus.co.uk

Overview

- About us
- Cloud deployment and application packaging, Containers, Sandbox containers, Unikernels
- Challenges of adopting unikernels
- urunc: a container runtime for unikernels
- Demos
- Evaluation

About us

- Team:
  - researchers, engineers & software developers
- Focus:
  - Virtualization stack
  - Container runtimes
  - Hardware acceleration

Containers have dominated

The de-facto solution for application packaging/deployment in Cloud & Edge

- Lightweight
- Fast spawn times
- Portable
- Usable
- Scalable

LXC

podman

Containers have dominated

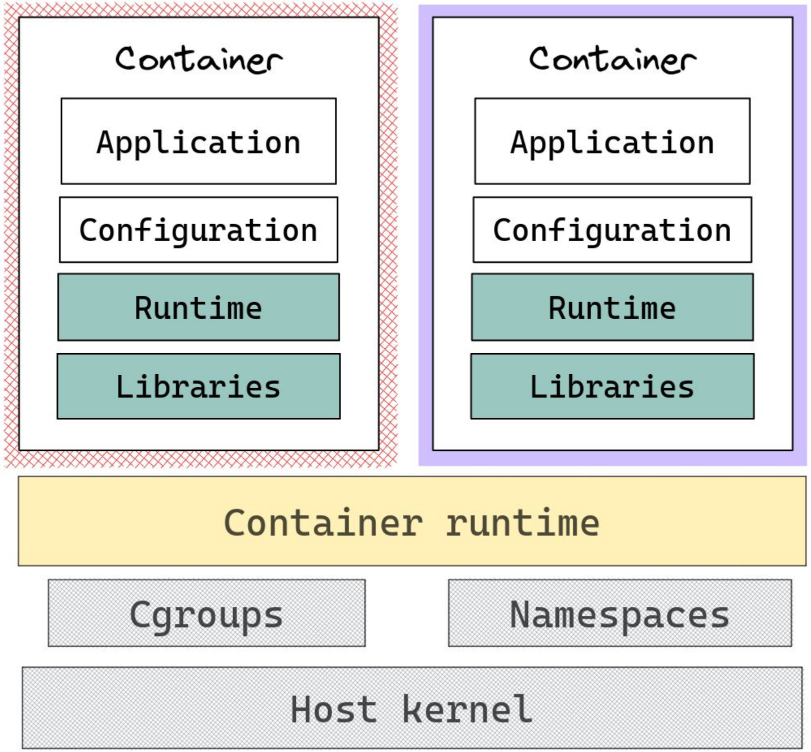
The de-facto solution for application packaging/deployment in Cloud & Edge

- Lightweight
- Fast spawn times
- Portable
- Usable
- Scalable

but…

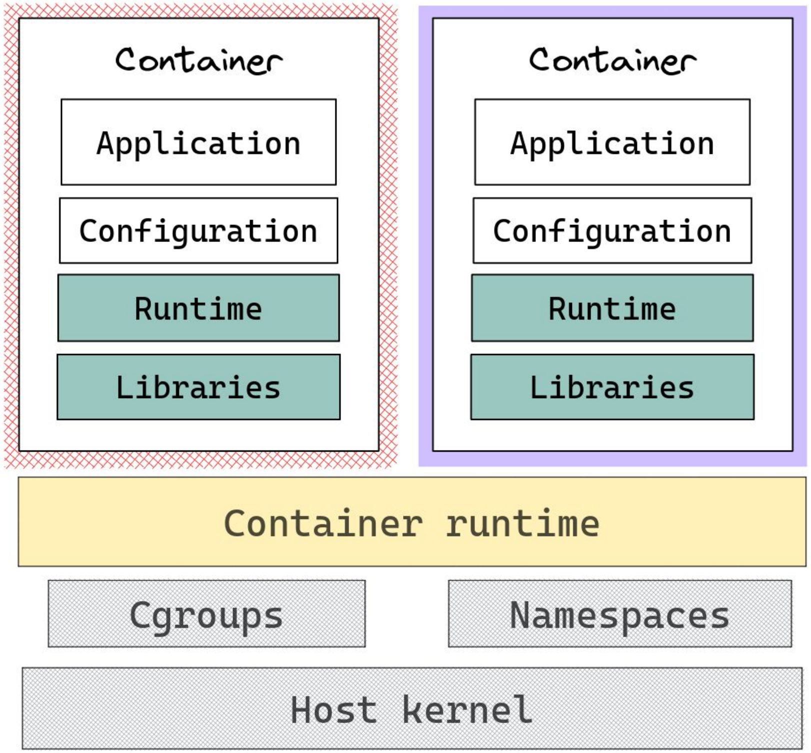
Containers have a major drawback

- Containers do not isolate:
  - Sharing the same kernel
  - Rely on software components for isolation
  - Numerous exploits



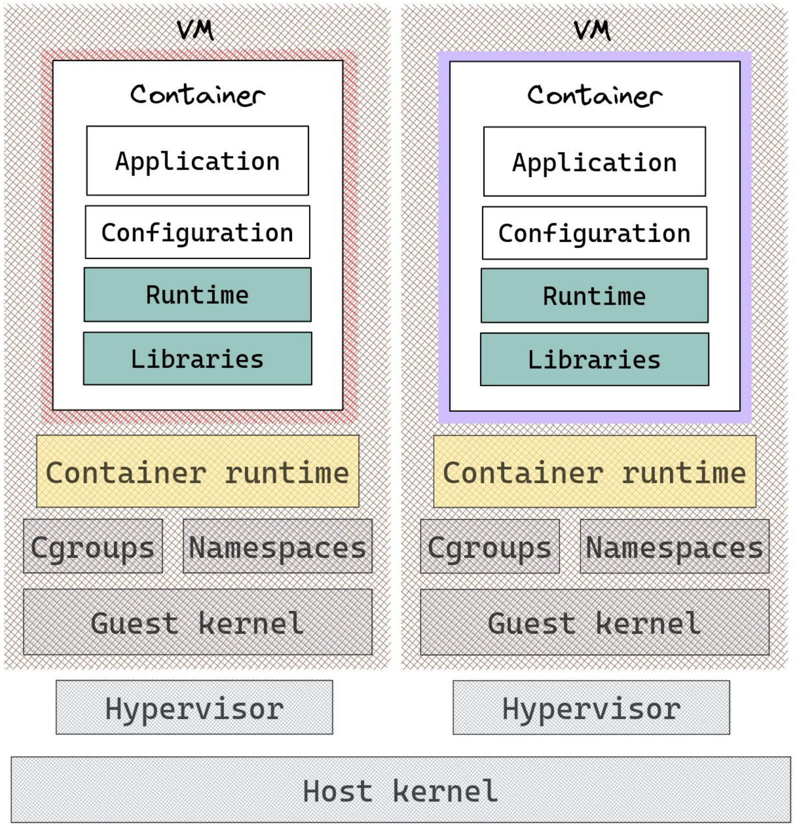
Containers have a major drawback

- Containers do not isolate:
  - Sharing the same kernel
  - Rely on software components for isolation
  - Numerous exploits



