KubeCon | CloudNativeCon

Europe 2024

Unikernels in K8s: Performance and Isolation for Serverless Computing with Knative

Anastassios Nanos & Ioannis Plakas

Research team: Charalampos Mainas, Georgios Ntoutsos

kubernetes

NUBIS

urunc

About us

NUBIS

- ➜ Young SME (inc. 2020) doing research in virtualization systems
- ➜ Involved in Research & Commercial projects
- ➜ Focus on systems software
  - ➜ Homogenize application deployment in heterogeneous infrastructure
  - ➜ Optimize application execution
  - ➜ Bring cloud-native concepts to Edge / Far-Edge devices

Charalampos (Babis) Mainas
Hypervisors / Unikernels
cmainas@nubis-pc.eu

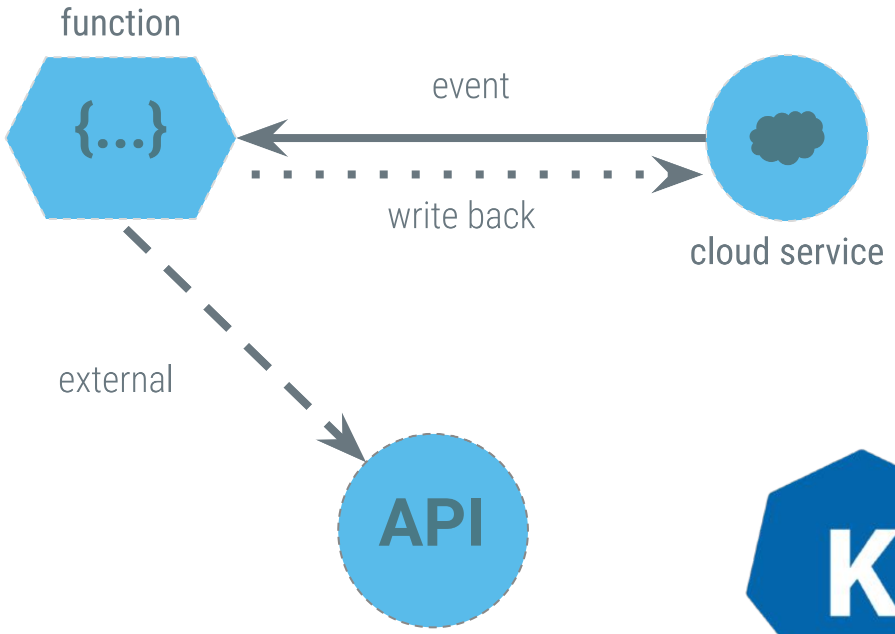
Anastassios (Tassos) Nanos
Hypervisors / Container runtimes
ananos@nubis-pc.eu

Georgios Ntoutsos
Container runtimes
gntouts@nubis-pc.eu

Ioannis Plakas
Orchestration
iplakas@nubis-pc.eu

FaaS & Serverless platforms

- Users:
  - write a function in a high-level language
  - pick the event to trigger the function
- The underlying framework handles:
  - instance selection, deployment, scaling, fault tolerance
  - monitoring, logging, security patches



FaaS platform requirements

Low end-to-end function execution latency:

➜ A function should complete with **minimal overhead** compared to its execution on a dedicated, bare-metal server

High throughput per CAPEX:

➜ To maximize throughput per capital expenditure, FaaS system software should serve a **high rate** of function execution requests **per server** to maximize utilization.

Energy efficiency:

➜ To minimize operational expenses — particularly energy consumption — the FaaS system should **minimize CPU cycles** for scheduling and executing functions.

Secure isolation:

➜ FaaS system software must **prevent untrusted** user function code from **tampering** with the **infrastructure** or accessing the **data** or **code** of other functions.

Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. **Function as a Function.** In Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23). Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3620678.3624648

Function as a Function

Tom Kuchler
ETH Zurich
Zurich, Switzerland
kuchlert@ethz.ch

Michael Giardino*
ETH Zurich
Zurich, Switzerland
mgiardino@ethz.ch

Timothy Roscoe
ETH Zurich
Zurich, Switzerland
troscoe@ethz.ch

Ana Klimovic
ETH Zurich
Zurich, Switzerland
aklimovic@ethz.ch

ABSTRACT

Function as a Service (FaaS) and the associated serverless computing paradigm alleviates users from resource management and allows cloud platforms to optimize system infrastructure under the hood. Despite significant advances, FaaS infrastructure still leaves much room to improve performance and resource efficiency. We argue that both higher performance and resource efficiency are possible — while maintaining secure isolation — if we are willing to revisit the FaaS programming model and system software design. We propose *Dandelion*, a clean-slate FaaS system that rethinks the programming model by treating serverless functions as pure functions, thereby explicitly separating computation and I/O. This new programming model enables a lightweight yet secure function execution system. It also makes functions more amenable to hardware acceleration and enables dataflow-aware function orchestration. Our initial prototype of Dandelion achieves 45× lower tail latency for cold starts compared to Firecracker. For 95% hot function invocations, Dandelion achieves 5× higher peak throughput.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Cloud computing**.

KEYWORDS

serverless, cloud computing, function as a service

ACM Reference Format:
Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3620678.3624648

*Currently with Computing Systems Lab, Huawei Technologies, Zurich.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624648

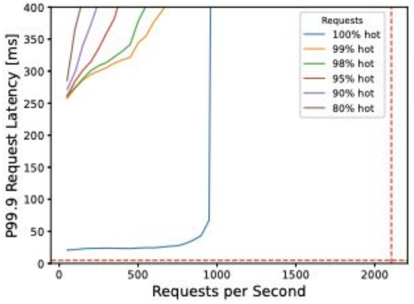


Figure 1: Round-trip tail latency for remote function execution with Firecracker, varying % hot requests. Red dotted lines show local bare-metal function execution latency (horizontal) and peak throughput (vertical).

1 INTRODUCTION

Serverless computing has the potential to become the dominant paradigm of cloud computing [58, 15], making cloud facilities easier to use and enabling cloud providers to more transparently optimize performance and energy efficiency of their infrastructure. With serverless, users develop applications as compositions of fine-grained functions, which execute independently while having access to shared remote storage. Users invoke functions on-demand and the cloud platform dynamically allocates the necessary hardware resources to execute them with an appealing pay-for-what-you-use cost model.

While this model holds promise, the system software infrastructure it uses is still rooted in the very different, more traditional execution model of long-running processes or virtual machines. Cloud providers typically provide function isolation by running them inside separate 'lightweight' VMs, which still incur significant startup times [62], context switch overheads [66], and memory duplication [56]. This practice of bundling each function with its own OS leads to a very general API, and the need to support this makes it hard for cloud providers to efficiently use their resources to run functions with low latency.

FaaS platform requirements

Low end-to-end function execution latency:

➜ A function should complete with **minimal overhead** compared to its execution on a dedicated, bare-metal server

High throughput per CAPEX:

➜ To maximize throughput per capital expenditure, FaaS system software should serve a high rate of function execution requests per server to maximize utilization.

Energy efficiency:

➜ To minimize operational expenses — particularly energy consumption — the FaaS system should minimize CPU cycles for scheduling and executing functions.

Secure isolation:

➜ FaaS system software must **prevent untrusted** user function code from **tampering** with the **infrastructure** or accessing the **data** or **code** of other functions.

Function as a Function

Tom Kuchler, ETH Zurich, Zurich, Switzerland, kuchlert@ethz.ch
Michael Giardino*, ETH Zurich, Zurich, Switzerland, mgiardino@ethz.ch
Timothy Roscoe, ETH Zurich, Zurich, Switzerland, troscoe@ethz.ch
Ana Klimovic, ETH Zurich, Zurich, Switzerland, aklimovic@ethz.ch

ABSTRACT

Function as a Service (FaaS) and the associated serverless computing paradigm alleviates users from resource management and allows cloud platforms to optimize system infrastructure under the hood. Despite significant advances, FaaS infrastructure still leaves much room to improve performance and resource efficiency. We argue that both higher performance and resource efficiency are possible — while maintaining secure isolation — if we are willing to revisit the FaaS programming model and system software design. We propose *Dandelion*, a clean-slate FaaS system that rethinks the programming model by treating serverless functions as pure functions, thereby explicitly separating computation and I/O. This new programming model enables a lightweight yet secure function execution system. It also makes functions more amenable to hardware acceleration and enables dataflow-aware function orchestration. Our initial prototype of Dandelion achieves 45× lower tail latency for cold starts compared to Firecracker. For 95% hot function invocations, Dandelion achieves 5× higher peak throughput.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Cloud computing**.

