Introduction to Containers II

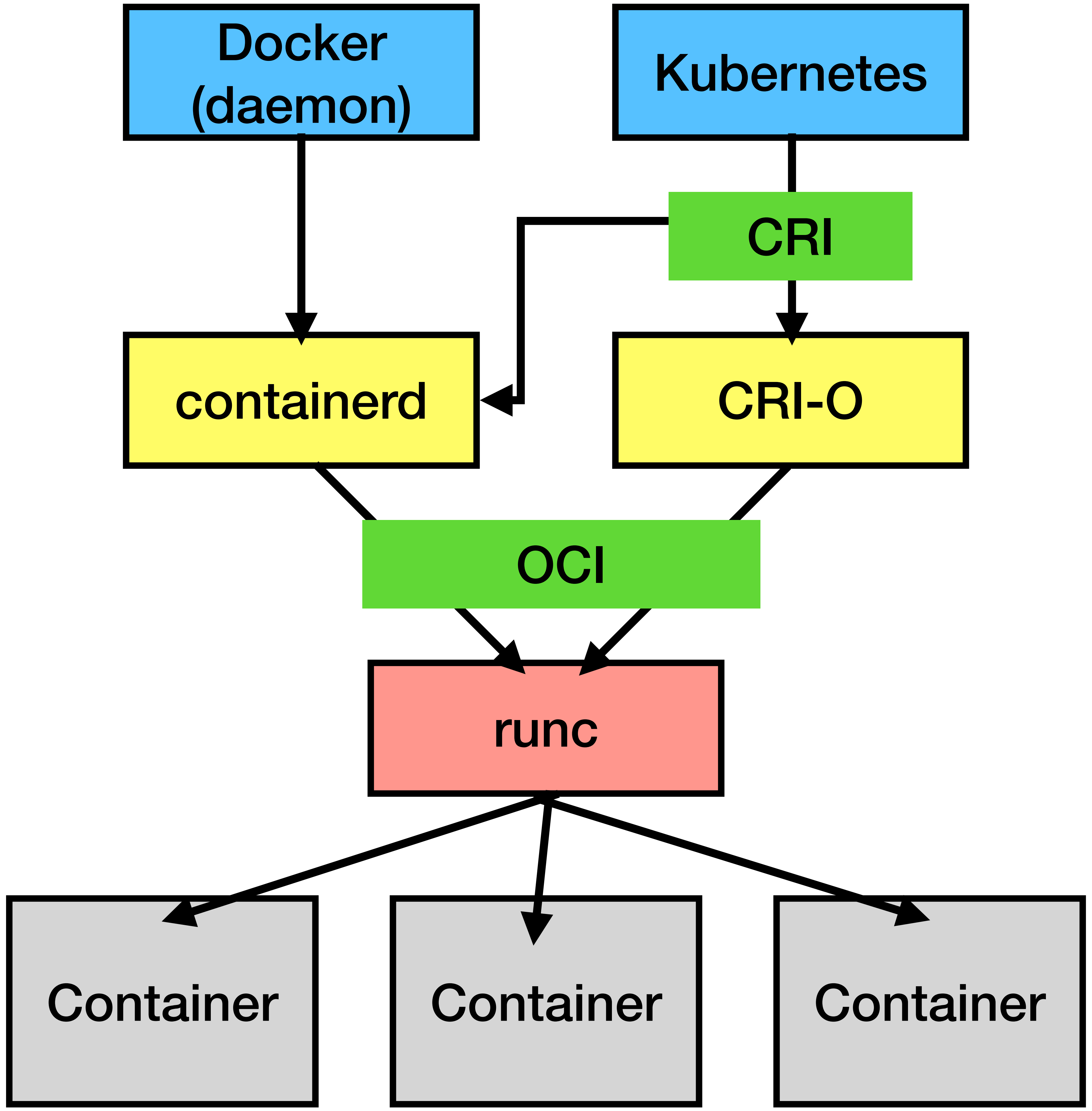
Prof. Shih-Wei Li

Department of Computer Science and Information Engineering
National Taiwan University

Agenda

- **Kubernetes and Docker Swarm**
- Serverless Computing
- Unikernels, MicroVMs, and Containers

Review: Container Ecosystem



Kubernetes

- Docker supports running containers in a single node (machine)
- Kubernetes (k8s) supports running containers across a cluster of nodes
  - Provides container orchestration and management

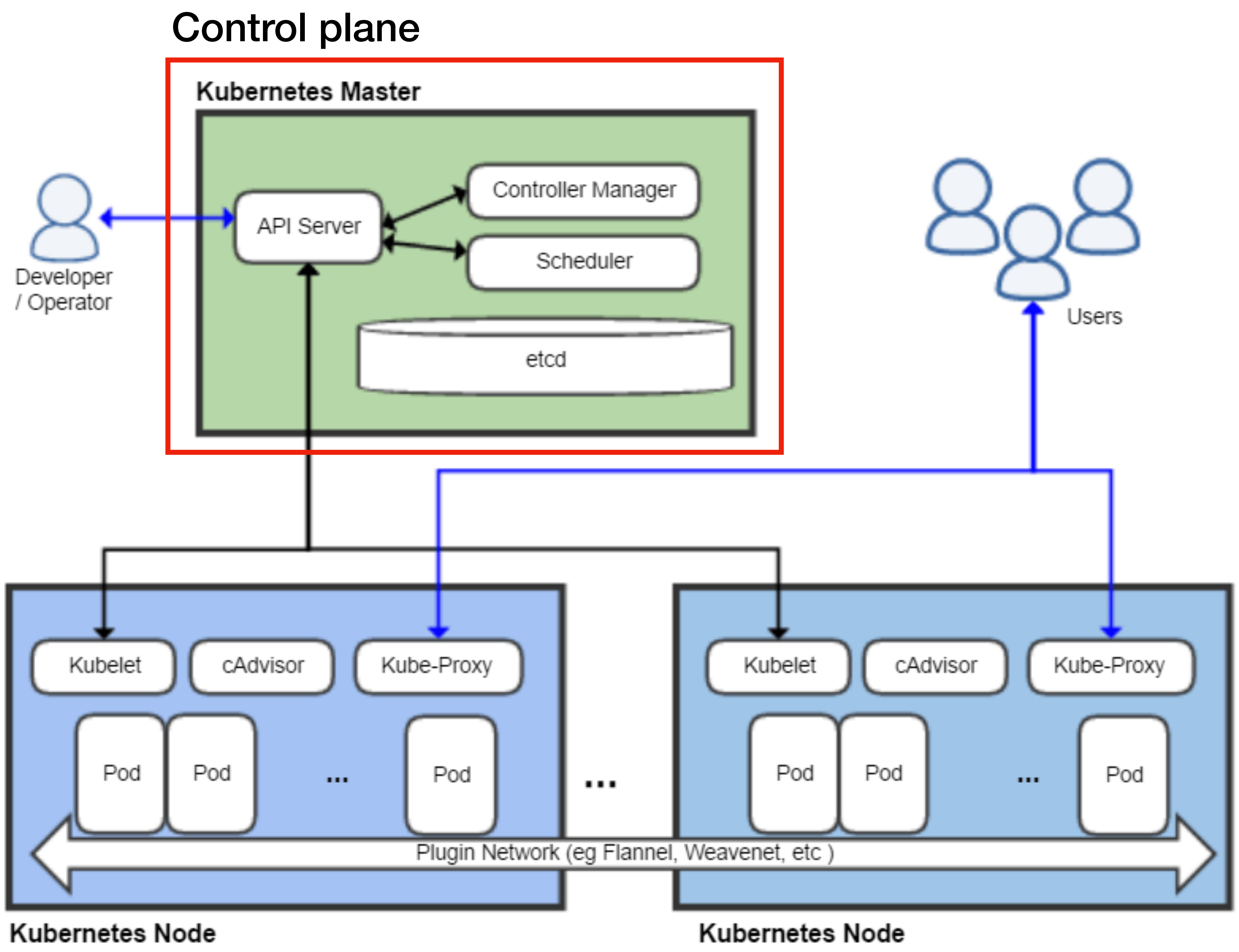
Kubernetes

- ***Service discovery and load balancing:***
    - Expose a container using the DNS name or IP address
    - Distribute workloads to containers for load balancing
- ***Storage orchestration:***
    - Kubernetes supports automatically mounting a storage system as local storages or cloud providers specific file systems

Kubernetes

- ***Automatic rollouts and rollbacks:***
  - Allow users to describe the desired state for the deployed containers
  - Can change the actual state to the desired state at a controlled rate
  - Example: automate Kubernetes to create new containers for one's deployment, remove existing containers and adopt all their resources to the new container
- ***Self healing:***
  - Restart and replace failing containers
  - Kill containers that failed to pass health check

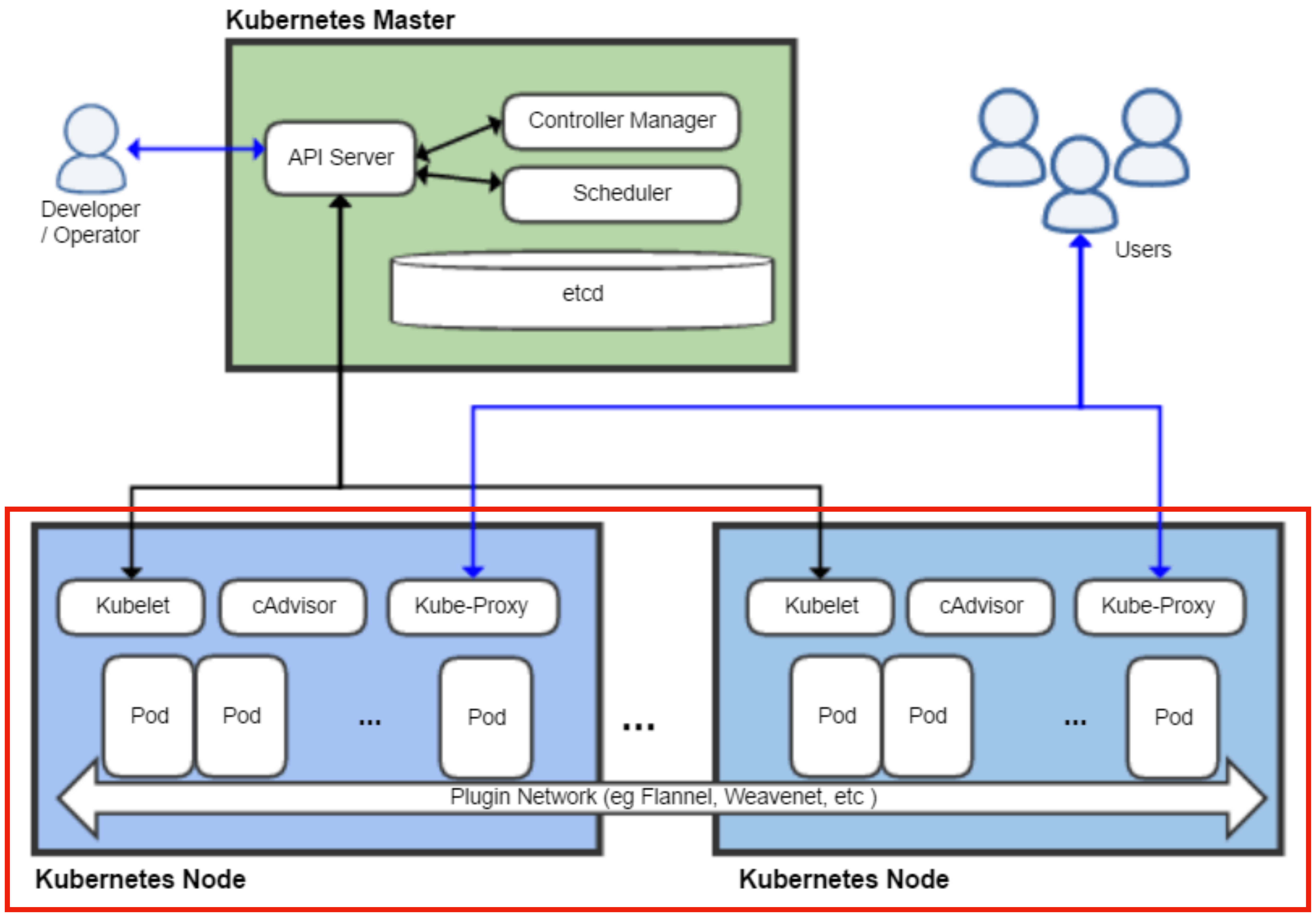
Kubernetes


Control plane
Kubernetes Master
API Server
Controller Manager
Scheduler
etcd
Developer / Operator
Users
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
...
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
Plugin Network (eg Flannel, Weavenet, etc)
Kubernetes Node
Kubernetes Node