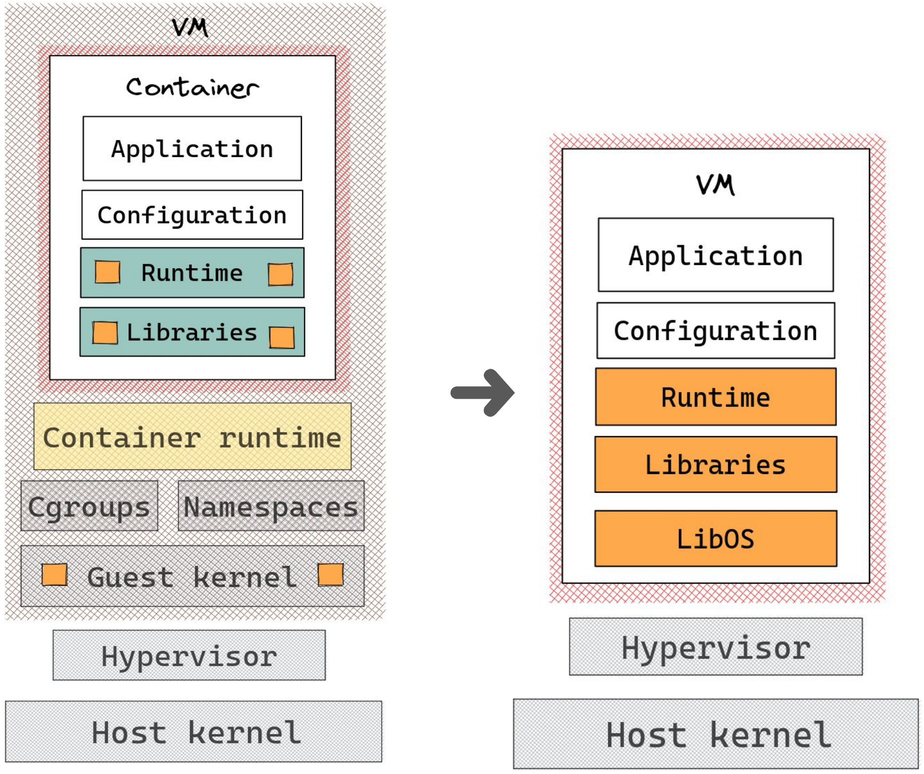
Back to (micro)VMs

- Combine containers and VMs
  - Keep the benefits of containers
  - Isolate containers inside Virtual Machines
- Side effects:
  - Higher overhead
  - Complex system stack



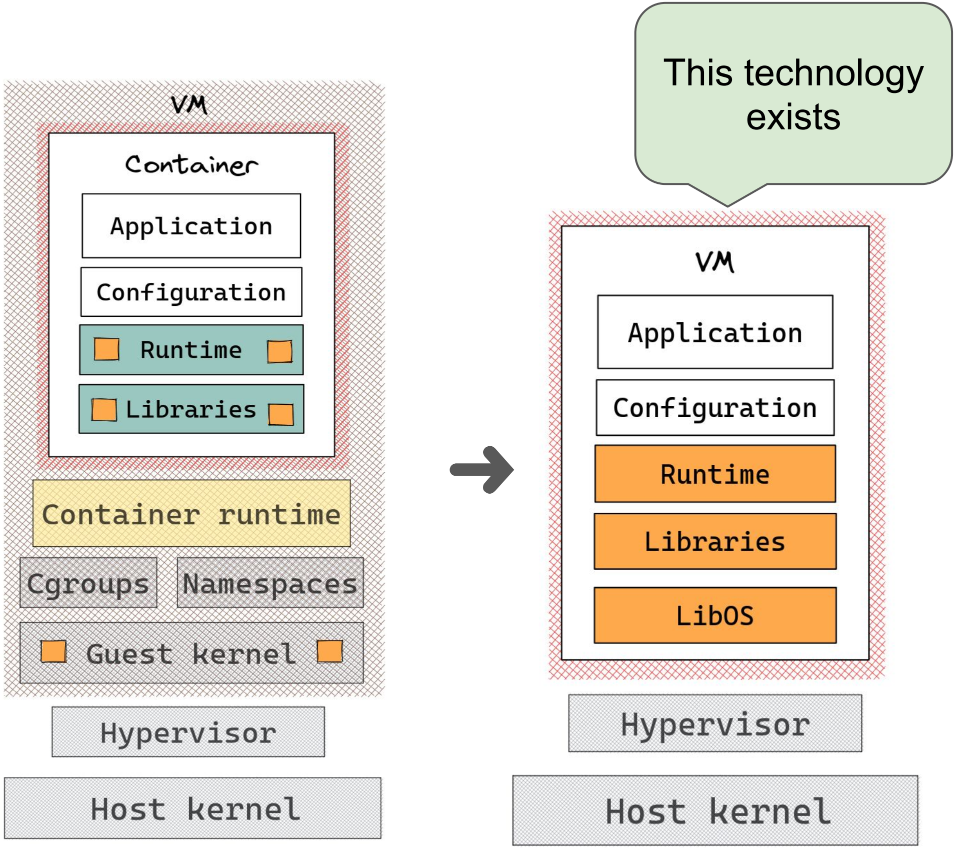
Back to (micro)VMs

- Combine containers and VMs
  - Keep the benefits of containers
  - Isolate containers inside Virtual Machines
- Side effects:
  - Higher overhead
  - Complex system stack



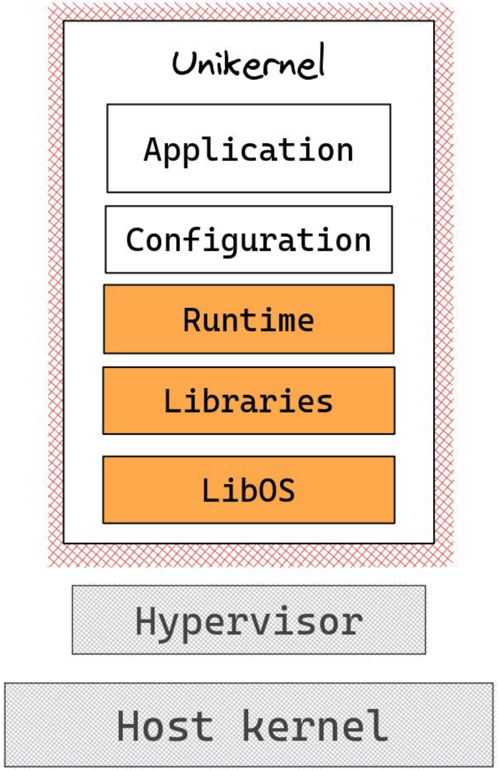
Back to (micro)VMs

- Combine containers and VMs
    - Keep the benefits of containers
    - Isolate containers inside Virtual Machines
- Side effects:
    - Higher overhead
    - Complex system stack