KEYWORDS

serverless, cloud computing, function as a service

ACM Reference Format:
Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. Function as a Function. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3620678.3624648

*Currently with Computing Systems Lab, Huawei Technologies, Zurich.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624648

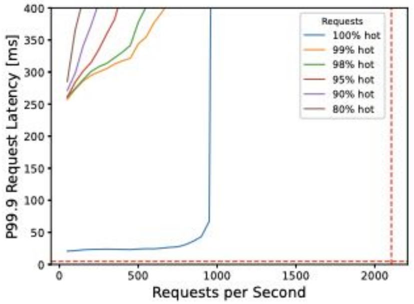


Figure 1: Round-trip tail latency for remote function execution with Firecracker, varying % hot requests. Red dotted lines show local bare-metal function execution latency (horizontal) and peak throughput (vertical).

1 INTRODUCTION

Serverless computing has the potential to become the dominant paradigm of cloud computing [58, 15], making cloud facilities easier to use and enabling cloud providers to more transparently optimize performance and energy efficiency of their infrastructure. With serverless, users develop applications as compositions of fine-grained functions, which execute independently while having access to shared remote storage. Users invoke functions on-demand and the cloud platform dynamically allocates the necessary hardware resources to execute them with an appealing pay-for-what-you-use cost model.

While this model holds promise, the system software infrastructure it uses is still rooted in the very different, more traditional execution model of long-running processes or virtual machines. Cloud providers typically provide function isolation by running them inside separate 'lightweight' VMs, which still incur significant startup times [62], context switch overheads [66], and memory duplication [56]. This practice of bundling each function with its own OS leads to a very general API, and the need to support this makes it hard for cloud providers to efficiently use their resources to run functions with low latency.

Tom Kuchler, Michael Giardino, Timothy Roscoe, and Ana Klimovic. 2023. **Function as a Function.** In Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC '23). Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3620678.3624648

FaaS systems software stack

Concerns about the systems software stack:

- retrofits legacy infrastructure
- presents high overhead when managing short-lived tasks

FaaS systems software stack

Concerns about the systems software stack:

- retrofits legacy infrastructure
- presents high overhead when managing short-lived tasks

FaaS systems software stack

Concerns about the systems software stack:

- retrofits legacy infrastructure
- presents high overhead when managing short-lived tasks

➜ k8s is still the dominant orchestration framework

➜ knative is a k8s-native serverless framework

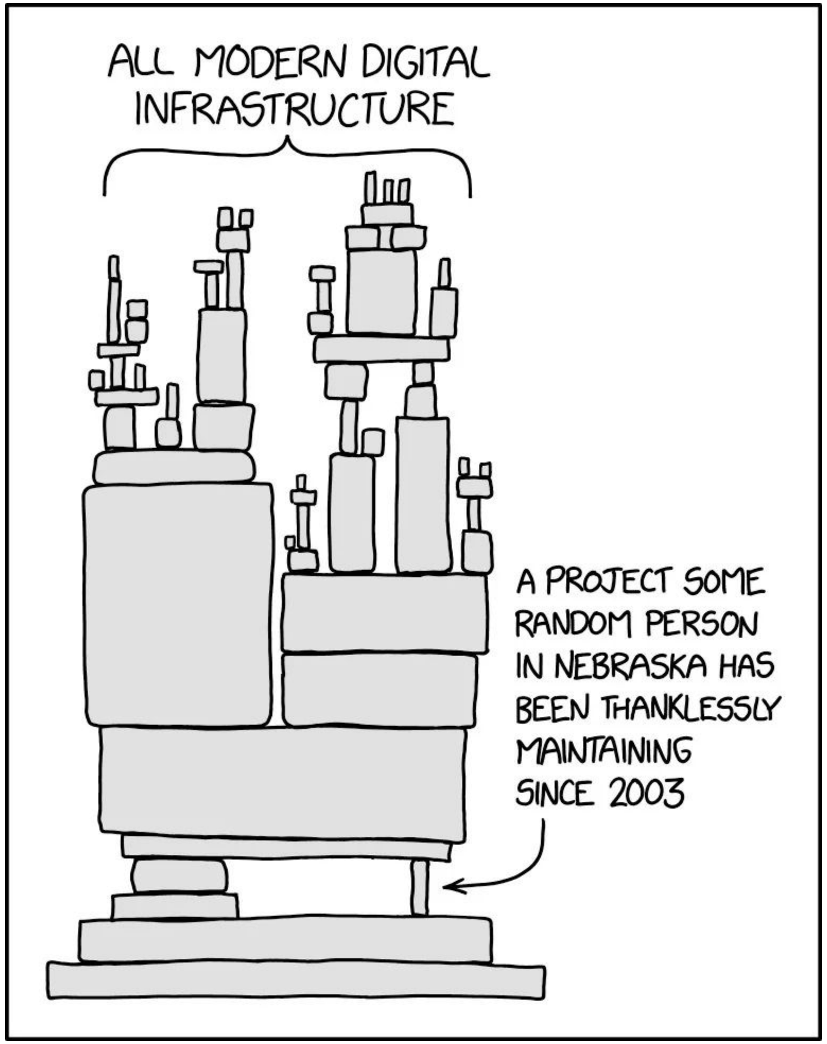
FaaS systems software stack

Concerns about the systems software stack:

- retrofits legacy infrastructure
- presents high overhead when managing short-lived tasks

➜ k8s is still the dominant orchestration framework

➜ knative is a k8s-native serverless framework



https://xkcd.com/2347/

FaaS systems software stack

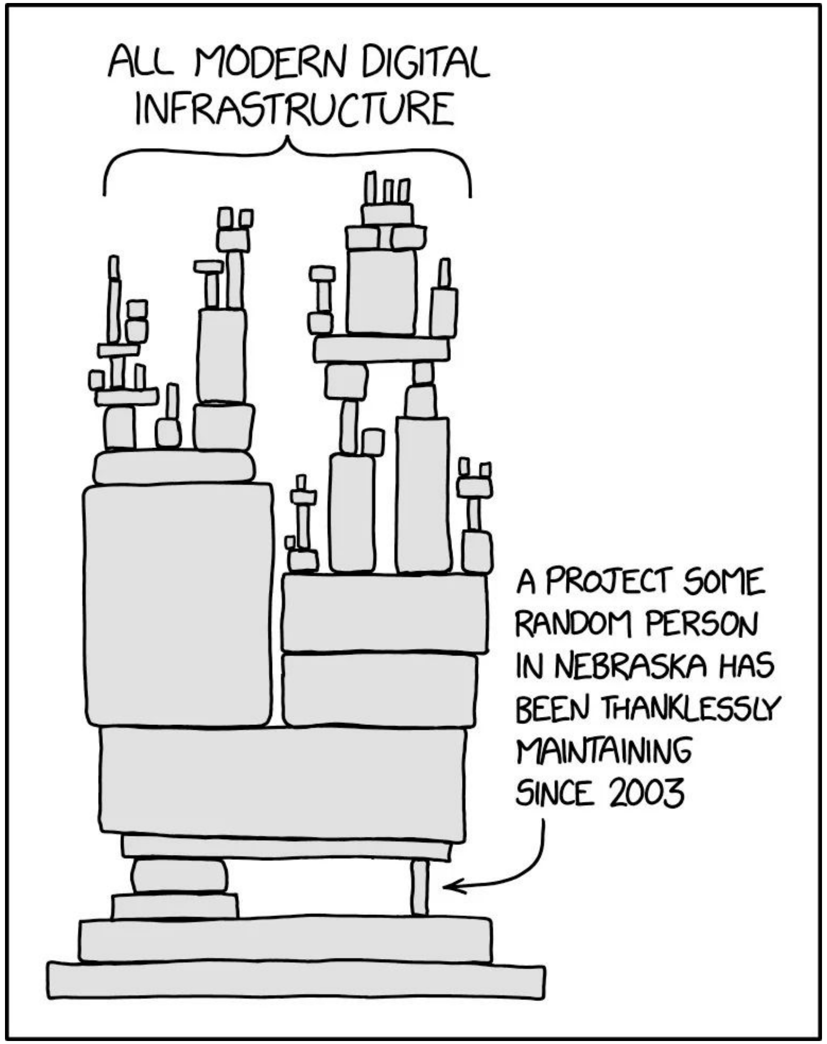
Concerns about the systems software stack:

- retrofits legacy infrastructure
- presents high overhead when managing short-lived tasks

➜ k8s is still the dominant orchestration framework

➜ knative is a k8s-native serverless framework

Let's try to optimize the parts of the stack we care about!