Kubernetes

- Control Plane (Kubernetes Master):
  - ***API Server (kube-apiserver):*** exposes Kubernetes API using JSON; processes and validates REST requests; allowing configuration of workloads and containers across worker nodes
  - ***etcd:*** consistent and highly-available key value store used as Kubernetes' backing store for cluster status data and network configs (buffer requests received by the API server)
  - ***Scheduler*** (kube-scheduler): selects which node an unscheduled pod runs on based on resource availability
  - ***Controller Manager:*** includes a set of controllers (example: node controller)

Kubernetes


Kubernetes Master
Developer
/ Operator
API Server
Controller Manager
Scheduler
etcd
Users
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
...
Kubelet
cAdvisor
Kube-Proxy
Pod
Pod
...
Pod
Plugin Network (eg Flannel, Weavenet, etc)
Kubernetes Node
Kubernetes Node

Kubernetes

- ***Kubernetes kubelet:*** an agent that runs on each node in the cluster
  - Takes care of starting, stopping, and maintaining application containers organized into ***pods*** as directed by the control plane
  - Communicate with the container runtime on the node
- ***Kubernetes Pods:*** a group of one or more containers on the same node for running single instance of a given application
  - Pods are the smallest deployable units of computing that you can create and manage in Kubernetes
  - Each has an unique ID and IP address
- ***Kubernetes Proxy:*** maintains rules for Pod network communication inside or outside the cluster

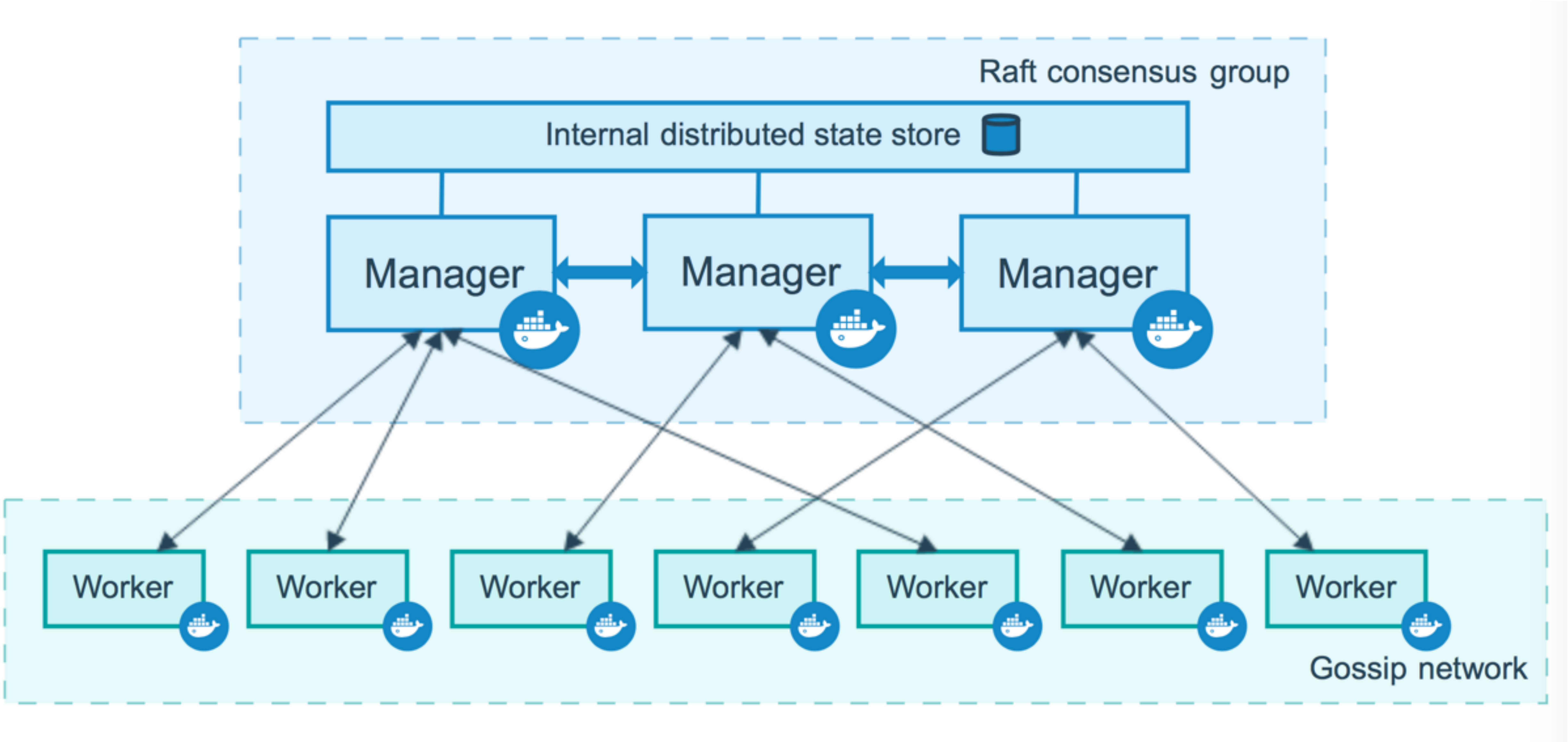
Kubernetes

- Kubernetes pod implementation
  - Containers in a pod share the same Linux namespaces, cgroups, IP addresses, etc.
  - Containers in a pod communicate via IPC or shared memory
- Two ways of Pods deployment:
  - ***A pod running a single container:*** most common Kubernetes use case; Pod as a wrapper around a single container for management
  - ***A pod running multiple containers:*** encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources (ex: storage)

Docker Swarm

- Docker swarm is Docker’s native technology for container orchestration
- A swarm consists of multiple Docker engines which run in swarm mode and act as managers and workers

Docker Swarm



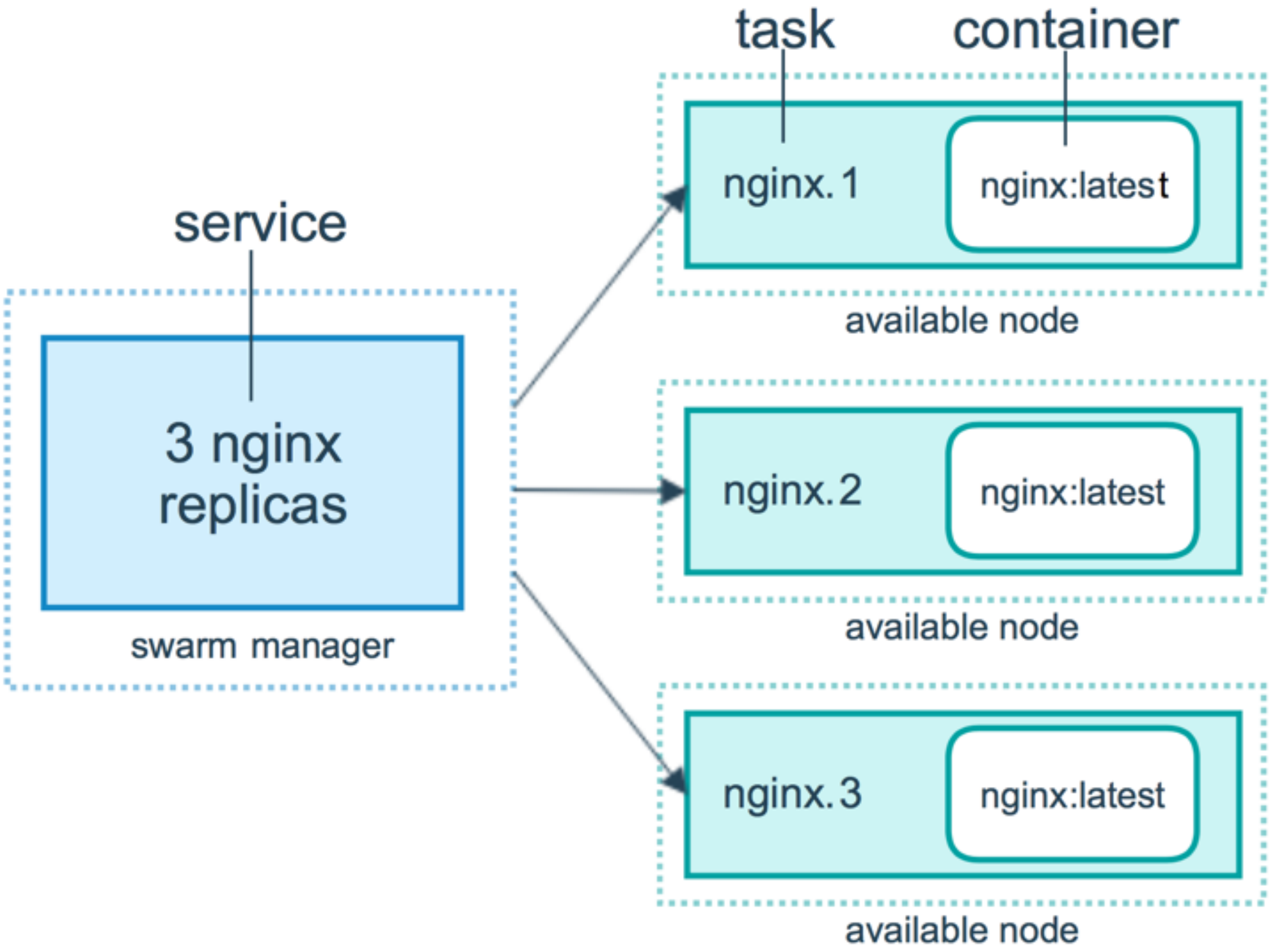
From: https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/

Docker Swarm

- To deploy an application image when Docker Engine is in swarm mode, you create a service:
  - ***Service:*** the definition of the tasks to execute on the manager or worker nodes; docker user specifies the container image and the number of replicas
  - ***Task:*** the units of work dispatched by swarm managers to the worker nodes within the docker swarm
    - The atomic unit of scheduling within a swarm

```
# docker service create --replicas 2 --name mynginx nginx
```

Docker Swarm



From: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/

Agenda

- Kubernetes and Docker Swarm
- **Serverless Computing**
- Unikernels, MicroVMs, and Containers

Serverless Computing

- A cloud computing execution model in which the cloud provider allocates machine resources on demand
  - Service providers take care of the servers on behalf of their customers
- Goals of serverless computing:
  - Relieve the cloud users from dealing with infrastructure and platforms — developers can focus on the application logic
  - Support pay-as-you-go billing model
  - Auto-scaling based on the user's demand