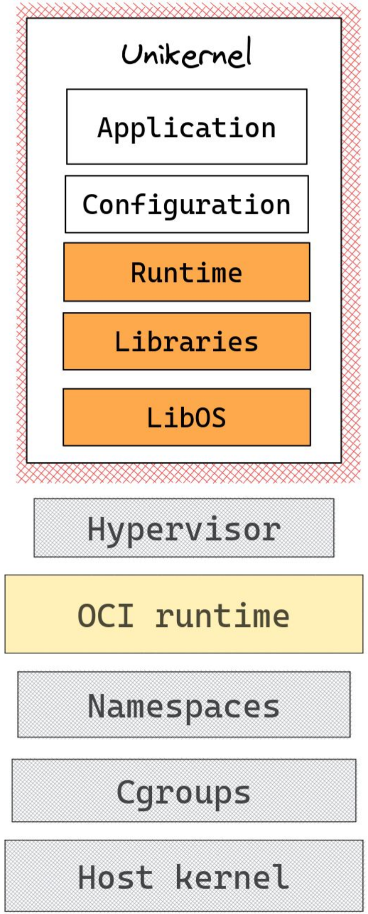
(Re)Introducing unikernels

- A unikernel is:
  - specialized
  - single address space
  - constructed using a LibOS
- Benefits:
  - Faster boot times
  - Reduced attack surface
  - Truly isolated
  - Smaller memory/disk footprint



Bringing unikernels to the cloud: What's missing?

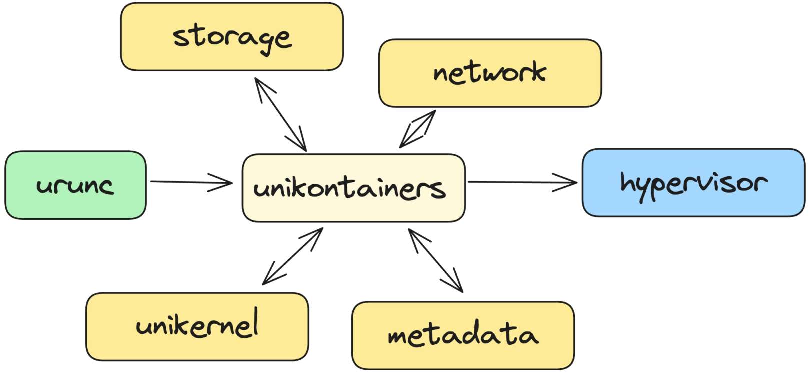
- Packaging: Unikernels should look like OCI images
  - OCI is a well defined and widely used format for container images
- **Deployment: Execution of Unikernels differs**
  - Container runtimes do not know how to execute Unikernels



urunc: the unikernel container runtime!

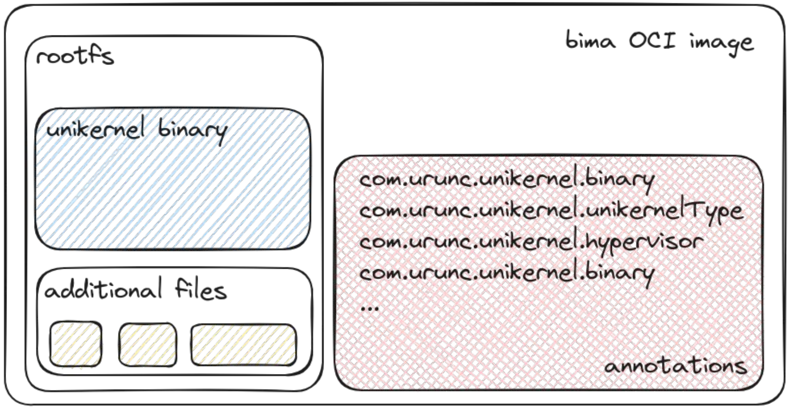
- **CRI-compatible** runtime written in Go
- Treats **unikernels** as **processes** -- directly manages applications
- Unikernel images for urunc are **OCI artifacts**
- Makes use of underlying hypervisors to spawn **unikernel VMs**

urunc



urunc: Unikernel OCI images

- Standard OCI images
- Can be managed and distributed using standard tooling (skopeo, umoci etc.) and registries (e.g. dockerhub)
- urunc makes use of specific annotations to function properly:
    - unikernel binary
    - unikernel type
    - hypervisor type
    - unikernel cmdline
    - initrd (optional)



urunc: Unikernel OCI images

To simplify image building, we built a **specialized image builder**, called **`bima`**.

`bima` uses a dockerfile-like syntax to create OCI images:

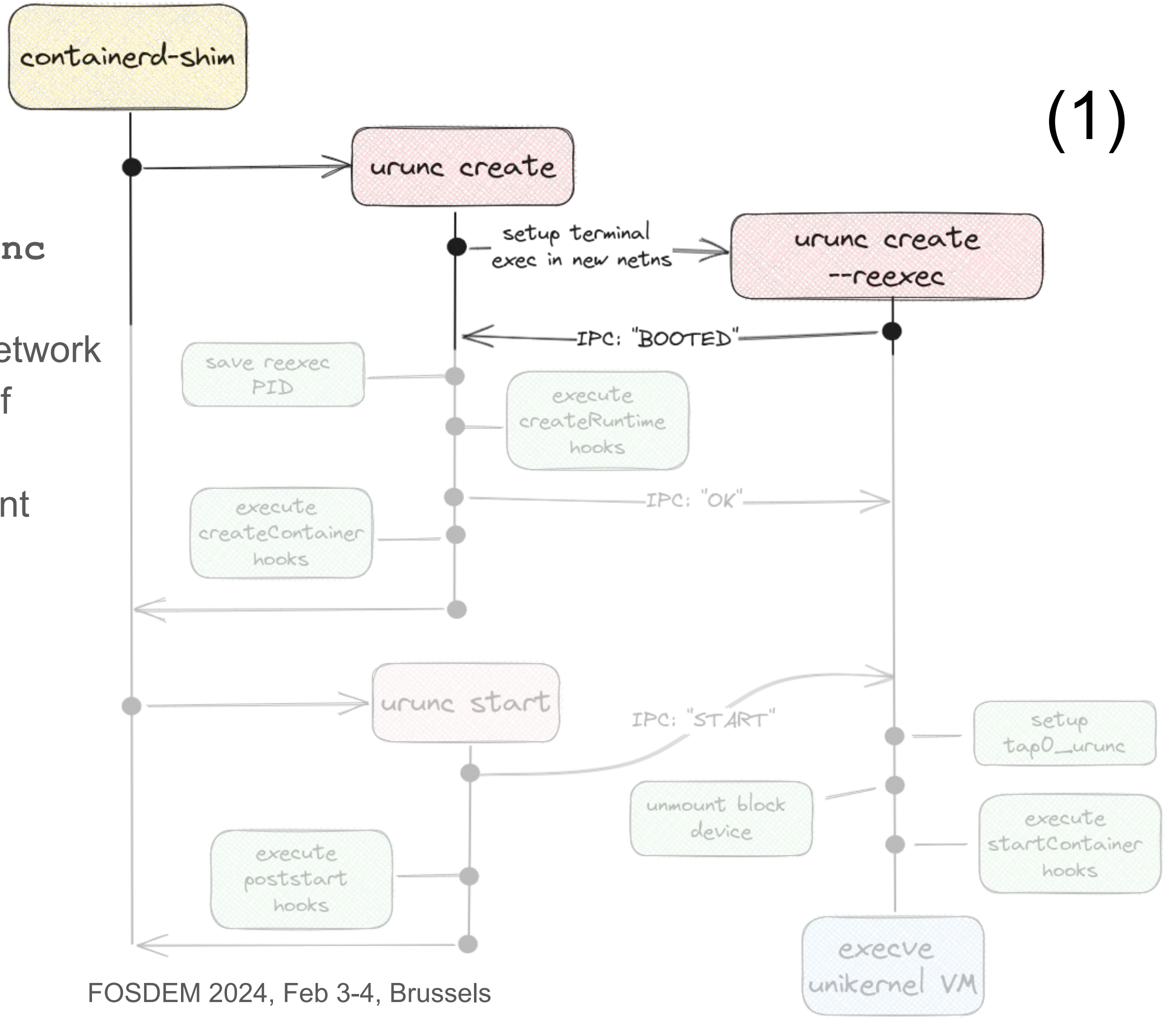
```
FROM scratch

COPY test-redis.hvt /unikernel/test-redis.hvt
COPY redis.conf /conf/redis.conf

LABEL com.urunc.unikernel.binary=/unikernel/test-redis.hvt
LABEL "com.urunc.unikernel.cmdline"='redis-server /data/conf/redis.conf'
LABEL "com.urunc.unikernel.unikernelType"="rumprun"
LABEL "com.urunc.unikernel.hypervisor"="qemu"
```

Sample **`bima`** invocation:

```
$ bima build -t image:tag .
```

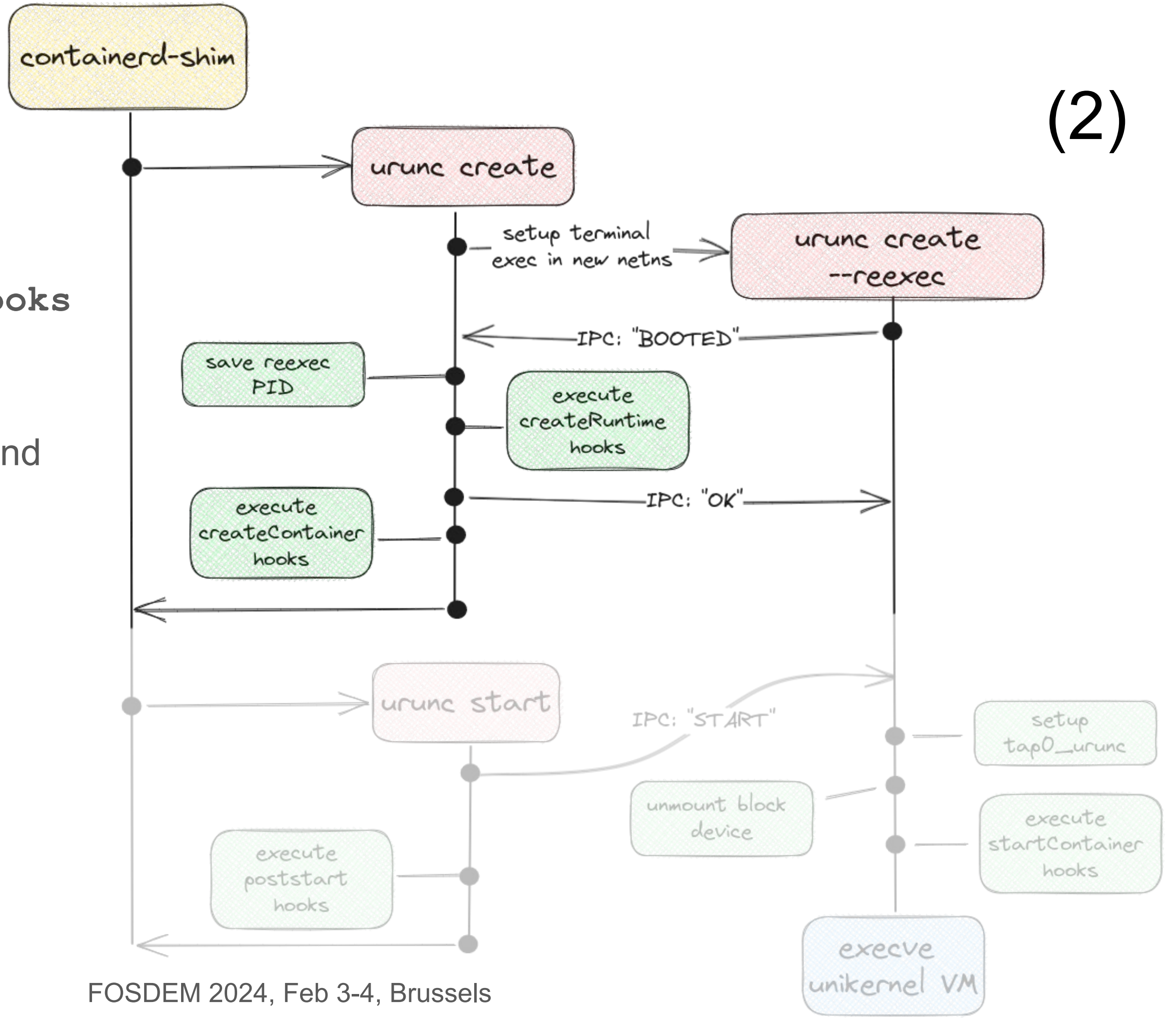
urunc: lifecycle (1)