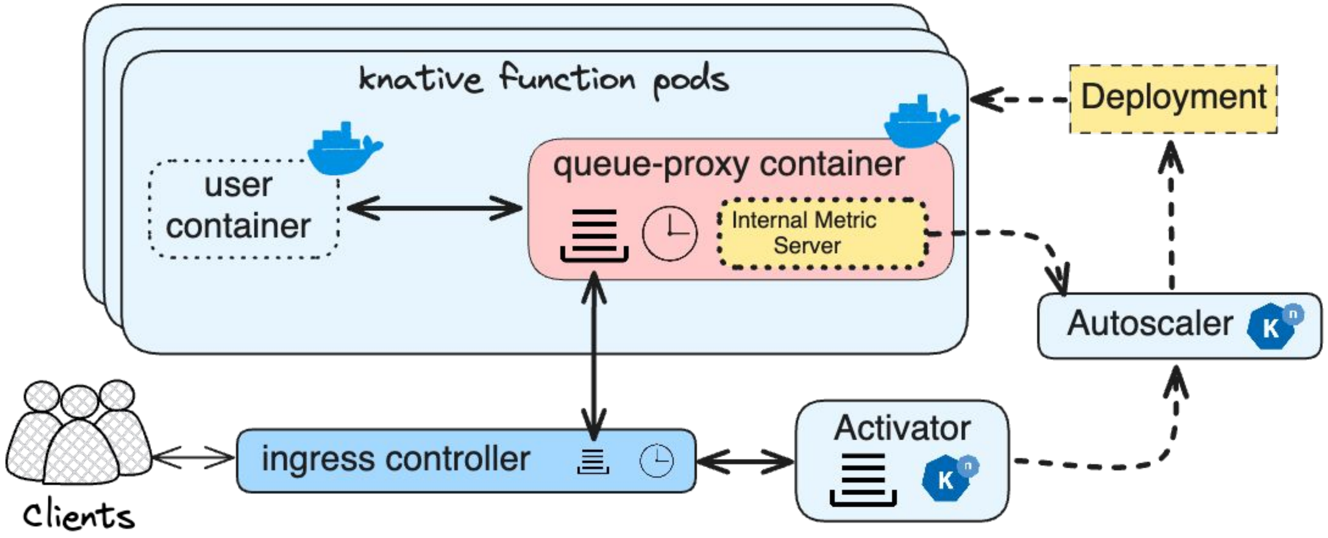


https://xkcd.com/2347/

Knative

Components:

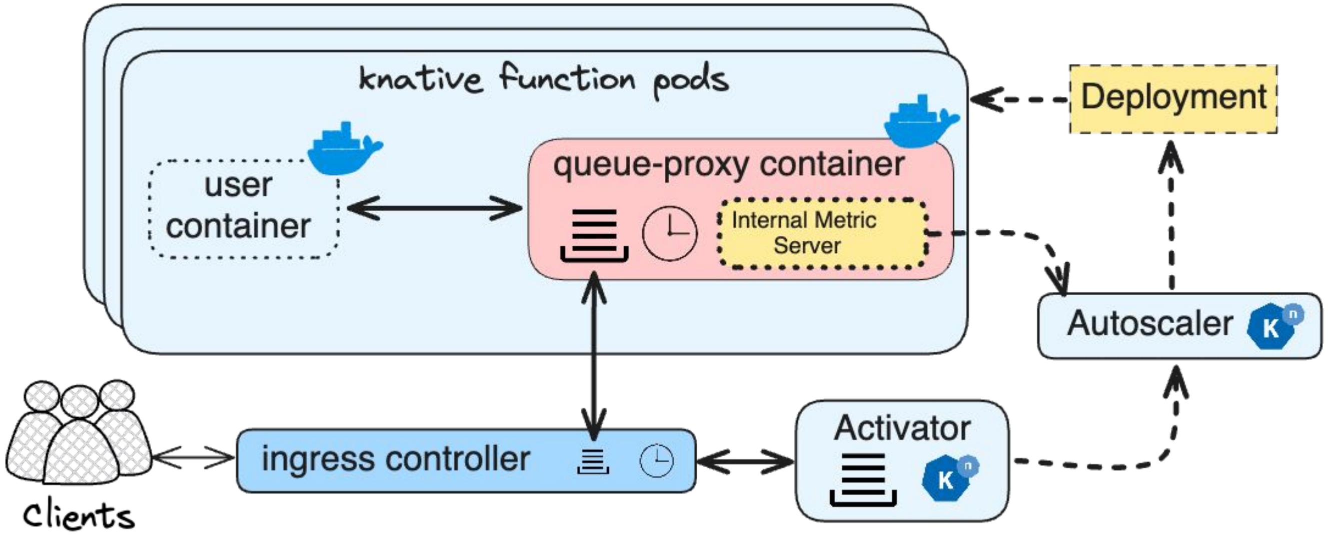
- Activator
- Autoscaler
- Function Pods:
  - queue-proxy
  - user-container



Knative

Components:

- Activator
- Autoscaler
- Function Pods:
  - queue-proxy
  - user-container



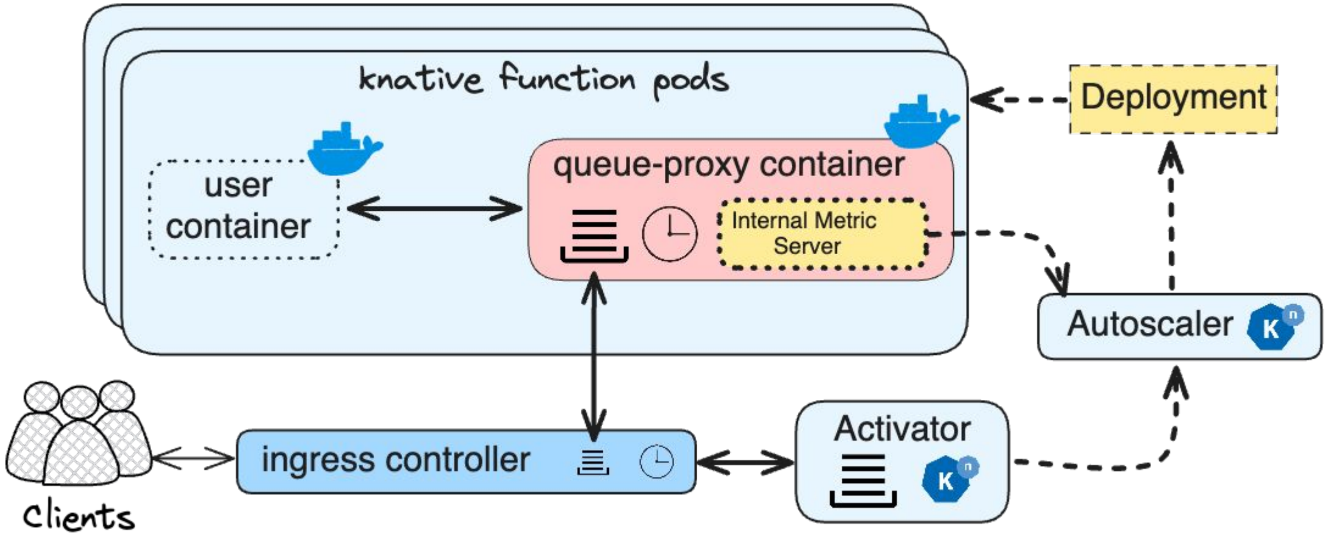
➜ Examine **isolation** issues

➜ sandbox user code

Knative

Components:

- Activator
- Autoscaler
- Function Pods:
  - queue-proxy
  - user-container



➜ Examine **isolation** issues

  ➜ sandbox user code

➜ Examine **response latency** issues

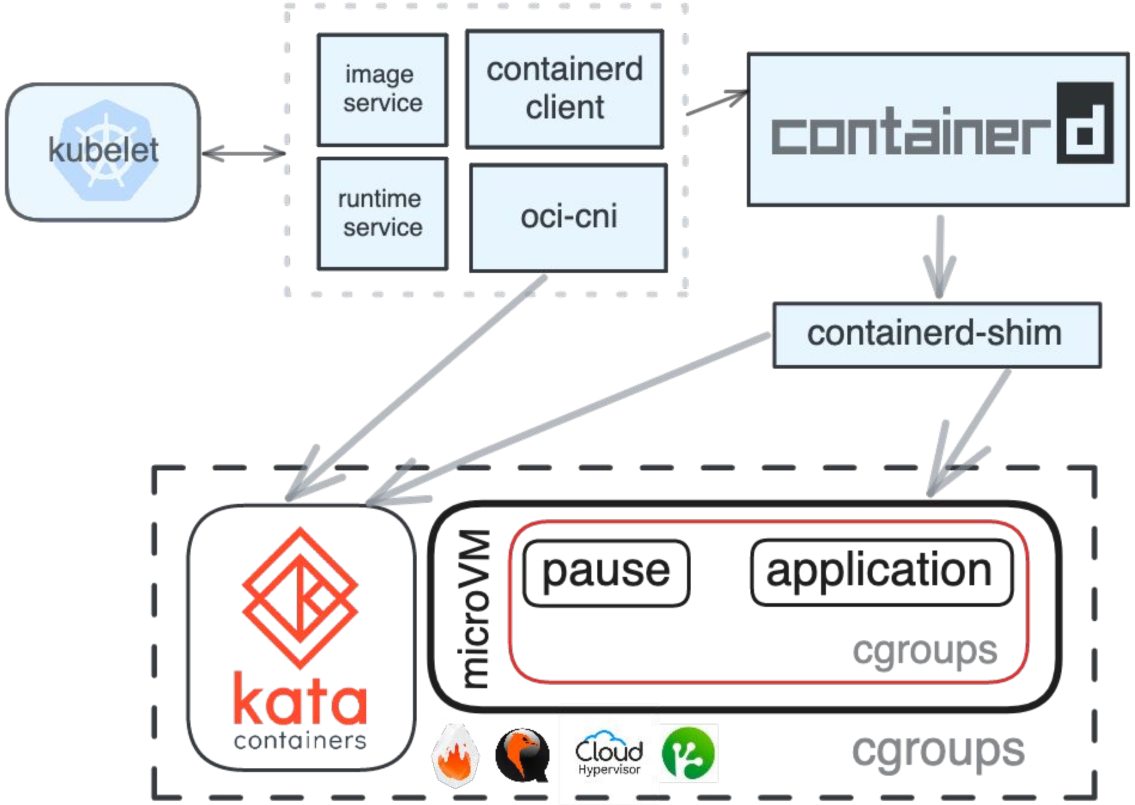
  ➜ cold boot times

Isolation: Sandboxed container runtimes

kata-containers:

- CRI-compatible
- spawn a sandbox / microVM
  - AWS Firecracker
  - QEMU
  - Cloud-hypervisor
  - Dragonball (runtime-rs)
- spawn all containers of a pod in the sandbox

Gvisor follows the same principle



Isolation: Knative with sandboxed containers

`RuntimeClass` option:

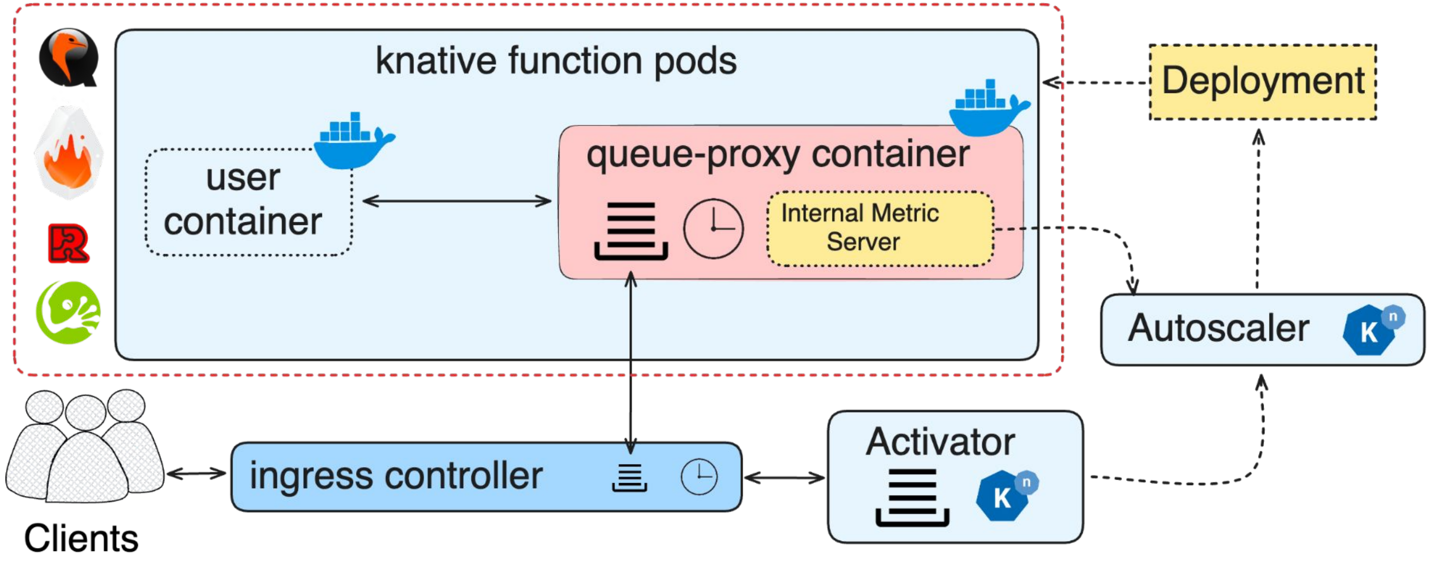
✓ protect the rest of the infrastructure from user-submitted code

✘ the **`queue-proxy`** container is still exposed to user-submitted code

✘ increased cold-boot overhead:

- ➜ spawn the microVM
- ➜ pass through container rootfs
- ➜ spawn the container

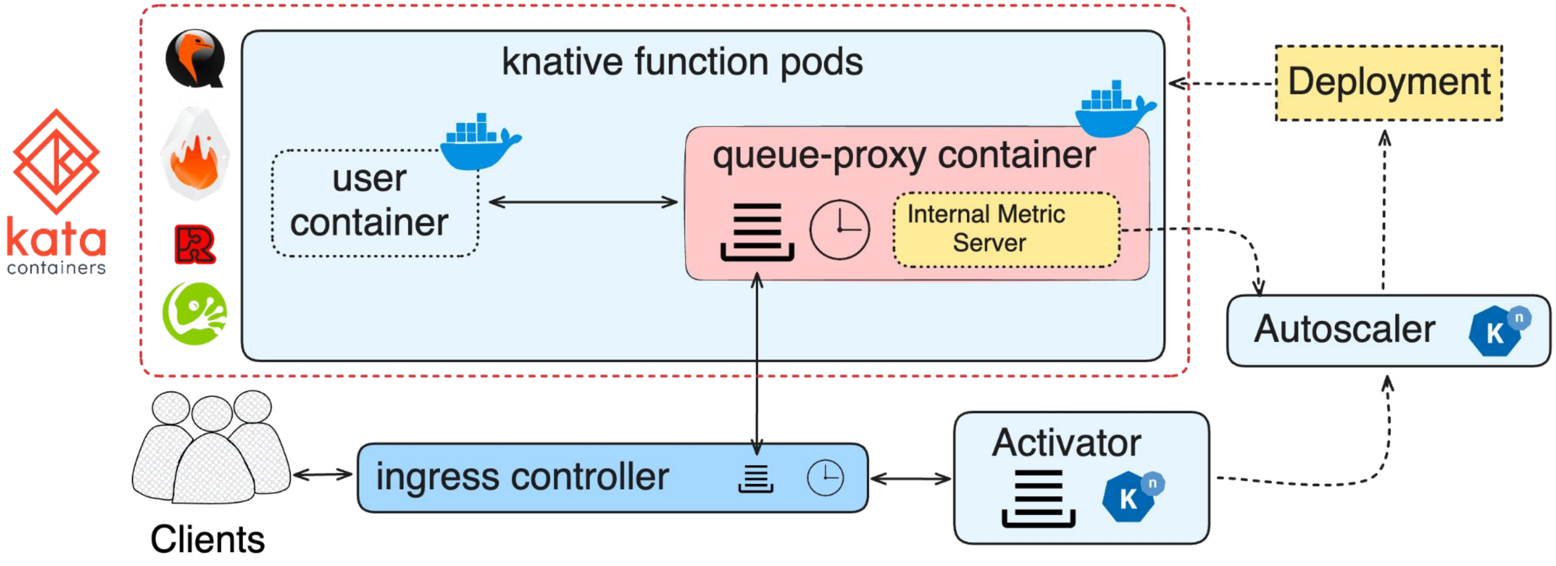
kata containers



Isolation: Knative with sandboxed containers

`RuntimeClass` option:

- ✓ protect the rest of the infrastructure from user-submitted code
- ✘ the `queue-proxy` container is still exposed to user-submitted code
- ✘ increased cold-boot overhead:
  - ➜ spawn the microVM
  - ➜ pass through container rootfs
  - ➜ spawn the container



What if we had a way to isolate the user-container from the rest of the stack

and

reduce cold-boot times...

