electronics

MDPI

Article

Enhancing Resource Utilization Efficiency in Serverless Education: A Stateful Approach with Rofuse

Xinxi Lu [1,*,†], Nan Li [1,†], Lijuan Yuan [2] and Juan Zhang [3]

[1] School of Software, Beihang University, Beijing 100191, China; 15131049@buaa.edu.cn
[2] Information Engineering College, Baoding University, Baoding 071000, China; yuanlijuan@bdu.edu.cn
[3] Department of Computer and Information Sciences, Northumbria University, Newcastle upon Tyne NE1 8ST, UK; juan.zhang@northumbria.ac.uk
* Correspondence: lxx@buaa.edu.cn
† These authors contributed equally to this work.

Abstract: Traditional container orchestration platforms often suffer from resource wastage in educational settings, and stateless serverless services face challenges in maintaining container state persistence during the teaching process. To address these issues, we propose a stateful serverless mechanism based on Containerd and Kubernetes, focusing on optimizing the startup process for container groups. We first implement a checkpoint/restore framework for container states, providing fundamental support for managing stateful containers. Building on this foundation, we propose the concept of "container groups" to address the challenges in educational practice scenarios characterized by a large number of similar containers on the same node. We then propose the Rofuse optimization mechanism, which employs delayed loading and block-level deduplication techniques. This enables containers within the same group to reuse locally cached file system data at the block level, thus reducing container restart latency. Experimental results demonstrate that our stateful serverless mechanism can run smoothly in typical educational practice scenarios, and Rofuse reduces the container restart time by approximately 50% compared to existing solutions. This research provides valuable exploration for serverless practices in the education domain, contributing new perspectives and methods to improve resource utilization efficiency and flexibility in teaching environments.

Keywords: serverless education; cloud-native; Kubernetes

Citation: Lu, X.; Li, N.; Yuan, L.; Zhang, J. Enhancing Resource Utilization Efficiency in Serverless Education: A Stateful Approach with Rofuse. *Electronics* **2024**, *13*, 2168. https://doi.org/10.3390/electronics13112168

Academic Editor: Carlo Mastroianni

Received: 29 April 2024
Revised: 28 May 2024
Accepted: 30 May 2024
Published: 2 June 2024

1. Introduction

The demand for computational resources has risen sharply in modern digital teaching and learning practices [1]. Educators and students increasingly require experimental environments that mirror the latest technological trends and simulate real-world challenges. Educational institutions should offer computational resources adaptable to various teaching activities. These resources should support teaching needs ranging from basic programming to advanced data science projects, catering to different scenarios like laboratory exercises, assignments, research, and remote learning. Each scenario presents unique requirements for technical support, thereby placing greater demands on the flexibility and responsiveness of the computing platform.

Moreover, the uncertainty of teaching activities necessitates that the management of computational resources be adaptable to changes in demand. For example, there may be a need to quickly provide independent experimental environments for numerous students or to suddenly allocate additional resources for processing large data sets in research projects. This need for rapid deployment, scalability, and cost-effectiveness imposes higher demands on cloud computing platforms tailored to educational practice scenarios.

In the early stages of cloud computing technology development, the construction of cloud platforms in education primarily relied on virtual machine technology, which

facilitated teaching and research. More specifically, virtual machine technology enabled the simulation of multiple independent operating systems on a single physical machine, reducing the need for additional hardware [2]. However, virtual machine solutions have notable limitations. The inclusion of a virtualization layer in virtual machine technology leads to inflexibility in resource management and performance degradation during system migration [3]. As teaching needs grow and diversify, and as learning methods evolve, traditional virtual machine technology fails to meet the demands for efficiency, flexibility, and simplicity in experimental environments. Therefore, exploring an efficient, scalable, and easily manageable alternative to virtual machines has become a key focus in cloud computing and educational technology.

Container technology, which has emerged in the past decades, is a lightweight operating system-level virtualization technology [4]. It enables software applications and their dependencies to run in isolated environments, independent of underlying resources [5]. Compared to traditional virtual machines, containers offer a higher-level operating system abstraction, where containers on the same host are isolated from each other while sharing the host's operating system kernel. This structure minimizes the redundancy of multiple operating system instances, significantly reducing resource consumption while facilitating faster startup times and more flexible deployment methods. The applications and their dependencies inside containers are packaged together, which greatly simplifies the configuration and management processes for experiments and eliminates the cumbersome installation and environmental discrepancies associated with virtual machines. These features make containers an ideal solution for the fast, dynamic, and continuous demands of modern educational practices. Emerging container management and orchestration tools such as Kubernetes [6] can further enhance the practicality of containers in large-scale deployments, ensuring effective resource allocation and fault recovery capabilities, making it possible to build a stable, scalable, and easily manageable educational experimental platform.

Kubernetes, a leading container orchestration platform, offers a range of management functions. However, the complexities of managing and utilizing resources in educational practice environments underscore the challenges of this platform. For example, the duration of a database experiment may be around 2 weeks. To ensure availability, the container needs to keep running during this period. Students may only need to spend a few hours using the experimental environment to complete the experiment within the 2 weeks. However, to ensure availability, the container must run continuously for about 2 weeks, which means that the container remains idle for most of the duration, causing a significant waste of resources. To solve this problem and achieve on-demand provision, the serverless mechanism can be considered to manage the container resources.

Serverless is a brand-new cloud computing model [7]. Under the serverless development paradigm, developers do not need to worry about the number and configuration of servers, or the details of specific networks, storage, and other deployment tasks. They only need to focus on the business logic of applications, as other tasks are automatically handled by the serverless platform [5]. The biggest feature of serverless is to provide services on demand. More specifically, it will be actively shut down when the container is not requested. However, when the container is requested again, it will restart on demand. Therefore, this shutdown/restart mode can minimize the waste of computational resources caused by container idling.

In traditional serverless architectures, services are usually stateless, meaning each request to a container receives a brand-new execution environment without the need to retain contextual information, enabling the service to achieve rapid scaling and flexible deployment. However, this stateless feature is not entirely applicable to educational practices, because educational experiments often need to save students' operational states and experimental data for recovery and reference in subsequent learning and experimentation processes. Moreover, when students use the experimental environment for course learning,

they may casually experiment and frequently change configurations, which can easily produce certain states that need to be managed and optimized by the cloud service platform.

On the other hand, compared to professional developers, students use standardized patterns less frequently. Consequently, experimental environments often need to adapt to irregular usage frequencies and times while maintaining certain states. This requires stateful serverless services to retain necessary state information while performing on-demand operations and managing resources dynamically. In other words, these services should quickly respond to experimental needs when containers are active and can actively shut down when containers are inactive to save resources. This makes the design and implementation of stateful serverless services more complex than those of simple stateless serverless services.

This research aims to explore and implement a stateful serverless mechanism to support the needs of state persistence and dynamic resource management in educational practices. The main contributions of this research are summarized as follows:

1. We propose a stateful serverless mechanism that addresses the challenges of resource wastage and state persistence in educational practice environments. This mechanism is built on Containerd and Kubernetes and focuses on optimizing the startup process for container groups.
2. We introduce the concept of container groups, which captures the characteristics of numerous similar containers on the same nodes in educational practices. Furthermore, we design and implement the Rofuse optimization mechanism, employing delayed loading and block-level deduplication techniques to enable the containers within the same group to reuse the cached filesystem data, significantly reducing container restart latency.
3. We conducted comprehensive experiments to evaluate the performance and robustness of our proposed stateful serverless mechanism and the Rofuse optimization. Our results demonstrate that Rofuse reduces the container restart time by approximately 50% compared to existing solutions, such as Dmigrate [8], and exhibits better adaptability to resource-constrained educational environments.

The structure of this paper is as follows. In Section 1, we introduce the background and main work of this paper. In Section 2, we discuss related work. Section 3 presents the base framework; this framework is built on Kubernetes and Containerd and is designed to achieve automatic saving and recovery of container states. In Section 4, we introduce the Rofuse optimization mechanism, which is tailored for educational practice scenarios and aims to optimize the restart time of containers in the base framework. In Section 5, we conduct extensive experimental validation to demonstrate the effectiveness of the Rofuse optimization mechanism and its advantages compared to existing solutions.