- containerd-shim invokes **`urunc create`**
- **`urunc`** forks itself in a new network namespace, setting up a **pty** if required, spawning a **`reexec`** process, and notifies the parent process



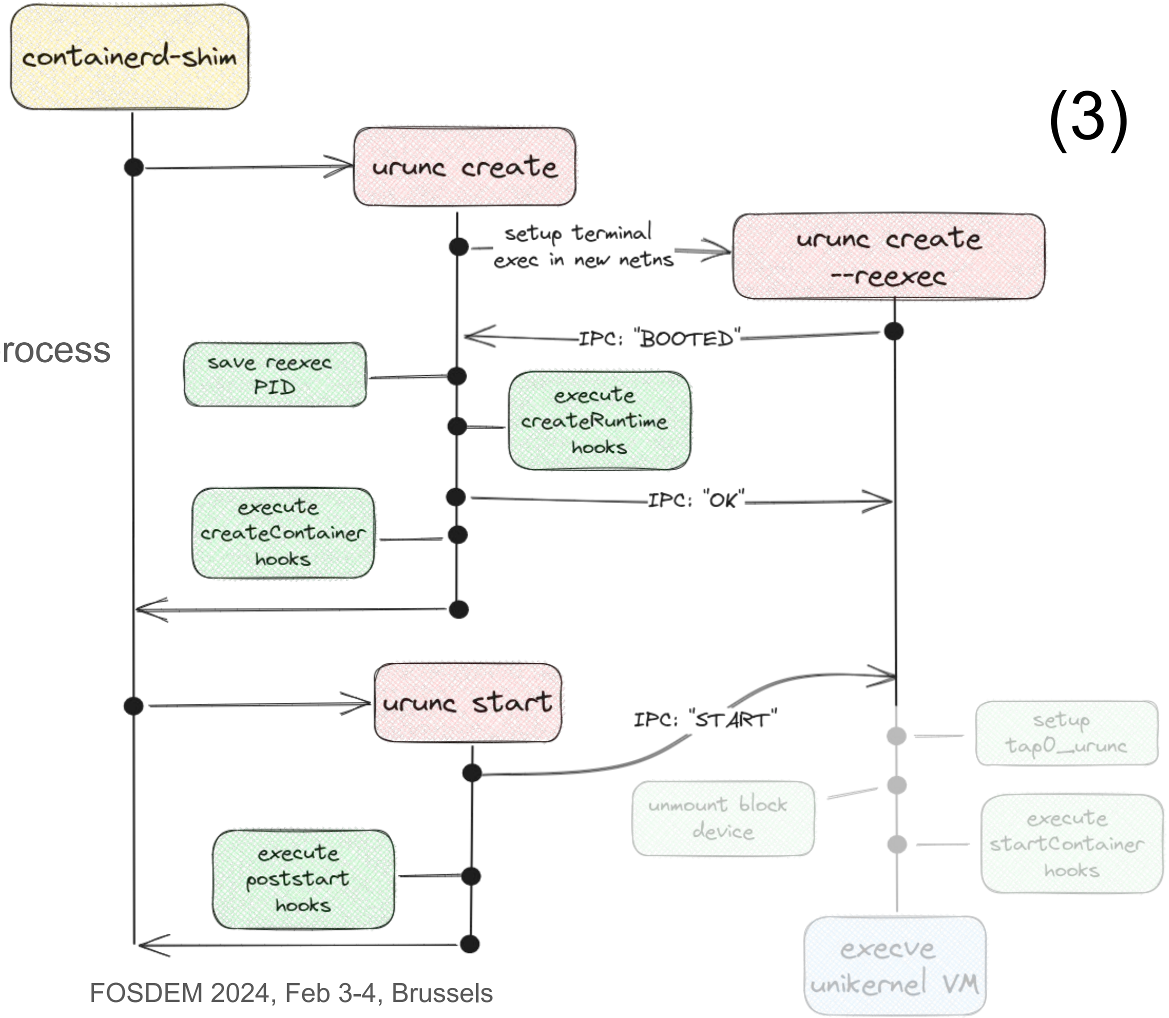
urunc: lifecycle (2)

- **`urunc`** saves the state and executes **`createRuntimeHooks`**
- **`urunc`** sends an ACK to the **`reexec`** process, executes **`createContainerHooks`** and exits gracefully.



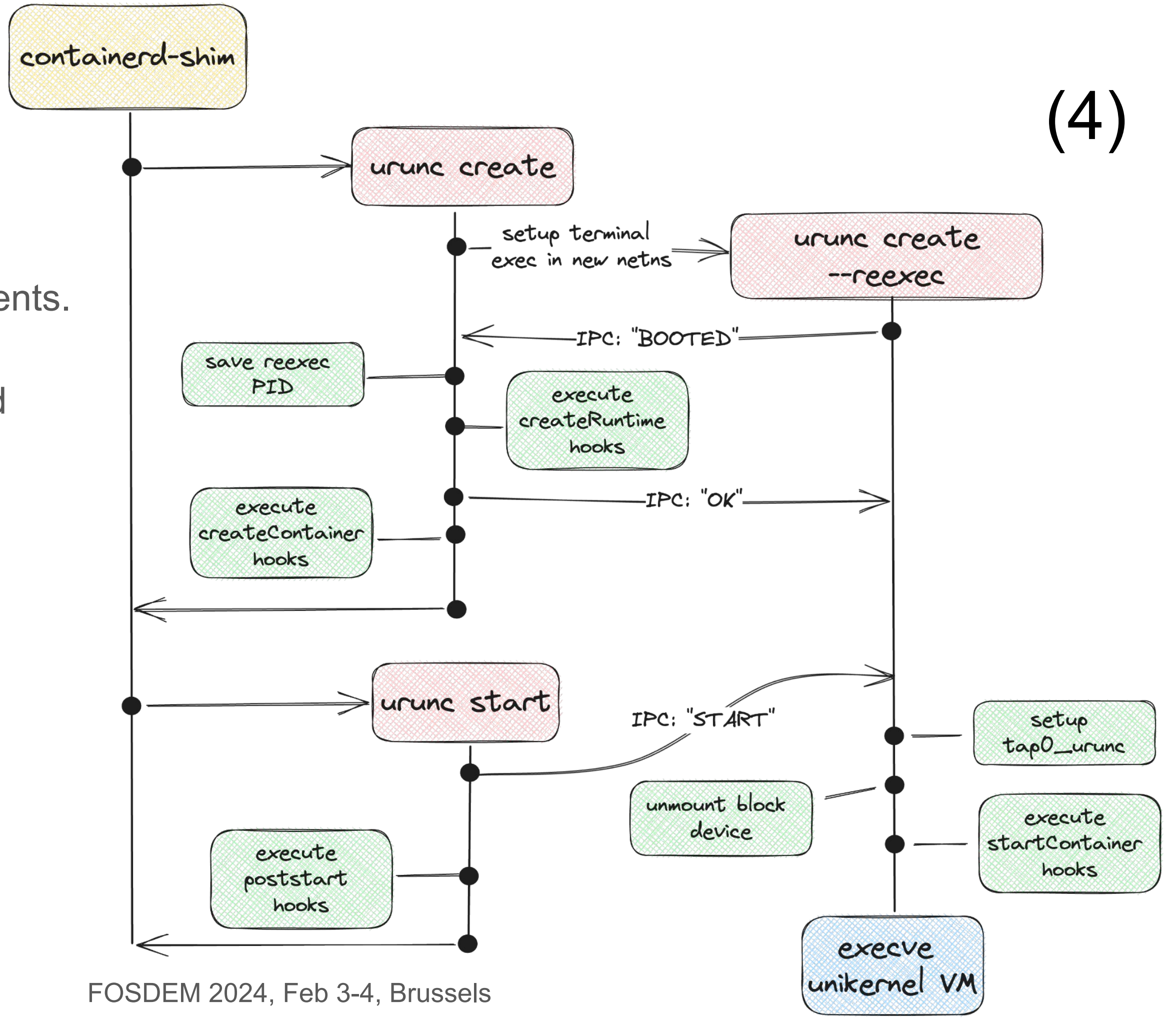
urunc: lifecycle (3)