Unikernels

A unikernel is:

- ➜ specialized
- ➜ single address space
- ➜ built using a LibOS

In other words:

- ➜ Tailored for a single application
- ➜ No kernel- / user-space separation (no mode switches)
- ➜ Contains the absolute minimum software components for the application to run

Unikernels

- Considered a research-y concept
    - "Unikernels are unfit for production"
- Lately, things are changing
- Many frameworks exist, tailored to specific use-cases

Unikernels

- ➜ Considered a research-y concept
  - ○ "Unikernels are unfit for production"
- ➜ Lately, things are changing
- ➜ Many frameworks exist, tailored to specific use-cases

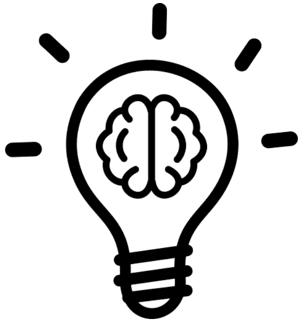
- ➜ Unikernels are *not* containers
  - ✘ can not use all the nifty container tools :(
- ➜ Unikernels are *not* typical VMs
  - ✘ can not integrate directly with sandboxed container runtimes

Cloud-native Unikernels

- ➜ OCI is a well defined and widely used format for container images
  - ✓ Unikernels should **look** like **OCI images**
- ➜ Container runtimes drive application execution in modern orchestration platforms
  - ✓ Container runtimes should know **how** to execute Unikernels

Cloud-native Unikernels

- ➜ OCI is a well defined and widely used format for container images
  - ✓ Unikernels should **look** like **OCI images**
- ➜ Container runtimes drive application execution in modern orchestration platforms
  - ✓ Container runtimes should know **how** to execute Unikernels

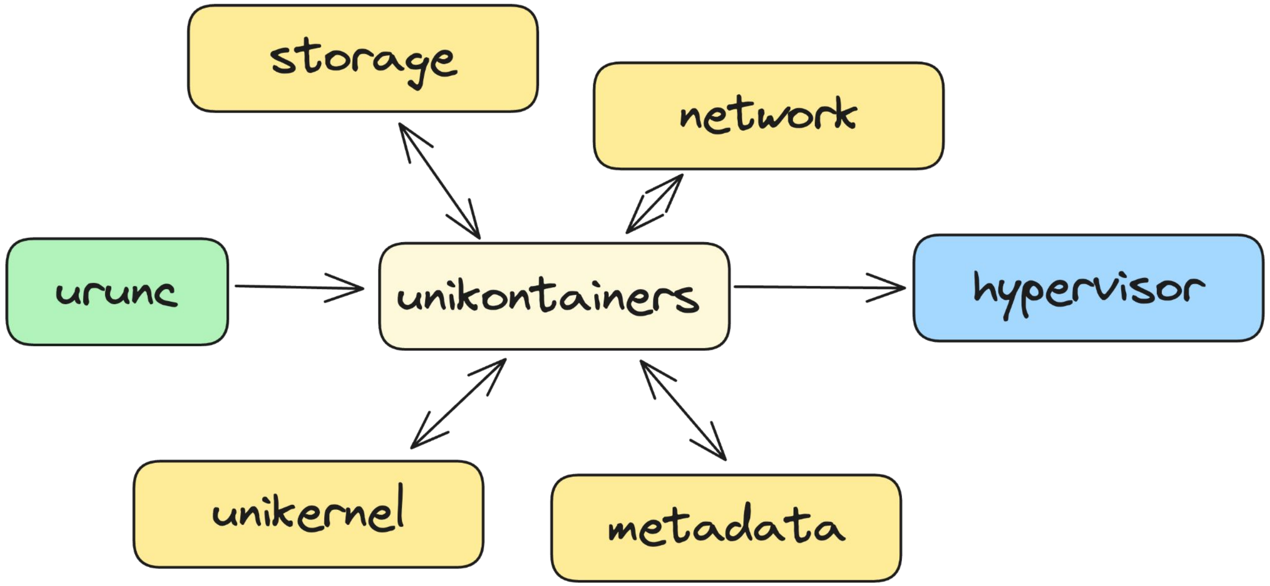


- ➜ Build a unikernel-compatible container runtime!

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**
- urunc makes use of **generic hypervisors** to spawn unikernel VMs
- Extensible, easy to add support for more unikernel frameworks & hypervisors

urunc



urunc: unikernel OCI packaging

- ➜ specialized image builder: `bima`
- ➜ Containerfile-like syntax:

```
FROM scratch

COPY httpreply_fc-aarch64 /unikernel/httpreply-unikraft

LABEL "com.urunc.uni.binary"="/unikernel/httpreply-unikraft"
LABEL "com.urunc.uni.cmdline"="-c /etc/httpreply/config.toml"
LABEL "com.urunc.uni.unikernelType"="unikraft"
LABEL "com.urunc.uni.hypervisor"="firecracker"
```

urunc

The image includes:

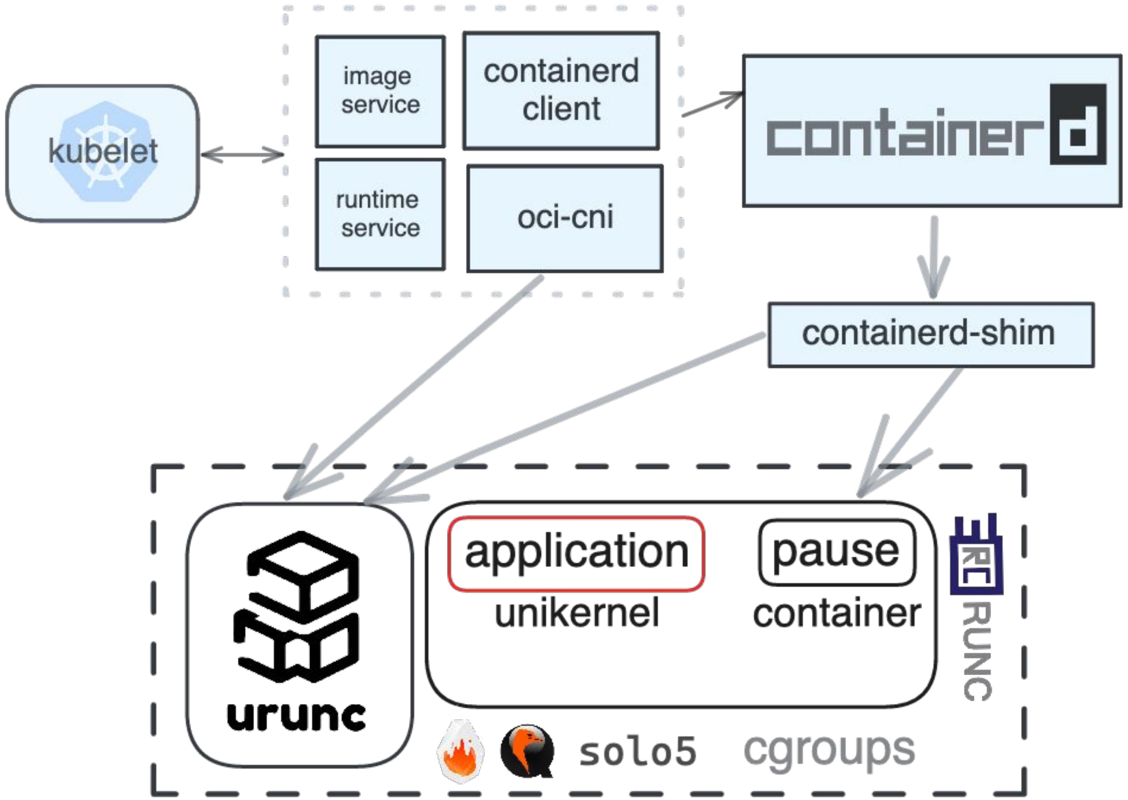
- ➜ the unikernel binary
- ➜ any extra files required (eg configuration, libraries)
- ➜ `urunc.json` containing urunc-specific metadata

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

- ➜ standard tooling (e.g. `skopeo`, `umoci`, `dive`) and container image registry support (e.g. `dockerhub`, `harbor` etc.).