2. Related Work

Serverless was initially designed to be stateless, meaning that the serverless platform does not actively save the runtime state of function instances. Here, state refers to the side effects generated during the process execution, including CPU, memory state, network state, changes to the filesystem, etc. The current function instance will not be affected by the previous instance, nor will it affect the next function execution. Moreover, multiple function instances running simultaneously will not interfere with each other. This stateless nature ensures that function instances can be easily scaled vertically or horizontally, enabling on-demand scaling capabilities.

However, in practical applications, purely stateless programs are rare, as most programs need to maintain their states during execution for coordination among internal components or interaction with external systems. There are two different scenarios where state-sharing occurs between function instances. In the first scenario, state-sharing occurs between two different function instances that exist simultaneously on the serverless platform, and these function instances can achieve mutual function invocation through methods such as RPC. In the second scenario, state-sharing occurs between function in-

stances created at different times, meaning that each time a new function instance starts, it depends on the state generated by previously launched instances.

Regarding state-sharing between different function instances, the state transfer inevitably involves performance overhead from state serialization, transmission, and deserialization. The academic community has proposed several solutions to minimize this additional overhead. Zhipeng Jia et al. proposed a serverless runtime system called Boki. This system provides an abstraction called LogBook, which allows state-sharing between function instances through a shared log API. The LogBook enables function instances to communicate and share state information, facilitating coordination and collaboration in a serverless environment while preserving the original semantics of the paper [9]. Zhang et al. built a low-latency, multi-tenant cloud storage system named Shredder, which allows direct execution of small computational units within storage nodes. It uses local code to manage data storage and network paths, ensuring performance; it interacts with data using a V8-specific intermediate representation to avoid expensive cross-protection domain calls and data copying [10]. Daniel et al. proposed a stateful JVM-based serverless programming framework called Crucial, which introduces a distributed shared memory layer in serverless and uses RDMA for inter-node communication to minimize the performance overhead of state exchange between function instances [11]. Sreekanti et al. proposed a stateful serverless platform called Cloudburst, which uses an auto-scaling key–value store for efficient state-sharing [12]. Simon et al. proposed a WebAssembly-based stateful serverless framework called Faasm, which introduces a more lightweight memory isolation scheme and enables state-sharing between different function instances through shared memory [13].

The above research solutions mostly explore the implementation of stateful serverless based on specific language runtimes. For example, Crucial [11] is built on JVM; Shredder [10] is built on the V8 engine; Faasm [13] is implemented based on WebAssembly. Moreover, most state-sharing schemes require the applications deployed on them to modify their code using the platform's proprietary APIs, which brings significant invasiveness to user-submitted tasks. For instance, both Cloudburst [12] and Faasm [13] can only be used normally with the help of platform-specific APIs. In educational practice scenarios, various programming languages, tools, and technology stacks are often involved, thus requiring more diverse containerization needs for the underlying runtime environment. Current solutions cannot be applied to stateful serverless frameworks in educational practice scenarios.

In regard to state-sharing between function instances starting at different times, current research studies provide the following solutions. Azure Functions offers the 'Durable Functions' feature, which allows developers to create stateful functions in a serverless environment by using event source state management and checkpoints to maintain state. This enables developers to write long-running, stateful workflows. Jonas et al. proposed a Python programming framework called PyWren, which provides a lightweight interface for managing memory, storage, and network resources. It achieves state-sharing between different function instances by reusing the same batch of containers to execute them [14]. Zhang et al. introduced a program execution framework in Picocenter that is suitable for programs that need to exist for a long time but are mostly idle. This framework preserves the function instance state by saving and restoring program state and filesystem changes [15].

Among the above research solutions, Durable Functions and PyWren [14] are only applicable to specific programming languages, and the 'Durable Functions' feature is tightly bound to the Azure platform. Although Picocenter [15] provides a solution for saving and restoring program state, its method for migrating container filesystem changes is relatively inefficient and struggles to adapt to the need for flexible and rapid deployment required in serverless environments.

3. Base Framework

In this chapter, the base framework for container state checkpointing and restoration is introduced for our work.

3.1. Challenges

Before we explore the stateful serverless mechanism suitable for educational practice environments, we need to choose an appropriate base framework. Kubernetes is an open-source container orchestration system that enables the automatic deployment, scaling, and management of containerized applications, offering high scalability and flexibility. Its rich feature set includes service discovery and load balancing, storage orchestration, automated deployment, rolling updates, automated resource management, and self-healing, greatly simplifying the complexity of container management and microservice architecture. Therefore, we selected Kubernetes as the foundation for our research.

Since Kubernetes adopts a modular design, any container runtime that implements the CRI (container runtime interface) can theoretically work with Kubernetes. We chose Containerd, the most widely used container runtime, for our research. Containerd provides the necessary functionality to manage the entire lifecycle of containers, including container creation, execution, pausing, resuming, and destruction. It also supports standard container images, allowing image pulling and storage operations. At the same time, Containerd is more concise and stable when compared to traditional Docker tools, making it very suitable for secondary modifications to facilitate research work. Unless otherwise specified, "Kubernetes" in this paper refers to "Kubernetes with Containerd as the container runtime".

Container runtimes (including Containerd) that work with Kubernetes commonly adopt union filesystems [16]. This technology divides the container filesystem into two parts: (1) the lower read-only layers stacked by image layers, and (2) the upper read–write layer. Changes in the filesystem during its execution normally happen in the upper read–write layer. The read–write layer is deleted simultaneously when the container is deleted. This design means that the changes in the filesystem made by the container during its execution, along with the container's CPU state, memory state, etc., will be deleted when the container is deleted. It is viable for stateless applications, as each invocation is independent and does not require saving the previous state.

However, state management and persistence become key challenges in stateful serverless applications for educational practice environments. The serverless framework typically destroys containers when they are no longer needed to release resources for other tasks and it quickly creates new container instances when the next request arrives. In this process, if the container state is not properly handled, the intermediate results from the previous run, user session data, and even some running configurations of stateful services will be lost, further affecting service continuity and user experience.

For example, in online programming environments used for educational practices, the state of code writing and testing during students' interactive sessions must be preserved so that they can seamlessly resume their work in a serverless environment. However, existing container runtimes and Kubernetes platforms do not directly support such scenarios, offering only some compromise solutions.

The most common solution involves using external storage services (including remote storage, databases, etc.) to maintain important state information [17]. For example, one approach is to mount paths in the container where filesystem changes occur as NFS paths, and then remount the same paths when the container is restarted later; alternatively, the program logic inside the container could be modified to write important data to an external database. However, both approaches present significant limitations in educational practice environments.

Using remote storage mounting as an example, paths that require write operations at the beginning of container startup should be identified, and these paths should be properly initialized in remote storage. However, the container environments required by teachers and students vary significantly, making it impossible to preset fixed writable paths. Moreover,

educational environments have unique characteristics: novice students may write or modify files in any path that is not predefined, making it impossible to predict all potential write paths in advance. As for writing state to a database, it inevitably requires changing the logic of the program inside the container, which contradicts the goal of educational practices that aim for generality and ease of use. Furthermore, external storage services can only ensure the persistence of the filesystem state, but cannot maintain CPU and memory states.

To overcome these difficulties, stateful pod features of Kubernetes are used for solutions, such as StatefulSet [18]. However, StatefulSet of Kubernetes does not support pause and resume operations, causing the container state to remain valid only within its current lifecycle. The capabilities of the serverless architecture cannot meet the on-demand service. Therefore, we propose a new container state checkpoint/restore framework to innovatively modify Containerd, which is a core component of Kubernetes. This framework can effectively realize container state checkpointing and restoration without interfering with the existing work on upper-layer components of Kubernetes.

3.2. Implementation Principle

In Linux, a container is essentially a group of isolated processes with limited resource usage. Therefore, checkpointing and restoring a container essentially means checkpointing and restoring a group of Linux processes. Since the process group of the container is completely isolated from other processes on the host machine through namespaces, it is entirely feasible to fully save a group of container process information without affecting other processes on the host.