- **containerd-shim** invokes **urunc start**
- **urunc** notifies the **reexec** process to start and executes **postStartHooks**



urunc: lifecycle (4)

- the **`reexec`** process sets up network and storage components.
- it executes **`startContainerHooks`** and spawns the unikernel.



urunc: Hypervisors

urunc features a extensible design, allowing easy integration for any underlying hypervisor, through the `hypervisors` package.

Currently, the following hypervisors are supported:

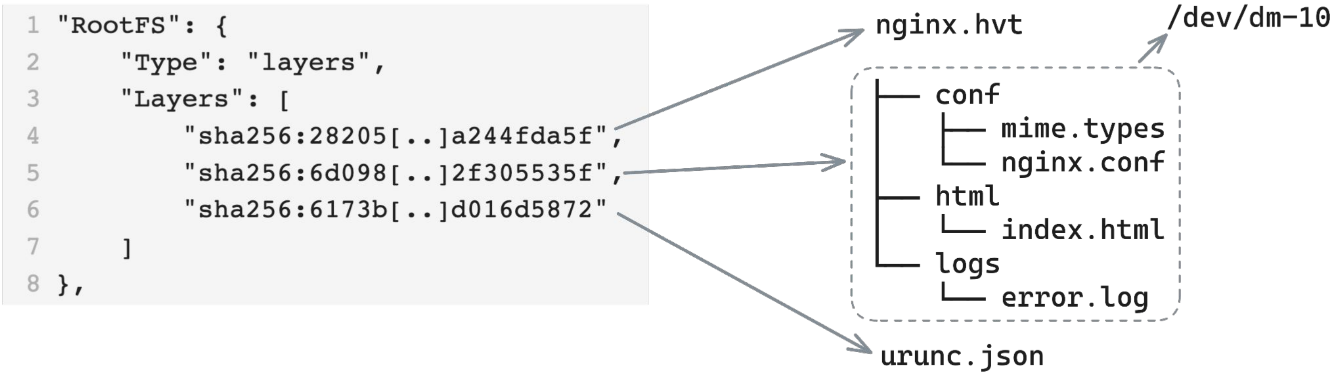
- solo5-hvt / solo5-spt
- QEMU
- firecracker

```
type VMM interface {
        Execve(args ExecArgs) error
        Stop(t string) error
        Path() string
        Ok() error
}
```

urunc: Storage

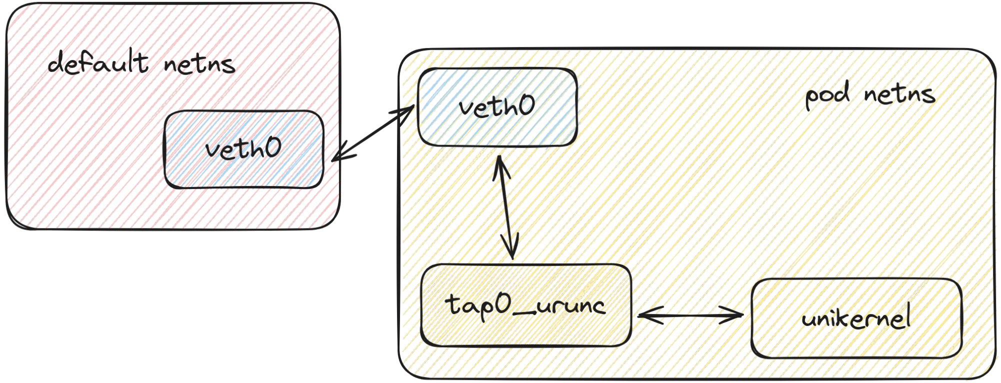
urunc provides storage to the unikernels via:

- Block device (devmapper snapshotter)
- Initrd (packed inside image rootfs)
- SharedFS