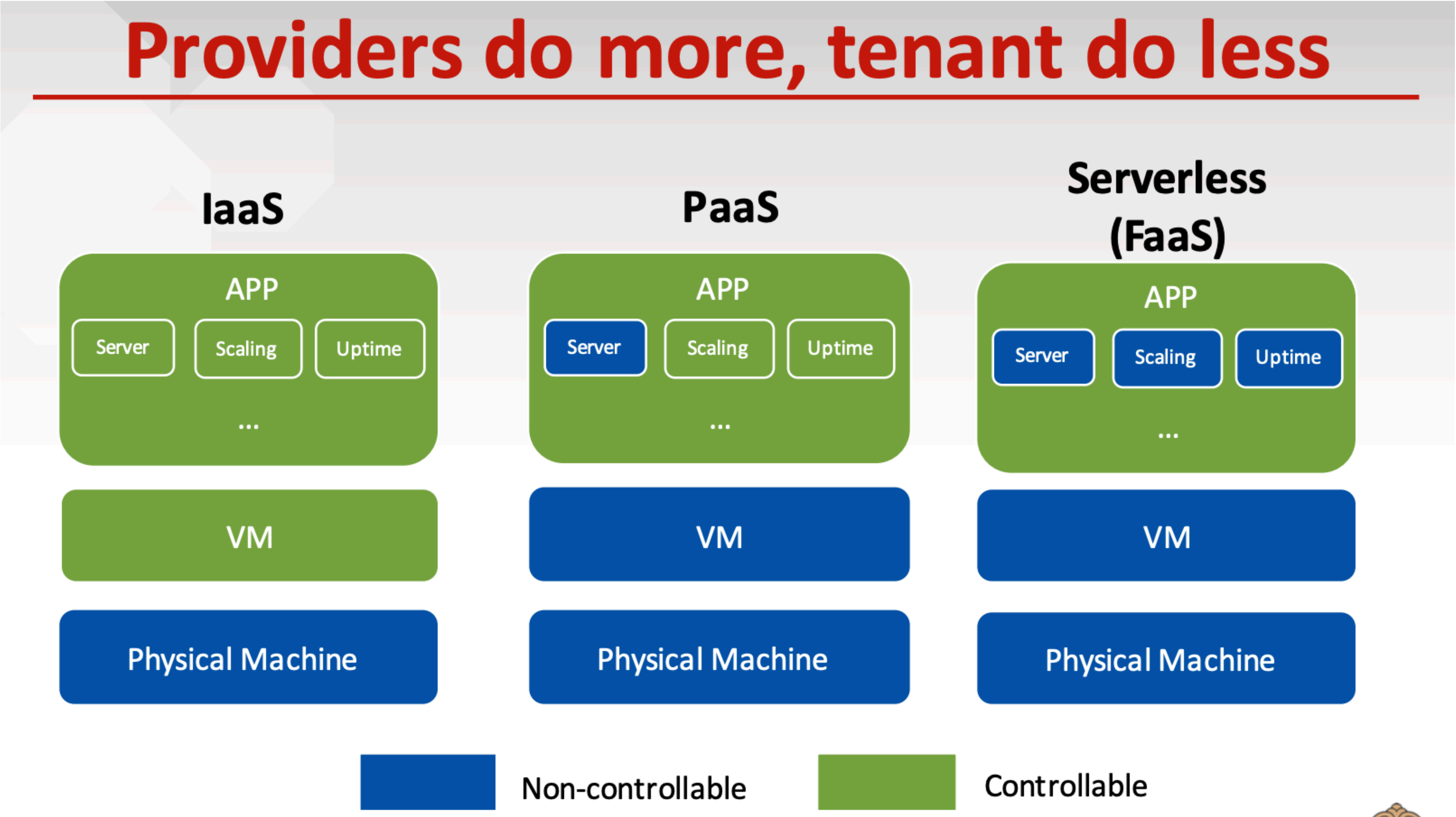
Serverless Computing

- Traditional deployment of web applications
  - Implement applications over a web framework (ex: Ruby on Rails)
  - Allocate a web server to run the code, setup/install environment, manage server utilization and scalability, manage OS and web stack
  - Require costs (allocated resources are often overkill for required by the functions) and uneasy management effort

Serverless Computing

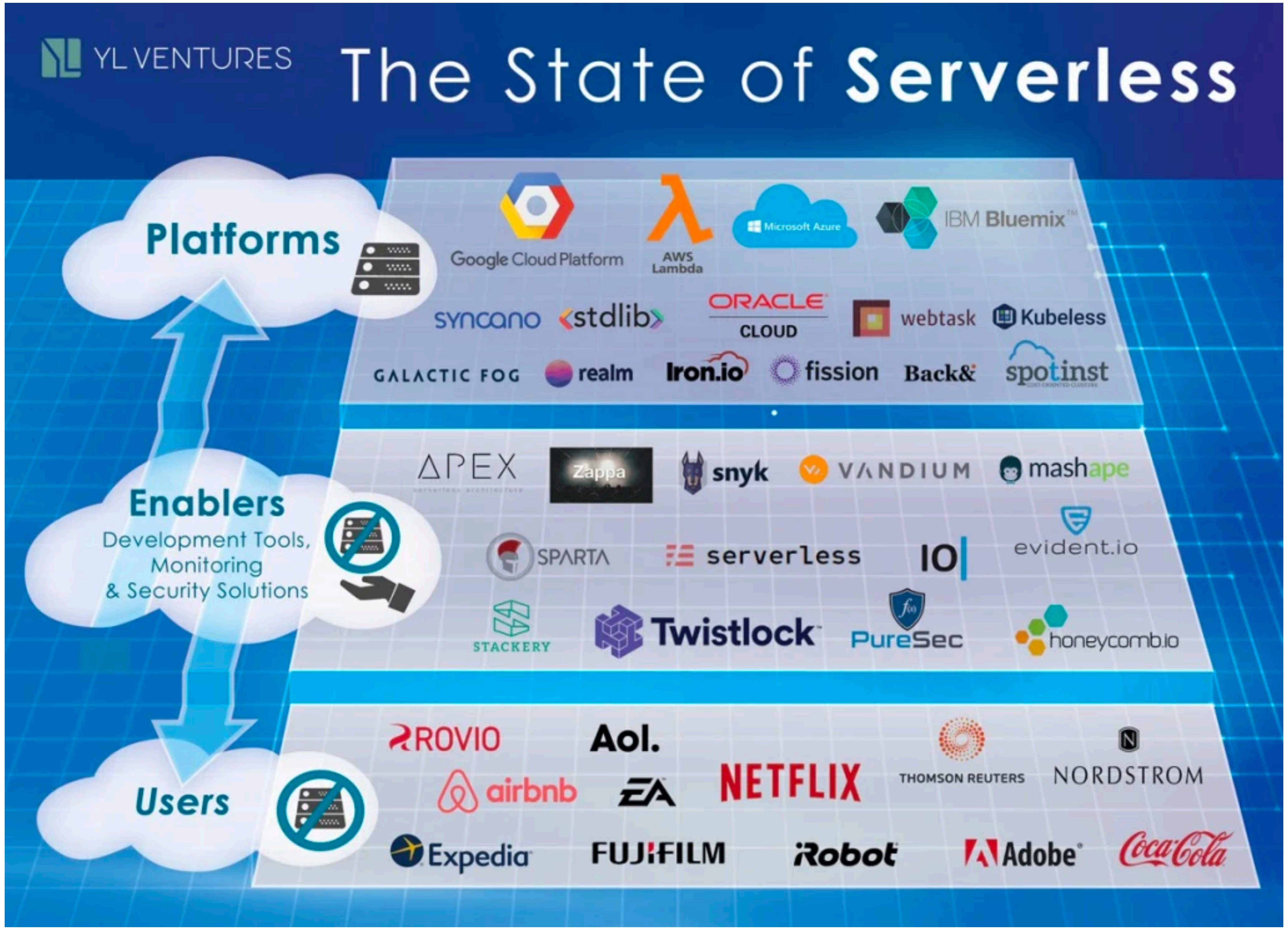
- Running serverless applications frees the developers from:
    - Capacity planning
    - Environment installation and configuration
    - Server management and maintenance
    - Manually scaling resources for containers, VMs, or physical servers
    - Cost management — developers only pay for the time the function is executing (triggered by user defined events)

Serverless Computing: Comparison



From: Peeking Behind the Curtains of Serverless Platforms [ATC 18]

Serverless Computing: Implementation



From: https://venturebeat.com/2017/10/22/the-big-opportunities-in-serverless-computing/

Serverless Computing: Implementation

- Cloud vendors provide their own serverless solutions

Amazon Lambda

Microsoft Functions

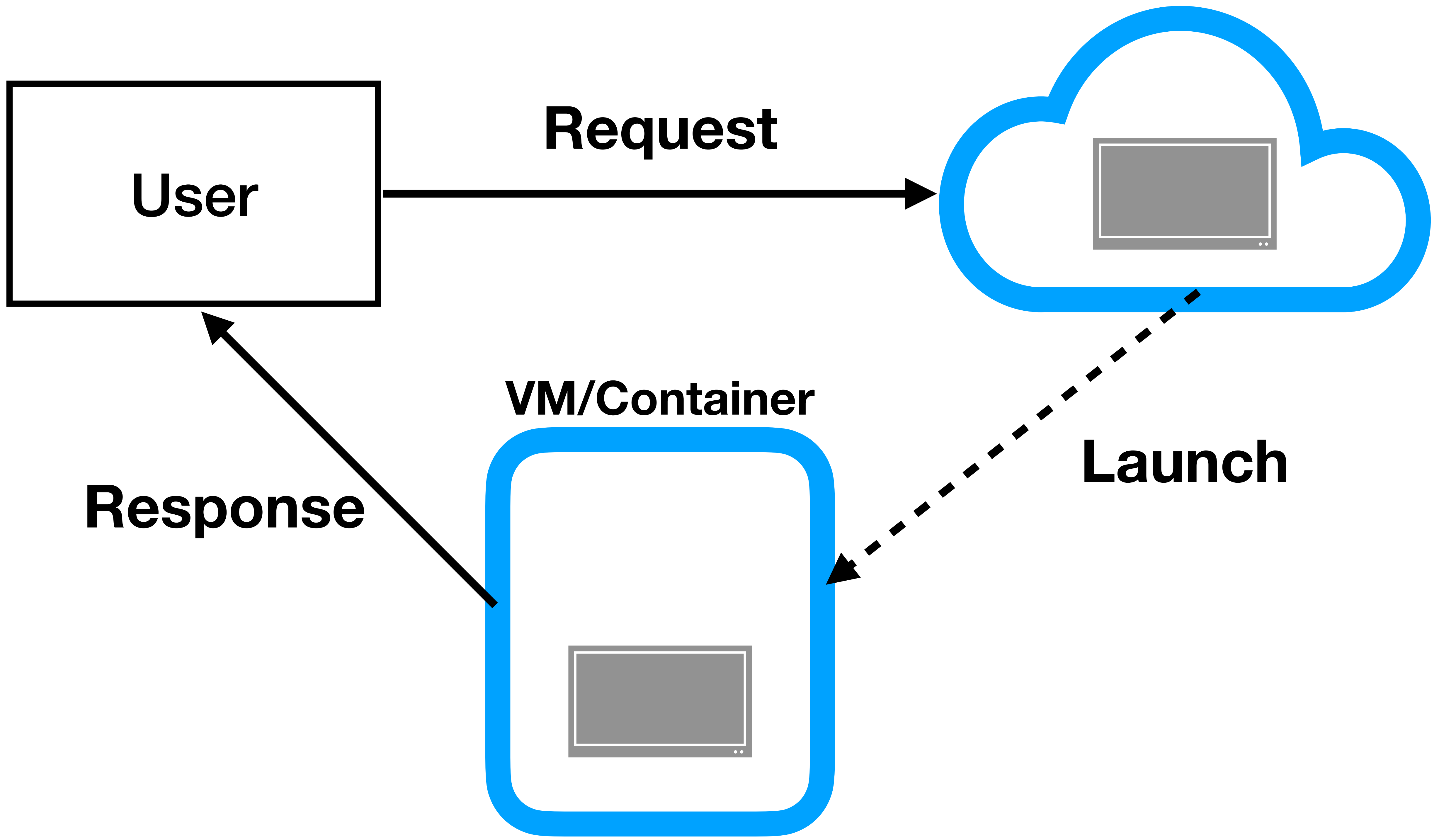
Google Cloud Functions

Serverless Computing: Implementation

- Cloud providers delegate the invocation of the serverless functions into a sandboxed environment:
    - To provide strong isolation between function instances
    - Sandboxed is usually backed by containers or VMs (MicroVMs)
    - Will discuss more in today's lecture later