During the checkpointing and restoration of a container, the state of the container primarily involves the memory of the process, CPU state, list of open file descriptors, environment variables, and other aspects. This information can be managed using the CRIU (checkpoint/restore in user space) tool. CRIU operates in the user space and primarily reads the process state information from the proc filesystem, controls process execution through the ptrace system call, and obtains and sets its registers to complete the saving of the process state. The process state information is saved by CRIU into IMG format files and then stored on the disk.

The execution of a process depends on a specific filesystem state, and the consistency of the filesystem needs to be ensured before and after process checkpointing and restoration. In a Kubernetes cluster, the checkpointing and restoration of stateful containers may occur on different nodes. Therefore, the migration of the container filesystem between different nodes should be completed while saving and restoring the critical runtime states of the process, such as memory and CPU.

The filesystem of a container is normally built in layers by a union filesystem [19]. First, the image layers of the container are stacked into read-only layers. Then the container runtime generates a temporary read–write directory mounted on top of the read-only layers of the image. Changes made by the container to the filesystem during its execution only occur in the topmost read–write layer. When checkpointing the filesystem of the container, the read-only layers can be directly migrated through images. In contrast, the read–write layer should save its contents to an archive file when checkpointing the container, such as the container runtime's changes to the filesystem. When restoring the container, the archived file is rewritten into a new read–write layer. The checkpointing and restoration processes of the filesystem migration methods are described in Figure 1.

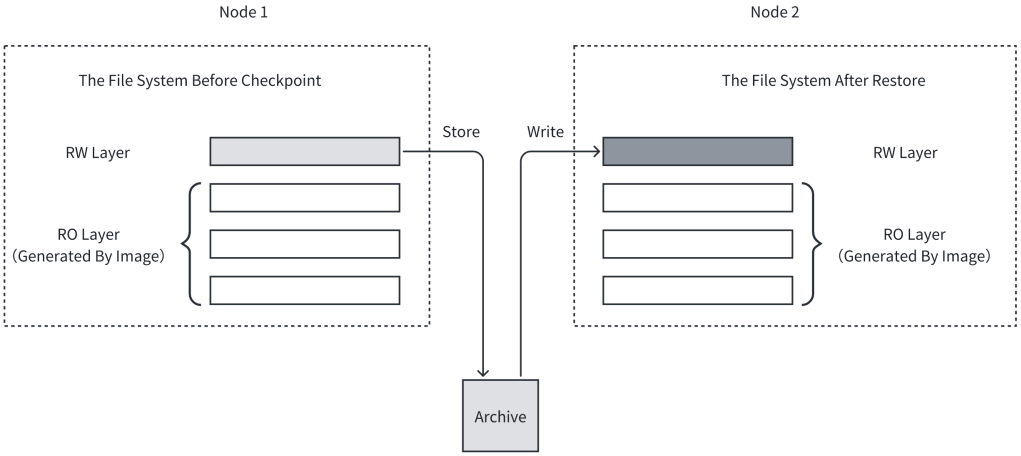

Figure 1. Filesystem migration methods during container state checkpointing and restoration processes.

3.3. Design

The CRI (container runtime interface) is proposed to enable Kubernetes to adapt to various container runtimes. Any container runtime that implements the CRI can be integrated into the Kubernetes ecosystem. The CRI mainly includes interfaces at the sandbox (PodSandbox) and container levels, where the sandbox corresponds to the concept of a Pod in Kubernetes.

When creating a Pod, the Kubelet on the node first invokes the "create sandbox" and "run sandbox" interfaces in CRI. As a response, the CRI service of Containerd correspondingly creates and runs a pause container to complete the sandbox creation. Subsequently, the Kubelet calls the "create container" and "run container" interfaces in sequence to create and run the actual main container. When deleting a Pod, the Kubelet on the node first calls the "stop container" and "delete container" interfaces in sequence to stop and delete the containers in the Pod. Among them, the CRI service of Containerd sends a stop signal (usually the SIGTERM signal) to the process group corresponding to the container to stop it and reclaim related resources when responding to the "stop container" interface. Finally, the Kubelet calls the "stop sandbox" and "delete sandbox" interfaces in sequence to ultimately remove the Pod resources on the node.

To facilitate the checkpointing and restoring of stateful containers, our framework mainly works on the interaction between Kubelet and Containerd on Kubernetes nodes. Specifically, this framework registers related hook functions in the CRI service of Containerd to monitor the CRI calls of Kubelet for creating and stopping containers; it modifies the behaviors of Containerd when creating and stopping containers. The architecture diagram of this framework is shown in Figure 2.

In this framework, the added component "snapshot manager" is responsible for maintaining the metadata information of the framework generated during the container checkpointing process. It contains a database service that stores information such as whether a container with a certain identifier has a snapshot, the historical versions of the snapshot, the storage location of the snapshot, etc. Moreover, it includes a simple HTTP service that can interact with the database service and provide necessary storage and query interfaces externally.

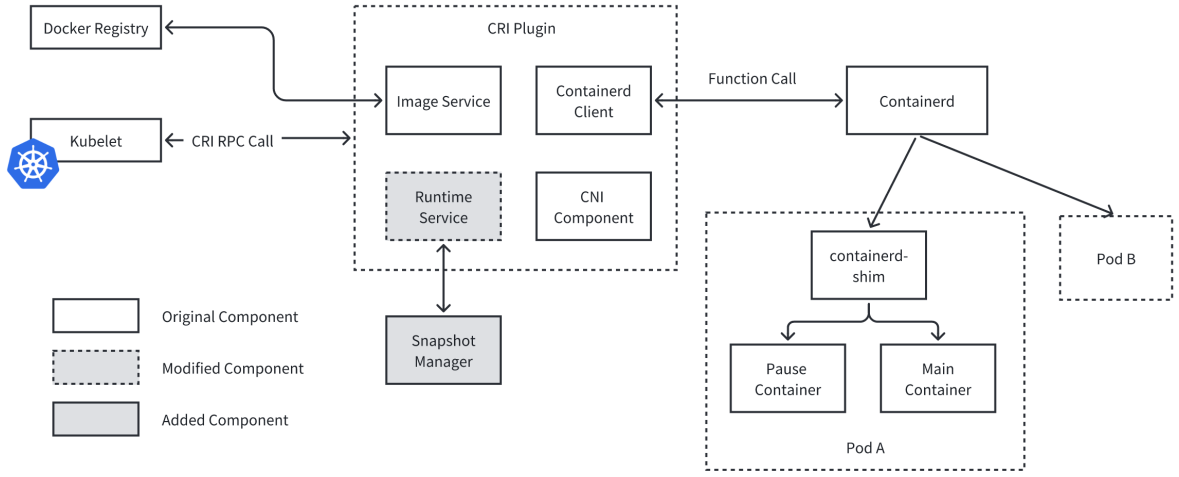

Figure 2. The architecture of the proposed framework. The proposed framework modifies the runtime service of the Containerd CRI plugin and adds the snapshotter manager component.

The modifications to the "runtime service" mainly comprise a series of hook functions, which include the following:

- "Checkpointing Function". This function listens to the "stop container" interface in the CRI service. It can create checkpoints of the CPU and memory states of the container process group using CRIU when Kubelet deletes a container, resulting in a container snapshot that is stored on disk. At the same time, it saves the read–write layer of the container filesystem to an archived file and writes it to the disk. This function stores the result after the checkpoint (including the name of the container at the checkpoint, the path of the files after the checkpoint, etc.) in the "snapshot manager".
- "Creating Function". This function listens to the "Create Container" interface in the CRI service. It allows for the preparation of container snapshot files locally by querying the "snapshot manager" when Kubelet creates a container; moreover, it pulls the archived file of the read–write layer of the previous container's filesystem from a remote service, and writes it into the read–write layer of the new container's filesystem.
- "Starting Function". This function listens to the "Start Container" interface in the CRI service. When Kubelet creates a container, it modifies the default behavior of Containerd, enabling Containerd to use CRIU to restore the container process group from the container snapshot.

