TÉCNICO LISBOA

Serverless Snapshot Orchestration

André Filipe do Pilar de Jesus

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno

Examination Committee

Chairperson: Prof. Andreas Miroslaus Wichert
Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno
Member of the Committee: Prof. Luís Manuel Antunes Veiga

October 2025

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First, I would like to express my deep gratitude to my parents, my sister, and my life partner Leonor. Although they may not fully understand the specifics of my work or field, they have always believed in me and my abilities. Their unwavering support during difficult times, along with their patience, presence, and encouragement, were essential throughout this journey. Without them, I could not have done it.

I also want to acknowledge my uncle and my grandparents, whose love and guidance have always been a source of strength for me. In particular, I want to honor my grandpa, who sadly could not see me finish this thesis. He was always my buddy and my friend, and his pride in my achievements was unmistakable. I am certain that he would be proud of this accomplishment, and I carry his encouragement with me always.

I would also like to thank my supervisor, Prof. Rodrigo Bruno, not only for the opportunity to work with him but also for his insight, guidance, and the generous sharing of his knowledge, which made this thesis possible. I extend this acknowledgment to Prof. Dmitrii Ustiugov, as well as Leonid Kondrashov, from the vHive team, for welcoming me into their project, allowing me to integrate this work into their platform, and offering their support and expertise throughout the process.

I am also deeply thankful to my friends and colleagues Nyckollas Brandão and André Páscoa, for our brotherhood and friendship, for your constant support, and for standing by me all these years. From the first year of our bachelor's to the end of our master's, you've been there, and I certainly could not have achieved all my academic, professional and life success without you two.

Last but not least, I would like to thank my colleagues at Cloudflare, especially those from the Radar team. Their support and understanding allowed me to balance my professional responsibilities while completing my studies, and thanks to their guidance, I continue to grow both professionally and personally.

I want to express my profound gratitude and appreciation to all of you, and to the many other friends, colleagues, and family members I haven't mentioned by name. You have shaped who I am, and I owe much of my growth and achievements to your presence in my life.

To each and every one of you—obrigado.

Abstract

Cold start latency remains a critical challenge in serverless computing, significantly affecting application responsiveness. Function snapshotting has become the *de facto* technique to mitigate cold starts. However, snapshots are typically large, encompassing disk and memory state, and thus impose substantial overhead in terms of storage and distribution. These challenges necessitate efficient mechanisms for caching and orchestration of snapshots, to ensure scalability and elasticity.

This work presents a snapshot orchestration system for serverless clusters, designed to enable efficient sharing, restoration, and management of snapshots. Main contributions include the integration of remote storage for centralized snapshot management, and a local caching mechanism on worker nodes paired with a snapshot-aware scheduler to mitigate snapshot retrieval latency. The system is built on top of the vHive platform, leveraging Knative for serverless orchestration, Firecracker microVMs for lightweight virtualization, and containerd as the container runtime. It integrates `stargz`, a containerd remote snapshotter, to enable lazy loading of container images.

Experimental evaluation shows that remote snapshot orchestration reduces cold start latency by about 70% compared to a baseline without snapshots, lowering average initialization time from 2507 ms to 757 ms. Around 73% of this latency comes from fetching, highlighting the need for caching. Overall, combining remote snapshot orchestration with caching and intelligent scheduling effectively reduces cold start overhead while improving resource efficiency and scalability in serverless environments. This work is being integrated into vHive, an open-source project for serverless systems research.

Keywords

Serverless Computing; Cold Start Latency; Snapshot Orchestration; MicroVMs.

Resumo

A latência de *cold start* permanece um desafio crítico na computação *serverless*, afetando a capacidade de resposta e a experiência do utilizador. A criação de *snapshots* de funções é a técnica *de facto* para mitigar esse problema. Contudo, estas *snapshots* frequentemente contêm grandes volumes de dados em disco e memória, resultando em tamanhos que dificultam o armazenamento e a distribuição. Tais desafios exigem mecanismos eficientes de *caching* e orquestração que reduzam a sobrecarga e garantam escalabilidade.

Este trabalho apresenta um sistema de orquestração de *snapshots* para *clusters serverless* distribuídos, concebido para enfrentar esses desafios através da partilha, restauração e gestão eficientes de *function snapshots*. As principais contribuições incluem a integração de armazenamento remoto para gestão centralizada de *snapshots* e um mecanismo de *caching* local nos nós, aliado a um *scheduler* otimizado que reduz a latência de recuperação. Construído sobre a plataforma vHive, o sistema utiliza *Firecracker microVMs* e *containerd* como tecnologias de virtualização *lightweight*, propondo um *containerd snapshotter* remoto para suportar o *lazy loading* eficiente de imagens.

A avaliação experimental demonstra que a orquestração remota reduz a latência de *cold start* em cerca de 70%, baixando o tempo médio de 2507 ms para 757 ms. Aproximadamente 73% desta latência deve-se à transferência de dados, destacando a importância do *caching*. A combinação de orquestração remota, *caching* local e agendamento inteligente melhora significativamente a eficiência e escalabilidade em ambientes *serverless*. Este trabalho está a ser integrado no vHive, um projeto *open-source* para investigação em sistemas *serverless*.

Palavras Chave

Computação *Serverless*; Latência de *Cold Start*; Orquestração de *Snapshots*; *MicroVMs*.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

API	Application Program Interface
C/R	Checkpoint/Restore
CRD	Custom Resource Definition
CRI	Container Runtime Interface
DAG	Directed Acyclic Graph
E2E	End-to-End
eBPF	Extended Berkeley Packet Filter
FaaS	Function-as-a-Service
HTTP	Hypertext Transfer Protocol
HDD	Hard Disk Drive
IaaS	Infrastructure-as-a-Service
I/O	Input/Output
KVM	Kernel-based Virtual Machine
LRU	Least Recently Used
LLM	Large Language Model
OCI	Open Container Initiative
OS	Operating System
P2P	Peer-to-Peer
PaaS	Platform-as-a-Service
SSD	Solid-State Drive
UI	User Interface
VM	Virtual Machine
VMM	Virtual Machine Monitor

WASI WebAssembly System Interface

Wasm WebAssembly

1

Introduction

Contents

Serverless computing [4] has revolutionized the way applications are deployed and scaled, offering flexibility and cost efficiency by allowing developers to execute code without managing the underlying infrastructure. Function-as-a-Service (FaaS) is a core component of serverless computing, enabling developers to build and deploy individual functions that automatically scale in response to demand. Several cloud providers offer FaaS [5–9] as an interface for usage-based, stateless (serverless) backend services.

1.1 Motivation

Despite the advantages, prior work [10–12] noted that one significant challenge in serverless environments is the latency caused by cold starts, which occur when a function is invoked without any

pre-existing execution environment. This results in considerable delays as the system initializes the necessary resources, such as loading the function's code and initializing runtime environments.

Addressing the issue of cold starts is crucial for the performance and user experience of serverless applications, which are increasingly being adopted for a wide range of purposes, from web services to data processing. Reducing cold start latency can significantly enhance the responsiveness and scalability of these applications, leading to improved user satisfaction and more efficient resource utilization.

One solution to mitigate cold starts is function snapshotting [12–17], also known as Checkpoint/Restore (C/R), which involves saving and reusing the state of function execution environments. However, snapshots are typically large, encompassing disk and memory state, making their management and efficient distribution complex and resource-intensive.

The challenge of reducing cold start latency is inherently difficult due to the unpredictability of function invocations and the complexity of managing large snapshots. Naive approaches, such as keeping all functions in a warm state, fail due to their impracticality in a scalable environment. They lead to excessive resource consumption and increased operational costs, undermining the cost-efficiency that makes serverless computing attractive. Furthermore, managing and distributing snapshots across a distributed system involves significant overhead, including data storage, transfer, and synchronization, all of which must be carefully balanced to avoid introducing new performance bottlenecks.

Prior work has often focused on fixed keep-alive strategies [10, 18] or pre-warming methods [10, 11] that do not adequately account for the diverse and dynamic nature of serverless workloads. Moreover, snapshot-based techniques [12, 13, 15, 19–21] have demonstrated potential in reducing cold start latency but face significant challenges, including large snapshot sizes, data duplication, and the overhead of efficient distribution in distributed environments.

1.2 Goals and Proposed Solution

This work proposes enhanced platform support for function snapshots, enabling their efficient sharing across nodes in a distributed serverless environment. At its foundation, the approach leverages centralized remote storage to optimize accessibility and coordination. Additionally, local caching is used to mitigate fetching delays, ensuring faster access to frequently used snapshot data. This is complemented by a snapshot-aware scheduler that enhances efficiency by redirecting requests to worker nodes with the required snapshot already cached locally.

In sum, the main goals of this work are the following:

- **Remote snapshot storage and sharing:** Enable centralized management and distribution of snapshots to improve accessibility and coordination across the cluster;

- **Local caching:** Reduce repeated snapshot transmissions by maintaining cached copies on worker nodes, minimizing network overhead and improving performance;
- **Snapshot-aware scheduling:** Implement a scheduling policy that prioritizes nodes with cached snapshots to further reduce cold start latency.

By combining these techniques, the approach not only minimizes resource overhead but also ensures scalability and responsiveness across distributed systems.

Built upon the vHive[1] platform (described in Section 2.3), the proposed approach leverages Firecracker microVMs [22] in conjunction with containerd [2] containers. This combination provides lightweight virtualization and efficient container runtime management (see Section 2.2 for more details). To further enhance container startup performance, the solution integrates a remote containerd snapshotter, such as Stargz [23], which supports lazy pulling to enable rapid on-demand loading of container data.

1.3 Contributions

In summary, the main contributions are:

- Design a snapshot orchestration system that enables sharing and restoration of snapshots across cluster nodes in a distributed serverless environment;
- Employ a local caching mechanism on each worker node, complemented by a snapshot-aware scheduler that redirects requests to nodes with locally cached snapshots to mitigate fetching latency;
- Implementation and integration of the proposed system into vHive [12], a state-of-the-art open-source serverless research platform. The implementation leverages modern virtualization technologies such as `Firecracker` and `containerd`.
- Conduct a comprehensive evaluation to demonstrate the effectiveness of the proposed system in reducing cold start times and improving resource efficiency, while analyzing its limitations in dynamic and large-scale workloads.

1.4 Organization of the Document

This thesis is organized as follows: Chapter 2 provides the necessary background on serverless computing, function snapshotting, and virtualization technologies. Chapter 3 reviews prior research and state-of-the-art systems for mitigating cold start latency, highlighting their limitations. Chapter 4 presents the

[1] This work was done in close collaboration with Prof. Dmitrii Ustiugov and his team from Nanyang Technological University (NTU) Singapore, who currently leads the vHive project.

proposed solution, detailing the architecture and design of the snapshot orchestration system. Chapter 5 describes the implementation of the system. Chapter 6 presents an evaluation of the solution's effectiveness, supported by detailed experimental results. Finally, Chapter 7 concludes the document by summarizing the contributions and outlining directions for future work.

2

Background

Contents

This chapter provides an overview of key concepts and technologies relevant to the proposed solution, including serverless computing, virtualization techniques, and the vHive platform.

2.1 Serverless Computing

Serverless computing [4] is an increasingly popular cloud computing model for developing and deploying online services, offering flexibility and cost efficiency by allowing developers to execute code without managing underlying infrastructure. FaaS is a core component of serverless computing, enabling developers to build and deploy applications, in which the functionality is divided into one or more stateless, event-driven functions executed and managed by the provider. All major cloud providers support serverless deployments, including Amazon Lambda [5], Azure Functions [6], and Google Cloud Functions [7].

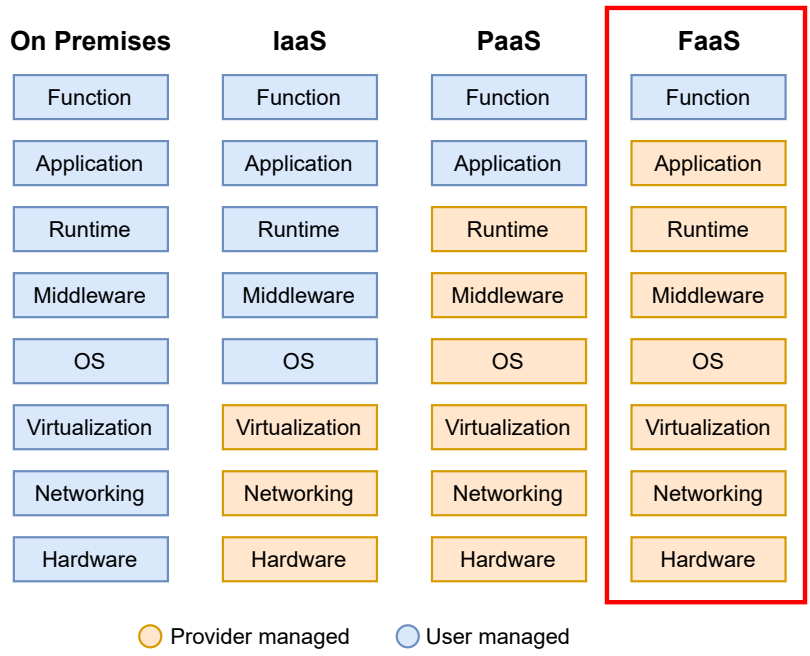

Figure 2.1: Cloud computing models and the division of management responsibilities. The figure compares On-Premises, IaaS, PaaS, and FaaS in terms of which layers are managed by the provider (orange) versus the user (blue). As abstraction increases from left to right, more infrastructure and platform responsibilities are shifted to the cloud provider, culminating in FaaS, where only the function logic is managed by the user.

Figure 2.1 illustrates the different levels of abstraction across cloud computing models, from on-premises deployments to Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and finally FaaS. The trend shows an increasing shift of responsibilities from the user to the provider. In traditional on-premises setups, the user manages all layers, including hardware, networking, and operating systems. With IaaS (e.g., Amazon EC2 [24], Google Compute Engine [25]), providers manage physical infrastructure, while users handle the operating system and above. PaaS (e.g., Google App Engine [26], Heroku [27]) further abstracts runtime and middleware, allowing developers to focus on application logic. At the highest level, FaaS (e.g., AWS Lambda [5], Azure Functions [6], Cloudflare Workers [9]) pushes almost all responsibilities—hardware, OS, middleware, runtime, and even parts of application management—to the provider, leaving the user responsible only for the function code.

This progression offers several advantages: it reduces operational complexity for developers, shortens time-to-market by eliminating infrastructure management, and improves scalability and elasticity since providers can automatically allocate resources. Additionally, it enables cost efficiency through fine-grained billing models, as users only pay for the execution time of functions rather than provisioning

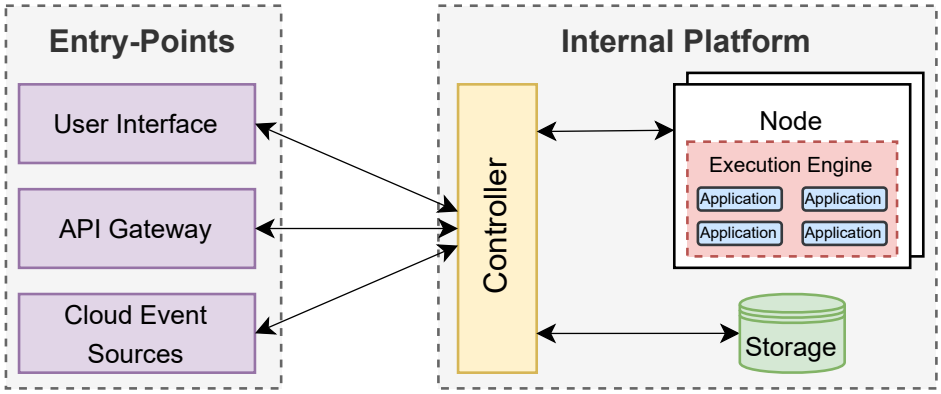

Figure 2.2: Typical FaaS platform architecture, highlighting key components: the frontend layer (entry points to the platform) and the internal platform, consisting of a controller, storage, and multiple nodes.

and maintaining servers.

2.1.1 Function-as-a-Service Architecture

The architecture of FaaS platforms revolves around the independent and event-driven execution of stateless functions. Developers deploy functions by registering them with the platform, specifying parameters such as trigger events and data bindings. These functions are typically grouped into applications, which serve as the primary units for resource allocation and scaling.

A typical FaaS system, illustrated in Figure 2.2, consists of several key components distributed across two main layers: the frontend layer and the internal layer. The frontend layer includes the system's entry points, such as the User Interface (UI), Application Program Interface (API) gateway, and cloud event sources, all of which interact with the controller located within the internal layer. The controller manages an event queue and includes a scheduler that consumes from the queue, directing incoming function requests to the appropriate nodes. Additionally, the controller functions as a scale controller and communicates with storage systems to manage function states and other necessary data. Each node is equipped with an execution engine responsible for running function instances within workers, with each worker isolated from others. These workers may represent Virtual Machines (VMs), containers, or processes that execute the code.

Each node hosts one or more application instances, which are isolated from one another using virtualization techniques, such as VMs or containers (see Section 2.2). This structure enables FaaS platforms to provide simple configuration with minimal user tuning. Additionally, the design emphasizes flexibility, supporting a wide range of use case, including API services, data processing, machine learning, and more.

2.1.2 Instance Orchestration

Scalability is a critical aspect of serverless computing, as it ensures that applications can efficiently handle varying workloads. In serverless systems, scalability is achieved through dynamic resource allocation, allowing the platform to automatically scale up or down based on demand.

Kubernetes

Kubernetes [28] is widely adopted as the orchestration layer in modern serverless environments. It manages clusters of machines and orchestrates containerized workloads, dynamically scaling resources in response to workload changes. At the core of Kubernetes' container management is containerd [2] (discussed further in Section 2.2.2), a container runtime responsible for pulling container images, creating and starting containers, and supervising their execution.

In Kubernetes, the smallest deployable unit is a pod, which encapsulates one or more containers that share storage, networking, and a specification for how to run the containers. Pods are ephemeral by design and are managed by higher-level abstractions such as Deployments, which control replication, updates, and autoscaling.