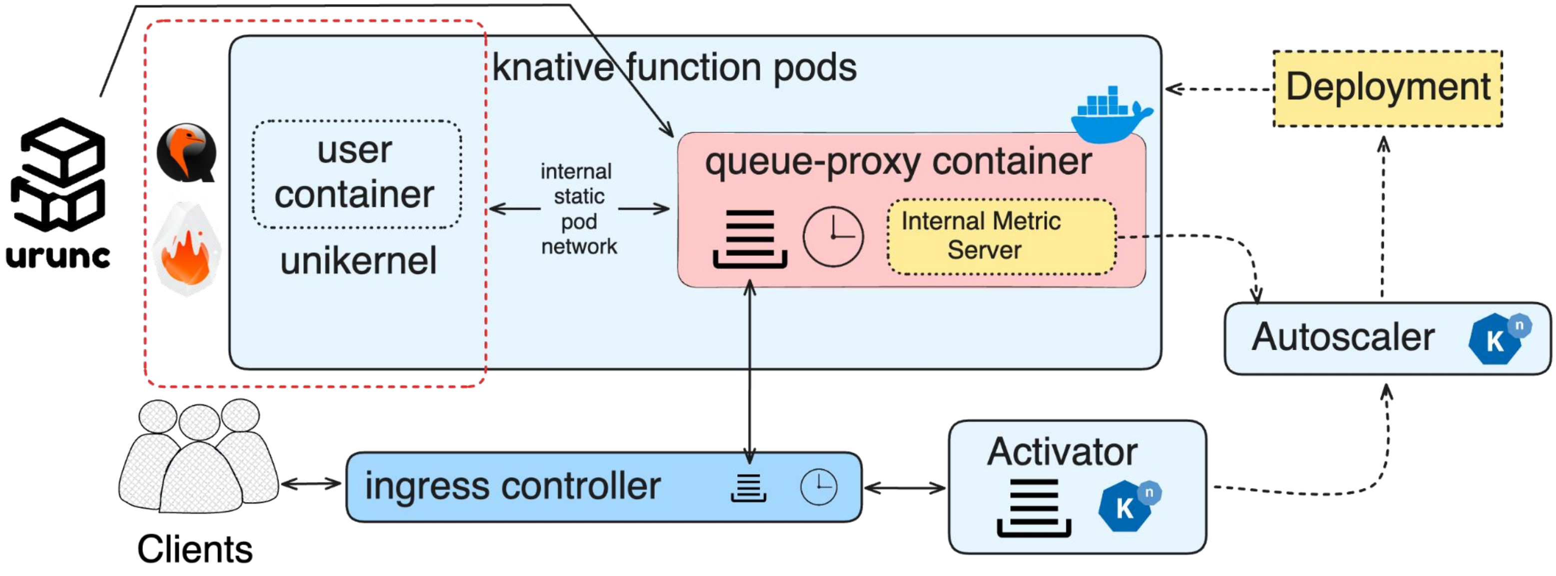
urunc: k8s integration

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



urunc: knative integration

- ➜ we build the user-code as a unikernel
- ➜ we package it using `bima` as an OCI artifact
- ➜ we create a Knative service using `urunc`'s **`RuntimeClass`**



The user code is spawned as a unikernel:

- ➜ hardware virtualization **isolation**
- ➜ **faster spawn times** than a sandboxed container

Benchmark Setup

Bare-metal server:

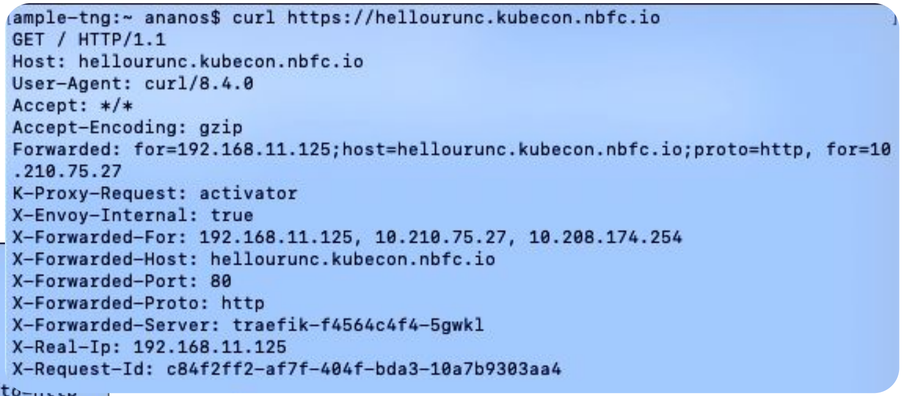
- ➜ AMD EPYC 7520P (Rome, 32 cores)
  - Turbo Boost disabled
  - CPU Frequency scaling disabled
- ➜ 128GB RAM

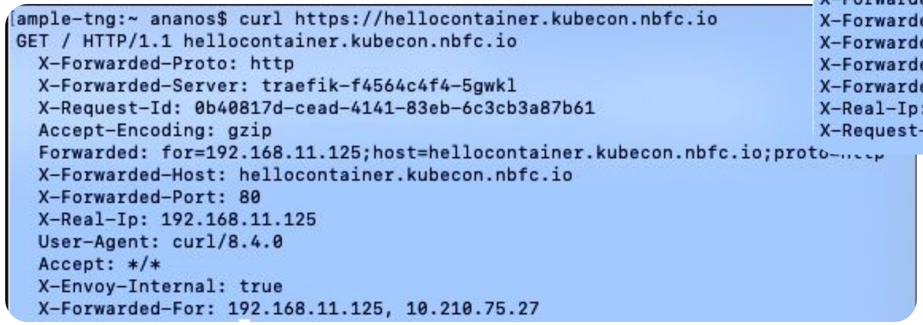
Software stack:

- ➜ Ubuntu 20.04
- ➜ K8s v1.28.2
- ➜ Knative v1.12
- ➜ kata-containers v3.2.0
- ➜ gvisor 20231113.0
- ➜ urunc v0.2

Experiment setup:

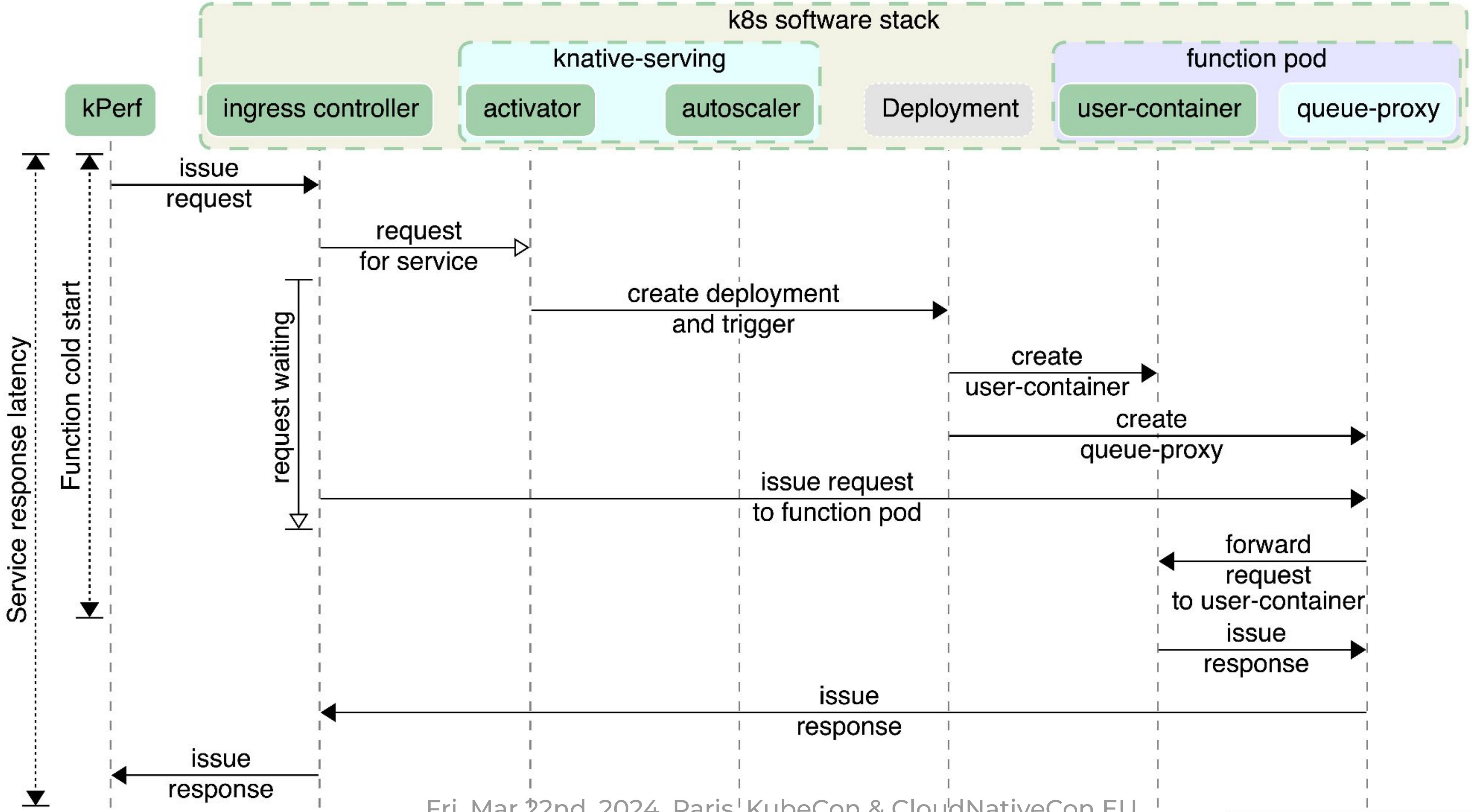
- ➜ Kperf
- ➜ Service: simple HTTP reply function
  - Go for generic/sandboxed containers
  - C for unikernels