4. Rofuse

Based on the base framework, it is worth noting that container restart time is relatively long, which seriously affects the user experience of the framework. To address this issue, we propose the Rofuse optimization mechanism to minimize the restart time of the container.

4.1. Challenges

The serverless platform can elastically scale container services and even destroy and recreate containers on demand, which means that the interaction and application performance should be consistent and stable, regardless of changes in resource allocation and management behind the scenes of the serverless platform. Therefore, this process should be transparent to users. When requesting services, users should not experience delays or interruptions, even if the container was previously deleted due to long-term activity.

Because of the above requirements, minimizing the time for a container to restart is essential to ensuring user experience. Users may encounter noticeable service delays if it takes a long time to recreate and start a container, which affects the user experience.

The restart time for stateless containers is primarily spent on pulling the image. However, the time consumed on the old container downloading the read–write layer archive file must also be considered for the base framework. The read–write layer of the container

continuously writes new data throughout the lifecycle. As the file size in the read–write layer increases, the size of the read–write layer archive file during container checkpointing also increases, and, accordingly, the container restart time increases linearly. Therefore, optimizing the restoration time of the read–write layer in legacy containers is crucial for minimizing overall container restart durations in stateful serverless architectures.

Optimizing the time required to restore the read–write layer of the old container is tantamount to enhancing the efficiency of container filesystem migration. Zhang et al. proposed a container filesystem migration scheme using btrfs in Picocenter [15], which compares the btrfs snapshots before the container runs and during the persistence phase to identify the changes made to the filesystem by the container. When the container restarts, only these changes need to be transferred. This scheme is essentially the same as the old container in the base framework that only transfers the read–write layer. Moreover, it cannot significantly reduce the amount of filesystem information that needs to be loaded during container restart. Nadgowda et al. proposed an on-demand loading scheme based on NFS in Voyager and used a background process to synchronize the container filesystem from NFS to the local system to reduce network overhead [20]. Dmigrate, proposed by Wang et al., introduced the concept of "file sets" based on the Voyager scheme and uses a multi-priority transmission scheduling parallel algorithm to improve container performance during container filesystem migration [8].

It can be observed that existing research on optimizing container filesystem migration generally treats containers on the same node as independent entities, ignoring the potential similarities between containers [8,15,20]. However, in educational practice scenarios, there are usually a large number of "similar containers" on the same host node. These containers may come from the same experimental course, be generated based on the same initial image, and execute similar program instructions. There are many duplicate files across the filesystems of these "similar containers." Under existing schemes, these duplicate files are repeatedly downloaded when a container restarts, incurring additional time and bandwidth overhead. Therefore, how to utilize the commonality between these "similar containers" and design an efficient filesystem sharing and reuse mechanism presents a challenging problem.

4.2. Solutions

Based on the above problems, we propose the container restart optimization mechanism Rofuse for educational practice scenarios. Rofuse can effectively reduce the container restart time by optimizing the recovery time of the container filesystem through on-demand loading and file chunking deduplication mechanisms to achieve superior performance than existing solutions.

4.2.1. On-Demand Loading Based on Union Filesystem

In the base framework, the recovery of the container filesystem includes two parts: (1) fully downloading and decompressing the container's initial image; (2) fully downloading the read–write layer archive file of the old container, decompressing it, and writing it to the read–write layer of the new container's filesystem. Harter et al. demonstrated in Slacker experiments that only 6.4% of image data are used during container startup [21], so it is unnecessary to fully download these files of the two parts at container startup. Therefore, Rofuse constructs a remote filesystem to implement on-demand loading of the container filesystem after restart. The working diagram of this remote filesystem is shown in Figure 3.

During the container restart process, Rofuse mounts the filesystem of the old container remotely and integrates it into the corresponding location of the new container, as shown in Figure 3. In this operation, there is no need to download all the data from the old container's filesystem, except for metadata. When the container attempts to read files from the old container's filesystem, these read requests are directed to the remote filesystem. The remote filesystem interprets the target file's name, offset, and length information in these read requests and fetches the corresponding data from the remote storage server.

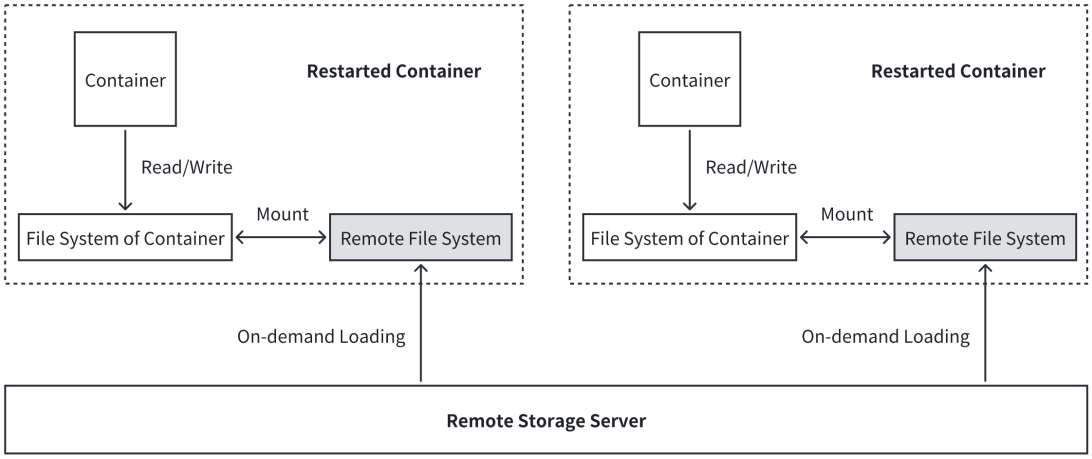

Figure 3. The process of supporting on-demand loading of the filesystem for containers via a remote filesystem.

Because of the diverse types of containers deployed by teachers and students in the teaching practice and the unpredictable file paths generated during their operation, the mount point location of the old container's filesystem in the new container cannot be known in advance. To ensure that the restarted container can correctly forward read and write requests from the filesystem of the old container to the remote filesystem, Rofuse needs to adopt a union filesystem. The process is shown in Figure 4.

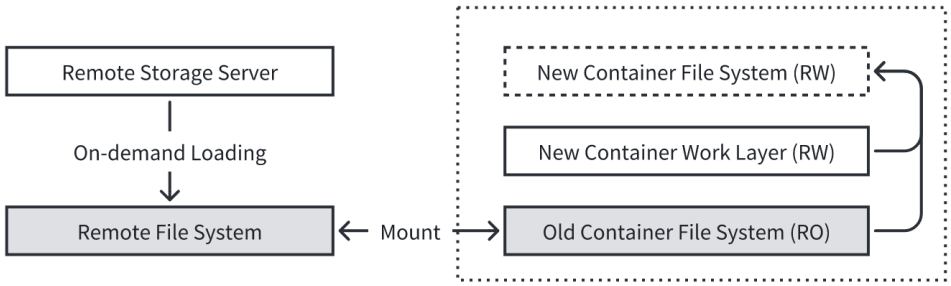

Figure 4. The process of Rofuse using a union filesystem to provide containers with a complete view of the filesystem.

A unified filesystem can mount multiple filesystems located at different physical locations into a single virtual file tree to form an integrated view that achieves seamless overlaying of files and directories. As described in Figure 4, Rofuse divides the filesystem of a new container into two parts, where the base layer is the filesystem of the old container and is set as read-only mode. However, the top layer, dedicated to the new container, possesses read–write capabilities. Rofuse uses the unified filesystem technique to merge the two layers of filesystems into one and mount it under the root filesystem directory of the new container to build a complete filesystem with read–write functionality. This filesystem hierarchy remains completely transparent to the container, ensuring that the container processes perceive only a complete and unified root directory. When a container process attempts to read a file, the read request will be passed down to the "filesystem layer of the old container" and ultimately loaded on demand through the remote filesystem.