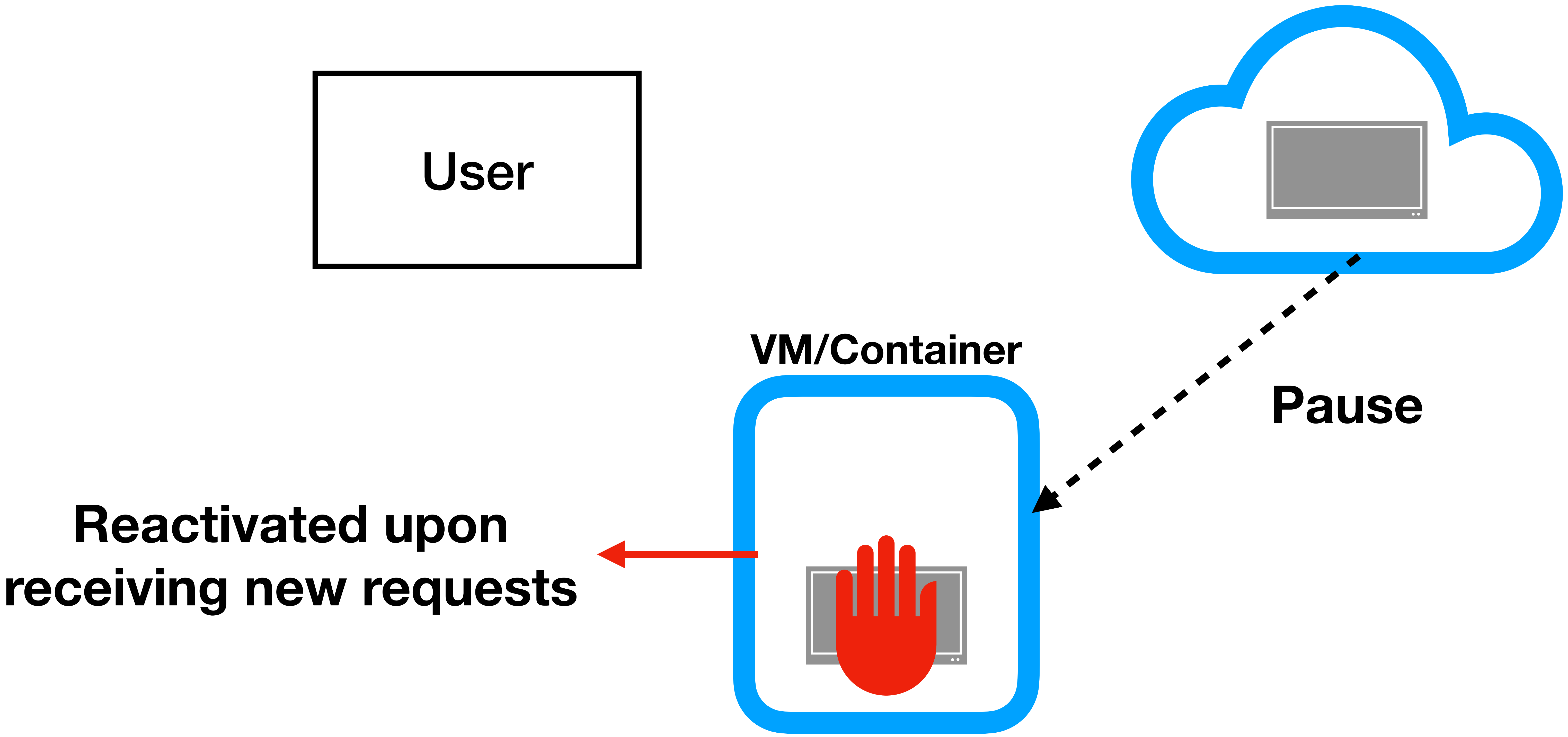
Serverless Computing: How it works

- A function runs in a container or VM (function instance) launched by the provider with limited CPU/memory/execution time



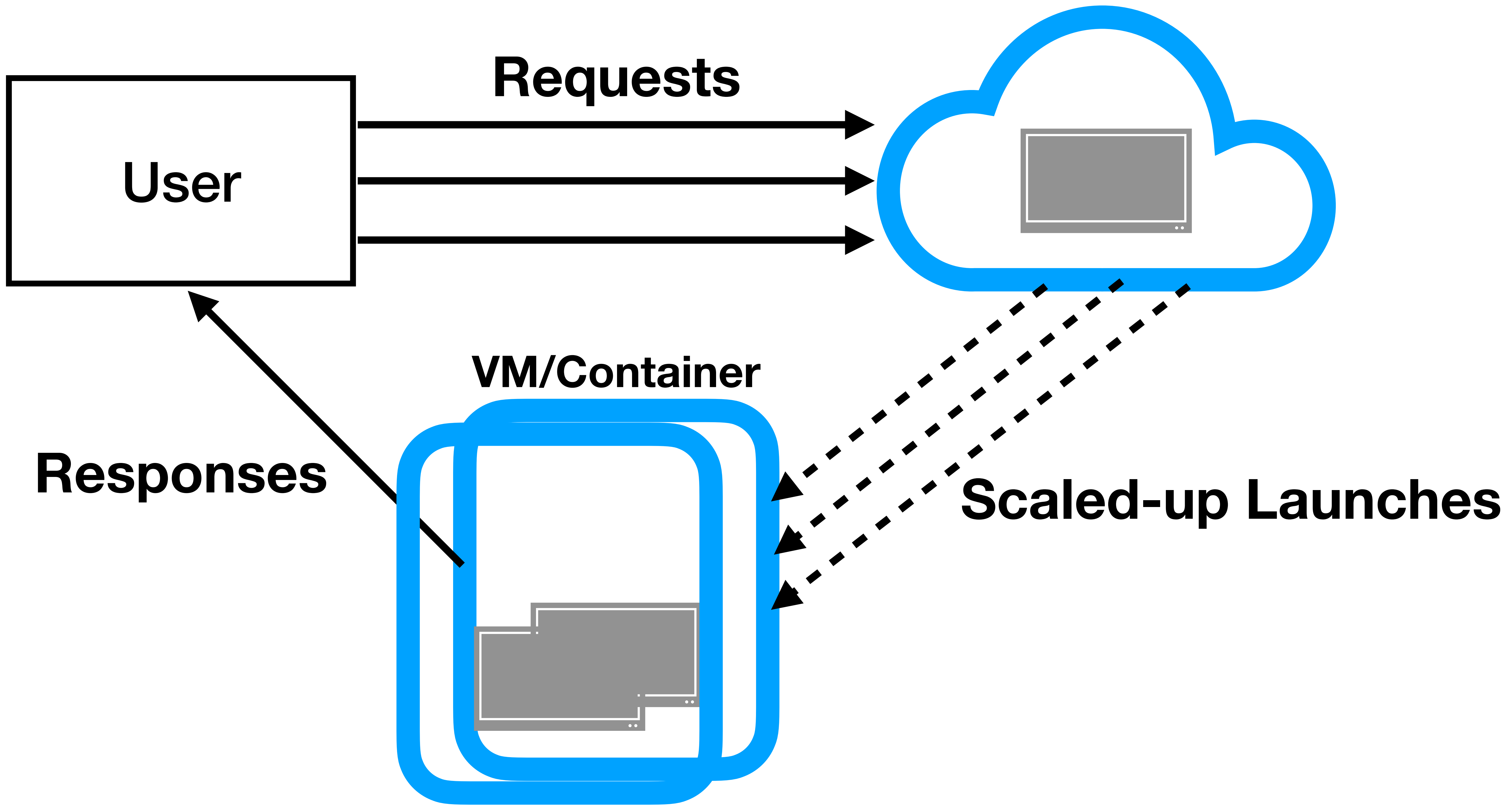
Serverless Computing: How it works

- The provider freezes the function instance when the invocation completes

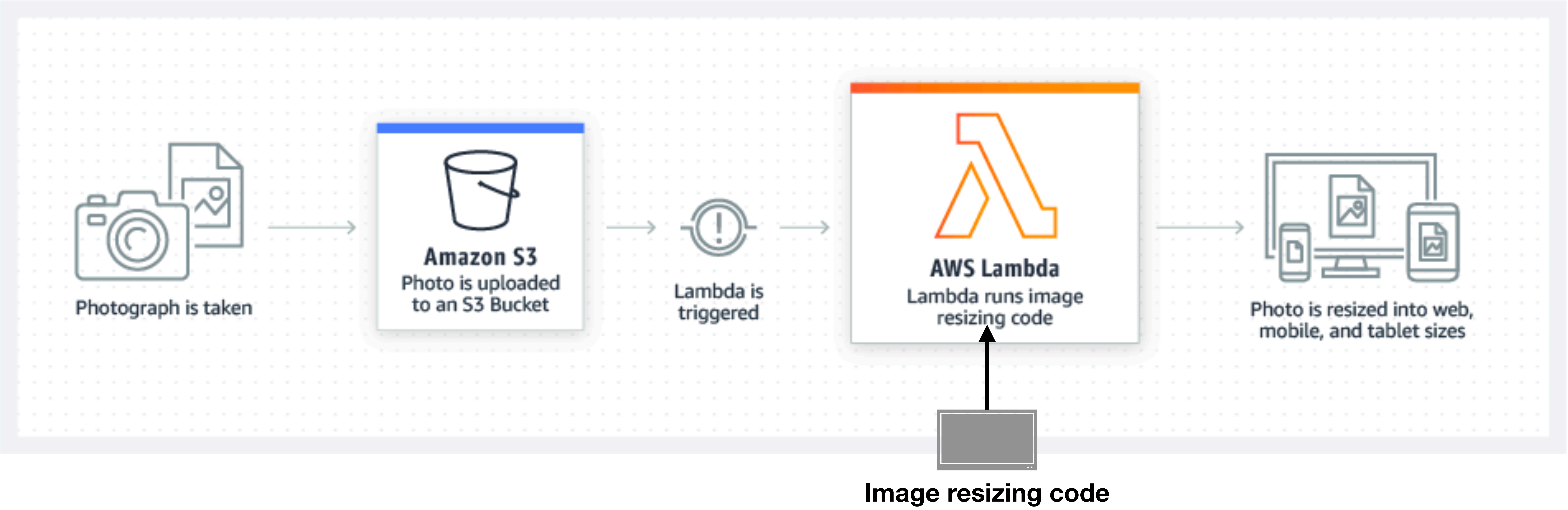


Serverless Computing: How it works

- The provider manages backend infrastructure and resources for users



Serverless Computing: Use Case



Serverless Computing: Limitations

- Deployed functions must be stateless — cannot depend on local states during executions
- Client program of serverless services might require redesign; clients have to track requests to services, and could become more complicated

Serverless Computing: Challenges

- Performance:
    - Serverless functions experience cold start issues
    - Infrequently used serverless code may suffer from more significant latency than code that is continuously running on a dedicated machine
        - Why? Service providers spin down the serverless code completely when not in use
    - Users cannot control how service providers schedule the functions

Agenda

- Kubernetes and Docker Swarm
- Serverless Computing
- **Unikernels, MicroVMs, and Containers**

Containers v.s. VMs

- Container offers lightweight virtualization solution compared to VMs, why?