Knative Request workflow

KubeCon CloudNativeCon Europe 2024



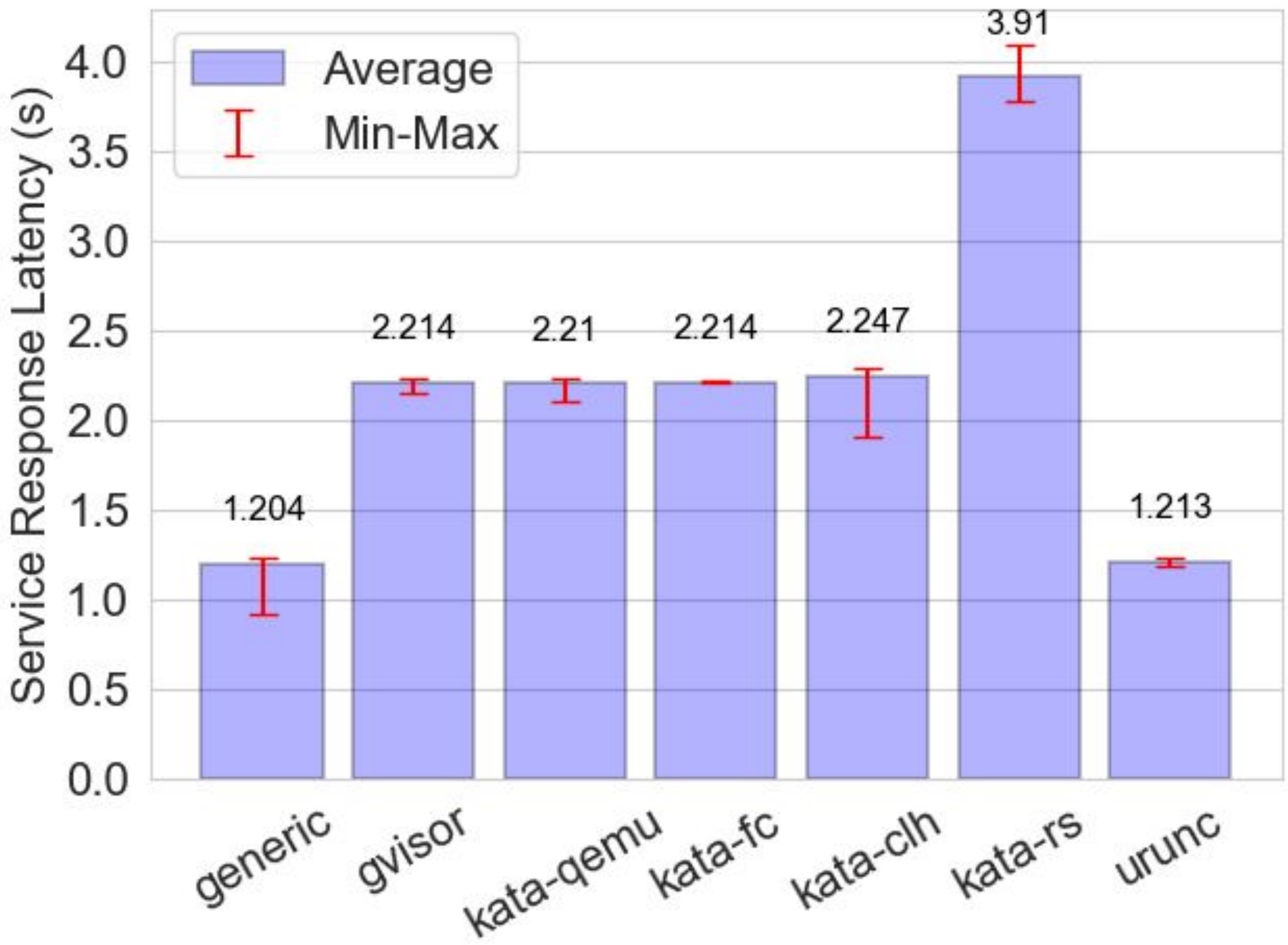
NUBIS

Service Response Latency (single instance)

KubeCon CloudNativeCon Europe 2024

Kperf using a single request trigger:

- ➜ measure "cold-boot" latency
- ➜ kata & gvisor 2x generic & urunc
- ➜ generic & urunc almost identical



Service Response Latency (single instance, 99th)

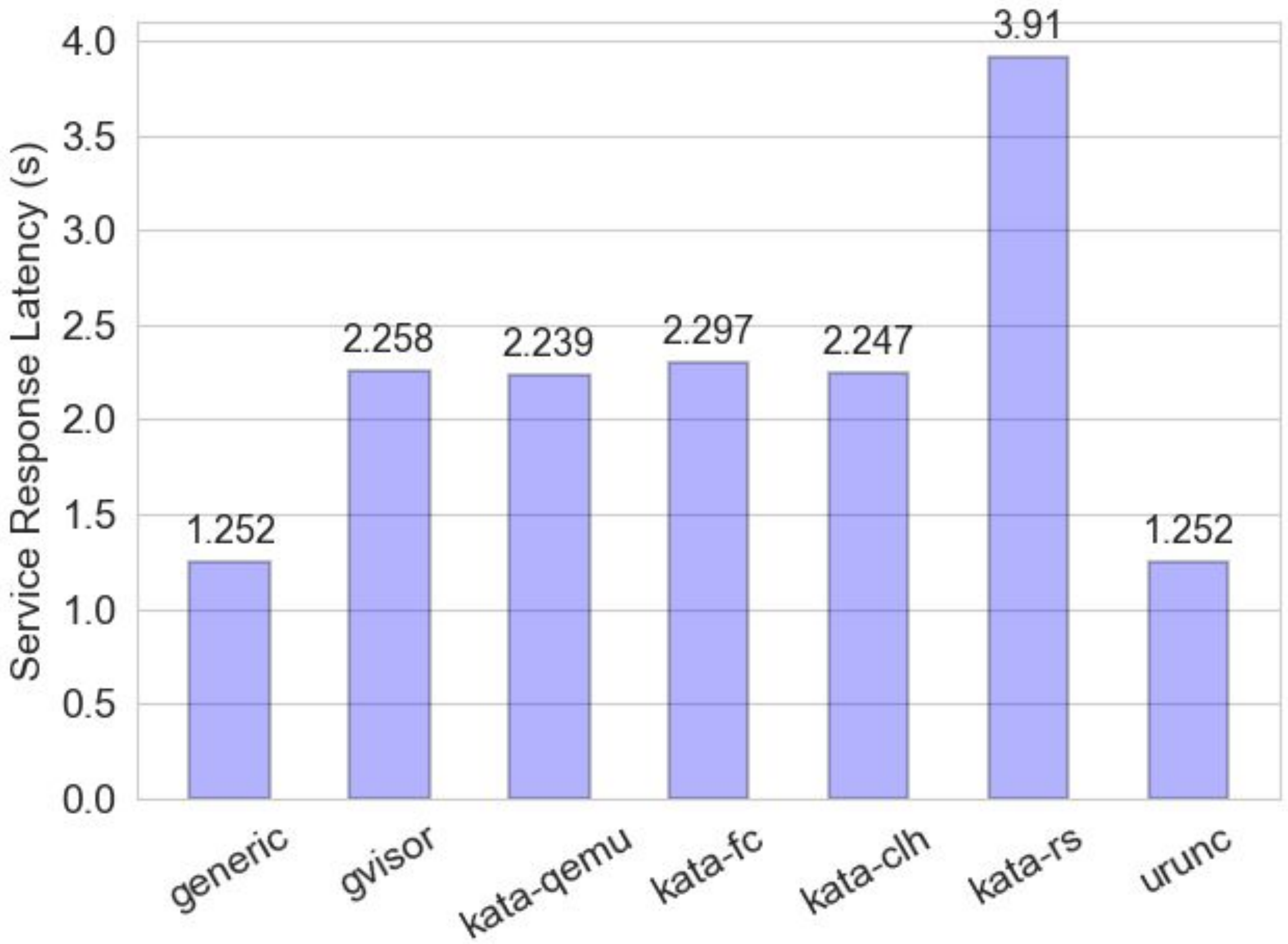
Kperf using a single request trigger:

➜ 99th percentile (slowest response)

identical behaviour:

➜ kata & gvisor 2x generic & urunc

➜ generic & urunc almost identical



Service Response Latency (concurrent)