A central component of Kubernetes' orchestration capabilities is the Kubernetes scheduler [29], which determines where new pods should run in the cluster. The scheduler performs this task by first filtering out nodes that do not meet the pod's basic requirements, such as insufficient resources, mismatched node selectors, taints, or affinity rules. Once a set of viable nodes is identified, the scheduler scores them based on various heuristics, such as resource availability, workload distribution, and custom priorities. Finally, the scheduler binds the pod to the most suitable node by updating the Kubernetes control plane, which then triggers container creation and execution on the selected node.

Knative

Knative [30] is a Kubernetes-based platform designed to bring serverless capabilities to containerized environments. In Knative, serverless functions are deployed as pods and are typically managed through Knative Services, which abstract the complexity of configuration, autoscaling, and traffic routing. This allows developers to deploy stateless functions that scale seamlessly based on demand.

Knative enhances scalability by providing automatic, traffic-driven autoscaling. When a function experiences a high volume of incoming requests, Knative increases the number of active pod instances to handle the load. Conversely, when demand drops, it scales down the pods, including the ability to scale to zero, thereby conserving resources and reducing operational costs. This dynamic scaling ensures efficient resource utilization while maintaining performance, even under highly variable or bursty

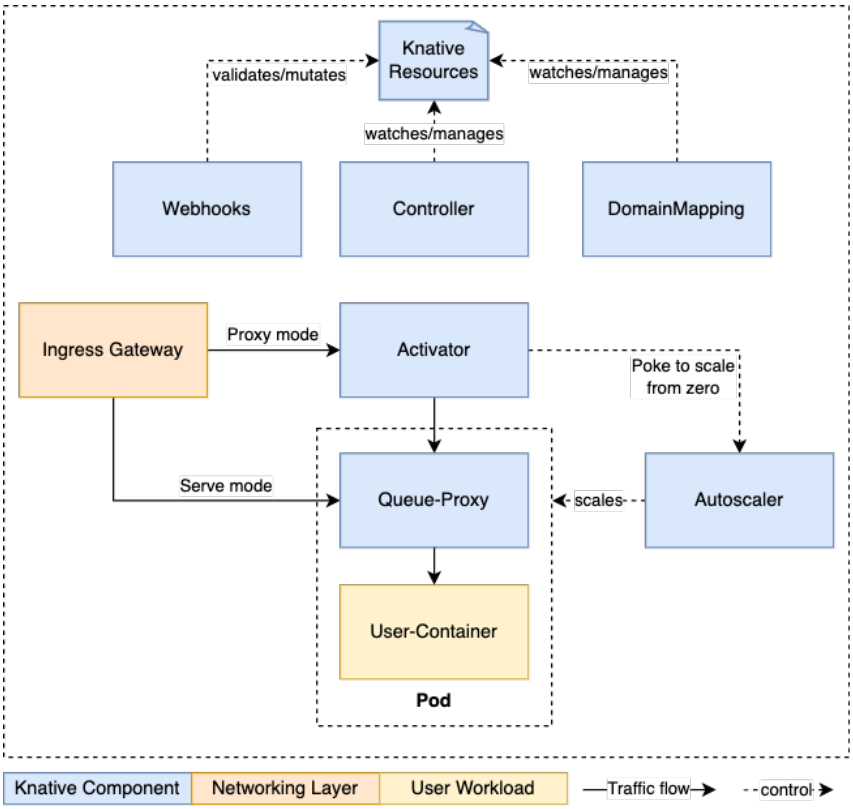

Figure 2.3: Knative Serving architecture and its core components. Source: Official Knative documentation [1].

workloads.

Figure 2.3 illustrates the architecture of Knative Serving and its key components:

- **Activator:** Handles requests when a service is scaled to zero, buffering them while the autoscaler provisions instances. It can also act as a request buffer during traffic bursts.
- **Autoscaler:** Scales Knative services up or down based on incoming traffic, resource usage, and configuration.
- **Controller:** Manages Knative resources, ensuring their desired state by handling lifecycle events and updates.
- **Queue-Proxy:** A sidecar container that enforces concurrency limits, collects metrics, and queues requests when necessary.

Together, Knative and Kubernetes provide a robust infrastructure for scaling serverless applications, allowing developers to focus on writing code while the platform automatically handles scaling and resource management.

2.1.3 The Cold Start Problem

Recent studies have focused on characterizing FaaS workloads, revealing key insights into the nature of serverless functions. For instance, Serverless in the Wild [10] provides a detailed analysis of real-world serverless workloads. It highlights that serverless functions are typically short-lived, averaging 670ms in duration, with 90% executing in less than 10 seconds. These functions are also invoked infrequently, with 80% being triggered less than once per minute. Additionally, the study shows that most functions consume small memory footprints, with over 90% using less than 300MB of virtual memory.

Similarly, ORION and the Three Rights [11] delves into the characteristics and optimization of serverless workloads, emphasizing the importance of efficient resource management. ORION's findings indicate that serverless functions are typically invoked less frequently than once per minute, with the majority being invoked at least once per week. Moreover, the study highlights a heavily skewed invocation pattern, where the top five most frequent Directed Acyclic Graphs (DAGs) account for 46% of all invocations. These frequently invoked DAGs benefit from optimized execution, resulting in significant cost savings. However, the study also reveals that 80% of DAGs are invoked fewer than 100 times per day, leading to a high percentage of cold starts. In contrast, DAGs with invocation frequencies of 100 or more times per day experience a much lower median cold start percentage of 0.35%.

A major challenge in serverless deployments is the memory occupation by idle function instances. To prevent inefficient memory use, providers typically limit the lifetime of function instances to 8-20 minutes after their last invocation [31]. This deallocation strategy, driven by the sporadic nature of function invocations, often leads to cold start delays, where the first invocation after a period of inactivity incurs significant start-up latency. Over the past few years, reducing cold start latencies has become a central problem in serverless computing and a key performance metric for evaluating serverless providers.

Cold start mitigation techniques

To reduce cold-start delays, researchers have proposed various techniques:

Runtime Recycling [10, 18, 32] In runtime recycling, also known as keep-alive technique, the function runtime is kept warm and ready for reuse after a function completes execution. This approach minimizes the need for repeated initialization. However, the downside is that keeping runtimes warm consumes memory and other resources, which can lead to higher operational costs and inefficient resource utilization, especially if the functions are invoked infrequently.

Pre-warming [10, 11] Pre-warming involves initializing and keeping a pool of function instances ready to handle incoming requests. By pre-warming, the system can reduce or eliminate cold-start latency

since the instances are already running and prepared to process requests. However, this method requires additional resource allocation and management, leading to higher operational costs and potential wastage of resources if the pre-warmed instances are not utilized effectively. Moreover, pre-warming is only beneficial for functions with predictable invocation patterns; new or highly irregular functions are difficult to anticipate and thus cannot easily leverage this technique.

Forking (Zygote) [33, 34] This technique involves forking an existing, running function instance to handle new incoming requests. By forking a hot runtime, the system can bypass the initialization phase, thereby reducing cold-start delays. The main disadvantage of forking a hot runtime is the potential for resource contention and increased complexity in managing multiple forked instances, which can lead to performance degradation and unpredictable behavior under heavy load.

Co-locating [35–39] Co-locating functions involves running multiple functions within the same runtime environment, as mentioned in the paragraph about memory isolates in Section 2.2. This approach leverages shared resources to reduce initialization overhead. The downside is the increased risk of interference and security issues, as multiple functions running in the same environment can affect each other's performance and potentially expose vulnerabilities.

Snapshotting [12, 13, 15, 19–21, 40] Snapshotting, or C/R, captures the state of a VM, including the Virtual Machine Monitor (VMM) and memory contents, storing this as files on disk. The host orchestrator can then quickly create a new function instance from these snapshots, ready to process incoming requests without high cold-start latency. As illustrated in Figure 2.4, the first execution involves snapshot creation, while subsequent executions can directly load and run the snapshot. Snapshots can reside in local storage (e.g., Solid-State Drive (SSD)) or be fetched from remote sources, providing flexibility in performance and resource usage. However, snapshotting introduces overhead in terms of storage capacity, potential duplication, and latency when fetching snapshots from remote locations. Another challenge lies in determining the optimal moment to take a snapshot [14, 41], as capturing too early or too late can affect performance.

This document focuses on snapshotting, as it provides a robust solution for reducing cold-start delays without requiring active memory allocation during function inactivity periods.

2.2 Virtualization for Serverless Platforms

Virtualization plays a critical role in serverless platforms by ensuring isolation, security, and efficient resource management across different workloads. Traditional solutions like VMs provide strong

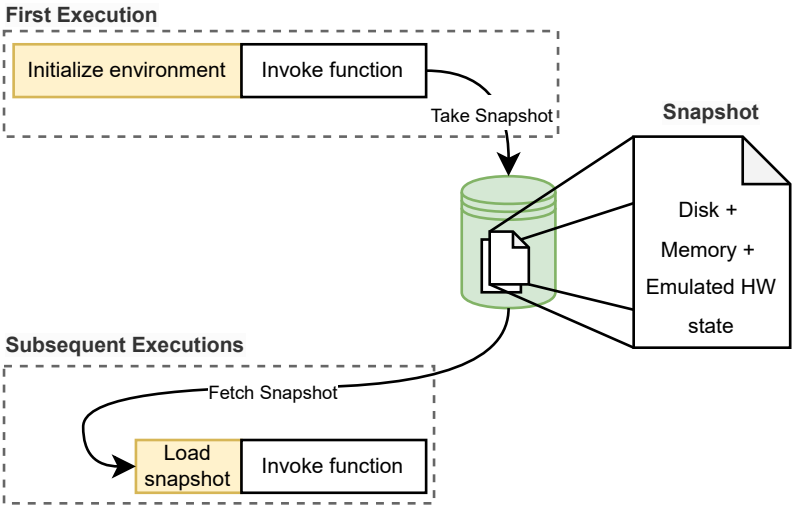

Figure 2.4: Snapshot orchestration in serverless platforms. The first execution captures the VM's memory, disk, and hardware state into a snapshot. Future invocations can bypass the full initialization by loading the saved snapshot, significantly reducing cold-start time.

isolation and enhanced security by running a full Operating System (OS) for each instance, but their heavyweight nature and long boot times make them less suited for the dynamic, elasticity demands of serverless architectures. Containers, in contrast, offer better performance and resource efficiency due to their lightweight nature, as they don't include a full OS; however, they share the host OS, leading to a wider attack surface and reduced security compared to VMs. To address the need for both performance and security in serverless environments, lightweight isolation solutions such as microVMs [22], unikernels [42], container sandboxes [43], and other virtualization technologies [38, 39, 44–46] have emerged. Figure 2.5 illustrates the architectural differences and layers of abstraction among some of these techniques.

Virtual Machines As discussed earlier, traditional VMs provide strong isolation and security guarantees by running a complete OS for each instance. This makes them well-suited for multi-tenant environments where isolation is critical. However, their heavyweight nature results in significant overhead, including large memory footprints and long boot times, which are less compatible with the elasticity and rapid scaling requirements of serverless workloads.

Containers Containers, also mentioned above, take a different approach by sharing the host OS kernel while isolating processes through namespaces and control groups. This design enables fast startup

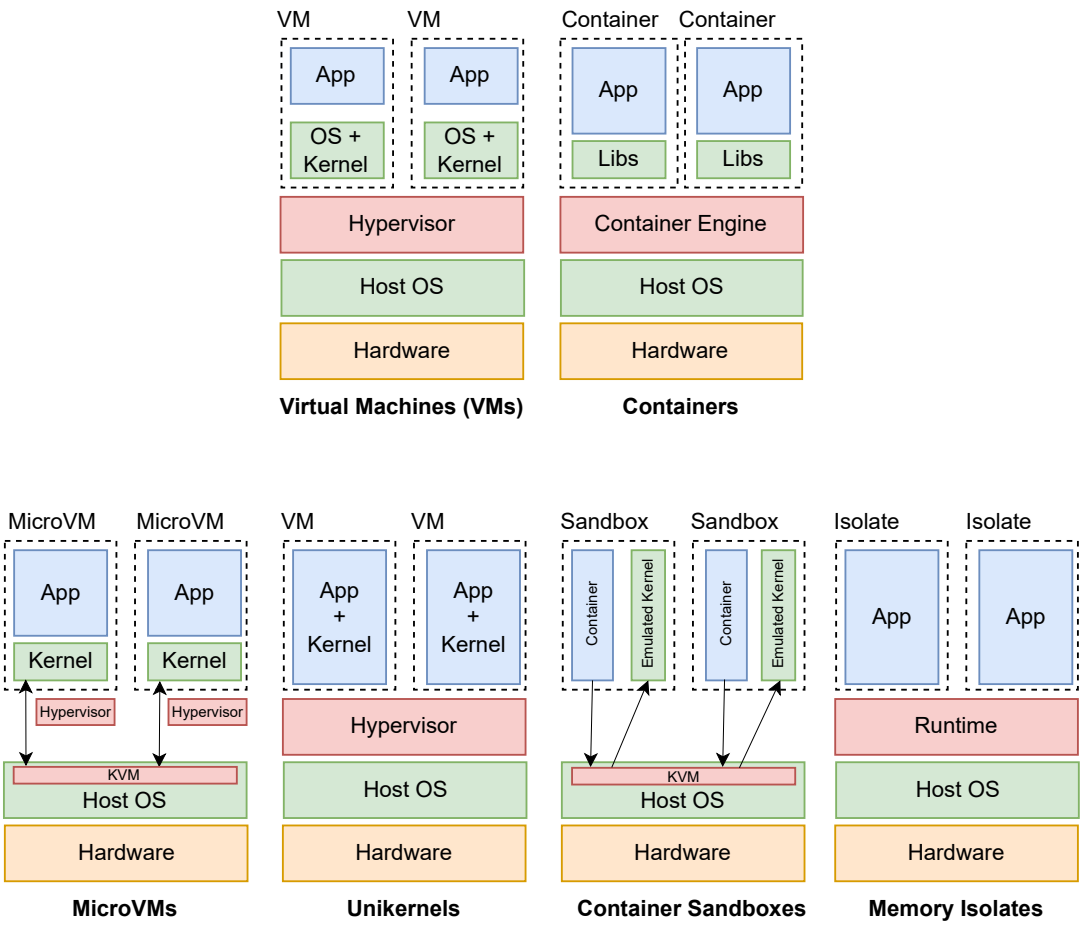

Figure 2.5: Comparison of virtualization techniques for lightweight isolation in serverless environments, including VMs, containers, microVMs, unikernels, container and memory sandboxes.

times and efficient resource usage, making containers a popular choice for microservices and cloud-native applications. Nonetheless, because they lack hardware-level isolation and share the host kernel, containers present a larger attack surface and weaker security guarantees compared to VMs. For more details on container management and runtime, see Section 2.2.2.

MicroVMs MicroVMs are lightweight virtualization solutions designed to deliver strong isolation with minimal resource overhead and fast startup times, making them ideal for serverless environments. They provide essential features such as hardware-level isolation, basic I/O virtualization (e.g., network and block devices), fast boot times, and a small memory footprint—often starting with only a few megabytes. Additionally, they support snapshots and leverage hardware virtualization extensions (Intel VT-x, AMD-V)

for performance and security.

However, microVMs deliberately omit non-essential components found in traditional hypervisors, such as full device emulation (e.g., GPUs, USB, audio), complex BIOS/UEFI firmware, legacy hardware support, and advanced management or hot-plug capabilities. By stripping down these features, microVMs reduce the attack surface, improve startup latency, and simplify resource management, while still ensuring strong isolation guarantees.

With this design philosophy, Amazon introduced Firecracker [22], a lightweight hypervisor (VMM) specifically built for serverless applications. Firecracker leverages the Kernel-based Virtual Machine (KVM) [47], reducing the emulation layer by supporting only essential device types and relying on the host OS for resource management. This approach achieves VM boot times as low as 125 milliseconds and a memory footprint of just 5 MB.

Unikernels Traditional operating systems are often large and complex, with the Linux kernel alone comprising around 28 million lines of code, over a third of the full OS. Despite the simplicity of many applications, the kernel remains a significant dependency, and most applications require only a fraction of its functionality. Unikernels address this by compiling applications together with only the necessary operating system components—such as minimal device drivers and essential libraries—into a single, immutable machine image. This approach eliminates unnecessary code, resulting in highly optimized, lightweight systems that improve both performance and security. Prominent examples include MirageOS [42] and Unikraft [48].

Container Sandboxes Container sandboxes, such as Google's gVisor [43], provide a lightweight virtualization environment focused on sandboxing and security. gVisor functions as a user-space kernel that intercepts system calls, acting as an intermediate layer for control and isolation without requiring a full virtual machine. This approach reduces the overhead typically associated with traditional VMs while still ensuring a secure execution environment for serverless functions.

Memory Isolates While VMs and containers provide isolation at the hardware and operating system levels, respectively, they often introduce overhead in terms of memory usage, communication, and startup time. To address these limitations, memory sandboxes—commonly known as isolates [44]—offer a lightweight alternative. Runtimes such as the V8 JavaScript engine [49] and GraalVM Native Image [38, 50] support isolates, which serve as execution sandboxes for running serverless functions. Unlike traditional VMs or containers, which require separate runtime instances for each function, isolates share a single runtime environment, enabling the concurrent execution of hundreds or thousands of functions. This model is widely adopted by edge providers, such as Cloudflare Workers [37]. While

isolates provide language-level isolation and are efficient in terms of resource utilization, they are less suited for multi-language support or access to native code.

In addition to the virtualization technologies mentioned above, emerging technologies like WebAssembly (Wasm) [45] and its extensions, such as WebAssembly System Interface (WASI) [51], are gaining traction as lightweight, portable alternatives for serverless computing. Wasm's low startup latency and minimal resource footprint make it ideal for edge and serverless platforms. Similarly, innovations like Extended Berkeley Packet Filter (eBPF) [46] offer lightweight kernel-level virtualization and enhanced security. These advancements further diversify the tools available for building efficient and scalable serverless architectures.

This document focuses on microVMs, specifically Firecracker, and containers managed with containerd, as they provide a lightweight yet powerful combination of strong isolation, efficient resource usage, and rapid startup times, making them well-suited for addressing the challenges of serverless computing.