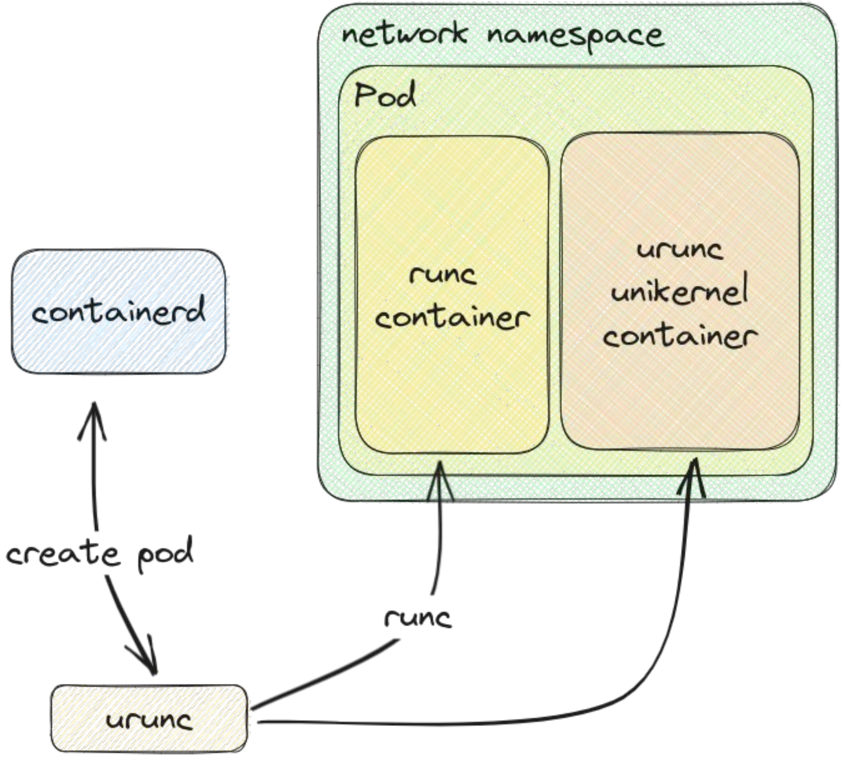
urunc: Network handling

- urunc creates a new tap device **`tap0_urunc`** inside the container netns
- CNI provides a `veth` endpoint inside the netns
- urunc maps all incoming traffic to the tap interface
- urunc maps all outgoing traffic to the veth endpoint



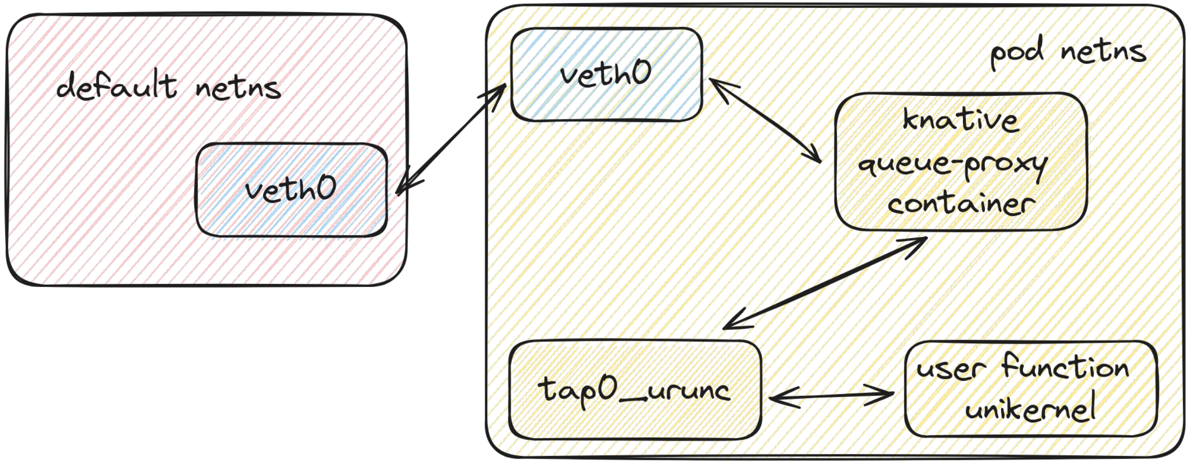
urunc: k8s integration

- to deploy **k8s pods**, we need to handle **non-unikernel** containers (eg pause, sidecar containers)
- **`urunc`** leverages **`runc`** to spawn generic containers
- **`urunc`** then spawns the unikernel container inside the Pod netns



urunc: intrapod unikernel - container communication

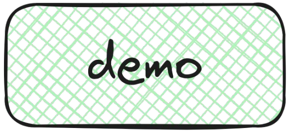
In some use cases, a normal container is required to communicate with the unikernel. To achieve this, we implement a static network configuration between the tap device and the unikernel.

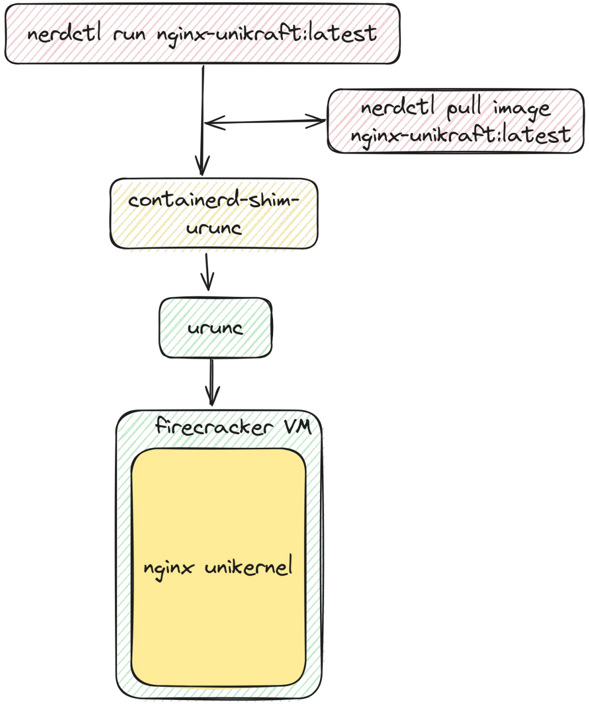


urunc in action: simple deployment

Simple nginx unikernel spawn

- nerdctl pulls image from registry
- nerdctl “calls” containerd
- containerd unpacks bundle and passes it to urunc
- urunc parses bundle and spawns firecracker VM with the provided unikernel

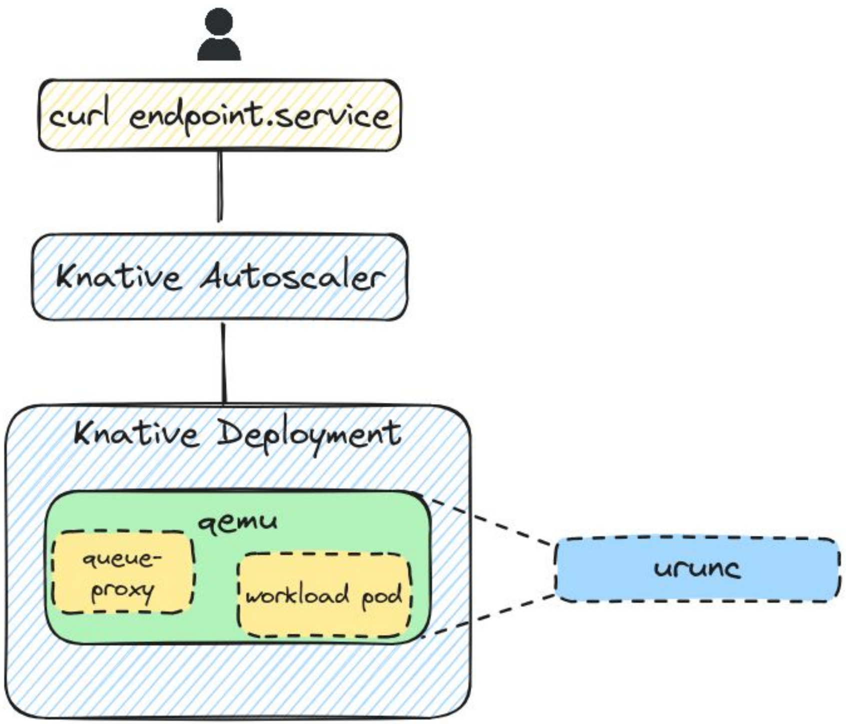




urunc in action: Knative function deployment

Simple Knative function deployment

- Define `urunc` runtime class
- Apply Kantive service `.yaml`
- curl endpoint
- Knative Service spawned
- urunc generates serverless workload