A concrete example is depicted for better understanding. For example, a container has written a file named file-1 to its read–write layer before checkpointing. Once the container is restarted, file-1 should be in the "old container filesystem" layer, as shown in Figure 4. If the restarted container attempts to read file-1, the unified filesystem will first examine the contents of the "new container work" layer. Since the "new container work" layer is empty in its initial state, the unified system will search downward until it reaches the "old

container filesystem" layer. Then the remote filesystem will respond to the read request and retrieve the required data from the remote storage server.

As for write operations, when attempting to write files after container restoration, Rofuse provides copy-on-write capabilities. For example, suppose a restarted container tries to modify file-1. In that case, the unified filesystem will read file-1 from the remote filesystem, copy it into the "new container work" layer, and subsequently execute write operations on file-1.

Since the remote filesystem is designed to be read-only, Rofuse is particularly well-suited for introducing caching mechanisms and implementing "prefetching" strategies. When a file is frequently read, Rofuse can pre-store parts or all of the file's contents, including data and metadata, in the local cache. Therefore, the number of requests and latency responses for future remote storage resources can be significantly reduced. This caching strategy can greatly improve the execution efficiency of system calls such as read, stat, and getxattr.

Additionally, based on the principle of locality, when a process accesses a specific data segment in a remote file, it often requests adjacent data segments. Thus, in addition to on-demand retrieval, Rofuse will intelligently download data from the surrounding areas of the requested point, achieving pre-loading of nearby block contents. This method can significantly improve the file reading speed and accelerate data processing.

4.2.2. Block-Level Deduplication of Old Container Filesystems

In educational practices, it is common that many "similar containers" appear on the same node. "Similar" means that these containers are generated from the same image, have similar operational goals, and execute similar program instructions after starting. Taking a database experiment as an example, each student will obtain a MySQL container, and all containers are created based on the same version of the MySQL image. During the experiment, students usually need to follow the same instruction manual to operate, such as entering the same or similar data, and writing and executing similar SQL statements for CRUD operations. To describe the "similar containers" in a Kubernetes cluster of an educational scenario, the concept of a "container group" will be introduced in this paper.

Containers within the same container group often store a large amount of duplicate files in their "old container filesystems" at the time of checkpointing. This duplication occurs because these containers are derived from the same image and files written by the containers often overlap. When these containers restart, containers on the same node may repeatedly need to obtain the same files through a remote filesystem. If these duplicate files can be shared within the container group, the average startup latency of containers can be reduced and the bandwidth pressure on remote storage servers in large-scale restart scenarios can be alleviated. However, file sharing may not be sufficient to eliminate duplicate data in some cases. For example, although students initially use uniform initial data provided by the teacher during container usage, they often make personalized edits and modifications during the experiment, resulting in differences in files, which makes it impossible to share these partially modified files within the container group after restarting.

Based on the above discussion, We propose a file block-level deduplication mechanism, in which the container's filesystem is split into multiple data blocks instead of directly storing complete files. This mechanism allows block-level data sharing among containers within the same container group, effectively reducing data redundancy and optimizing container restart latency. Figure 5 shows the block-level deduplication and data sharing of the container filesystems, where container-1 and container-2 belong to the same container group after restarting on the same node.

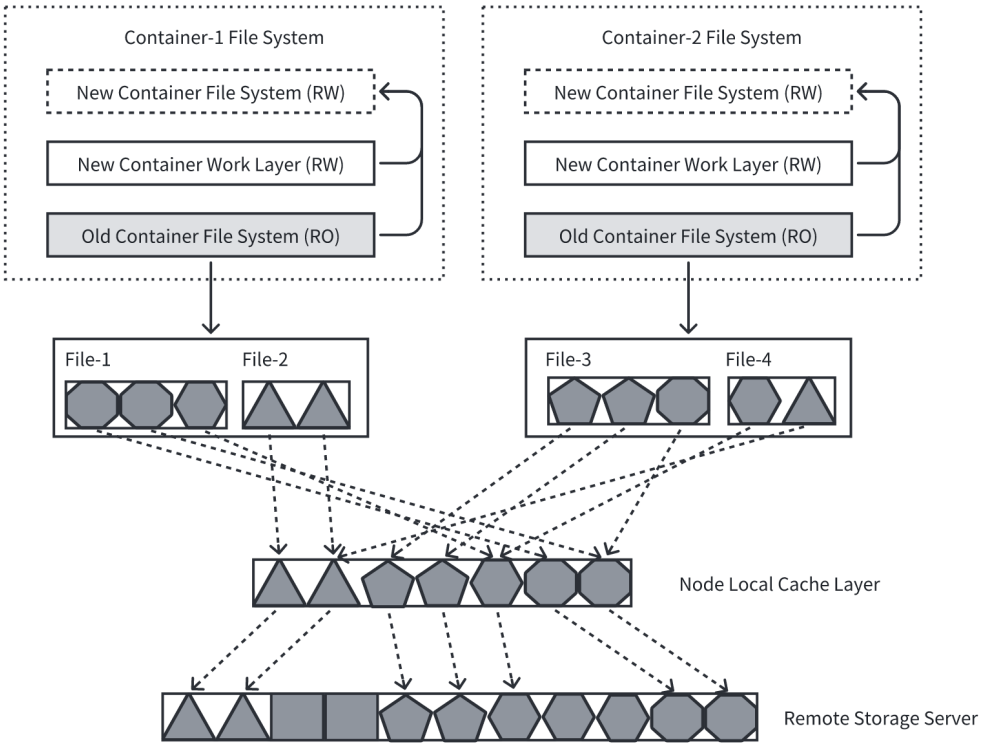

Figure 5. The process of block-level data sharing between containers on the same node within a container group. We use different shapes to represent data blocks. These shapes do not have specific meanings but are used to indicate the degree of similarity in data content across different files. Similar usage of data block shapes will apply in subsequent figures in this paper.

In Figure 5, the files in the "old container filesystem" layer of container-1 and container-2 have undergone a chunking process. These data blocks are then hashed, and the blocks with the same hash value are stored and shared in the local cache layer of the node, which is adaptable for a CAS (content-addressable storage) strategy using the hash value of the data block to determine its storage location. Across different nodes, the information of the data blocks in the cache layer originates from the remote storage server. The specific working mechanism of this process will be discussed in detail in the next section.

4.3. Design

4.3.1. Overview

Rofuse consists of three main components, as shown in Figure 6:

1. Data block metadata manager. This component operates at the cluster level and manages the data blocks throughout all container files. It records the container details for data block creation and tracks all containers that reference these blocks. Therefore, this metadata manager handles the ability to delete blocks when their reference count drops to zero. This precise management can ensure the efficiency and orderliness of the data block lifecycle.
2. Remote storage server. This component also works on the cluster level and is responsible for storing the data blocks of all container files. In this paper, we use NFS as the specific implementation of the remote storage server.
3. Rofuse remote filesystem. The remote filesystem is a read-only FUSE filesystem. Each restarted container corresponds to an instance of the remote filesystem.

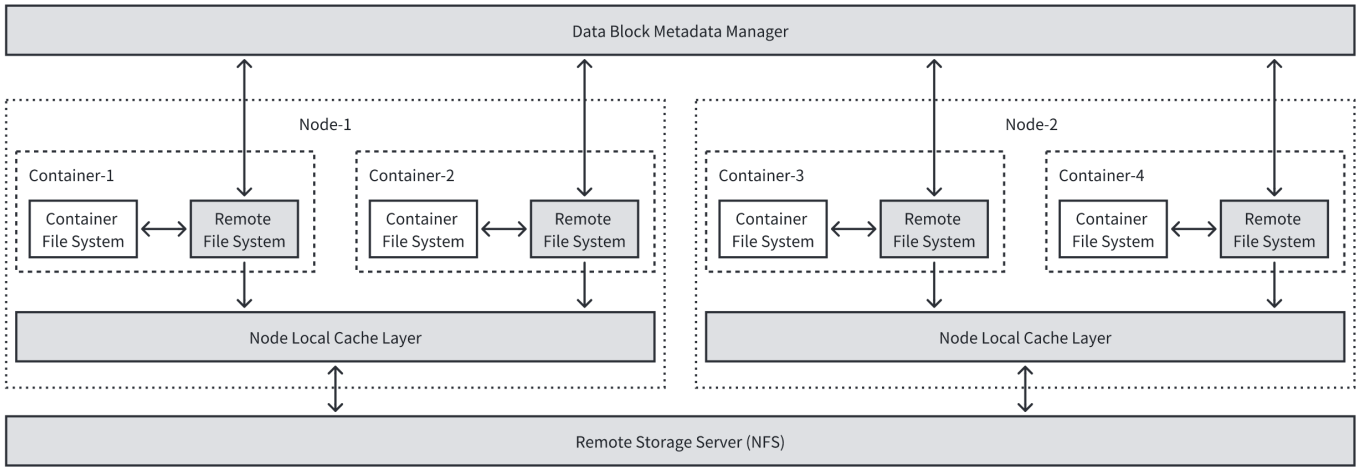

Figure 6. The architecture of Rofuse. The Rofuse consists of three components: the data block metadata manager, the remote storage server, and the Rofuse remote filesystem.