Containers v.s. VMs

- Container offers a lightweight virtualization solution compared to VMs
  - Containers achieve faster boot time and smaller resource footprint
  - VMs offer better resource isolation with hardware virtualization support

Container Security

- Security Issue: huge host system call interface exposed to containers
  - Linux includes 400+ system calls, making it hard to secure
  - Why? Didn’t we just talk about Seccomp?
- Challenge: can one design virtual machines to achieve similar performance as containers? Must satisfy the following requirements [LightVM 17]
  - Fast instantiation: containers boot in 100+ milliseconds or less
  - High instance density: 10000 containers can run simultaneously
  - Pause/unpause: containers can be paused/unpaused quickly

Container Security: VM Size Challenges

- VMs suffer from worse performance and scalability compared to containers because of their size
    - Gigabytes versus Megabytes scale
    - Solution for VM optimization: reduce VM disk and memory usage

Unikernels

- Observation: VMs and containers in the cloud often run a single-purpose application
- ***Unikernels*** are specialized, single-address-space machine images constructed by using *library operating systems* (from unikernel.org)
    - Support only the kernel functionality needed by a target application, in contrast to general-purpose OS kernels like Linux and Windows
- First introduced by researchers from the University of Cambridge:
    - "Unikernels: Library Operating Systems for the Cloud" (in ASPLOS 13)
    - Introduced MirageOS written in OCaml

Unikernels

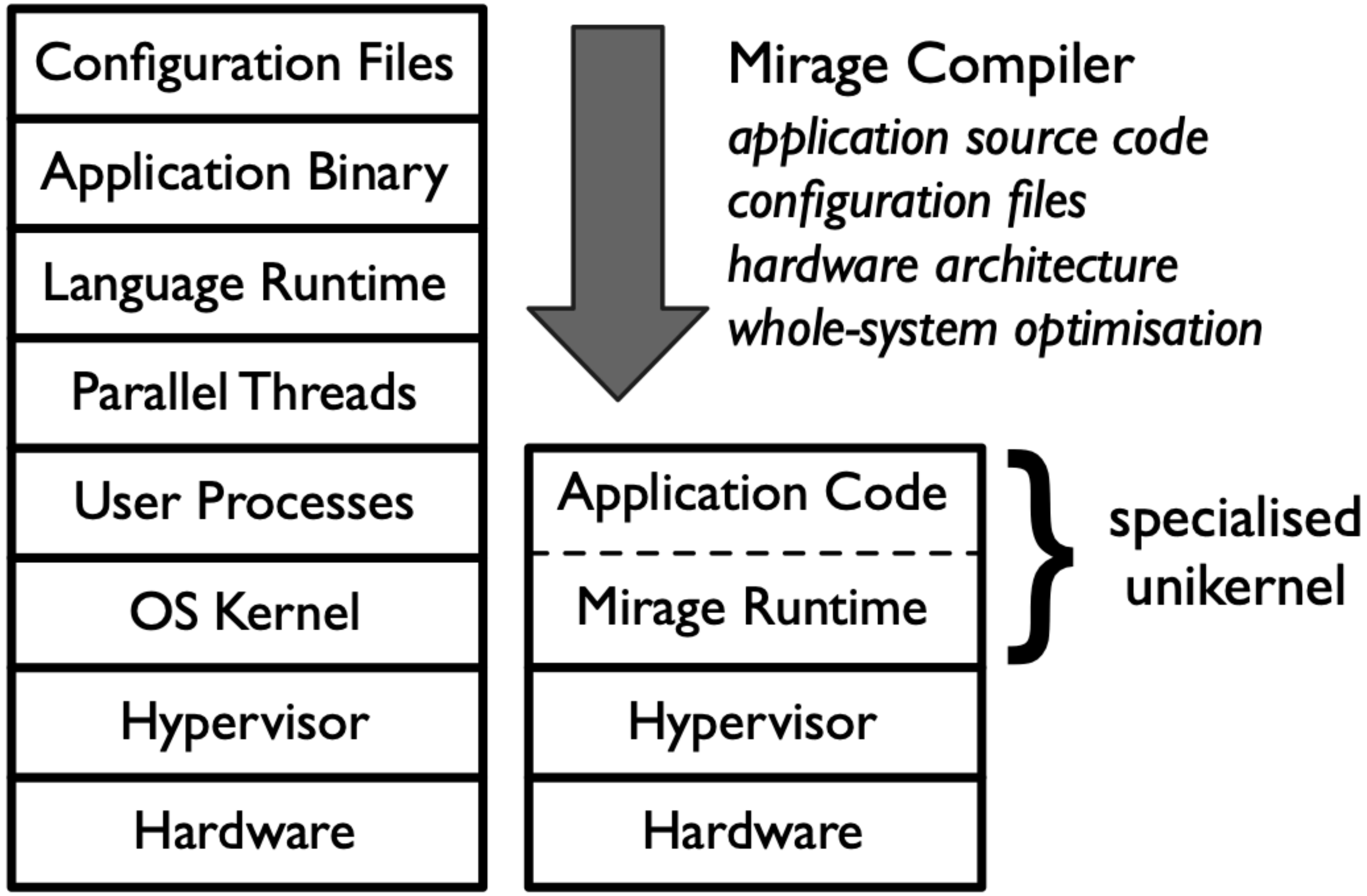


Figure 1: Contrasting software layers in existing VM appliances *vs*. unikernel's standalone kernel compilation approach.

Unikernels

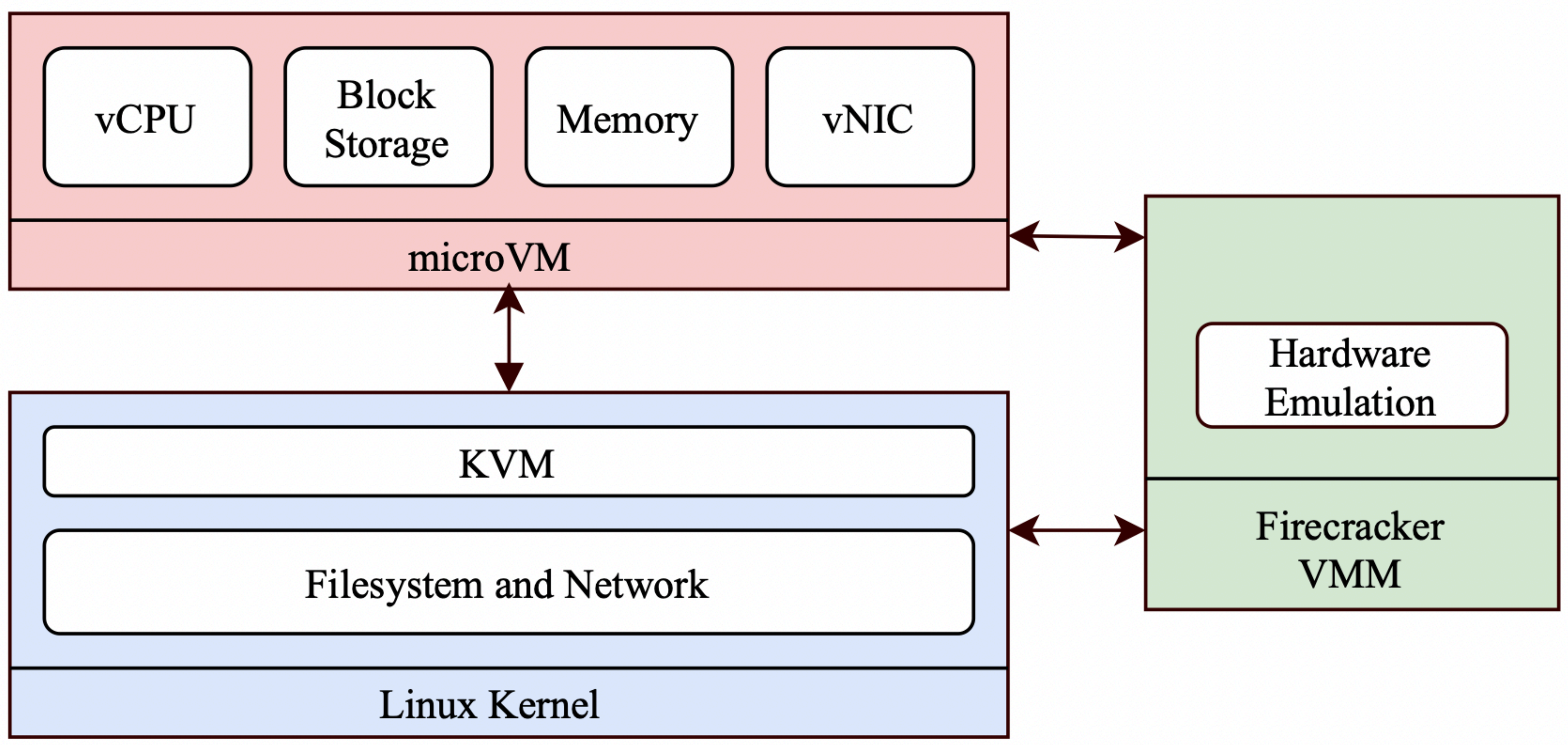
- Hypervisors provide virtualized “generic” virtual hardware abstractions
  - Less concern in unikenels about the hardware heterogeneity
- Run unikernels in VMs
  - Hypervisor enforces VM isolation and schedules VMs running Unikernels that focus on supporting the generic virtual device driver as opposed to attempting to support various hardware
  - Achieve must faster boot time and resource footprint than Linux VMs

Unikernels

- Limitation:
  - Unikernels are highly specialized, unsuitable for general purpose and multi-user environment
  - Unikernels are immutable: dynamically adding additional functionality or altering a compiled unikernel is not possible; require recompile and redeploy for updates

MicroVMs [Firecracker '20]

- ***AWS Firecracker:*** from Amazon; running containerized workloads in a microVM



From: Blending Containers and Virtual Machines: A Study of Firecracker and gVisor

MicroVMs [Firecracker ’20]

- Firecracker builds on KVM to support secure and performant MicroVMs
- Firecracker replaces QEMU with a new Rust based user space “VMM”
  - The VMM serves stripped-down functionality with a minimal device model to support the guest Linux OS

MicroVMs [Firecracker 20]

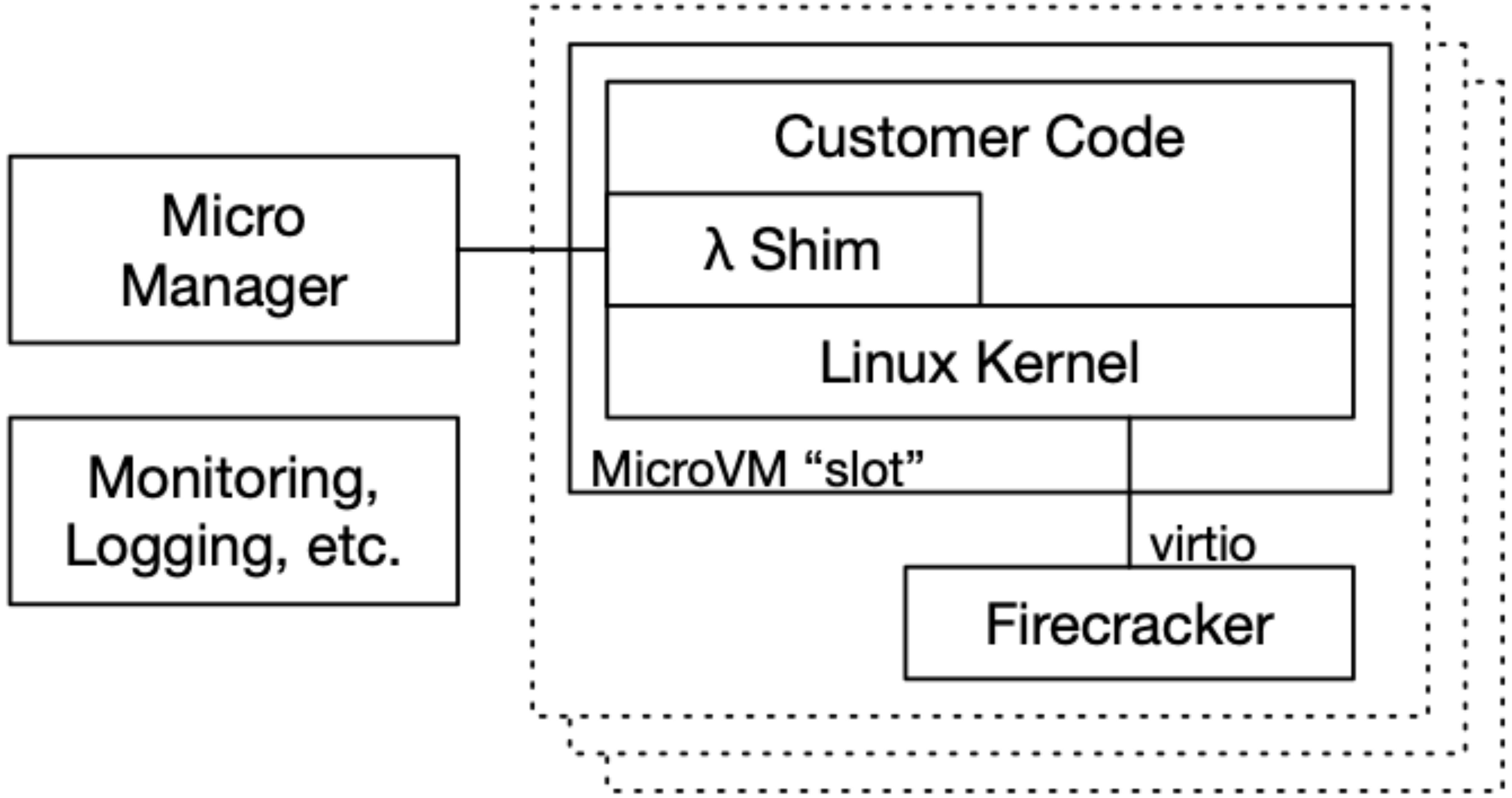


Figure 3: Architecture of the Lambda worker

MicroVMs [Firecracker 20]