2.2.1 Firecracker MicroVMs

Firecracker [22, 52] is a lightweight VMM developed by Amazon to cater specifically to the needs of serverless computing and containerized environments. It leverages the Linux KVM to provide strong isolation while maintaining performance close to that of containers. Unlike traditional hypervisors, Firecracker is designed with minimalism in mind, supporting only essential features required for cloud-native workloads.

Firecracker operates by launching microVMs—lightweight VMs stripped of unnecessary components, such as advanced device emulation and multi-CPU support. This approach reduces the boot time to as little as 125 milliseconds and ensures that each microVM has a memory footprint of approximately 5 MB. These optimizations make Firecracker ideal for use cases requiring rapid instantiation and efficient scaling, such as FaaS workloads and edge computing.

Function Snapshots

One key feature of Firecracker is its snapshot support [53]. Snapshots capture the entire state of a running microVM, including its memory and device states, enabling near-instantaneous creation of new instances by restoring from these snapshots. This capability is useful for addressing the cold start problem in serverless computing, where startup latency directly impacts user experience.

A Firecracker snapshot consists of multiple files that collectively represent the full microVM state:

- **Guest Memory**: A file containing the memory state of the guest;

- **Emulated Hardware State**: Files capturing the state of both the KVM and Firecracker's emulated hardware components.

If the microVM uses disk files, these are not automatically included in the snapshot and must be managed separately. Typically, disks can be reused across snapshots and shared as read-only when possible, reducing duplication and optimizing resource utilization.

When loading a snapshot, Firecracker employs a technique where, instead of fully loading the memory file into memory during the resume process, it creates a `MAP_PRIVATE` mapping of the memory file. This approach allows for runtime on-demand loading of memory pages. Any writes to memory during execution are directed to a copy-on-write anonymous memory mapping. While this strategy significantly accelerates snapshot loading times, it requires retaining the guest memory file for the entire lifetime of the resumed microVM.

Firecracker's design emphasizes security through strong workload isolation. Each microVM runs its own kernel and userspace, ensuring that workloads are fully isolated from each other. Additionally, Firecracker uses a jailer [54] process to constrain and sandbox microVMs, further reducing the attack surface and adhering to principles of least privilege.

2.2.2 Container Management and Runtime

Container images are the fundamental units of deployment in modern cloud-native environments. They package applications together with all necessary dependencies into a portable artifact, simplifying development, deployment, and scaling across heterogeneous environments.

The Open Container Initiative (OCI) [55] plays a central role in this ecosystem by defining open standards for container images and runtimes. These specifications ensure interoperability across platforms and runtimes, enabling container images to be consistently built, stored, and executed. For instance, the OCI Image Specification defines the format for container layers and metadata, while the Runtime Specification standardizes how containers are launched and managed. Popular registries like Docker Hub [56] follow these standards, ensuring compatibility with container runtimes such as Docker [57] and containerd [2].

containerd Architecture

containerd is an industry-standard, open-source container runtime designed to manage the full container lifecycle, from pulling images to executing containers and managing storage. While Docker originally bundled all these capabilities in a monolithic design, it now delegates low-level container management to containerd, which is optimized for efficiency and modularity. Higher-level platforms like Kubernetes

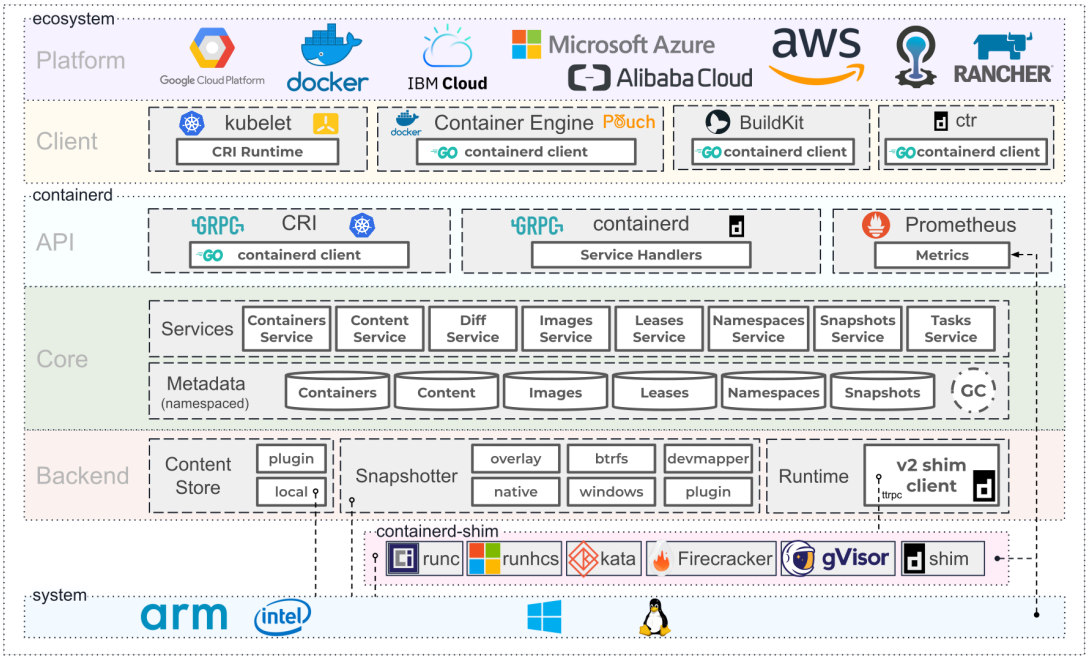

Figure 2.6: Overview of the containerd ecosystem architecture. Source: Official containerd documentation [2].

integrate with containerd through the Container Runtime Interface (CRI), while Docker and other engines rely on it as their core runtime.

Figure 2.6 illustrates the layered architecture of containerd:

- **Client Layer:** Exposes interfaces for orchestration tools (e.g., Kubernetes via CRI, Docker CLI) to interact with containerd.
- **API Layer:** Provides a gRPC-based API and service handlers for communication with clients and plugins.
- **Core Services:** Implements essential functionalities such as managing container metadata, images, snapshots, and tasks (running containers).
- **Backend:** Includes the *Content Store* for managing image data, *Snapshotters* for layered filesystem operations, and *Runtimes* (e.g., `runc`, Firecracker, gVisor) for container execution.

This modular design allows containerd to support a variety of snapshotters and runtimes, enabling flexibility in how containers are stored and executed. For instance, lightweight virtualization technologies like Firecracker can integrate seamlessly with containerd as a runtime plugin, providing enhanced isolation while preserving container-based workflows.

containerd Snapshotters

Snapshotters expose these layers to containerd using *mounts*, which describe how the layers should be attached to the filesystem. A mount is essentially a structured representation of the parameters that would be passed to the Linux `mount` syscall [58]—such as the source path, target path, filesystem type, and mount options—but in a serialized format that containerd can store and transmit. This abstraction allows containerd to apply the same interface regardless of the underlying filesystem type.

For example, when preparing a snapshot, the snapshotter creates an empty directory, associates it with metadata, and returns a *mount*. The directory is then populated with the layer's content by the Diff Service. For subsequent layers, the snapshotter clones the parent snapshot's content, applies changes from the new layer, and updates the cumulative filesystem view. This layered approach minimizes storage overhead and promotes resource sharing across containers.

containerd supports multiple snapshotters tailored to various workloads. One widely used snapshotter is overlayFS, the default snapshotter in containerd. OverlayFS employs a layered filesystem approach to minimize redundant storage and enhance performance. It functions similarly to Docker's "overlay2" driver, providing high efficiency and reliability in managing container layers.

Another snapshotter is the device mapper (`devmapper`), which is block-based and leverages the device mapper kernel framework to manage ext4 or xfs volumes. While historically significant in enabling container storage, `devmapper` has been deprecated due to concerns over performance limitations and maintainability [59].

Image pulling is often one of the most time-consuming steps in container startup, accounting for up to 76% of startup latency [60]. Traditional snapshotters require downloading all image layers from a registry before container startup, leading to delays, especially with large images. Remote snapshotters in containerd address this bottleneck by allowing snapshotters to reuse snapshots stored in remote shared locations. These snapshots, called *remote snapshots*, enable containers to start without pulling layers directly from registries. By preparing snapshots out-of-band and leveraging shared storage, remote snapshotters can significantly reduce the time required for image pulls.

One notable implementation is the Stargz snapshotter [23, 61]. Stargz introduces *lazy pulling*, allowing containers to begin execution without waiting for the full image to download. Instead, only the necessary portions of an image are fetched on demand, dramatically reducing startup latency. This functionality is achieved by combining containerd's remote snapshotter capabilities with the eStargz lazily-pullable image format. The stargz image format, originally developed by Google CRFS (Container Registry Filesystem) [62], enables efficient image access and is compatible with OCI-compliant registries. This makes Stargz a practical choice for cloud-native environments seeking to optimize startup times for large-scale workloads.

In summary, containerd snapshotters are pivotal for efficient image management. By leveraging

layered filesystems and remote fetching mechanisms, they optimize resource utilization, reduce redundancy, and enhance container startup times. These features are particularly crucial in large-scale serverless and cloud-native environments.

2.2.3 Bridging MicroVMs and Containers

Integrating containers inside microVMs combines the strengths of both technologies, offering the enhanced security and isolation of microVMs while preserving the flexibility and portability of containers. firecracker-containerd [63] enables this by combining the lightweight virtualization of Firecracker with the container orchestration capabilities of containerd, allowing developers to harness the benefits of both technologies.

In firecracker-containerd, each container runs inside a Firecracker microVM, achieving strong isolation without sacrificing the operational simplicity of container-based deployments. This architecture leverages containerd's snapshotting and runtime features while delegating workload execution to Firecracker microVMs.

Core Architecture

firecracker-containerd implements the V2 runtime API of containerd, which is responsible for configuring and running containerized processes. This architecture is built around several key components:

- **Control Plugin:** Manages the lifecycle of Firecracker microVMs and implements the control API. The plugin is compiled into a specialized containerd binary tailored for firecracker-containerd.
- **Runtime:** Connects containerd to Firecracker and serves as an intermediary for both VM lifecycle operations and container lifecycle operations. This runtime is implemented as an out-of-process shim communicating with Firecracker using ttRPC [64], a gRPC [65, 66] for low-memory environments [64].
- **Agent:** Runs inside the microVM and is responsible for executing commands, emitting events and metrics, and proxying STDIO for container processes. The agent uses runC [67] via containerd's shim to manage Linux containers within the microVM.

When a container is launched, the control plugin initializes a Firecracker microVM, pulls the required container image, and uses the runtime to start the container inside the microVM. This process ensures compatibility with the containerd ecosystem while maintaining the enhanced isolation provided by Firecracker. The microVM relies on two essential components: a **kernel image**, which contains the Linux kernel to boot the microVM, and a **root filesystem (rootfs)**, which serves as the base filesystem for

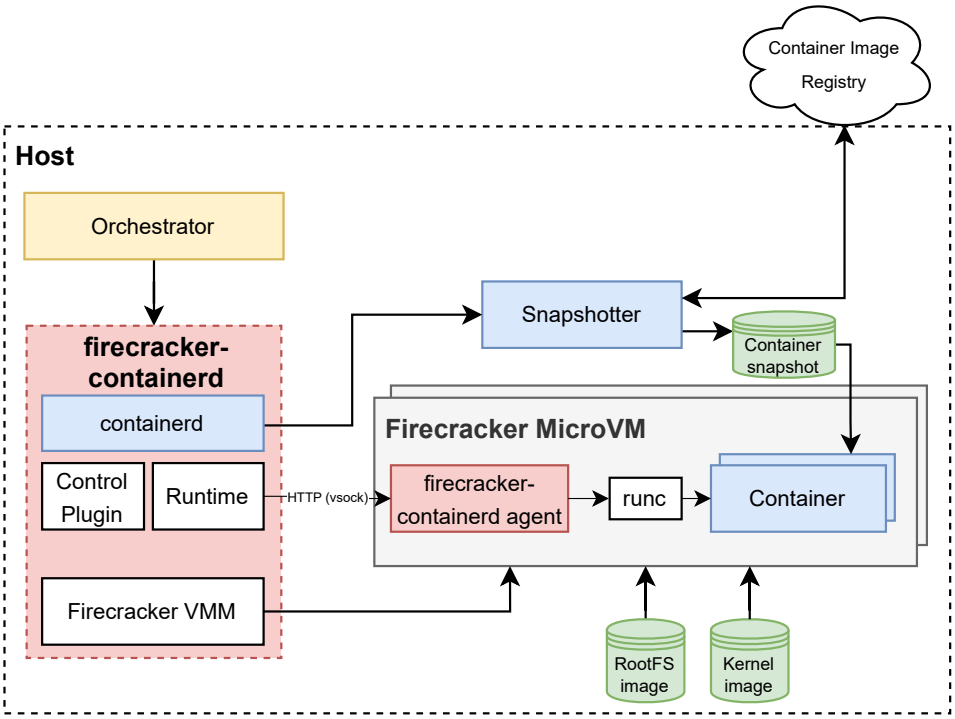

Figure 2.7: Architecture of firecracker-containerd. Adapted from the official firecracker-containerd documentation [3].

the virtual machine. These images are typically pre-provisioned on the host and can be obtained from official distributions or built from minimal Linux images. Figure 2.7 illustrates this architecture.

Support for Remote Snapshotters

firecracker-containerd also supports remote snapshotters, which allow containerd to reuse snapshots stored in a shared remote location rather than always pulling image layers from a registry and reconstructing them locally. Unlike local snapshotters, which require downloading and unpacking the entire image before the container can start, remote snapshotters can leverage pre-prepared snapshots or on-demand fetching, significantly reducing startup time and network overhead. A prominent example is the Stargz snapshotter, which enables *lazy image loading*: containers can start running as soon as the necessary parts of the image are fetched, while the rest of the data is pulled in the background.

However, Firecracker microVMs lack direct support for Virtio-FS, making it challenging to expose host filesystem-based mount points inside the microVM. To overcome this limitation, firecracker-containerd moves the entire snapshotter inside the microVM. This approach ensures that the lazy-loaded filesys-

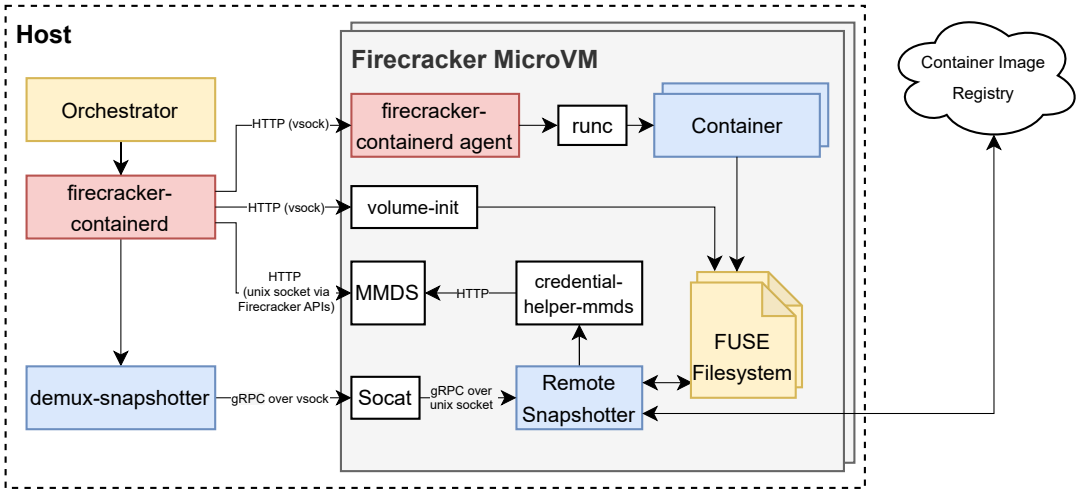

Figure 2.8: Architecture of firecracker-containerd with remote snapshotter support. Adapted from the official firecracker-containerd documentation [3].

tem operates within the guest and can be directly mounted as a container root filesystem. Figure 2.8 illustrates this extended architecture.

The extended architecture introduces new components to enable remote snapshotting:

- **Remote Snapshotter:** Each microVM hosts its own instance of a remote snapshotter to handle lazy loading of container images. This differs from the traditional approach where a single snapshotter runs on the host;
- **Demux-Snapshotter:** With a dedicated snapshotter in each microVM, a host-side proxy is required to manage interactions. The demux-snapshotter acts as this proxy, presenting itself as a single snapshotter to containerd, routing snapshot requests to the appropriate microVM;
- **Socat:** Exposes the remote snapshotter's socket inside the microVM as a vsock service, making it accessible from the host;
- **Credential Helper:** Uses Firecracker's microVM Metadata Service (MMDS) [68] to provide credentials for pulling container images from remote registries.

With these innovations, firecracker-containerd supports modern remote snapshotting mechanisms, ensuring fast container startups, efficient resource usage, and compatibility with cloud-native environments.

In conclusion, by combining Firecracker's strong isolation and fast startup capabilities with containerd's portability and compatibility with the broader container ecosystem, firecracker-containerd offers a robust solution for serverless and cloud-native environments.