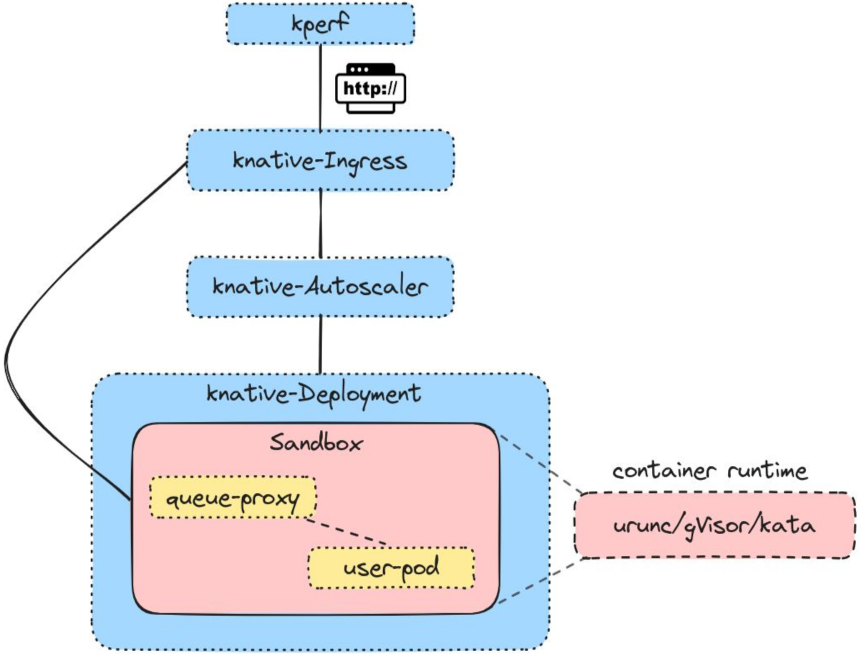


Evaluation: Serverless Workloads Spawning

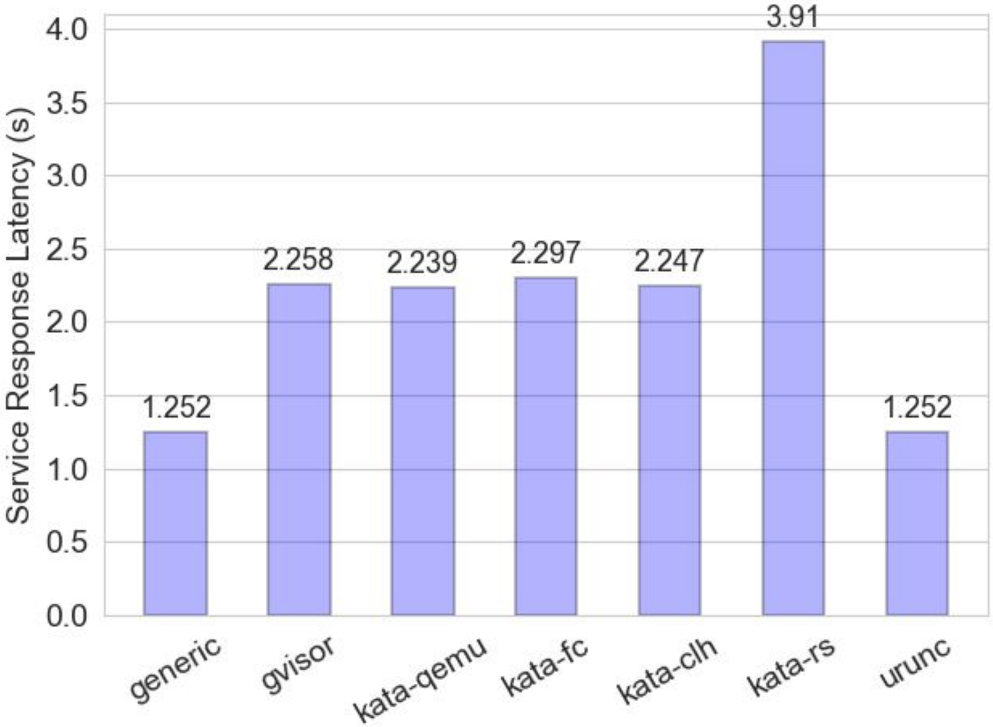
- Compared **urunc** with various container runtimes:
  - **runc**
  - **gVisor**(runsc)
  - **Kata-containers**{Firecracker, DragonBall, QEMU, Cloud Hypervisor}
- Utilized **Kperf** – “A benchmarking tool to evaluate Knative performance”
  - Generating and Triggering Knative Services
  - Reporting *Service Response Latency*
- Used *HTTP-reply* image as workload

Evaluation: Serverless Workloads Spawning

- Establish *Scale-from-Zero* Evaluation Scenario:
  - For *N iterations :*
    - Scale Knative Service (Workload Pod from 0 to 1)
    - Report avg Response Latency
      for every container runtime
      (~cold boot time)



Evaluation: Serverless Workloads Spawning



- (most) *sandbox* container runtimes require **2-2.5** seconds for servicing a request
- generic(**runc**) and **urunc** container runtime, request is being served in approximately **1.20** seconds
- early version of **urunc** is on par with generic container runtime(**runc**)

This work is partially funded through Horizon Europe actions, MLSysOps (GA: 101092912) and DESIRE6G (GA: 101096466)

Summary

- containers are great, but lack isolation
- unikernels as an alternative option
- urunc, the missing component for executing Unikernels, as easy as containers
- urunc and generic appear identical in terms of response latency
- unikernels can achieve the same or better performance than generic containers when it comes to serverless functions!

Check out the code on github:

- https://github.com/nubificus/urunc
- https://github.com/nubificus/bima

Check out the evaluation blog post:

- https://blog.cloudkernels.net/posts/knative-runtime-eval