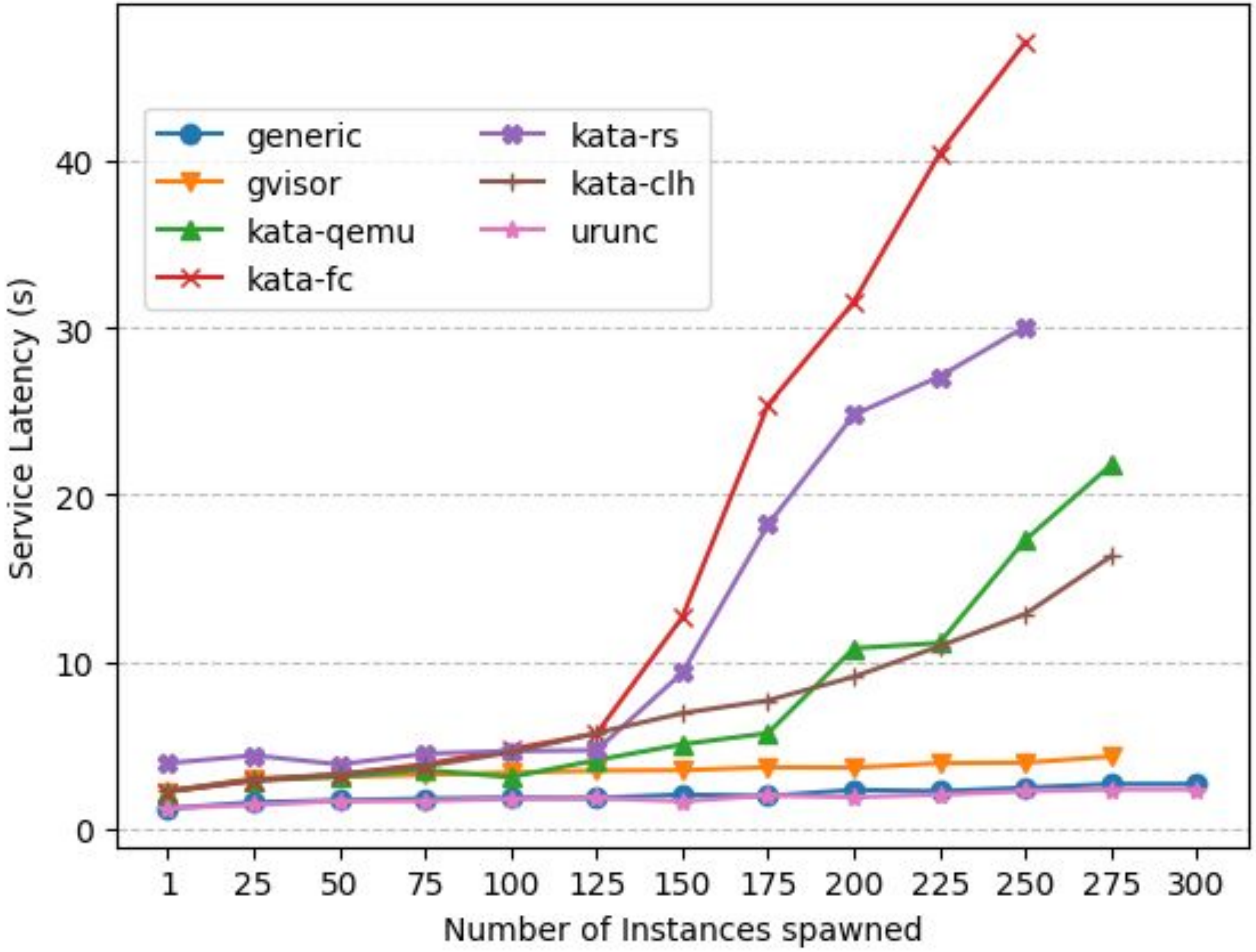
tweaked Kperf to map each request to a distinct function:

➜ measure concurrent cold-boot spawns & sustainable response times

similar behaviour

➜ kata & gvisor 2x generic & urunc (up to 125 instances)

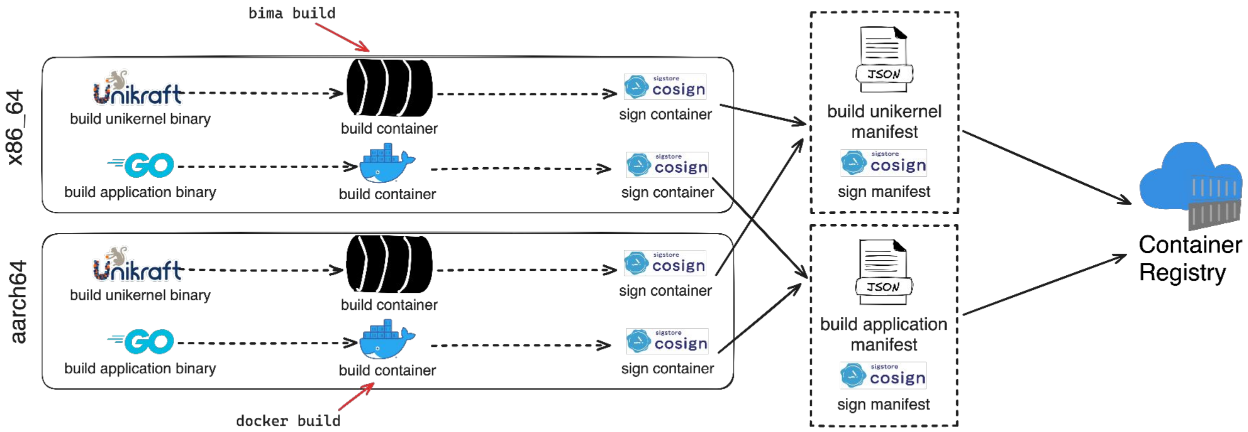
➜ generic & urunc almost identical



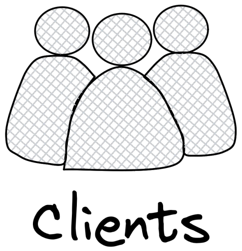
Demo

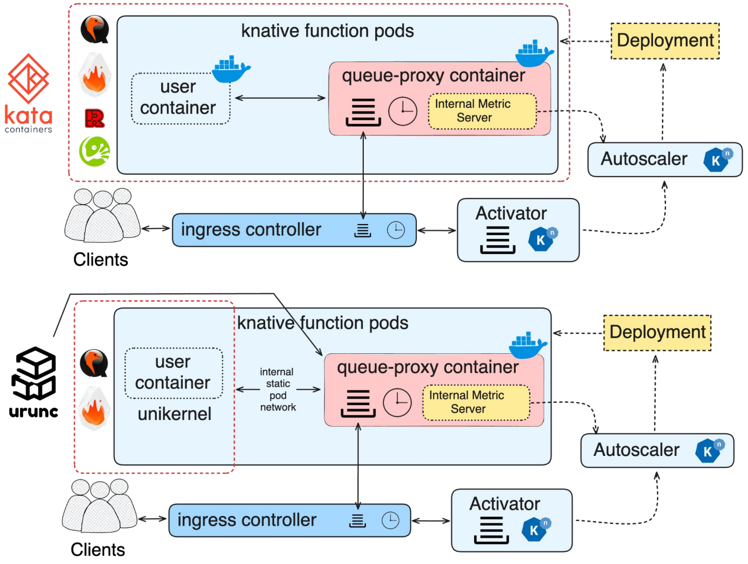
- Build workflow
  - automate the process of building unikernel & generic functions just like container images

github workflow



- cold-boot triggers & node overhead
  - capture memory overhead when spawning 10s of functions on an Edge device (eg NVIDIA Jetson) using generic, sandboxed container runtimes & urunc.





Acknowledgements

KubeCon | CloudNativeCon Europe 2024

ML SysOps

EMPYREAN

DESIRE6G

- This work is partially funded through Horizon Europe actions, MLSysOps (GA: 101092912), DESIRE6G (GA: 101096466), and EMPYREAN (GA: 101136024)

HORIZON EUROPE

European Commission

Co-funded by the European Union

6GSNS

Summary & Feedback

- ➜ containers offer **hassle-free** development & execution in diverse environments
  - ➜ orchestration platforms such as k8s are tightly coupled with the container ecosystem
- ➜ sandboxing containers to ensure isolation brings **overhead**, especially in FaaS setups where **short-lived tasks** dominate
- ➜ unikernels **reduce** the **attack surface** & **spawn times**, but are not cloud-native
- ➜ **urunc** appears as the missing component, enabling the use of unikernels in **FaaS** frameworks such as **Knative**

Check out the code on github:

- ➜ https://github.com/nubificus/urunc
- ➜ https://github.com/nubificus/bima
- ➜ https://github.com/nubificus/unikernel-demo

Check out the evaluation blog post:

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

Summary & Feedback

- containers offer **hassle-free** development & execution in diverse environments
  - orchestration platforms such as k8s are tightly coupled with the container ecosystem
- sandboxing containers to ensure isolation brings **overhead**, especially in FaaS setups where **short-lived tasks** dominate
- unikernels **reduce** the **attack surface** & **spawn times**, but are not cloud-native
- **urunc** appears as the missing component, enabling the use of unikernels in **FaaS** frameworks such as **Knative**

Thanks!

Check out the code on github:

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

Check out the evaluation blog post:

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