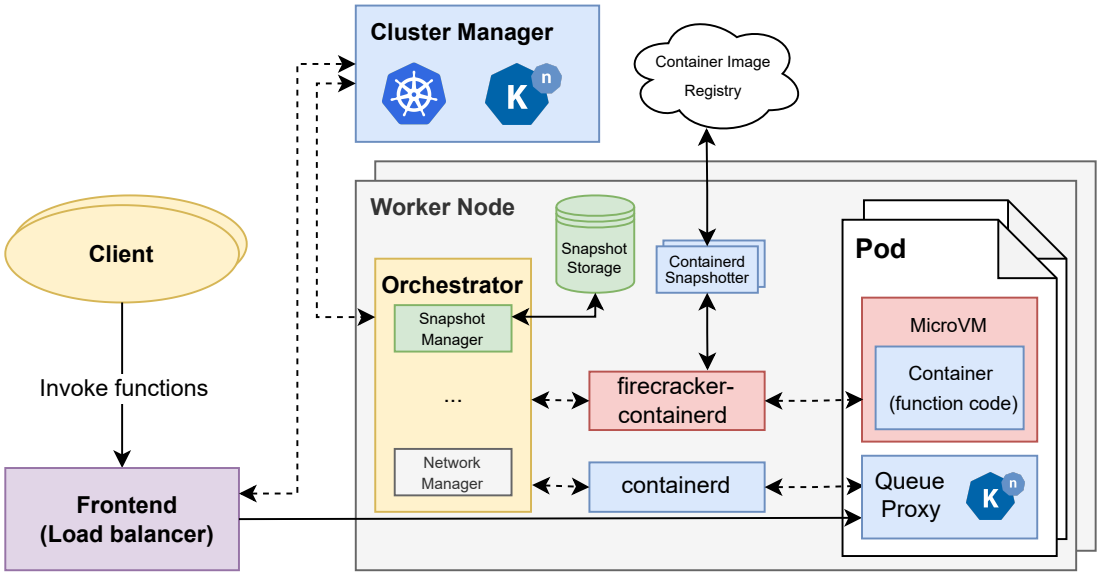

Figure 2.9: Overview of the vHive architecture. Solid arrows represent the data plane, while dashed arrows indicate the control plane.

2.3 vHive: A Framework for Serverless Experimentation

vHive [12, 69] is an open-source ecosystem for research and innovation in serverless cloud systems. It provides a full-stack framework for benchmarking and development in serverless computing environments. Figure 2.9 illustrates the architecture of vHive.

The vHive platform is designed to closely mirror real-world serverless cloud environments. Clients initiate function invocations as Hypertext Transfer Protocol (HTTP) requests received by frontend servers, which routes them to functions running in Kubernetes-managed containers. The architecture consists of a master node, which serves as the Kubernetes cluster master, and several worker nodes forming the cluster.

vHive employs Knative (see Section 2.1.2) to handle function autoscaling based on invocation traffic. When a function receives an invocation and has active instances, the front-end server routes the request to one of these instances. If no active instance exists, the load balancer contacts the cluster manager, and the Knative autoscaler selects a worker node to launch a new instance of the function. This instance is created as a pod, the Kubernetes's scaling unit, which contains a Firecracker microVM (as detailed in Section 2.2.1), running a container with the function code. This setup allows container images to be used as application packages while leveraging Firecracker for strong isolation. Additionally, each pod includes a Knative queue proxy that manages incoming requests.

To implement the control plane, vHive implements a CRI orchestrator service that integrates two containerd (see Section 2.2.2) instances: the standard containerd distribution (responsible for managing regular containerized workloads) and a Firecracker-specific fork, firecracker-containerd (discussed in Section 2.2.3) for managing microVM-based workloads. This orchestrator processes requests from the Kubernetes control plane, invoking the appropriate containerd services to manage containers and microVMs. It also oversees VM networking, the network manager, and snapshot management through a snapshot manager.

vHive supports Firecracker snapshots, allowing the state of a microVM to be saved and quickly restored. However, vHive currently only supports local snapshots—that is, snapshots stored on each node individually. This setup requires each node to maintain its own copies, leading to duplicated data and higher storage costs, which limits scalability as nodes must allocate significant storage for redundant snapshot data. Furthermore, the existing scheduler is oblivious to the location of these local snapshots, resulting in unnecessary snapshot transfers across nodes and additional latency. These challenges illustrate the need for a more scalable and resilient solution that enables remote snapshot orchestration combined with location-aware scheduling to support efficient serverless function deployment across distributed clusters.

3

State-of-the-Art

Contents

There is a fast-growing body of literature on serverless computing, particularly on FaaS platforms. This chapter reviews recent and relevant works on cold start mitigation and resource orchestration, which are closely related to this thesis. The papers are presented chronologically within each category to highlight the evolution of approaches. While works are categorized by primary technique, many combine pre-warming, caching, co-location, and snapshotting methods. Additionally, these solutions are compared and analyzed to identify their opportunities and limitations concerning the research goals.

3.1 Cold Start Mitigation

Several strategies have been proposed to mitigate cold starts, including pre-warming, caching, co-location, and snapshotting.

3.1.1 Pre-Warming Strategies

Serverless in the Wild [10] introduces a hybrid pre-warming policy that adapts to the invocation patterns and frequencies of individual applications. The method involves dynamically adjusting pre-warming and keep-alive windows based on the application's idle times (ITs), using a range-limited histogram to track IT distributions. Additionally, the policy includes fallback mechanisms such as standard keep-alive and time-series analysis for cases where the histogram is not representative. By tailoring pre-warming to specific application behavior, this approach effectively balances cold-start latency with resource utilization. Experimental results showed that the hybrid policy reduced cold starts by 32.5% on average and memory consumption by 15.6%, compared to a standard 10-minute fixed keep-alive policy, while adding only 835.7 μs of latency to the system.

ORION [11] mitigates cold starts by leveraging the DAG structure of serverless applications. The technique involves identifying optimal pre-warming delays for VMs associated with different stages of a DAG. By balancing End-to-End (E2E) latency and resource utilization, ORION minimizes the time taken to start the VMs just before they are needed while avoiding unnecessary resource consumption. Experimental results show that ORION's dynamic adjustment of pre-warming delays results in significant performance improvements, achieving up to 42% lower E2E latency and 57% lower costs compared to alternatives. The system consistently maintains efficient performance throughout the execution of complex, multi-stage serverless workflows, effectively mitigating the impact of cold starts without sacrificing resource utilization.

IceBreaker [70] proposes a heterogeneity-aware pre-warming strategy that uses both costly and cheaper nodes to improve latency and cost efficiency in serverless platforms. Unlike traditional approaches that keep functions warm only on expensive nodes, IceBreaker selects the most cost-effective node type based on invocation probability. This allows more functions to remain warm within the same budget, reducing cold starts. Evaluations with real workloads show a 45% reduction in keep-alive cost and a 27% decrease in execution time, demonstrating that heterogeneous infrastructures can improve both performance and cost.

While pre-warming strategies, such as those used in Serverless in the Wild and ORION, have demonstrated significant improvements in reducing cold-start latency, these approaches are not without limitations. They require maintaining idle resources, which increases memory and compute consumption, especially in systems with low or unpredictable invocation patterns. Additionally, these methods rely on predictable workloads, making them less effective for irregular or infrequent usage and potentially leading to resource wastage or operational overhead. Balancing latency reduction with resource efficiency remains a key challenge.

3.1.2 Caching and Forking Techniques

Alongside pre-warming, several systems utilize caching and forking techniques to enhance function initialization. Zygote [33], originally developed for Android, caches pre-initialized Java applications, improving the startup times of new instances.

SOCK [34] tackles cold-start latency by introducing serverless-optimized containers that enable rapid function provisioning. It adopts a generalized Zygote strategy—pre-initializing Python interpreters—to bypass redundant library loading, and streamlines container startup by replacing heavyweight kernel features with lightweight isolation techniques. Combined with a three-tier caching system, SOCK achieves up to $45\times$ faster cold-starts and significantly lowers overhead in serverless platforms.

FaasCache [18] employs a caching-inspired Greedy-Dual [71] keep-alive policy to prioritize functions based on initialization overhead, invocation frequency, and resource footprint. By optimizing the eviction of containers with the lowest priority, FaasCache reduces cold-start latency by $3\times$, doubles the number of warm function invocations, and improves application latency by $6\times$. Additionally, its resource provisioning techniques decrease memory usage by 30%, offering scalability for dynamic workloads.

Flame [72] advances the caching paradigm by introducing a centralized cache controller for serverless computing. Unlike prior systems that rely on decentralized, local cache policies, Flame employs a global `CacheManager` with visibility into the entire cluster, enabling hotspot-aware caching (identifies frequently-invoked functions using an exponentially-decaying scoring model) and adaptive instance placement (detects redundancy and dynamically reclaims over-provisioned hotspot instances). Each server runs a lightweight agent (`Cachelet`) that synchronizes state with the controller, allowing dynamic, fine-grained cache decisions. By classifying functions as "hotspot" or "non-hotspot" based on exponentially decaying invocation scores, Flame allocates protected memory for high-frequency functions while opportunistically caching less-used ones. This hybrid caching strategy, combined with a redundancy reclamation algorithm, reduces cold-start ratio by $7\times$ and cuts cache memory usage by 36% compared to state-of-the-art methods like FaasCache.

RainbowCake [32] introduces a layer-wise container caching and sharing approach. It decouples container initialization into three stages (Bare, Lang, and User layers), enabling finer-grained caching and sharing strategies. By leveraging invocation history to make sharing-aware, real-time caching decisions, RainbowCake reduces cold-start latency by 68% and memory waste by 77% compared to state-of-the-art solutions.

However, caching solutions also face challenges. They often consume additional memory to maintain pre-warmed resources, and their effectiveness can be diminished when workloads are highly variable or infrequent. Therefore, a more scalable and resource-efficient approach is needed to address the cold-start latency in diverse serverless environments.

3.1.3 Co-locating Functions

Co-locating functions involves running multiple functions within the same runtime environment, allowing sharing of resources.

SAND [36] enhances serverless performance by introducing application-level sandboxing and a hierarchical message bus. Unlike traditional models where each function runs in an isolated container, SAND allows multiple functions of the same application to share a container while maintaining process-level isolation. This design reduces cold-start latency and improves resource efficiency. Additionally, its hierarchical message bus optimizes communication for intra-application calls, resulting in up to 43% performance gains over systems like OpenWhisk.

Faasm [35] introduces Faaslets, a lightweight isolation abstraction using WebAssembly for memory isolation and efficient state sharing. Faaslets enable in-memory state sharing across co-located functions while maintaining isolation through software fault isolation (SFI). Faasm reduces cold-start latency to microseconds, achieves a 2× speed-up in machine learning training with 10× less memory usage, and doubles throughput for inference workloads while cutting tail latency by 90%. Faasm's two-tier state architecture co-locates functions with their required data, minimizing network transfers and resource overheads, making it a highly efficient approach for stateful serverless computing.

Photons [39] introduces an ultra-lightweight execution context for serverless functions, enabling safe co-location of multiple concurrent invocations of the same function within a shared runtime. By virtualizing both the language runtime and application state, Photons allows functions to share common components like libraries and datasets while maintaining strict data isolation through per-invocation state separation. Implemented atop OpenWhisk, Photons reduces per-invocation memory usage by 25% to 98%, lowers cluster-wide memory utilization by 30%, and cuts cold starts by 52%, all without degrading performance.

Nightcore [73] focuses on co-locating latency-sensitive microservices within the same runtime while maintaining low overhead. It rethinks scheduling, communication, and I/O threading models to minimize isolation costs while supporting multiple languages (C/C++, Go, Node.js, Python). Nightcore achieves 1.36× to 2.93× higher throughput and reduces tail latency by up to 69%, demonstrating that efficient co-location can make serverless viable for interactive microservices requiring sub-millisecond responsiveness.

Hydra [74] extends co-location by introducing a virtualized, multi-language runtime designed for high-density serverless platforms. Unlike prior systems that rely on bloated virtualization stacks or target single-language runtimes, Hydra consolidates multiple sandboxes within a shared process, enabling concurrent execution of functions written in different languages. To optimize performance, Hydra features a caching layer of pre-allocated instances that eliminates cold starts, and a snapshotting mechanism to checkpoint and restore individual sandboxes. Evaluation results show that Hydra improves function

density (ops/GB-sec) by 2.41× on average compared to OpenWhisk and by 1.43× compared to Knative. When replaying the Azure Functions trace, Hydra reduces memory footprint by 21.3–43.9% relative to OpenWhisk and by 14.5–30% compared to Knative. Most notably, Hydra eliminates cold starts entirely, reducing p99 latency by 45.3–375.5× over OpenWhisk and by 1.9–51.4× over Knative.

The downside of function co-location is the increased risk of interference and security issues, as multiple functions running in the same environment can affect each other's performance and potentially expose vulnerabilities.

3.1.4 Snapshotting

Snapshotting mechanisms have gained traction as a method to optimize serverless performance by reducing the time required to load functions. These methods allow for the preservation of the runtime environment in a snapshot, which can be quickly restored during execution. By doing so, snapshots bypass both runtime and function initialization, eliminating the need to re-create execution environments from scratch and significantly reducing cold start latency.

SEUSS [15] explores the use of unikernel snapshots to enable rapid function deployment. The key technique, called Snapshot Stacks, creates a lineage of snapshots that captures differences between states over time, enhancing memory efficiency. This method uses copy-on-write semantics to only capture modified pages, reducing storage requirements. Additionally, Anticipatory Optimizations (AO) are applied to pre-warm the system, reducing startup and execution times of functions. This work builds on previous research in process-level checkpointing, but uniquely applies it to a serverless environment using unikernels.

Similarly, Catalyzer [13] focuses on minimizing snapshot restoration times in the context of gVisor [43] virtualization technology. It uses lazy paging to only load memory when needed, and a novel primitive called `sfork` to directly reuse the state of a running sandbox. Evaluation results show that Catalyzer can achieve startup latencies as low as 0.97ms and reduce end-to-end latency by up to 67x, showcasing its effectiveness across various real-world serverless applications.

Another innovative approach to reducing cold-start delays in serverless computing, is the REAP (Record-and-Prefetch) [12] method. REAP operates in two phases: the Record phase, where it traces and records page faults during the initial invocation of a function, and the Prefetch phase, where it preloads the necessary memory pages for subsequent invocations, significantly reducing latency. By leveraging the Linux `userfaultfd` mechanism, REAP efficiently handles page faults in userspace, achieving 1.04 to 9.7 times faster invocations compared to baseline snapshots. This method is particularly effective for functions with frequent invocations, making it a valuable addition to the optimization techniques in serverless environments.

FaaSnap [19] builds on these ideas by introducing a snapshot-based VM restoration approach specif-

ically tailored for serverless platforms. By employing optimizations such as concurrent paging, per-region memory mapping, and hierarchical memory groups, FaaSnap significantly reduces cold-start latency and disk access overhead. These optimizations enable faster snapshot restoration by prefetching compact working sets and handling page faults more effectively. FaaSnap achieves up to a 3.5× speedup over systems like REAP [12] and exhibits performance on par with memory-cached snapshots, making it a scalable and efficient solution for serverless workloads

While these systems focus on VM-based snapshots, there is also growing interest in snapshot-based techniques in the context of deep learning, particularly with Large Language Models (LLMs). ServerlessLLM [20] adapts snapshot mechanisms for serverless inference tasks, reducing cold-start latency in large-scale AI systems by utilizing efficient checkpoint loading and scheduling techniques. As a result, it significantly reduces the latency overheads associated with cold starts, demonstrating a 10 to 200 times improvement over existing systems in various LLM inference workloads.

These snapshot-based solutions, while effective, often focus on specific use cases or environments. A comprehensive solution that integrates snapshot orchestration with efficient resource management and distribution is still lacking in the broader serverless computing space.

3.2 Resource and Storage Management

Efficient resource and storage management are critical for optimizing serverless platforms, particularly to reduce cold-start latency and support elastic provisioning. In systems that rely on frequent snapshotting of microVMs or containers, techniques such as deduplication and compression play a key role.

FAASNET [75] addresses inefficiencies in provisioning custom containers by using a function tree structure and an I/O-efficient fetching mechanism. This system reduces provisioning times and scales effectively, making it a promising solution for handling bursty workloads. The adaptive function tree structure and efficient Input/Output (I/O) mechanisms employed by FAASNET offer promising strategies for optimizing snapshot distribution processes. However, implementing these strategies would necessitate an additional communication layer between nodes to ensure rapid and scalable data handling.

Another prominent technique in snapshot management is block-level deduplication, which optimizes storage and retrieval by breaking data into smaller blocks, identifying duplicate blocks, and storing only the unique ones. AWS Lambda has implemented this technique for on-demand container image loading [76]. Their findings indicate that approximately 80% of newly uploaded Lambda functions have zero unique chunks, demonstrating high deduplication efficiency. For the remaining 20%, the average upload contains only 4.3% unique chunks. This method has been shown to significantly reduce storage costs and improve cache effectiveness, with potential reductions in storage by up to 23x.

Efficient snapshot management is a key factor in optimizing serverless environments. In this context,

SnapStore [16] introduces a memory-region-aware snapshot deduplication and retrieval system that enhances storage efficiency. By dividing snapshots into distinct memory regions—such as non-runtime, read-only file-backed, and write-modified—SnapStore applies a region-based deduplication strategy that achieves significant reductions in both deduplication time (46% on Hard Disk Drives (HDDs)) and retrieval latency (82.6% on HDDs). Moreover, SnapStore improves serverless function end-to-end latency by 25.9% and achieves a 2.4× reduction in storage requirements compared to conventional systems like FaaSnap [19], making it highly effective in environments with large numbers of function snapshots.

Traditional software-based compression algorithms, though effective, often introduce considerable CPU overhead during snapshotting and restoration processes—posing a bottleneck in latency-sensitive serverless applications. Recent advances in hardware-accelerated compression have demonstrated promising improvements in this area. For instance, Sabre [17] introduces a hardware-assisted snapshot compression framework tailored for microVMs in serverless platforms. By leveraging Intel's In-Memory Analytics Accelerator (IAA), Sabre achieves up to 4.5× compression ratios and speeds up decompression by up to 10× compared to software approaches, all without imposing additional CPU load. This allows for significantly faster microVM restorations and reduces end-to-end cold start times by approximately 20%, highlighting the benefits of offloading snapshot processing to specialized hardware.

3.3 Discussion

The current landscape of serverless cold-start mitigation and snapshot orchestration techniques highlights various strengths and weaknesses:

- **Pre-warming strategies:** These demonstrate effective latency reduction for predictable workloads, but struggle with resource inefficiencies and fail to adapt to irregular or bursty invocation patterns. These limitations underscore the need for solutions that can dynamically respond to unpredictable workloads without incurring high resource costs.

- **Colocation strategies:** Placing related functions or services on the same node can reduce data transfer and startup latency. The downside of function co-location is the increased risk of interference and security issues, as multiple functions running in the same environment can affect each other's performance and potentially expose vulnerabilities.