- Amazon Lambda started adopting Firecracker in 2018
  - MicroVMs run as Lambda workers to execute serverless functions

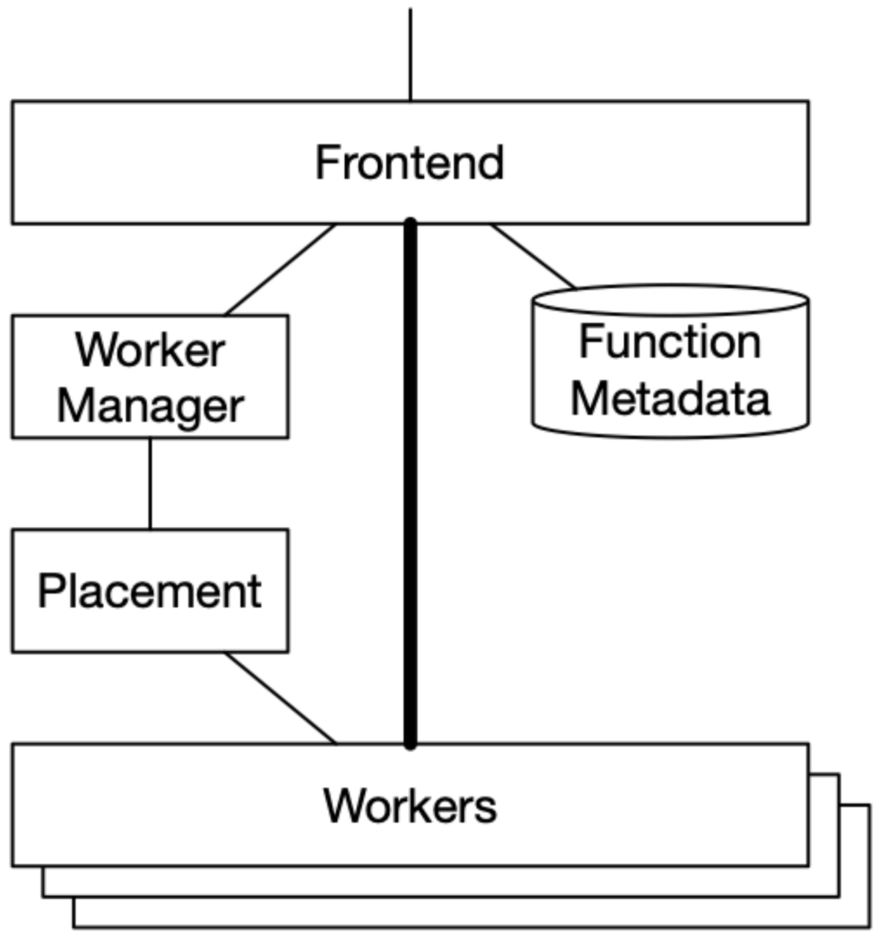


Figure 2: High-level architecture of AWS Lambda event path, showing control path (light lines) and data path (heavy lines)

MicroVMs [Firecracker 20]: More details

- One Firecracker (VMM) process is launched per MicroVM — responsible for creating and managing the MicroVM, providing device emulation and handling VM exits.
- Firecracker implements "*jailer*" to protect against unwanted VMM behavior (such as a bug that allows the guest to inject code into the VMM)
  - Sandbox the Firecracker process into a jailer before booting a VM
  - The jailer process runs in a chroot, uses namespaces and cgroups to protect system resources that require elevated permissions

MicroVMs [Firecracker 20]

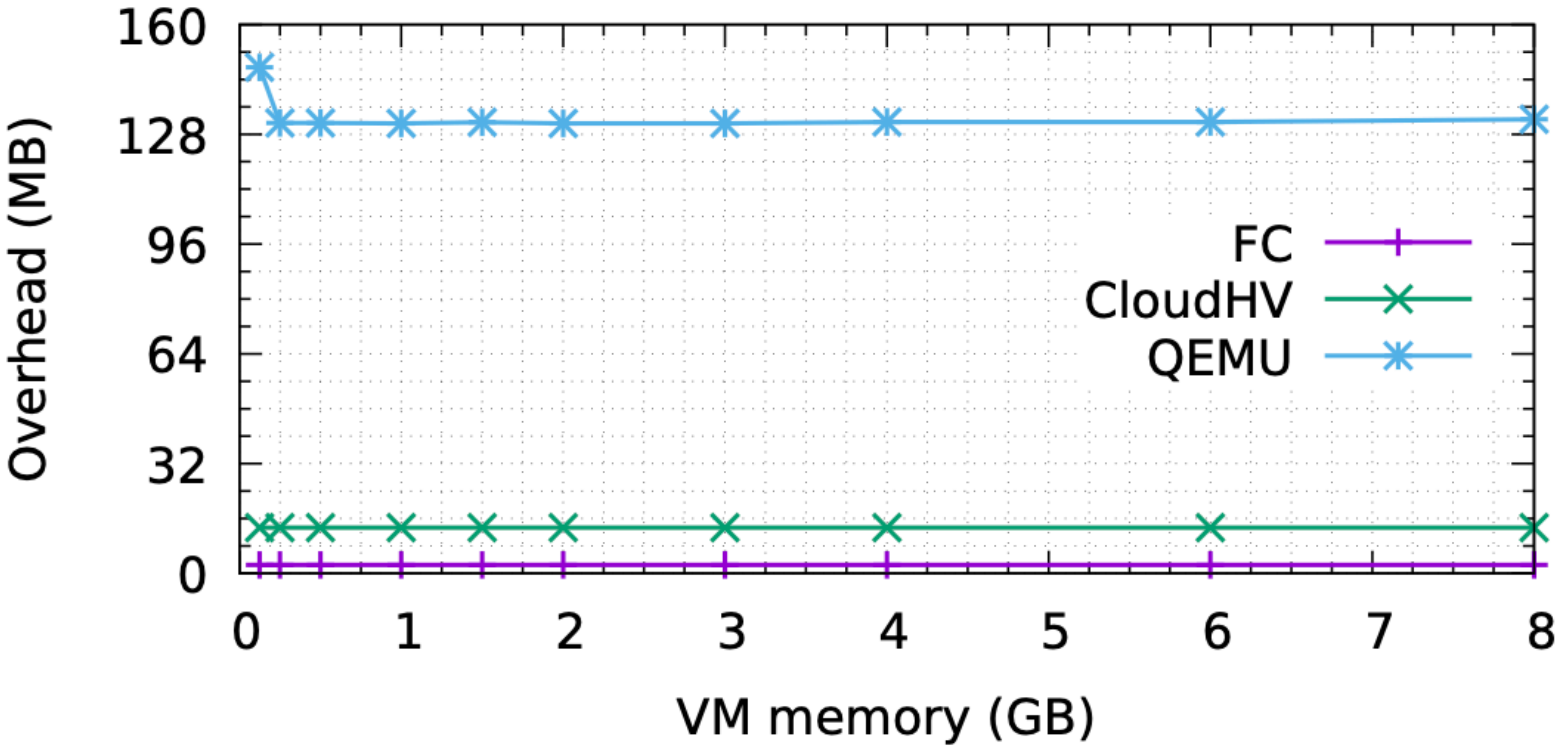


Figure 7: Memory overhead for different VMMs depending on the configured VM size.

MicroVMs [Firecracker 20]: Performance

- Firecracker has large performance overhead in block and network I/O

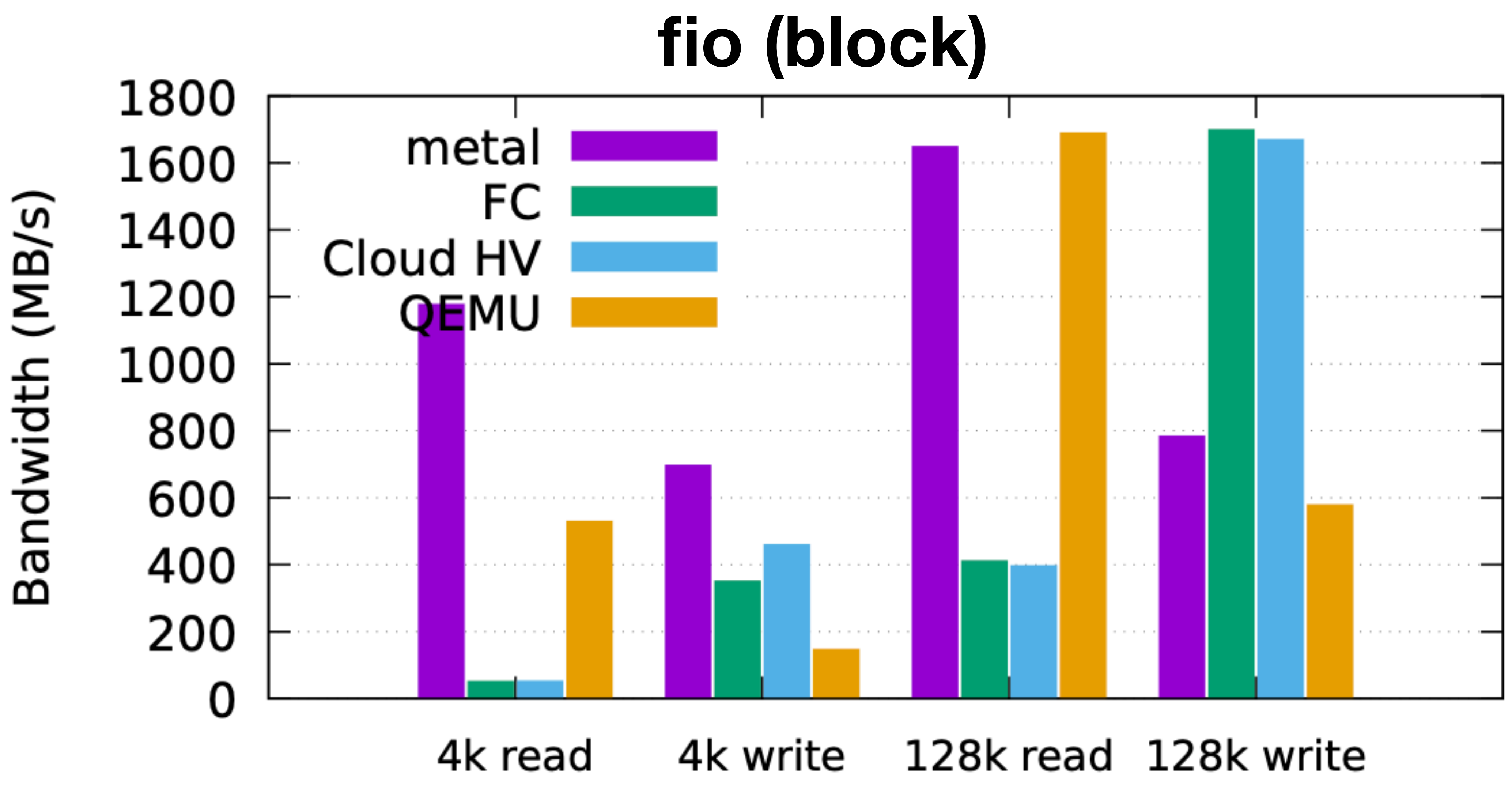


Figure 8: IO throughput on EC2 m5d.metal and running inside various VMs.