4.3.2. Rofuse Remote Filesystem

The Rofuse remote filesystem is a read-only FUSE filesystem. It is the core component of Rofuse. Its architectural diagram is shown in Figure 7. Upon container restart, an instance of the Rofuse remote filesystem is generated. This instance mounts itself into the "old container filesystem" layer through the kernel FUSE module. Consequently, read operations initiated from the union filesystem toward the old container files are redirected to the Rofuse remote filesystem.

In addition to the kernel FUSE module, the Rofuse remote filesystem also includes a FUSE server running in the user space. The FUSE server makes an open system call to the device file /dev/fuse at startup so that a unique FUSE handle can be obtained. The FUSE server listens to the changes in the contents of the FUSE file through the FUSE handle, from which it reads and parses file operation requests from the kernel FUSE module. For read-type operation requests, the FUSE server pulls the corresponding data on-demand from the node's local cache layer or remote storage server (NFS) and then writes it back to the kernel FUSE module through the FUSE handle.

Files in Linux contain two parts: file metadata and file data. File metadata include the file's creation date, author, size, file type, and other attributes that can help build a complete filesystem structure. When mounting the "old container filesystem" layer, the union filesystem frequently performs system calls such as stat and getxattr to obtain the organizational structure of the underlying filesystem. If file metadata are also stored remotely and loaded on-demand, the Rofuse remote filesystem would need to frequently pull file metadata over the network at startup, which incurs significant performance overhead. On the other hand, file metadata are usually very small, and even if downloaded in full before a container restart, it would not result in a noticeable time cost. Therefore, Rofuse separates the file metadata from file data to ensure that the Rofuse remote filesystem can obtain the complete file information before startup while implementing on-demand loading for file data.

In Rofuse, the separation of the metadata and data segments of files is completed after the state checkpointing of the container, which will be discussed in detail later. Once the separation is completed, an archive file (in TAR format) containing the metadata of all files in the old container's filesystem and a set of data blocks are obtained. This set of data blocks is written to the remote storage server (NFS) in a CAS manner, and then the metadata archive file is uploaded to the Docker Registry as part of the container snapshot image. Before startup, the FUSE server pulls the metadata archive file from the remote server and decompresses it. By traversing the metadata of all files after decompression, the FUSE server can build a file tree of the old container's filesystem, in which each leaf node corresponds to a non-directory file. More specifically, each leaf node contains the

file type, size, and other attributes, and the file tree resides in the memory of the FUSE server. When the FUSE server receives stat, open, and other requests forwarded from the kernel FUSE module, it can quickly search the corresponding file node in the file tree and respond accordingly.

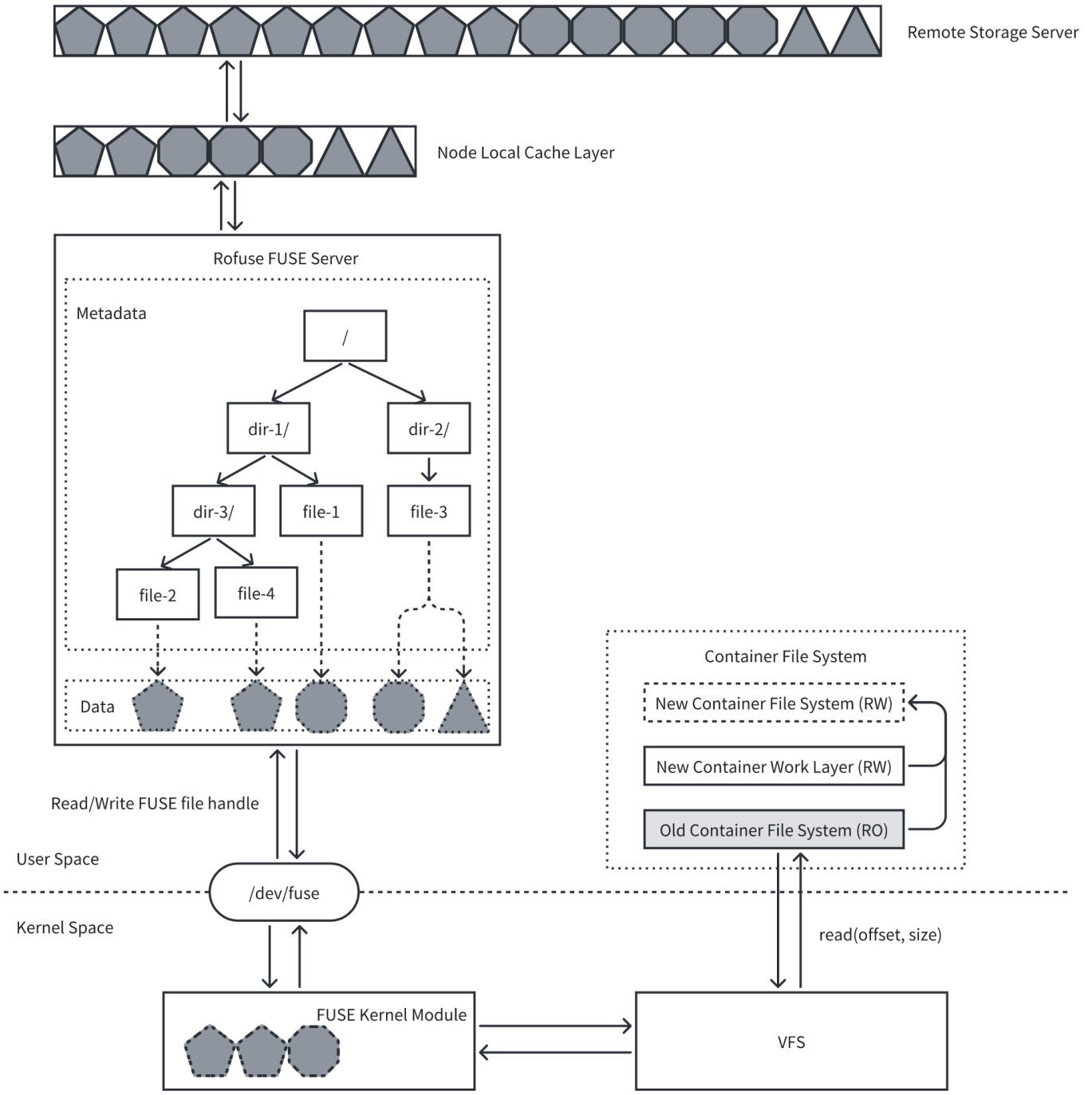

Figure 7. The architecture of the Rofuse remote filesystem. Rofuse remote filesystem is a fuse-type filesystem that runs in the user space and receives the IO requests from the kernel fuse module via the /dev/fuse file handle.