- **Snapshotting mechanisms:** Current snapshotting mechanisms show significant promise in reducing cold-start latency. However, these systems are often limited by their focus on isolated environments or the complexity of managing and distributing large snapshots efficiently across a cluster.

- **Resource management solutions:** Approaches such as FAASNET offer valuable strategies for improving provisioning and memory management. However, they primarily focus on optimizing specific aspects of the serverless pipeline, such as container fetching or memory preloading, without providing a holistic approach to snapshot orchestration and distribution.

In summary, while each of these works addresses different aspects of the cold-start problem, none fully resolves the need for a distributed snapshot orchestration system that integrates remote storage, local caching of snapshots, and snapshot-aware scheduling to enable scalable and resource-efficient cold-start mitigation. Although existing approaches offer valuable insights and partial solutions, they fall short of achieving the comprehensive objectives outlined in this proposal. This gap underscores the novelty and significance of the proposed solution, which aims to unify these capabilities into a cohesive framework to effectively address the challenges of serverless computing at scale.

4

Solution Architecture

Contents

This chapter outlines the proposed solution for a serverless snapshot orchestration system. The system's components are described, along with an explanation of how they interact to meet the defined requirements.

4.1 Architecture Overview

The proposed system builds upon vHive, the serverless platform described in Section 2.3, which provides a robust foundation for managing microVMs with containers using Firecracker and containerd. Figure 4.1 highlights the new components introduced in the design, contrasting them with the original vHive architecture.

To enhance vHive's snapshot management capabilities, the proposed solution extends its snapshot manager module to support both local and remote snapshots. While maintaining compatibility with fully

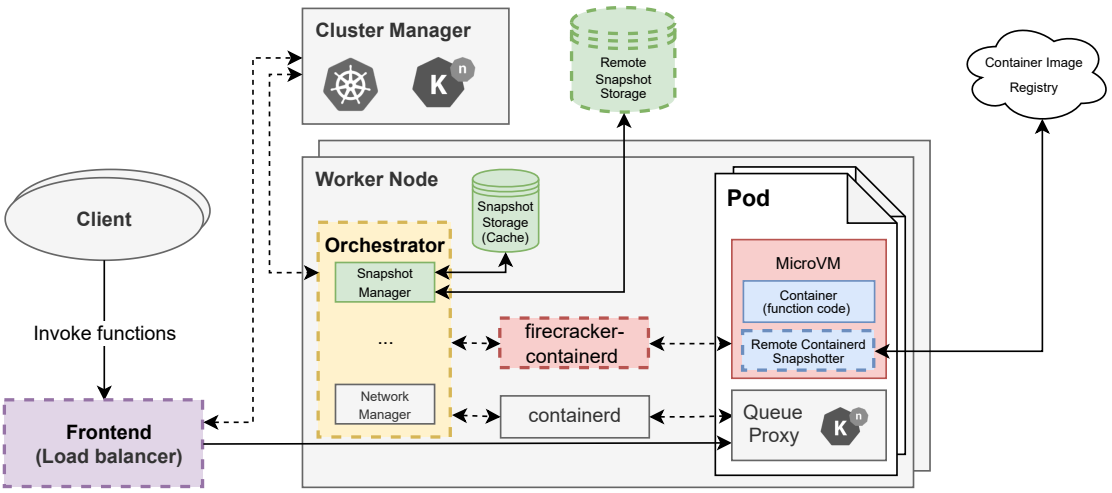

Figure 4.1: System Architecture: Extensions to vHive for Remote Snapshot Management. Grayed components represent unmodified elements from the original vHive platform, while colored and dashed components highlight the additions and modifications introduced by this solution, respectively.

local snapshots, the solution introduces the option to store snapshots in a remote centralized storage system (e.g., Amazon S3 [77]). This remote storage capability reduces cold start latencies in distributed environments by avoiding redundant snapshot creation and enabling reuse.

Since vHive leverages firecracker-containerd to package and run applications within microVMs, the system integrates a remote containerd snapshotter, such as Stargz [61] or Nydus snapshotter [78], to enable lazy loading of container images. This approach reduces container image fetch times by loading only the image parts required at launch, thus improving startup performance at scale.

To further reduce snapshot retrieval latency, a snapshot-aware scheduler is introduced. This scheduler is co-designed with a local snapshot cache on each worker node and makes placement decisions by considering whether a snapshot is already cached locally. By preferring nodes with cached snapshots, the scheduler increases cache hit rates, reduces access latency from remote storage, and accelerates microVM cold starts.

To manage local snapshot caches efficiently, each worker node maintains a bounded-size cache with an Least Recently Used (LRU)-based eviction policy. The snapshot manager monitors cache usage, estimates snapshot sizes before storing them, and evicts the least recently used snapshots when space is insufficient.

In summary, the key components added to the architecture are:

- **Remote Snapshot Storage**: A remote centralized storage solution that enables snapshot reuse across the cluster.
- **Extension of Snapshot Manager:** An extension to support both local and remote snapshots,

including a bounded local snapshot cache with LRU-based eviction.

- **Remote containerd Snapshotter:** Integration of a remote containerd snapshotter for lazy loading of container images, reducing fetch times.
- **Snapshot-Aware Scheduler:** A scheduler paired with a local snapshot cache on each node, optimizing node placement decisions to reduce access latency from remote storage.

Together, these enhancements build upon the existing vHive framework to deliver a scalable, low-latency snapshot orchestration system for serverless environments, introducing innovations in snapshot management, storage optimization, and scheduling intelligence.

4.2 Snapshot Lifecycle

Snapshotting involves two main phases: *snapshot creation* and *snapshot restoration*. Figure 4.2 illustrates the entire snapshotting workflow, showing a sequence diagram that covers both initial execution and subsequent executions.

4.2.1 Snapshot Creation

During the first execution of a serverless function, the orchestrator on the worker node initializes a *fresh VM* to run the function. This requires a Linux kernel and a root filesystem. The kernel is usually obtained from official distribution repositories, while the root filesystem can be prepared in advance from a minimal distribution image. A common approach is to download an official rootfs tarball (e.g., `ubuntu-base` [79]), extract it into a directory, and then convert it into an `ext4` image suitable for the VM.

After the VM is booted and the function invocation has completed, instead of immediately shutting down the instance, a snapshot is created to preserve the warm state of the execution environment. This snapshot captures both the VM state and the filesystem state. Once created, the snapshot is uploaded to remote storage, allowing it to be reused by other nodes and eliminating the need for redundant environment setups in future invocations.

The detailed steps of this process are shown in Algorithm 4.1. First, the VM is paused using the Firecracker API to ensure a consistent state. Then, another Firecracker API call occurs to generate the memory and microVM state snapshot files. Next, the container's filesystem state is captured, either manually or automatically depending on the snapshotter being used. Afterward, metadata describing the snapshot (e.g., file paths, revision number) is serialized into an info file. Finally, all these files are uploaded to a remote storage system.

In summary, a complete snapshot is composed of four files:

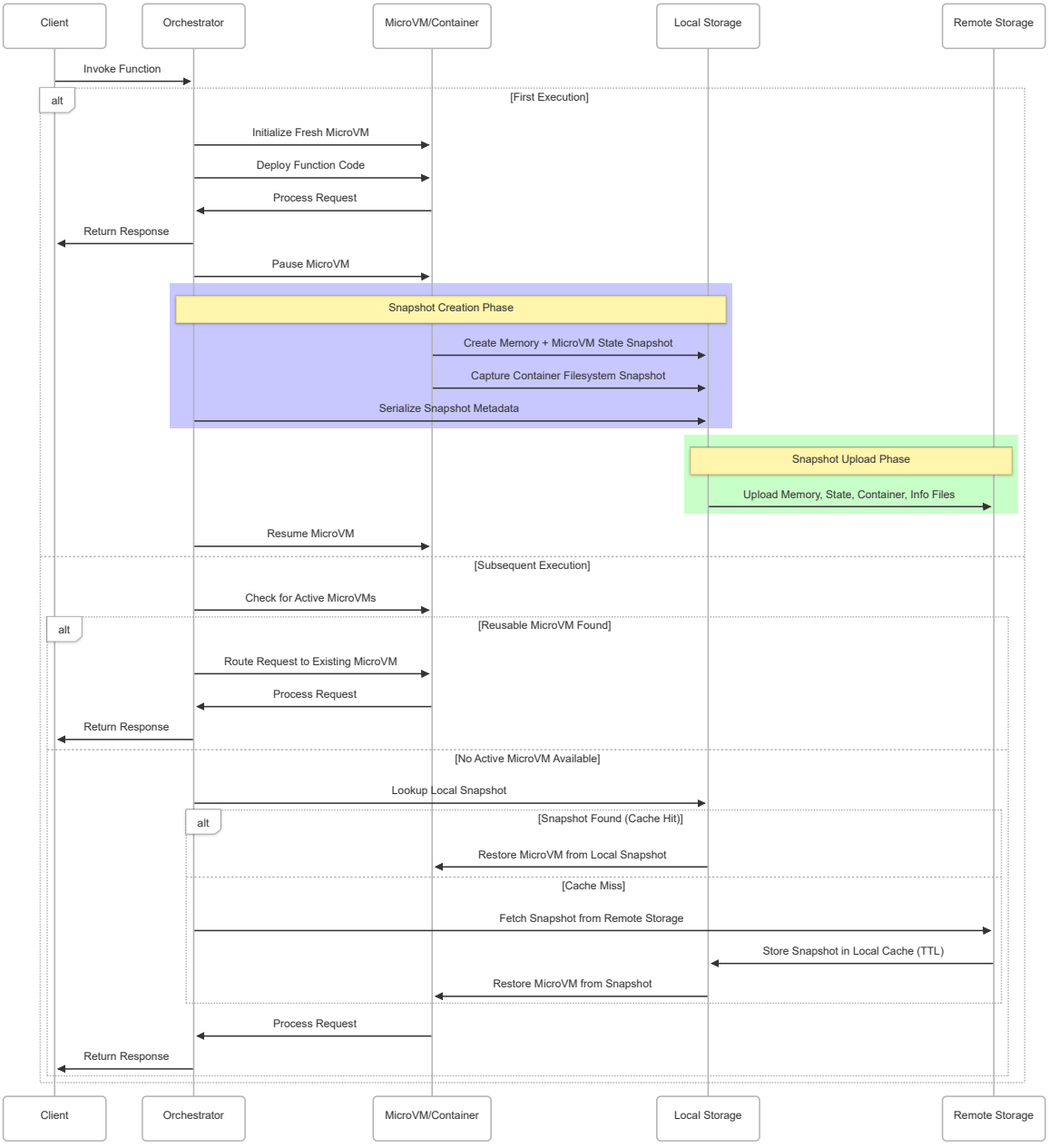

Figure 4.2: Snapshot workflow: snapshot creation, storage, and restoration.

- **Memory file:** Captures the contents of the VM's memory;
- **MicroVM state file:** Stores processor and device state information;
- **Container snapshot file:** Encodes the state of the container's filesystem;
- **Info file:** Contains metadata such as file paths, snapshot revision, and configuration details.

Algorithm 4.1: Snapshot Creation.

Input: Running microVM with deployed serverless function
Output: Snapshot stored in local cache and remote storage
begin
`// 1. Pause the VM`
PauseVM()
`// 2. Create Firecracker snapshot`
$(memFilePath, vmStateFilePath) \longleftarrow$ CreateFirecrackerSnapshot()
`// 3. Capture container filesystem state`
$containerSnapPath \longleftarrow$ CaptureContainerState()
`// 4. Serialize snapshot metadata`
$infoFilePath \longleftarrow$ SerializeSnapshotInfo(memFilePath, vmStateFilePath, containerSnapPath)
`// 5. Upload snapshot files and metadata to remote storage`
UploadToRemote(memFilePath, vmStateFilePath, containerSnapPath, infoFilePath)

The method used to capture the container state depends on the containerd snapshotter and the architecture in use. Some snapshotters store the container state directly within the memory snapshot, which simplifies snapshot management by eliminating the need for separate filesystem capture. Otherwise, manual capture is required and can be performed in two main ways:

- Using Docker's `commit` command [80] to create a new container image that includes any filesystem modifications;
- Accessing the mounted disk containing the container's filesystem and either copying it or creating a differential (diff) file representing the changes since initialization.

4.2.2 Snapshot Restoration

Subsequent executions of the same serverless function leverage the previously created snapshot to avoid redundant initialization. When a worker node is selected to run the function, it first queries the local cache for the required snapshot. If the snapshot is found, it can be used immediately to restore the microVM. Otherwise, the orchestrator downloads the necessary snapshot files from the remote storage and caches them locally for future reuse.

The restoration process is outlined in Algorithm 4.2. First, the system checks for the presence of the snapshot corresponding to the given revision. If the snapshot is not cached locally, it is downloaded from remote storage, starting with the info file. This metadata file is then parsed to determine the paths of the memory, microVM state, and container snapshot files. Once all components are present, the microVM is restored using the Firecracker API.

This workflow enables efficient reuse of warm execution environments, significantly reducing cold start times. The use of snapshot revisions ensures consistency and cacheability, while the local cache

Algorithm 4.2: Snapshot Restoration.

Input: Snapshot revision
Output: Restored microVM ready to execute function
begin
  `// 1. Load or download snapshot metadata`
  if not ExistsLocally(revision) **then**
    $infoPath \longleftarrow$ DownloadInfoFile(revision)
  else
    $infoPath \longleftarrow$ GetLocalInfoFile(revision)
  `// 2. Parse metadata to get file paths`
  $(memPath, vmStatePath, containerSnapPath) \longleftarrow$ ParseInfo(infoPath)
  `// 3. Ensure snapshot files are cached locally`
  if not ExistsLocally(memPath, vmStatePath, containerSnapPath) **then**
    DownloadFromRemote(memPath, vmStatePath, containerSnapPath)
  `// 4. Restore microVM from snapshot`
  RestoreFirecrackerVM(memPath, vmStatePath, containerSnapPath)

avoids unnecessary remote fetches during repeated invocations.

4.3 Optimized Scheduling via Snapshot Caching

Fetching snapshots from remote storage introduces non-negligible latency, particularly in cold start scenarios where microVMs are launched frequently and dynamically across a distributed cluster. To minimize this latency, the system incorporates a snapshot-aware scheduling mechanism paired with a local caching layer on each worker node.

Figure 4.3 illustrates the integration of snapshot-awareness into the Kubernetes scheduling architecture. The design introduces two new components:

- **NodeSnapshotCache (Custom Resource):** Each worker node maintains a local cache of frequently used snapshots and exposes its contents through a Kubernetes Custom Resource named `NodeSnapshotCache`. This resource serves as a declarative representation of the node's cache state, allowing cluster-level components to query which snapshots are locally available without direct file system interactions.

- **Snapshot-Aware Scheduler Extender:** Integrated into the Kubernetes scheduling workflow, this extender augments the default scheduler by considering snapshot locality during pod placement decisions. It queries the `NodeSnapshotCache` resources of all nodes and assigns higher scheduling scores to nodes that already contain the required snapshot, thus minimizing cold start latency.

This architecture ensures that scheduling decisions are informed by snapshot availability, reducing

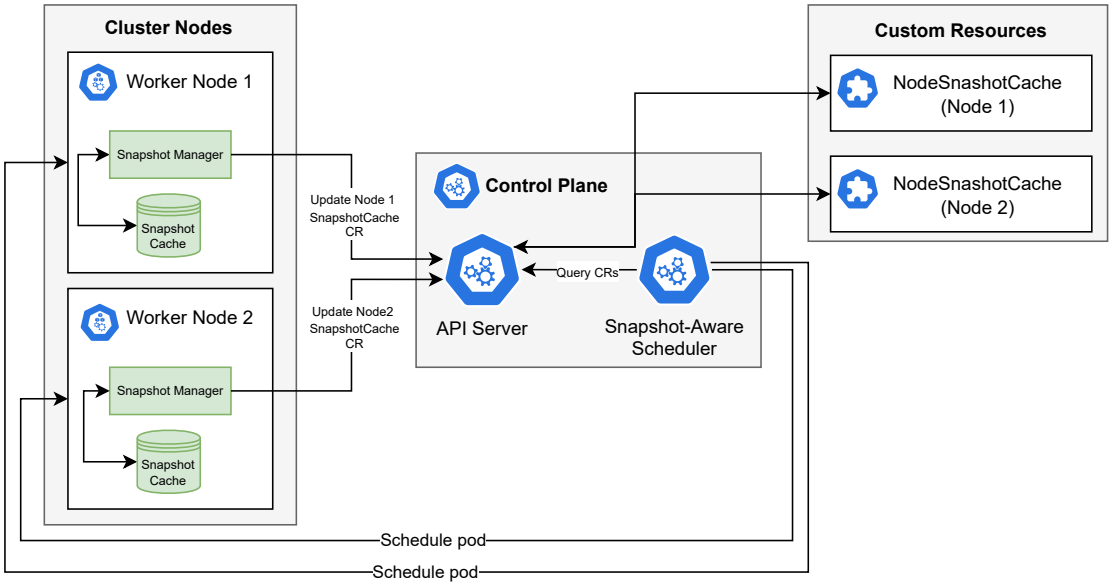

Figure 4.3: Architecture of the snapshot-aware scheduling system. Each node maintains a local snapshot cache and exposes its contents through a custom `NodeSnapshotCache` CRD. The scheduler extender queries these CRDs to assign higher scores to nodes that already cache the required snapshot, thereby reducing cold start latency during pod scheduling.

network transfers and improving startup performance. The following subsections describe these components in more detail.

4.3.1 Tracking Cached Snapshots

To enable snapshot-aware scheduling, the system must maintain an accurate view of which snapshots are locally cached on each node. This information allows the scheduler to make informed placement decisions, prioritizing nodes that already store the required snapshot and thereby reducing remote fetch latency.

For this purpose, the design introduces a custom Kubernetes [28] resource definition (Custom Resource Definition (CRD)) [81] named `NodeSnapshotCache`. Each `NodeSnapshotCache` object represents a single node and contains a list of snapshot identifiers currently available in that node's local cache. By abstracting this information as a Kubernetes resource, the cluster achieves a declarative and easily queryable mechanism for snapshot tracking without requiring direct interaction with the underlying file system.