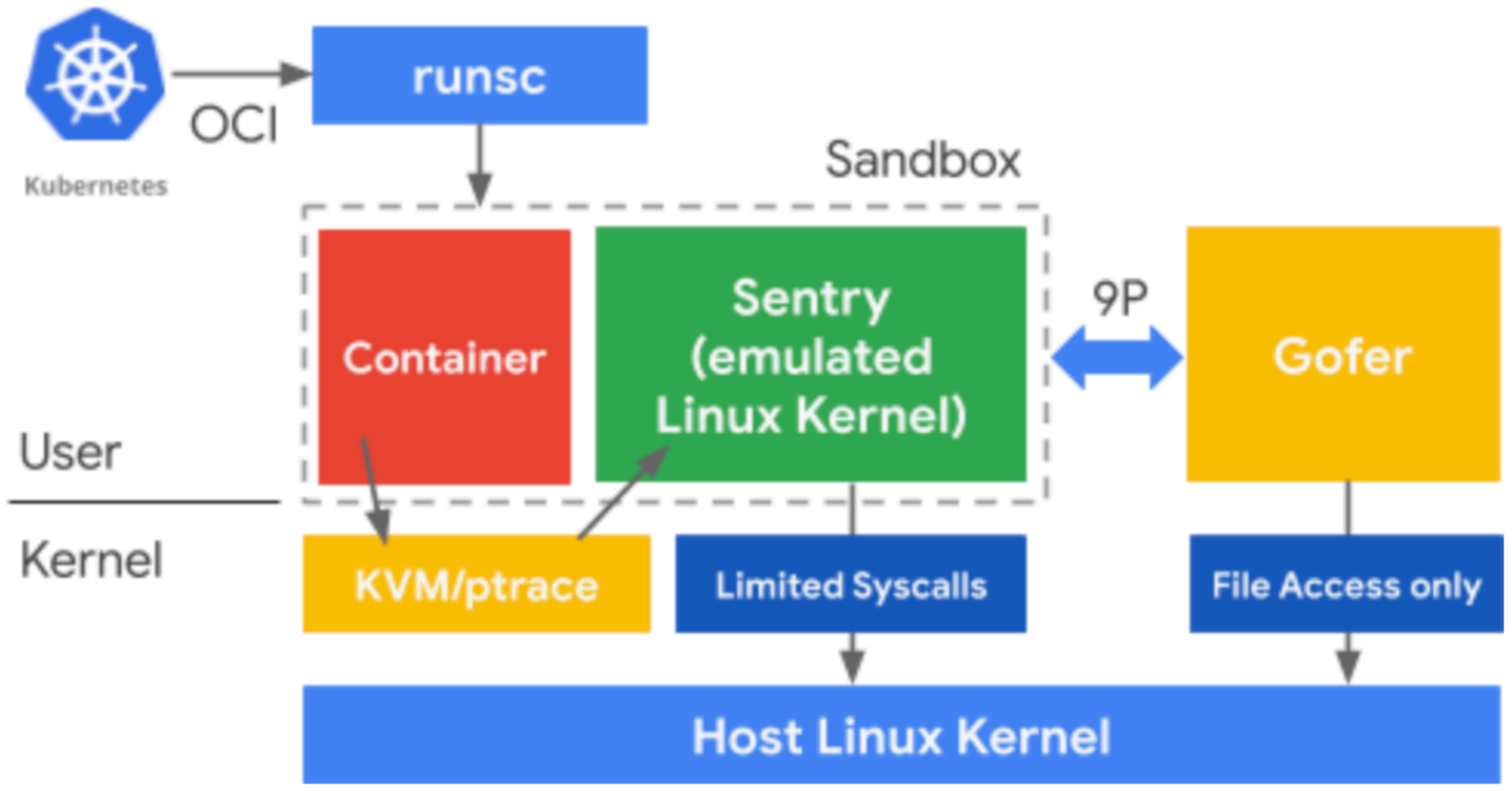
iperf (network)

VMM	1 RX	1 TX	10 RX	10 TX
loopback	44.14	44.14	46.92	46.92
FC	15.61	14.15	15.13	14.87
Cloud HV	23.12	20.96	22.53	N/A
Qemu	23.76	20.43	19.30	30.43

Table 1: `iperf3` throughput in Gb/s for receiving (RX) in the VM and transmitting (TX) from the VM for a single and ten concurrent TCP streams.

Google gVisor

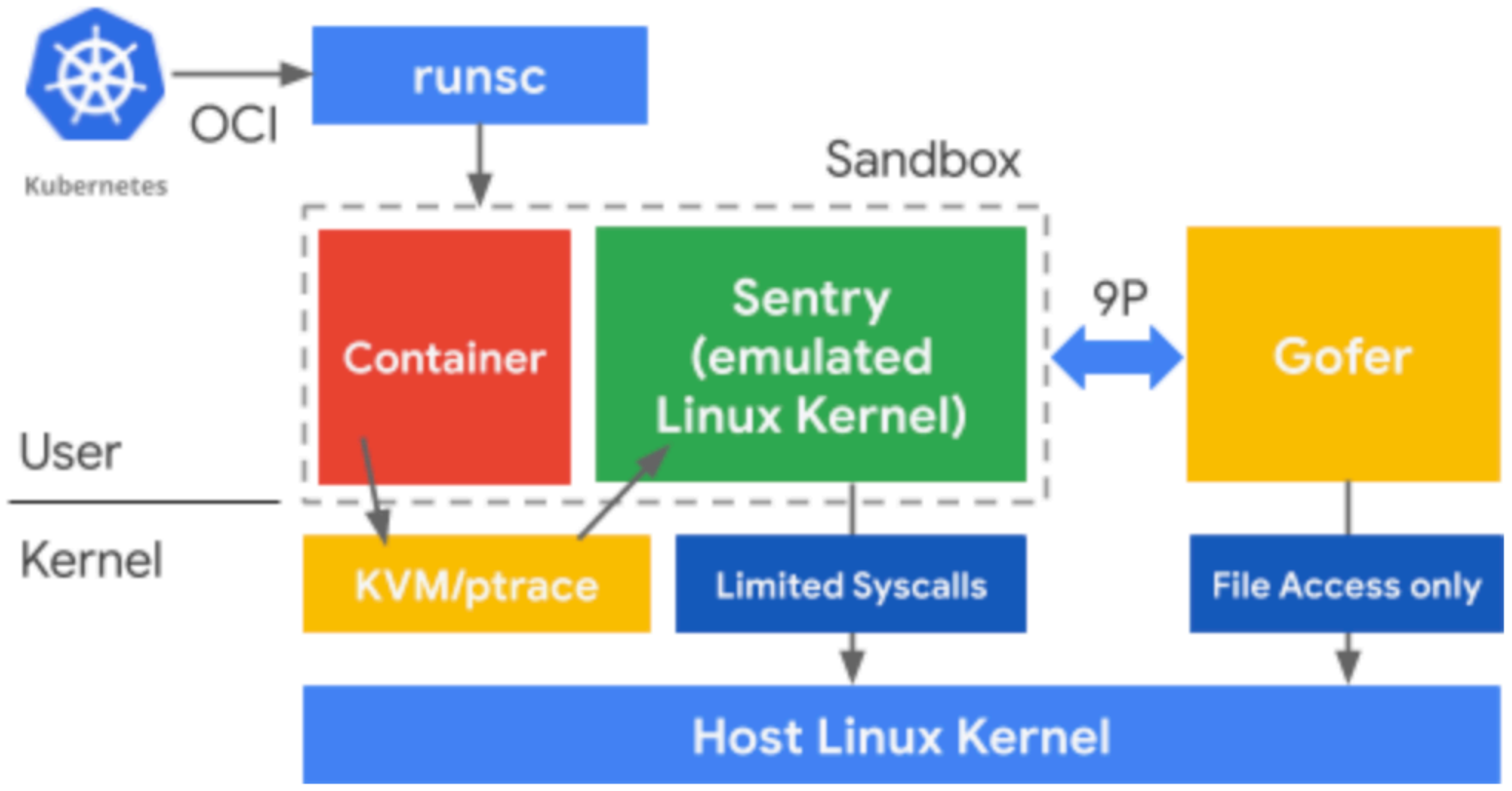
- gVisor allows the execution of untrusted containers, preventing them from adversely affecting the host or other containers
- Interpose system calls to forbid direct container interaction with the host OS



From: https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/

Google gVisor

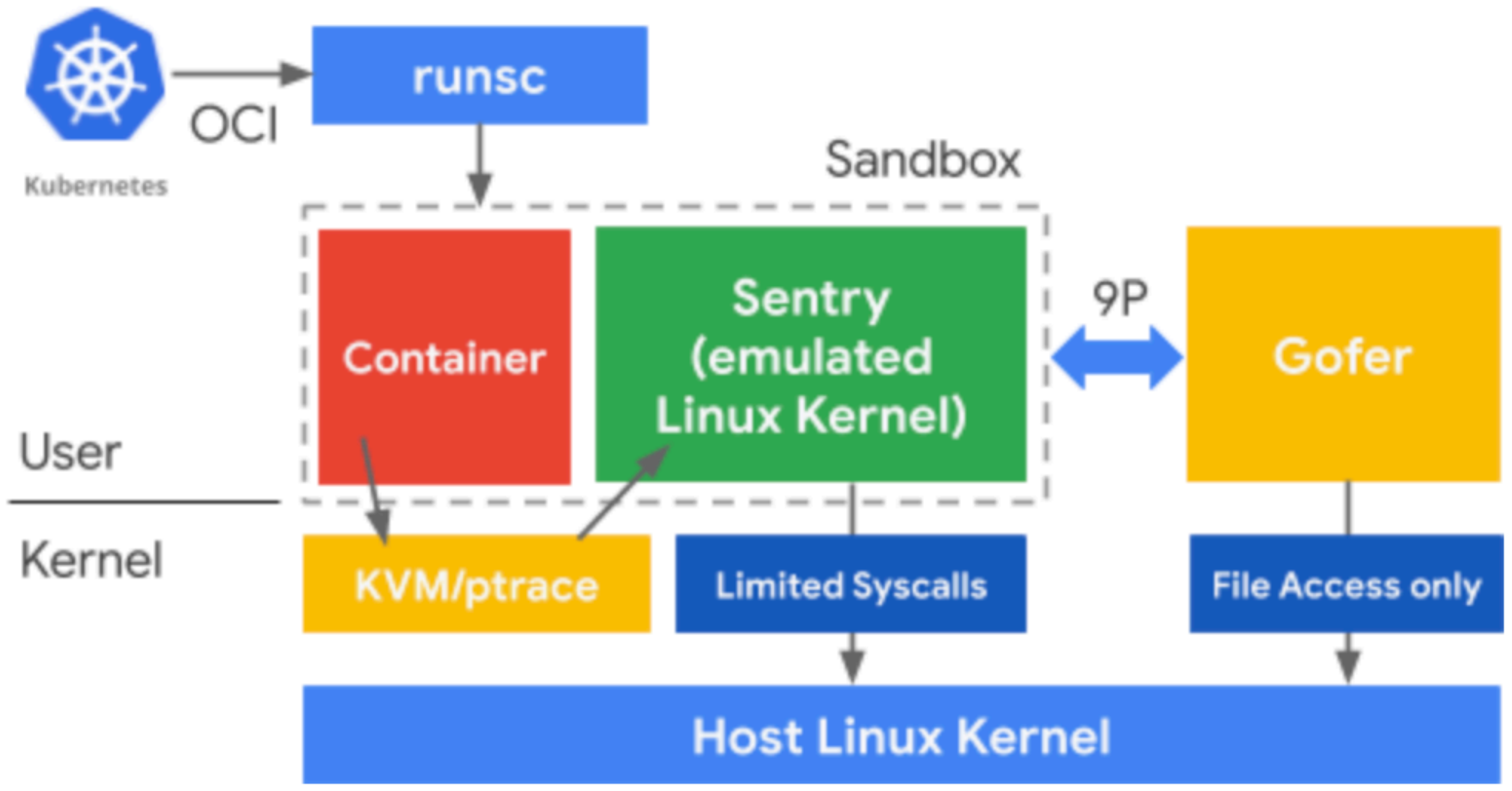
- gVisor includes an OCI compatible runtime called “runsc”
- runsc integrates with Docker and Kubernetes



From: https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/

Google gVisor

- Sentry:
  - The “application kernel” in user space that serves the untrusted application in the container:
  - Each container has its own Sentry
  - The Sentry handles system calls (200+), routes I/O to gofers, and manages memory and CPU
  - The Sentry is allowed to make limited, filtered system calls to the host OS