For "regular files", the leaf nodes of the file tree store a list of metadata corresponding to file data blocks, where the primary piece of the metadata is the hash value of the data block. Since data blocks are stored using CAS, Rofuse can directly derive the storage path

of the data block based on its hash value and read out its contents. Specifically, as for reading requests received for regular files, the FUSE server performs the following steps:

1. Parses the read request obtained from the FUSE handle to obtain the information of the target file, including the inode number, the offset position where the read request starts, and the number of bytes to be read.
2. Finds the target file node in the file tree and reads out the node information to obtain the list of data blocks involved in this read operation from its data block metadata list.
3. Performs a read operation on each data block in the data block list. Let the data block that needs to be read in a certain iteration be *Block*.
4. Searches for the *Block* in the local cache layer of the node based on its hash value. Since the node's local cache layer uses CAS storage, this search can be simplified to the file reading after concatenating the path.
5. If step 4 fails to find the *Block*, the search occurs in the remote storage server (NFS). Since NFS provides complete POSIX file operation semantics, the concatenated path can be directly used to search within the local mount point of NFS. The data blocks reading from NFS while responding to the read request will also be asynchronously written to the node's local cache layer.

4.3.3. Cache and Prefetching Strategies

To minimize network overhead and reduce the latency of reading requests in container processes when fetching data from remote filesystems, Rofuse implements a three-level storage architecture to locally cache data blocks as much as possible. The three-level storage architecture is shown in Figure 8.

Rofuse adopts a hierarchical storage strategy and constructs a three-layer storage structure consisting of memory, node-local cache, and a remote storage server (NFS), as shown in Figure 8. When the FUSE server process of Rofuse starts on a fresh node for the first time, both the memory and local cache of the node are empty. In this scenario, the read requests initiated by container processes must be fetched by the FUSE server from the remote storage server (NFS). The fetched data is then sent to the requesting container and simultaneously cached in both the FUSE server's memory and the node-local cache for future access.

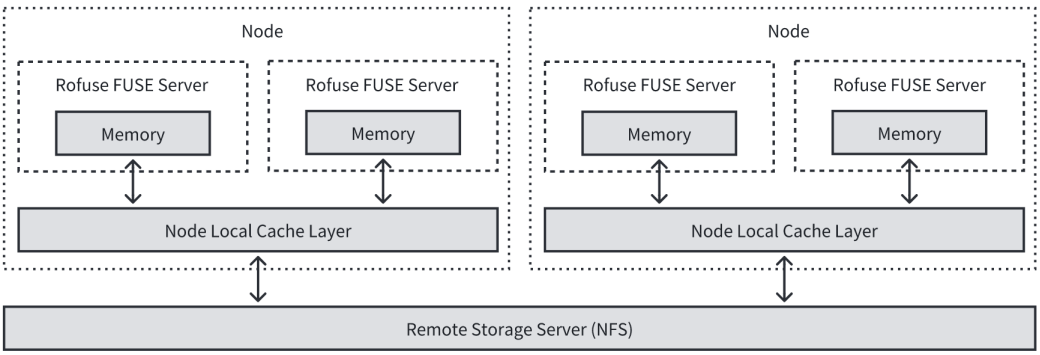

Figure 8. The storage architecture of Rofuse is divided into three layers. From top to bottom, these are the memory within the Rofuse FUSE server process, the node-local cache layer, and the remote storage server. The architecture is designed to optimize performance by leveraging local caching while ultimately storing data on remote servers for persistence and sharing across nodes.

In the memory cache, the writing and eviction of data blocks are performed based on the hash values of the data blocks. Since a FUSE server is dedicated to serving a single container, a simple LRU (least recently used) algorithm is adopted to manage the memory cache. Each Rofuse FUSE server can be regarded as a "sidecar" type program attached to each container, which is assigned a fixed memory threshold according to the container's resource request at startup. When the cache size of a FUSE server exceeds the threshold, the data blocks used at the lowest frequency are removed from memory.

As for the node-local cache, each data block is stored as a separate file, with the path determined by the hash value of the data block. For instance, if the root directory of the node-local cache is set to /var/rofuse/blocks and the hash value of a data block is sha256-xxx, then the data block file will be saved at the path /var/rofuse/blocks/sha256-xxx. Before writing a data block to the node-local cache, Rofuse first calculates its path and proceeds with the writing only if no corresponding file exists at the target path. Unlike the memory cache, the node-local cache adopts a multi-level LRU strategy, as shown in Figure 9.

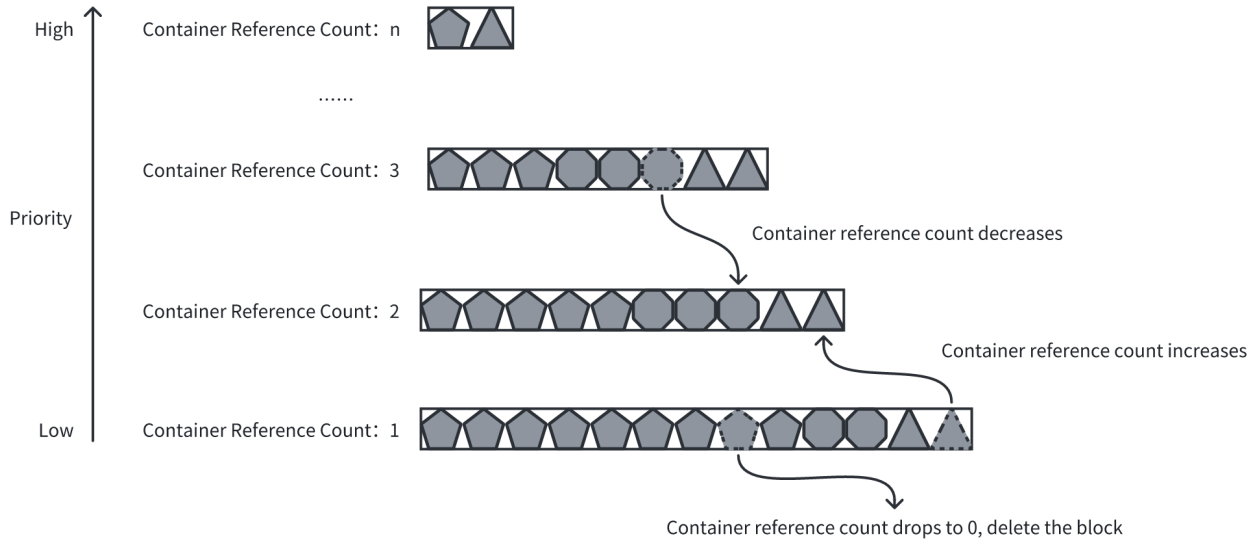

Figure 9. The multi-level LRU policy for node-level cache.

In Figure 9, the node-local cache performs hierarchical cache management by identifying the degree of data sharing blocks among different containers. Specifically, Rofuse determines the importance of a data block based on the number of containers sharing it; if numerous containers share a data block, it is considered as having higher importance and correspondingly higher priority in the cache. We quantified the degree of data-sharing blocks with the "container reference count". For example, since the local cache is empty, the FUSE server process will place the written data blocks into the LRU queue with a "container reference count" of 1, when a container starts on a new node for the first time. Subsequently, when additional containers are initiated and their associated FUSE servers access the same data blocks as previously encountered, the reference count for these blocks increases. This increment elevates their caching priority, facilitating their migration to an LRU (least recently used) queue where they are maintained with an enhanced reference count. Conversely, the termination of a container results in a decrease in the reference count of the data blocks it can access, thereby reducing their caching priority. If the reference count of a data block decreases to zero, it will consequently be purged from the node's local cache.

The capacity of the node's local cache is subject to a predefined threshold. When the amount of written data blocks exceeds this limit, the node's local cache will gradually remove the least frequently used data blocks, prioritizing from low to high priority. To prevent high-priority queues from abnormally expanding and causing prolonged starvation in low-priority queues, Rofuse sets independent thresholds for each priority queue. Under the fixed total threshold premise, the thresholds for each priority queue are allocated using an arithmetic sequence method based on the container's reference count. As the maximum container reference count changes, the thresholds of the queues are dynamically adjusted proportionally. For example, when the maximum container reference count is 2, the threshold for the queue with a reference count of 1 is set to two-thirds of the total

threshold; when the maximum reference count increases to 3, the threshold for that queue is adjusted to half of the total threshold.