Listing 4.1 illustrates an example `NodeSnapshotCache` resource for a node that caches snapshots for two functions: a simple `hello-world` function and an image-processing function. The scheduler can use this data to favor nodes that already have the necessary snapshot, significantly reducing cold start latency and network overhead.

Listing 4.1: Example `NodeSnapshotCache` CRD instance representing cached snapshots for a specific node

```
apiVersion: example.com/v1
kind: NodeSnapshotCache
metadata:
  name: node-000.cluster.example.com
spec:
  nodeName: node-000.cluster.example.com
  snapshots:
    - hello-world-00001
    - image-resize-00001
```

This decentralized and declarative approach provides a lightweight yet powerful mechanism for exposing cache state to external components such as the scheduler, enabling efficient snapshot-aware scheduling decisions across the cluster.

4.3.2 Extending the Scheduler with Snapshot-Aware Scoring

To leverage cache information during scheduling, the default Kubernetes scheduler [29] is extended using a scheduler extender [82]. This extender integrates into the Kubernetes scheduling pipeline via HTTP callbacks, influencing pod placement decisions without modifying core logic.

The extender operates exclusively during the `prioritize` phase of scheduling. At this point, the default scheduler has already performed node filtering (e.g., based on CPU, memory, taints, and affinity constraints) and assigned its own scores using built-in policies. The extender does not override these scores; instead, it augments them by adding a bias toward nodes that cache the required snapshot.

Upon receiving a scheduling request, the extender:

1. Parses pod metadata to identify the function and associated snapshot.
2. Queries all `NodeSnapshotCache` CRDs to determine which nodes have the snapshot locally.
3. Assigns additional scores to those nodes, boosting their chances of selection without excluding others.
4. Returns a ranked list of candidate nodes to the scheduler, which then combines these scores with existing priorities.

This approach ensures that snapshot locality is considered alongside other metrics, not in isolation. For example, if a node already caches the snapshot but is heavily loaded or violates affinity rules, the default scheduler's priorities will still dominate, preventing inefficient placement. Conversely, when

multiple nodes are otherwise equivalent, the extender gives preference to nodes that minimize remote snapshot fetches.

In summary, the integration of a node-local snapshot cache, CRD-based visibility, and an extender-enhanced scoring policy introduces snapshot-awareness without disrupting Kubernetes' multi-metric decision-making process.

4.3.3 Snapshot Cache Eviction Policy

To ensure that local snapshot caches remain within a bounded size, each worker node enforces a capacity limit on its snapshot cache, typically configured in the order of several gigabytes (e.g., 10–20 GB) to balance performance benefits against disk resource constraints. This limit prevents uncontrolled disk usage while allowing frequently accessed snapshots to remain available for reuse.

When a new snapshot is fetched or generated, the snapshot manager estimates the total size of the snapshot before committing it to disk. Since the memory state file constitutes the majority of the snapshot size, this estimation primarily considers the memory footprint of the corresponding microVM, while the metadata and device state files are negligible in comparison. If sufficient space is not available, the node evicts one or more existing snapshots based on a LRU policy.

The LRU policy is implemented by tracking a usage timestamp for each snapshot in the cache, which is updated every time a snapshot is used for restoring a function. When eviction becomes necessary, the snapshot manager identifies the least recently accessed snapshots and deletes them to free space. Eviction involves removing all files associated with the snapshot from disk and updating the corresponding `NodeSnapshotCache` CRD to reflect the new state.

The complete procedure for estimating snapshot size, evicting old snapshots if necessary, storing a new snapshot, and updating usage information is summarized in Algorithm 4.3. The algorithm proceeds as follows: first, the snapshot size is estimated (primarily considering the memory file); second, the algorithm checks available cache space and evicts the least recently used snapshots until sufficient space is freed; third, the new snapshot is downloaded and recorded in the cache; finally, the usage timestamp is updated to support future LRU decisions.

Although the current design uses a simple LRU approach, the policy can be extended in future work to incorporate additional factors such as snapshot access frequency, usage trends, or predictive models that anticipate future demand. These enhancements would enable more informed eviction decisions, improving cache hit rates while maintaining efficient resource utilization.

Algorithm 4.3: Snapshot Cache Eviction and Storage Policy.

Input: New snapshot metadata S, cache capacity C (bytes)
Output: Snapshot S stored in local cache
begin

```
// 1. Estimate size of the new snapshot (primarily memory file)
```
$size \longleftarrow$ EstimateSnapshotSize(S)
```
// 2. Check available space
```
while FreeSpace() $< size$ **do**
```
    // Evict least recently used snapshot
```
    $oldSnap \longleftarrow$ GetLRUSnapshot()
    DeleteSnapshotFiles($oldSnap$)
    UpdateNodeSnapshotCacheCRD(Remove=$oldSnap$)
```
// 3. Store the new snapshot
```
DownloadSnapshotFiles(S)
UpdateNodeSnapshotCacheCRD(Add=S)
```
// 4. Update usage timestamp for future LRU decisions
```
UpdateTimestamp(S)

5

Implementation

Contents

This chapter bridges the gap between the proposed architecture and its practical realization. It details the technical implementation of the solution, including the extension of the vHive platform to support remote snapshotting and other related optimizations. In total, the implementation involved adding/-modifying approximately 4000 lines of code, configuration, and documentation, spread across multiple repositories, including vHive, and firecracker itself. Key challenges encountered during the development process are highlighted, along with the strategies employed to address them. By the end of this chapter, readers will gain a clear understanding of how the conceptual design was translated into a functional prototype within the vHive ecosystem.

5.1 Extending firecracker-containerd for Snapshot Support

At the time this work was conducted, firecracker-containerd did not officially support the Firecracker snapshotting API. This functionality was previously implemented by the vHive team through forks of three Firecracker-related repositories:

- **firecracker**: Modified to add the `container_snapshot_path` parameter for loading snapshot requests. This change addressed the non-deterministic nature of container snapshot paths, which differ between snapshot creation and restoration. Since Firecracker does not allow renaming resources during restore, the vHive implementation substituted the VM state path of the block device with the new container snapshot path received from the `LoadSnapshot` request.
- **firecracker-go-sdk**: Extended to support the `container_snapshot_path` parameter, aligning with the Firecracker base modifications.
- **firecracker-containerd**: Enhanced to enable Firecracker snapshot restore by introducing:
  - A new `CreateSnapshot` request.
  - Additional parameters for snapshot loading in the `CreateVM` request (snapshot loading is implemented as a variant of VM creation with snapshot options).

  For VMs created from a snapshot, container snapshot drives are already mounted, so drive mount stubs and mounting logic were skipped.

These changes were implemented prior to this project by the vHive team and not as part of this work. However, in the context of this work, adjustments were required to make these existing capabilities compatible with the new architecture. For instance, in the Firecracker base repository, additional logic was introduced to correctly handle container snapshot devices for the Stargz snapshotter, specifically matching both `"snap"` and `"ctrstub"` device paths during snapshot restoration.

5.2 Stargz Snapshotter Integration

The implementation began with the addition of support for remote containerd snapshotters, specifically integrating the Stargz snapshotter into vHive. This work is documented in PR #1091 and the associated issue #1090.

Stargz is an optimized snapshotter for containerd that introduces *lazy image loading*, meaning container images can start running before the entire image is downloaded. It uses a seekable compressed format (stargz) that allows on-demand fetching of image layers, significantly reducing startup time for

large container images—a feature particularly valuable in serverless and microVM-based environments where cold-start latency is critical.

vHive did not support remote snapshotters such as `stargz` in conjunction with `firecracker-containerd`, limiting its ability to leverage lazy image loading. While `stargz` was supported in standard Kubernetes deployments, it was not integrated into vHive's Firecracker-based container runtime.

To address this, the architecture described in the official firecracker-containerd documentation for remote snapshotters (discussed in Section 2.2.3) was adopted. This required several components and modifications to be introduced:

- **New dependencies:**
  - `http-address-resolver`: A lightweight service that maps container namespaces to snapshotter addresses.
  - `demux-snapshotter`: A proxy that intercepts snapshotter requests from containerd and routes them to the correct remote snapshotter instance running inside the target microVM.
- **Updated binaries:**
  - `default-rootfs.img`: Rebuilt to include Stargz.
  - `vmlinux-5.10.186`: Updated kernel version with FUSE enabled.
  - `Firecracker binary`: Upgraded to enable dynamic stub drive path updates.

To ensure reliability in this integration, a set of both unit and integration tests was also implemented.

5.3 Supporting Remote Snapshots

Following the addition of support for the Stargz snapshotter, the remote snapshot logic was implemented. Initially, snapshot management was controlled via a simple boolean flag (`-snapshots`). This was replaced by a string-based enumeration, allowing users to explicitly choose among `"disabled"`, `"local"`, and `"remote"` snapshot modes.

To implement remote storage, **MinIO** [83] was used as the S3-compatible solution. A MinIO instance is deployed within the vHive Kubernetes cluster and serves as the remote store for snapshot metadata and files.

Snapshot handling was modified to support the following workflow:

- **During snapshot creation:** After the snapshot is written to local disk, it is uploaded to MinIO for remote availability.

- **During snapshot restoration:** When a snapshot revision is requested, the orchestrator first checks the local cache. If the required files are missing, it retrieves the snapshot metadata (info file) from MinIO, followed by the associated memory, state, and container snapshot files.

For simple solutions, this MinIO instance is running on a single node; however, it can be easily scaled and distributed across multiple nodes for a production-ready solution.

Again, a set of integration tests was added to test the remote snapshot integration.

5.4 Scheduler Extension for Cache-Aware VM Placement

To optimize snapshot distribution and reduce cold start latency, the default Kubernetes scheduler was extended with a snapshot-locality-aware scoring mechanism. This extension prioritizes nodes that already cache the requested snapshot, minimizing remote fetch times and improving overall system responsiveness.

5.4.1 Defining the Snapshot Cache CRD

The first step is defining a Kubernetes CRD that tracks which snapshots are cached locally on each node. The CRD schema specifies two main fields:

- `nodeName`: Identifies the node associated with this cache resource.
- `snapshots[]`: An array of snapshot revision identifiers currently stored in the local cache.

These fields provide a declarative view of cache state that is accessible through the Kubernetes API, avoiding direct file system queries.

Listing 5.1 shows the CRD definition, which includes the OpenAPI schema to enforce structure.

Listing 5.1: Definition of the `NodeSnapshotCache` CustomResourceDefinition

```
1 apiVersion: apiextensions.k8s.io/v1
2 kind: CustomResourceDefinition
3 ...
4 spec:
5   ...
6   versions:
7     - name: v1
8       ...
9       schema:
```

```
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              nodeName:
                type: string
                description: Name of the node where the snapshot cache resides.
              snapshots:
                type: array
                description: List of locally cached snapshot revisions.
                items:
                  type: string
            required:
              - nodeName
              - snapshots
```

In parallel, the Go types for the CRD were generated using the Kubernetes code generation tools. The snapshot manager was then modified to update the CRD whenever it commits or deletes a snapshot. This interaction was implemented using the Kubernetes Go API client.

5.4.2 Implementing the Scheduler Extender

To influence pod placement, a scheduler extender was implemented with support for the `prioritize` phase only. During this phase, the extender scores each node based on whether it has the required snapshot already cached.

The core logic is shown in Listing 5.2. It retrieves all nodes being considered for scheduling, checks the corresponding `NodeSnapshotCache` resource, and awards an higher score if the snapshot is found locally.

Listing 5.2: Simplified `prioritize` handler in the snapshot-locality scheduler extender

```
func (s *SnapshotLocalityExtender) prioritize(w http.ResponseWriter, r *http.Request) {
    ...
    for _, node := range nodes.Items {
        score := int64(0)
        nodeCache := &k8s.NodeSnapshotCache{}
        err := s.client.Get(r.Context(), client.ObjectKey{Name: node.Name}, nodeCache)
```

```
        if err == nil {
            for _, cached := range nodeCache.Spec.Snapshots {
                if cached == snapshotRevision {
                    score = 100 // Cache hit
                    break
                }
            }
        }
        hostPriorities = append(hostPriorities, schedulerapi.HostPriority{
            Host:  node.Name,
            Score: score,
        })
    }
    ...
}
```

This method ensures that pod scheduling is skewed toward nodes with warm caches, thus significantly reducing initialization latency in a serverless context.

Finally, the cluster setup scripts were extended to deploy the snapshot-locality scheduler and register the extender. The Kubernetes scheduler configuration includes a reference to the extender's service endpoint and scoring method, as shown in Listing 5.3.