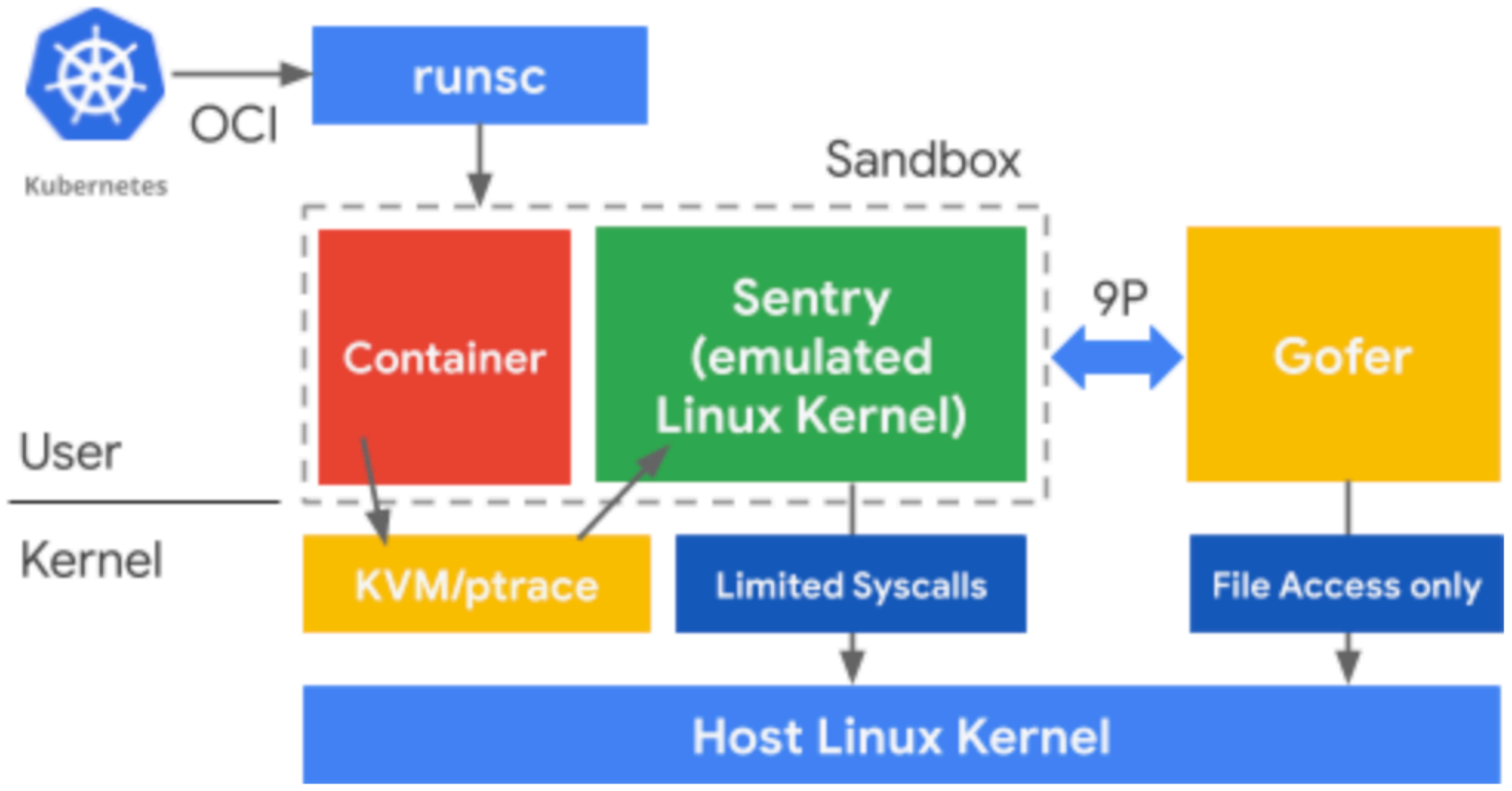


From: https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/

Google gVisor

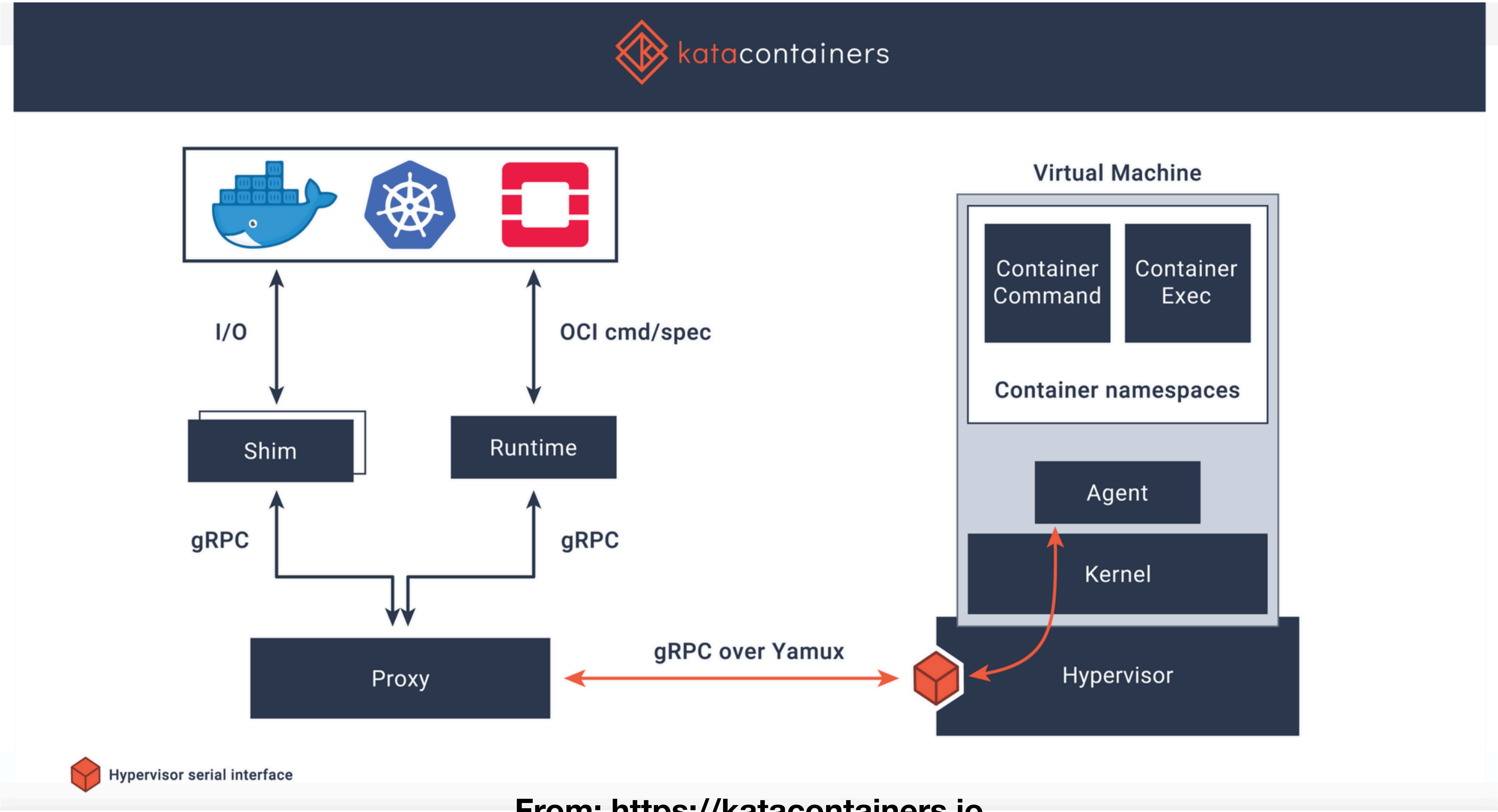
- Gofer: a process that specifically handles different types of I/O for the Sentry (disk I/O)

- Gofers are also allowed to make filtered system calls to the Host OS



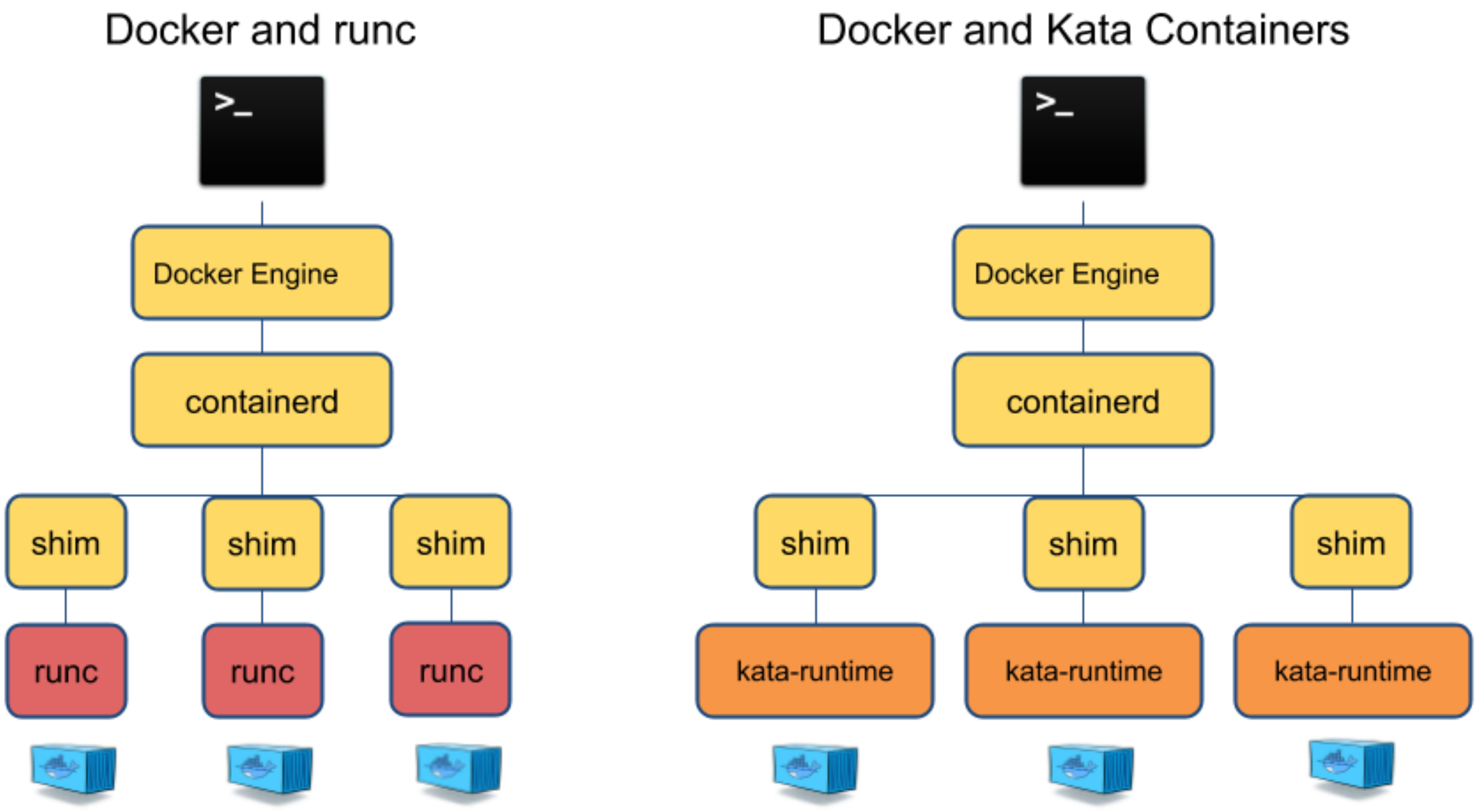
From: https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/

Kata Containers



From: https://katacontainers.io

Kata Containers



From: https://github.com/kata-containers/documentation/blob/master/design/architecture.md

Kata Containers

- Supports KVM, Firecracker, and Cloudhypervisor
- Supports Linux based distribution (Clear Linux, Fedora, CentOS)
- Kata Containers exposes an OCI compliant runtime
    - Can be run by existing container tools like Docker and Kubernetes

Container Ecosystem [Revisited]