Listing 5.3: Kubernetes scheduler configuration with snapshot-locality extender

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - schedulerName: snapshot-locality-scheduler
extenders:
  - urlPrefix: "http://snapshot-locality-extender.kube-system.svc.cluster.local:8080"
    prioritizeVerb: "prioritize"
    weight: 100
    nodeCacheCapable: false
```

This configuration ensures that the extender is invoked for each scheduling decision, augmenting the default scheduler's scoring logic without modifying its core behavior.

In summary, this scheduler extension integrates tightly with the snapshot caching layer to provide low-latency, locality-aware VM placement across the cluster.

6

Evaluation

Contents

This chapter presents a comprehensive evaluation of the system, focusing on performance. It begins by outlining the evaluation goals and defining the key metrics used to assess the system. The experimental setup and selected benchmarks are then described to provide context for the testing process. Finally, the results of the experiments are presented and analyzed, followed by a discussion addressing the system's effectiveness, trade-offs, and limitations.

6.1 Goals

Numerous evaluation criteria are available for assessing modern serverless platforms. This evaluation concentrates on three primary objectives:

- **Measure the effectiveness of the remote snapshot system in reducing cold start latency without caching.** This establishes a baseline understanding of the performance impact of retrieving snapshots directly from remote storage, and compares it against both the baseline vHive (no snapshots) and vHive with local snapshots. The expectation is that even without caching, remote snapshots may still offer faster startup than initializing a fresh instance.
- **Evaluate the benefits of caching remote snapshots locally.** Once a remote snapshot has been retrieved and cached on a node, subsequent cold starts for the same function should avoid repeated remote fetches. This goal focuses on quantifying the performance improvement from caching and determining how much it narrows the gap between local and remote snapshot systems.
- **Evaluate the impact of introducing snapshot-aware scheduling.** This will be assessed against vHive with both local and remote snapshots, but without the enhanced scheduler. The hypothesis is that the snapshot-aware scheduling approach will effectively reduce cold start times and mitigate fetching latency by preferentially scheduling functions on nodes where the relevant snapshots are already available locally.

The evaluation compares the enhanced vHive system with remote snapshots against two alternative configurations: the baseline system with local snapshots and a version of vHive without snapshotting. This comparison highlights the benefits introduced by remote snapshot orchestration. In addition, the impact of using the `stargz` snapshotter instead of `devmapper` is analyzed.

6.2 Metrics

To evaluate these goals, the metric used was **initialization latency** *(Unit: milliseconds, ms)*, which was captured and analyzed under different conditions. Initialization latency is defined as the time between when an invocation is registered in the system and when the function code begins execution. This metric was chosen instead of overall response time because the latter is influenced by additional factors such as Kubernetes readiness probes, namespace creation, and the function's actual execution time. By excluding these factors, initialization latency isolates the startup overhead.

6.3 Experimental Setup

Experiments were conducted on the CloudLab [84] platform using a controlled environment with uniform hardware configurations for worker nodes. Table 6.1 summarizes the hardware and software specifications used for each node in the cluster.

Component	Specification
CPU Model	Intel Xeon Silver 4114 Deca-core 2.20 GHz
CPU Cores	10 physical cores, 20 threads
Memory	64 GB DDR4
Architecture	x86_64
OS Image	Ubuntu 24.04

Table 6.1: Node Configuration Used in Experiments (CloudLab `c220g5`).

The evaluation was performed on a Kubernetes cluster consisting of four nodes: a master node, a loader node responsible for invoking functions, and two worker nodes executing the functions. To support remote snapshot storage, a Kubernetes deployment of MinIO was installed within the cluster, providing object storage services required by the system.

6.3.1 Evaluation Tools

The evaluation was conducted using a combination of small-scale experiments and large-scale trace-driven workloads. Initially, smaller and more controlled experiments were executed by invoking the vHive orchestrator directly. These experiments served two purposes: first, to obtain preliminary performance results before running full-scale traces, and second, to enable a more fine-grained analysis of individual function invocations.

For large-scale evaluation under realistic conditions, In-Vitro [85] was used, a set of tools for analyzing the performance of serverless cluster deployments designed to simulate data-intensive workloads. In-Vitro comprises two components: the sampler, which generates representative workload summaries based on production traces, and the loader, which reconstructs invocation traffic from a given trace and directs it toward functions deployed in the target serverless cluster. In-Vitro also supports vSwarm benchmarks. Its seamless integration with the vHive ecosystem makes it particularly well-suited for evaluating the performance of the proposed system. To enable evaluation of remote snapshot orchestration and the custom scheduler, In-Vitro was extended by modifying its configuration and setup scripts.

6.3.2 Benchmarks

The evaluation employed a trace containing four functions with distinct behaviors: three triggered by queues and one triggered by a timer. The trace duration was 30 minutes, resulting in a total of 60 invocations (15 invocations per function).

The trace pattern alternates one minute of function invocations followed by one minute of inactivity, repeating this cycle. Given the default scale-down period of one minute, spacing invocations with a minute (e.g., 1,0,1,0,...) effectively triggers cold starts for each invocation.

Table 6.2: Average cold start latencies (ms) across snapshotters and configurations. Each value is the mean of 15 cold-start invocations.

Snapshotter	No Snapshots	Local Snapshots	Remote Snapshots (no cache)
Devmapper	2507	355	N/A
Stargz	6698	197	757

6.4 Experiments

This section presents the experiments conducted to evaluate the proposed system. The evaluation is structured to progressively analyze the system's contributions: (i) the benefit of remote snapshots over local snapshots and no snapshots, and (ii) the impact of the snapshot-aware scheduling policy. The goals and metrics are described in Section 6.1 and Section 6.2, and the experimental setup is outlined in Section 6.3.

6.4.1 Controlled Snapshot Performance Experiments

Before running large-scale trace-driven experiments, a set of smaller, controlled measurements was performed by invoking the vHive orchestrator directly. These controlled experiments had two goals: (i) to produce early, micro-level evidence about the benefits and costs of snapshotting, and (ii) to enable a fine-grained analysis of the startup path (for example, separating VM restoration from remote fetch time). All latency values reported for the controlled experiments are averages of 15 independent cold-start invocations for each configuration.

Table 6.2 summarizes the average cold-start latencies measured for each snapshotter and configuration. Note that remote snapshots were not implemented for the `devmapper` snapshotter in the system (marked as N/A). This limitation was one of the motivations for adopting `stargz` for the full remote-snapshot implementation.

Key observations and interpretation

- **What the numbers mean.** Each entry in Table 6.2 is the average end-to-end cold-start latency measured for the named snapshotter and snapshotting mode. For example, "Stargz – Remote Snapshots = 757 ms" means that with the `stargz` snapshotter and the prototype remote-snapshot retrieval enabled, the average cold-start latency across 15 independent cold starts was 757 ms.
- **Devmapper vs. stargz (no-snapshot baseline).** The "no snapshots" baseline is substantially slower on `stargz` (6698 ms) than on `devmapper` (2507 ms). Intuitively, `stargz` would be expected to perform equally or faster due to lazy pulling; however, in the system setup, the `stargz` baseline exhibited significant additional overhead. Possible (non-exclusive) explanations include additional

components in the system setup, extra resolution or initialization steps introduced by the `stargz` stack, or configuration-specific overheads in the testbed. The precise cause of the slower performance in the plain ("vanilla") `stargz` path is unknown and would require further root-cause profiling to confirm.

- **Local snapshots: large improvements.** With local snapshots, both snapshotters show dramatic reductions in cold-start latency compared to their vanilla baselines. Specifically:
  - `Devmapper`: 2507 ms $\rightarrow$ 355 ms, an $\approx 85.8\%$ improvement.
  - `Stargz`: 6698 ms $\rightarrow$ 197 ms, an $\approx 97.1\%$ improvement.

  Interestingly, `stargz` is faster than `devmapper` in the local case, although the reason for this difference is unclear from the experiments.

- **Remote snapshots: substantial but fetch-limited improvements.** For `stargz`, remote snapshots reduce latency from 6698 ms $\rightarrow$ 757 ms, an $\approx 88.7\%$ improvement over vanilla. However, roughly
$$\frac{556.6}{757} \approx 73.5\%$$
of the remote-snapshot time is spent fetching data from MinIO, showing that network/storage transfer dominates. As a result, remote snapshots are slower than local snapshots, but still far outperform the vanilla baseline.

The controlled measurements therefore establish three important micro-level facts: (i) snapshotting dramatically reduces cold-start latency when snapshots are local (up to $\approx 97\%$ reduction with `stargz`); (ii) the snapshotter implementation and configuration materially affect both baseline and snapshot restore times; and (iii) remote snapshotting provides significant improvements (nearly 89% vs. vanilla), but its performance is limited by backend transfer latency, explaining why it is significantly slower than local snapshotting.

6.4.2 Remote vs Local Snapshot Performance

To validate the findings under realistic conditions, large-scale experiments were conducted using In-Vitro. These experiments compared four configurations: (a) the baseline without snapshots (Vanilla), (b) local snapshots, (c) remote snapshots without caching, and (d) remote snapshots with caching. The inclusion of the no-cache variant reflects an environment where local disk capacity is limited and caching is not feasible.

The expectation is that remote snapshots achieve performance similar to local snapshots. Although retrieving a remote snapshot introduces additional latency, this overhead should occur only during the

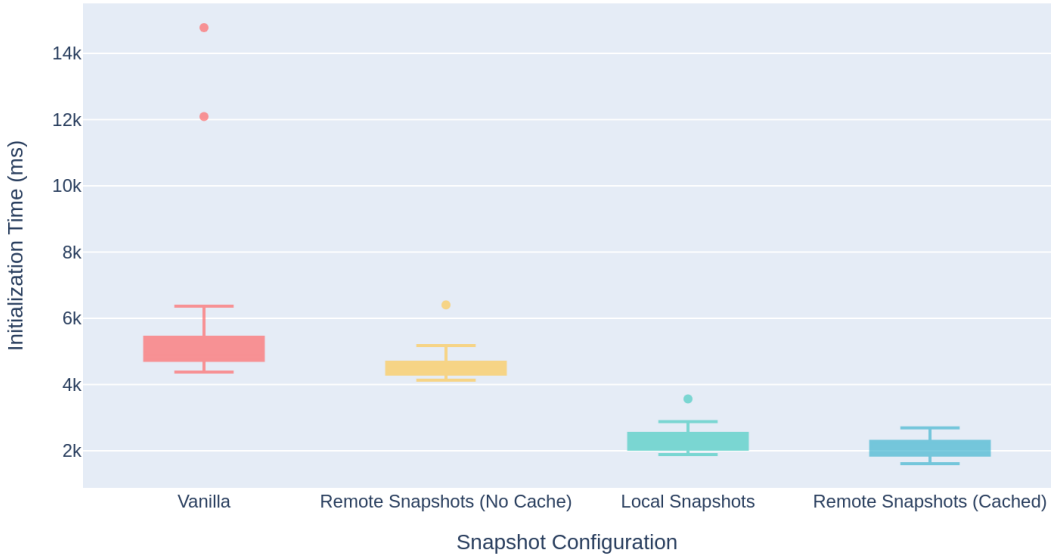

Figure 6.1: Box plots showing cold start initialization latency distributions for baseline vHive (no snapshots), local snapshots, remote snapshots (no cache), and remote snapshots (cached).

initial fetch on a given worker node, after which the snapshot remains cached locally. Furthermore, this one-time fetch latency is expected to be lower than the latency incurred when starting a fresh VM without snapshots.

As shown in Figure 6.1, all snapshot-based configurations reduce cold start latency compared to the baseline. The average initialization latency for the baseline is approximately 5398 ms. Local snapshots reduce this to around 2317 ms, corresponding to a 2.33$\times$ speedup. Remote snapshots without caching achieve an average latency of 4541 ms (1.19$\times$ speedup), while remote snapshots with caching provide the best performance among these configurations at approximately 2088 ms (2.58$\times$ speedup).

Remote snapshots without caching still outperform the baseline by avoiding full VM boot, but they suffer from higher tail latency and increased variability compared to cached snapshots. Caching plays a critical role in minimizing remote fetch overhead and ensuring predictable performance.

Outliers in the baseline configuration exhibit significantly higher latency spikes, reflecting the unpredictability of cold starts without snapshots. Snapshot-based strategies reduce both median latency and the spread of values, although the no-cache remote snapshots show greater variance than cached ones.

Figure 6.2 further illustrates tail latency behavior across configurations. Both local snapshots and cached remote snapshots substantially reduce P90–P99 latencies compared to the baseline. Remote snapshots without caching, however, exhibit higher tail latencies, highlighting the negative impact of

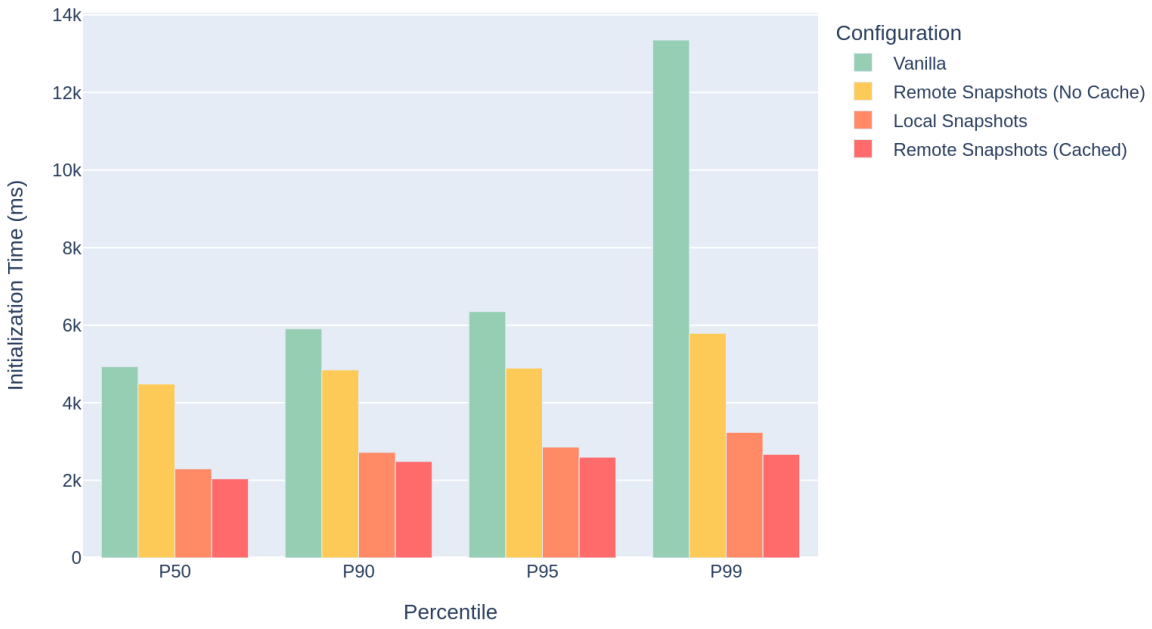

Figure 6.2: Tail latency analysis (P50, P90, P95, and P99) of cold start initialization times across snapshot strategies.

repeated remote fetches under load.

In summary, remote snapshots with caching achieve the best balance between latency reduction and flexibility, enabling cross-node snapshot sharing while delivering performance comparable to local snapshots. Without caching, remote snapshots provide only limited benefits and are less suitable for latency-sensitive environments without local storage availability.

6.4.3 Impact of Snapshot-Aware Scheduling

This experiment evaluates the performance benefit of enabling the snapshot-aware scheduler in conjunction with remote snapshots. The expectation is that the snapshot-aware scheduling will have similar, and potentially better, performance than the regular scheduler, since it mitigates the fetching latency.

Figure 6.3 shows that the snapshot-aware scheduler reduces cold start latency compared to the default scheduler when both employ remote snapshots. The average initialization latency decreases from approximately 2088 ms to 2078 ms, indicating that cache-aware scheduling further optimizes startup times by placing functions on nodes with relevant snapshots already cached.

As illustrated in Figure 6.4, the snapshot-aware scheduler narrows the latency distribution and reduces tail latencies, improving both the consistency and predictability of cold starts. By considering

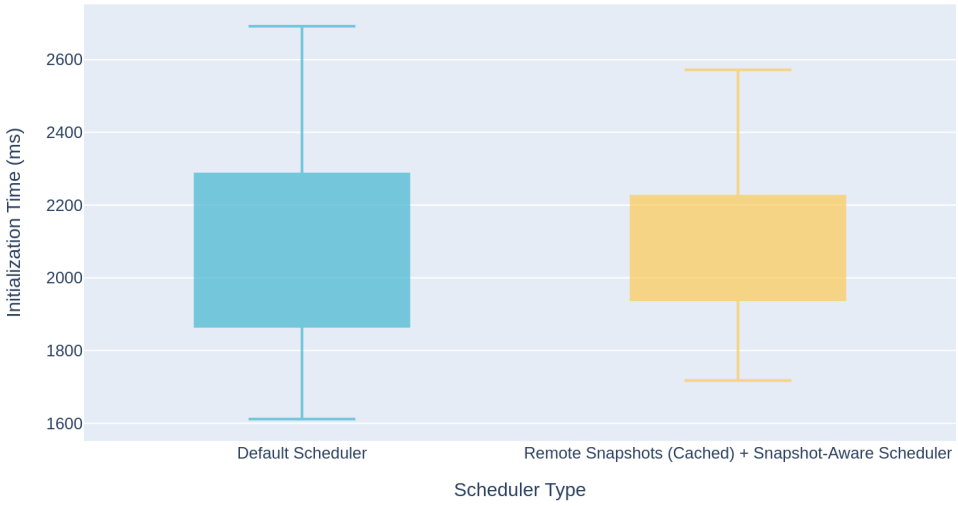

Figure 6.3: Box plots comparing initialization latency distributions between the default scheduler and the snapshot-aware scheduler under the remote snapshot configuration. The snapshot-aware scheduler leverages snapshot cache locality to optimize function placement and reduce cold start latency.

snapshot cache locality, the scheduler effectively minimizes overhead caused by fetching snapshots remotely.

6.5 Discussion

The experimental results clearly demonstrate the advantages of remote snapshot orchestration and snapshot-aware scheduling in reducing cold start latencies in serverless environments.

Figure 6.5 summarizes the cumulative performance benefits of the proposed system. The integration of remote snapshots with caching and snapshot-aware scheduling achieves the lowest average initialization latency (2078 ms) and reduces variability compared to all other configurations, demonstrating the synergy between snapshot sharing and intelligent scheduling.

Remote snapshots with cache achieve more than a 2x reduction in initialization latency compared to the baseline without snapshots, while maintaining competitive performance with local snapshots and enabling flexible snapshot sharing across nodes. This flexibility is crucial for scaling in distributed systems where local snapshot availability may be limited.

The snapshot-aware scheduler further enhances performance by intelligently leveraging snapshot cache locality, reducing both the average latency and its variance. This results in more predictable cold

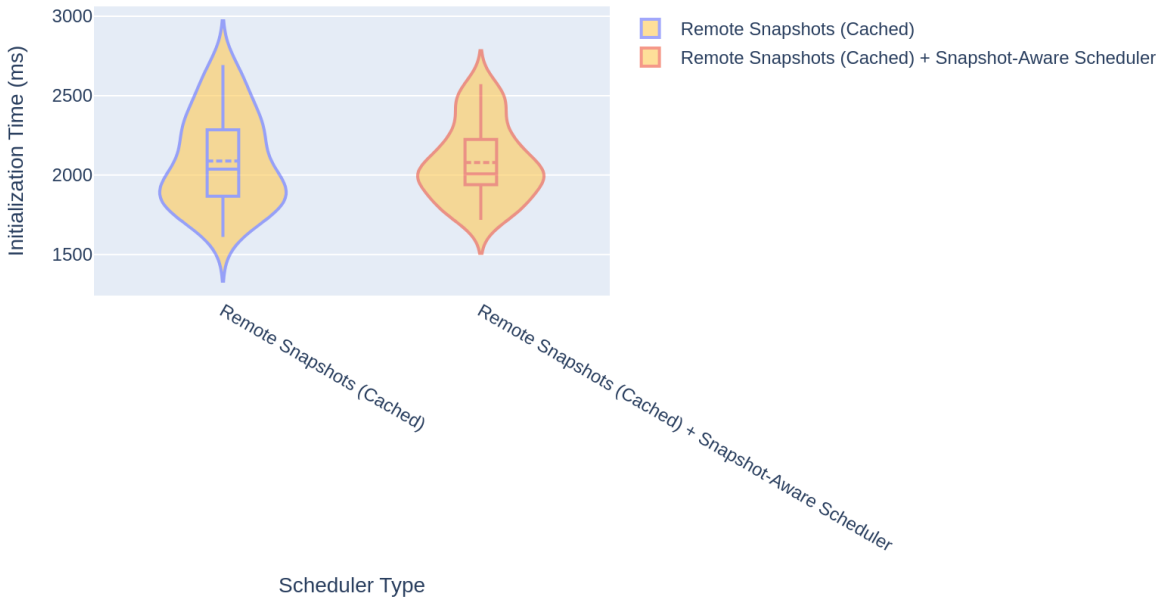

Figure 6.4: Violin plots showing the full distribution of cold start initialization latencies under the default and snapshot-aware schedulers. The snapshot-aware scheduler not only reduces median latency but also decreases variance by prioritizing nodes with cached snapshots.

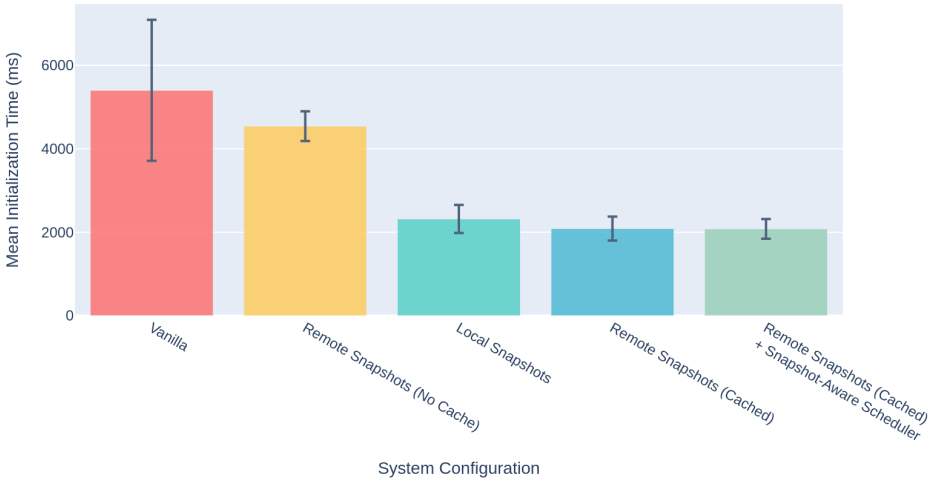

Figure 6.5: Bar chart comparing mean cold start initialization times with standard deviation error bars across five configurations: baseline (no snapshots), local snapshots, remote snapshots (no cache), remote snapshots (cached), and remote snapshots with snapshot-aware scheduler.

start behavior, which is vital for meeting quality-of-service requirements in production deployments.

Overall, the combined system not only improves average initialization times but also significantly reduces tail latencies, addressing both the efficiency and reliability challenges of cold starts. These findings validate the design choices of the remote snapshot orchestration framework and underscore the importance of scheduling policies that are aware of snapshot placement.

Future work could explore adaptive scheduling heuristics that dynamically learn snapshot usage patterns and integrate them with workload forecasting to further optimize cold start performance.

7

Conclusion

Contents

This chapter recaps the core contributions and findings of the work presented. It reflects on the effectiveness of the implemented solution, emphasizing the outcomes observed through evaluation. Additionally, it acknowledges current limitations and outlines potential directions for future research and system enhancements.

7.1 Conclusions

This work explored cloud computing with a focus on the FaaS programming model, identifying cold start latency as a critical bottleneck impacting serverless platform responsiveness. After reviewing various mitigation techniques, snapshotting emerged as the most promising approach to address cold start delays. However, existing state-of-the-art solutions suffer from challenges related to scalability, storage efficiency, and snapshot distribution overhead.

To address these limitations, a snapshot orchestration system tailored for distributed serverless environments was proposed. Built on the vHive platform, the system combines remote snapshot storage, local caching, and snapshot-aware scheduling to enable efficient snapshot sharing and restoration across worker nodes.

This system facilitates efficient sharing and restoration of snapshots across worker nodes within a distributed deployment. A centralized remote snapshot store separates snapshot creation from access, while local caching minimizes redundant data transfers and reduces fetch latency.

Additionally, a snapshot-aware scheduler was introduced to reduce cold start latency by directing invocations to nodes with the required snapshot cached locally. This combined orchestration and scheduling strategy effectively balances performance improvements with resource efficiency.

Experimental evaluation demonstrated that remote snapshot orchestration reduces cold start latency by approximately 61.3% compared to the baseline without snapshots, decreasing average initialization time from 5398 ms to 2078 ms, and achieves slightly better performance than local snapshots. Integration with the snapshot-aware scheduler further reduces cold start latency compared to remote snapshots with the default scheduler, lowering the average initialization time to 2078 ms. This highlights the combined benefit of snapshot orchestration and intelligent scheduling.

These results validate snapshot orchestration as a scalable, effective strategy for minimizing cold start overhead in modern serverless platforms while maintaining efficient resource utilization.

7.2 Limitations and Future Work

While the proposed system addresses key performance bottlenecks in FaaS cold starts, several limitations remain.

First, prior work [10,11] has shown that function invocations often exhibit temporal and spatial locality, making them amenable to prediction. This opens the opportunity for integrating predictive prefetching techniques that proactively load snapshots into a node's local cache before they are needed, thereby reducing cold start latency caused by on-demand snapshot retrieval.

Second, storage optimization techniques such as deduplication and compression were not implemented in the current system but represent promising avenues for future enhancement. By identifying and eliminating redundant data across snapshots, deduplication can significantly reduce storage footprint and network transfer overhead. Compression techniques can further decrease the size of stored snapshots, improving storage efficiency and reducing snapshot fetch latency. Integrating these optimizations would contribute to better scalability and resource utilization, particularly in large-scale deployments with numerous and frequently updated snapshots.

Moreover, the current system implements only a basic snapshot eviction strategy, and comprehen-

sive lifecycle management is still lacking. As snapshots accumulate over time, both remote and local storage risk becoming congested with outdated or infrequently accessed snapshots. While the existing approach uses a simple policy to reclaim space, future work could explore more sophisticated eviction mechanisms that account for additional factors such as usage frequency trends, temporal access patterns, and workload periodicity. Such enhancements would improve cache efficiency and ensure that storage resources are used optimally.

Finally, the current design relies on a centralized remote storage system, which may present scalability or availability bottlenecks in larger clusters. Future work could explore decentralized or hybrid models that incorporate Peer-to-Peer (P2P) snapshot sharing among nodes. Such designs could reduce latency, distribute load more evenly, and improve fault tolerance, especially in edge or geographically distributed deployments.

Addressing these limitations would enhance the robustness, scalability, and adaptability of the proposed system in real-world serverless environments.

Bibliography

[1] "Architecture - Knative — knative.dev," https://knative.dev/docs/serving/architecture, [Accessed 31-08-2025].

[2] "containerd — containerd.io," https://containerd.io/, [Accessed 08-11-2024].

[3] "firecracker-containerd/docs/architecture.md at main · firecracker-microvm/firecracker-containerd — github.com," https://github.com/firecracker-microvm/firecracker-containerd/blob/main/docs/architecture.md, [Accessed 26-12-2024].

[4] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Comput. Surv.*, vol. 54, no. 11s, nov 2022. [Online]. Available: https://doi.org/10.1145/3510611

[5] "Understanding Lambda function invocation methods - AWS Lambda — docs.aws.amazon.com," https://docs.aws.amazon.com/lambda/latest/dg/lambda-invocation.html, [Accessed 10-07-2024].

[6] "Azure Functions – Serverless Functions in Computing — Microsoft Azure — azure.microsoft.com," https://azure.microsoft.com/en-us/products/functions/, [Accessed 10-07-2024].

[7] "Cloud Functions — Google Cloud — cloud.google.com," https://cloud.google.com/functions/?hl=en, [Accessed 10-07-2024].

[8] "IBM Cloud Code Engine — ibm.com," https://www.ibm.com/products/code-engine, [Accessed 10-07-2024].

[9] "Cloudflare Workers© — workers.cloudflare.com," https://workers.cloudflare.com/, [Accessed 03-08-2024].

[10] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[11] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, “ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs,” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 303–320. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/mahgoub

[12] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, “Benchmarking, analysis, and optimization of serverless function snapshots,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 559–572.

[13] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, “Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481. [Online]. Available: https://doi.org/10.1145/3373376.3378512

[14] S. Kohli, S. Kharbanda, R. Bruno, J. Carreira, and P. Fonseca, “Pronghorn: Effective checkpoint orchestration for serverless hot-starts,” in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 298–316. [Online]. Available: https://doi.org/10.1145/3627703.3629556

[15] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “Seuss: skip redundant paths to make serverless fast,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3392698

[16] A. Panda and S. R. Sarangi, “Snapstore: A snapshot storage system for serverless systems,” in *Proceedings of the 24th International Middleware Conference*, 2023, pp. 261–274.

[17] N. Lazarev, V. Gohil, J. Tsai, A. Anderson, B. Chitlur, Z. Zhang, and C. Delimitrou, “Sabre: Hardware-Accelerated snapshot compression for serverless MicroVMs,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 1–18. [Online]. Available: https://www.usenix.org/conference/osdi24/presentation/lazarev

[18] A. Fuerst and P. Sharma, “Faascache: keeping serverless computing alive with greedy-dual caching,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 386–400. [Online]. Available: https://doi.org/10.1145/3445814.3446757

[19] L. Ao, G. Porter, and G. M. Voelker, “Faasnap: Faas made fast using snapshot-based vms,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 730–746. [Online]. Available: https://doi.org/10.1145/3492321.3524270

[20] Y. Fu, L. Xue, Y. Huang, A.-O. Brabete, D. Ustiugov, Y. Patel, and L. Mai, “{ServerlessLLM}:{Low-Latency} serverless inference for large language models,” in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024, pp. 135–153.

[21] M. Brooker, “Lambda Snapstart, and snapshots as a tool for system builders - Marc's Blog — brooker.co.za,” https://brooker.co.za/blog/2022/11/29/snapstart, [Accessed 03-05-2025].

[22] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: lightweight virtualization for serverless applications,” in *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, ser. NSDI’20. USA: USENIX Association, 2020, p. 419–434.

[23] K. Tokunaga, “Startup Containers in Lightning Speed with Lazy Image Distribution on Containerd — medium.com,” https://medium.com/nttlabs/startup-containers-in-lightning-speed-with-lazy-image-distribution-on-containerd-243d94522361, [Accessed 11-01-2025].

[24] “Amazon EC2 - Capacidade de computação em nuvem - AWS — aws.amazon.com,” https://aws.amazon.com/pt/ec2/?trk=1274ba19-09b0-4244-ac34-7736160c5885&sc_channel=ps&ef_id=CjwKCAjwt-_FBhBzEiwA7QEqyO9QgVldIYYIDnuRV-2otxddAhW-Fx1Yoz_GQ_uQBzQzKRypS_LSrhoCt0MQAvD_BwE:G:s&s_kwcid=AL!4422!3!589857500061!e!!g!!amazon%20ec2!16395977659!136815002307&gad_campaignid=16395977659&gbraid=0AAAADjHtp8d3j3tMWblTtxDSGOVlSnX0&gclid=CjwKCAjwt-_FBhBzEiwA7QEqyO9QgVldIYYIDnuRV-2otxddAhW-Fx1Yoz_GQ_uQBzQzKRypS_LSrhoCt0MQAvD_BwE, [Accessed 06-09-2025].

[25] “Compute Engine — cloud.google.com,” https://cloud.google.com/products/compute, [Accessed 06-09-2025].

[26] “App Engine Application Platform — Google Cloud — cloud.google.com,” https://cloud.google.com/appengine, [Accessed 06-09-2025].

[27] “Home — heroku.com,” https://www.heroku.com/, [Accessed 06-09-2025].

[28] M. Luksa, *Kubernetes in action*. Simon and Schuster, 2017.

[29] “Kubernetes Scheduler — kubernetes.io,” https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/, [Accessed 29-06-2025].

[30] J. Chester, *Knative in Action*. Simon and Schuster, 2021.

[31] B. Strehl, “Serverless Benchmark 2.0—Part I — medium.com,” https://medium.com/elbstack/serverless-benchmark-2-0-part-i-f23acb8e8a29, [Accessed 02-01-2025].

[32] H. Yu, R. Basu Roy, C. Fontenot, D. Tiwari, J. Li, H. Zhang, H. Wang, and S.-J. Park, “Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing,” in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS ’24. New York, NY, USA: Association for Computing Machinery, 2024, p. 335–350. [Online]. Available: https://doi.org/10.1145/3617232.3624871

[33] “Overview of memory management — App quality — Android Developers — developer.android.com,” https://developer.android.com/topic/performance/memory-overview, [Accessed 12-08-2024].

[34] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Sock: rapid task provisioning with serverless-optimized containers,” in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’18. USA: USENIX Association, 2018, p. 57–69.

[35] S. Shillaker and P. Pietzuch, “Faasm: Lightweight isolation for efficient stateful serverless computing,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 419–433. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shillaker

[36] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, “Sand: towards high-performance serverless computing,” in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’18. USA: USENIX Association, 2018, p. 923–935.

[37] “How Workers works · Cloudflare Workers docs — developers.cloudflare.com,” https://developers.cloudflare.com/workers/reference/how-workers-works/#isolates, [Accessed 11-08-2024].

[38] “Native Image — graalvm.org,” https://www.graalvm.org/latest/reference-manual/native-image/, [Accessed 02-01-2025].

[39] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 45–59.

[40] M. Brooker, A. C. Catangiu, M. Danilov, A. Graf, C. MacCárthaigh, and A. Sandu, “Restoring uniqueness in microvm snapshots,” *arXiv preprint arXiv:2102.12892*, 2021.

[41] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, “From warm to hot starts: leveraging runtimes for the serverless era,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 58–64. [Online]. Available: https://doi.org/10.1145/3458336.3465305

[42] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, “Unikernels: library operating systems for the cloud,” in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 461–472. [Online]. Available: https://doi.org/10.1145/2451116.2451167

[43] “gVisor — gvisor.dev,” https://gvisor.dev/, [Accessed 11-08-2024].

[44] “Fine-Grained Sandboxing with V8 Isolates — infoq.com,” https://www.infoq.com/presentations/cloudflare-v8/, [Accessed 26-10-2024].

[45] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with webassembly,” *SIGPLAN Not.*, vol. 52, no. 6, p. 185–200, Jun. 2017. [Online]. Available: https://doi.org/10.1145/3140587.3062363

[46] M. A. Vieira, M. S. Castanho, R. D. Pacífico, E. R. Santos, E. P. C. Júnior, and L. F. Vieira, “Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications,” *ACM Computing Surveys (CSUR)*, vol. 53, no. 1, pp. 1–36, 2020.

[47] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the linux virtual machine monitor,” in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[48] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici, “Unikraft: fast, specialized unikernels the easy way,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 376–394. [Online]. Available: https://doi.org/10.1145/3447786.3456248

[49] “V8 JavaScript engine — v8.dev,” https://v8.dev/, [Accessed 02-01-2025].

[50] S. Wang, “Thin serverless functions with graalvm native image,” Master’s thesis, ETH Zurich, 2021.

[51] “Standardizing WASI: A system interface to run WebAssembly outside the web – Mozilla Hacks - the Web developer blog — hacks.mozilla.org,” https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, [Accessed 28-12-2024].

[52] “Firecracker — firecracker-microvm.github.io,” https://firecracker-microvm.github.io/, [Accessed 11-01-2025].

[53] “firecracker/docs/snapshotting/snapshot-support.md at main · firecracker-microvm/firecracker — github.com,” https://github.com/firecracker-microvm/firecracker/blob/main/docs/snapshotting/snapshot-support.md, [Accessed 22-12-2024].

[54] “firecracker/docs/jailer.md at main · firecracker-microvm/firecracker — github.com,” https://github.com/firecracker-microvm/firecracker/blob/main/docs/jailer.md, [Accessed 11-01-2025].

[55] “Open Container Initiative - Open Container Initiative — opencontainers.org,” https://opencontainers.org/, [Accessed 11-11-2024].

[56] “Docker Hub Container Image Library — App Containerization — hub.docker.com,” https://hub.docker.com/, [Accessed 07-12-2024].

[57] J. Nickoloff and S. Kuenzli, *Docker in action*. Simon and Schuster, 2019.

[58] “mount(2) - Linux manual page — man7.org,” https://man7.org/linux/man-pages/man2/mount.2.html, [Accessed 11-01-2025].

[59] “Deprecated features — docs.docker.com,” https://docs.docker.com/engine/deprecated/#device-mapper-storage-driver, [Accessed 17-11-2024].

[60] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Slacker: Fast distribution with lazy docker containers,” in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, Feb. 2016, pp. 181–195. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter

[61] “GitHub - containerd/stargz-snapshotter: Fast container image distribution plugin with lazy pulling — github.com,” https://github.com/containerd/stargz-snapshotter, [Accessed 23-11-2024].

[62] “GitHub - google/crfs: CRFS: Container Registry Filesystem — github.com,” https://github.com/google/crfs, [Accessed 11-01-2025].

[63] “GitHub - firecracker-microvm/firecracker-containerd: firecracker-containerd enables containerd to manage containers as Firecracker microVMs — github.com,” https://github.com/firecracker-microvm/firecracker-containerd, [Accessed 15-12-2024].

[64] “GitHub - containerd/ttrpc: GRPC for low-memory environments — github.com,” https://github.com/containerd/ttrpc, [Accessed 04-01-2025].

[65] K. Indrasiri and D. Kuruppu, *gRPC: up and running: building cloud native applications with Go and Java for Docker and Kubernetes*. O'Reilly Media, 2020.

[66] "gRPC — grpc.io," https://grpc.io/, [Accessed 04-01-2025].

[67] "GitHub - opencontainers/runc: CLI tool for spawning and running containers according to the OCI specification — github.com," https://github.com/opencontainers/runc, [Accessed 04-01-2025].

[68] "firecracker/docs/mmds/mmds-user-guide.md at main · firecracker-microvm/firecracker — github.com," https://github.com/firecracker-microvm/firecracker/blob/main/docs/mmds/mmds-user-guide.md, [Accessed 04-01-2025].

[69] "The vHive Ecosystem for Serverless Systems Research — vhive-serverless.github.io," https://vhive-serverless.github.io/, [Accessed 26-12-2024].

[70] R. B. Roy, T. Patel, and D. Tiwari, "Icebreaker: warming serverless functions better with heterogeneity," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 753–767. [Online]. Available: https://doi.org/10.1145/3503222.3507750

[71] L. Cherkasova, "Improving www proxies performance with greedy-dual-size-frequency caching policy," 01 1998.

[72] Y. Yang, L. Zhao, Y. Li, S. Wu, Y. Hao, Y. Ma, and K. Li, "Flame: A centralized cache controller for serverless computing," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ser. ASPLOS '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 153–168. [Online]. Available: https://doi.org/10.1145/3623278.3624769

[73] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 152–166. [Online]. Available: https://doi.org/10.1145/3445814.3446701

[74] S. Ivanenko, V. Lanko, R. Horn, V. Jovanovic, and R. Bruno, "Hydra: Virtualized multi-language runtime for high-density serverless platforms," 2025. [Online]. Available: https://arxiv.org/abs/2212.10131

[75] A. Wang, S. Chang, H. Tian, H. Wang, H. Yang, H. Li, R. Du, and Y. Cheng, "FaaSNet: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute," in

2021 USENIX Annual Technical Conference (USENIX ATC 21). USENIX Association, Jul. 2021, pp. 443–457. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/wang-ao

[76] M. Brooker, M. Danilov, C. Greenwood, and P. Piwonka, "On-demand container loading in aws lambda," in *USENIX ATC 2023*, 2023. [Online]. Available: https://www.amazon.science/publications/on-demand-container-loading-in-aws-lambda

[77] "Amazon S3 - Cloud Object Storage - AWS — aws.amazon.com," https://aws.amazon.com/s3/, [Accessed 26-12-2024].

[78] "GitHub - containerd/nydus-snapshotter: A containerd snapshotter with data deduplication and lazy loading in P2P fashion — github.com," https://github.com/containerd/nydus-snapshotter/, [Accessed 06-09-2025].

[79] "Base - Ubuntu Wiki — wiki.ubuntu.com," https://wiki.ubuntu.com/Base, [Accessed 30-08-2025].

[80] "docker container commit — docs.docker.com," https://docs.docker.com/reference/cli/docker/container/commit/, [Accessed 11-11-2024].

[81] "Custom Resources — kubernetes.io," https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, [Accessed 21-06-2025].

[82] "design-proposals-archive/scheduling/scheduler_extender.md at main · kubernetes/design-proposals-archive — github.com," https://github.com/kubernetes/design-proposals-archive/blob/main/scheduling/scheduler_extender.md, [Accessed 29-06-2025].

[83] I. MinIO, "MinIO — S3 Compatible Storage for AI — min.io," https://min.io/, [Accessed 05-05-2025].

[84] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The design and operation of CloudLab," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 1–14. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/duplyakin

[85] D. Ustiugov, D. Park, L. Cvetković, M. Djokic, H. He, B. Grot, and A. Klimovic, "Enabling in-vitro serverless systems research," in *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless (WORDS 2023)*. ACM, 2023.