To maximize the cache hits of FUSE server processes when accessing data blocks, Rofuse adds a "prefetch" strategy to cache management, which means that Rofuse will download and store data blocks from the remote storage server to the local cache before the container accesses them. Rofuse's prefetching strategy includes the following two aspects.

According to the principle of locality, it can be assumed that if a container currently accesses a portion of a file, it is likely to access other parts of the file shortly. Therefore, when the Rofuse remote filesystem receives a read request for a file, it responds not only to the immediate request but proactively downloads the remaining segments file into the local cache, as shown in Figure 10.

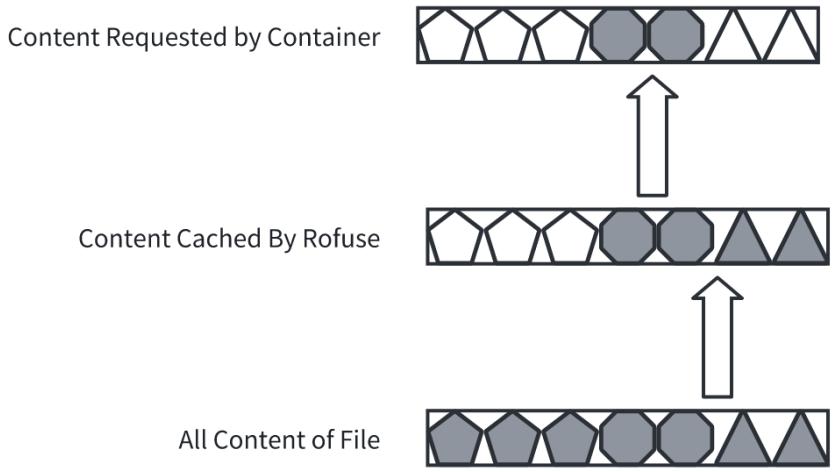

Figure 10. Rofuse intelligently pre-fetches data blocks of files to the local machine, instead of pulling them only when the filesystem actually requests them. This takes full advantage of the principle of program locality, improving the read performance of the Rofuse filesystem.

On the other hand, Wang et al. mentioned that applications tend to access the contents of the same directory within a period of time [8]. When the Rofuse framework handles a container's read request for a file, it not only reads that file but also preloads data blocks of other files in the same directory into the cache. Considering that a directory may contain a large number of files, each FUSE service process has an additional work queue to temporarily store the prefetching requests that wait in line to avoid excessive concurrent downloads of data causing pressure on the remote server. At the same time, the FUSE service also runs multiple background processes, which are responsible for periodically extracting prefetching requests from the work queue and pulling the corresponding data blocks from the remote server.

4.3.4. Block Splitting of File Data

When checkpointing the state of a container, block splitting and storage of the container's filesystem data occur. This process consists of two parts: (1) separating file metadata from file data, and (2) storing the file data in blocks. After processing, a filesystem metadata archive file (in TAR format) and a set of data blocks are obtained. This section first illustrates how data block splitting is performed using a single file as an example, and then introduces how the various components in Rofuse coordinate during this process.

The splitting process of a single regular file into data blocks is shown in Figure 11.

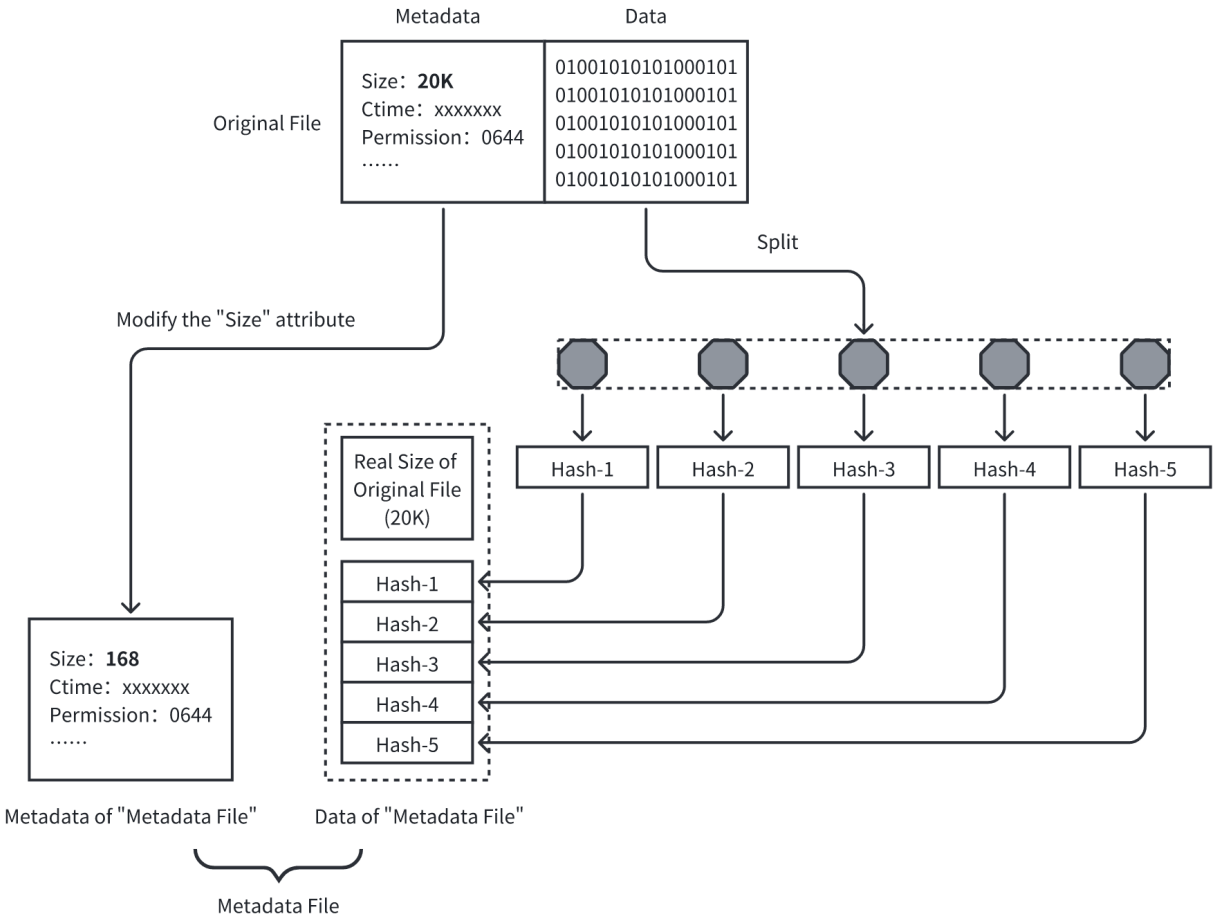

Figure 11. The process of splitting a single regular file into data blocks in Rofuse. The result of a complete regular file split will yield a metadata file and several data blocks.

In Figure 11, a regular file is split into a "metadata file" and several data blocks. During the processing, the data part of the file is handled first. Taking a 20 KB file as an example, Rofuse splits its data part into five 4 KB data blocks. Subsequently, Rofuse applies the SHA-256 hashing algorithm to each data block, generating five 32-byte hash values. These hash values, along with the original file size of 20 KB (expressed as a 64-bit integer), are then compiled into a 168-byte sequence. This byte sequence forms the data of the new "metadata file". Next, the original file's metadata attribute for file size is modified to 168, creating the metadata part of the "metadata file", thereby completing the creation of the full "metadata file". In particular, for sparse files stored in a sparse format on disk, the zero-value parts of the file consistently yield a fixed hash value after being processed by the hashing algorithm when split into data blocks. Since the remote storage server uses a CAS approach to store data block files, the zero-value parts of different sparse files share the same data block representation and do not cause additional storage overhead.

In the above splitting process, the additional storage data generated include the metadata of each data block file and the data portion of the metadata file. Taking the metadata size of a data block file as 256 bytes, for a regular file with an original size of N bytes, the additional storage data generated is approximately $(N/4096)256 + 168$ bytes, and the additional data are about 7% to 8% of the original file size.

Furthermore, when performing data block splitting on the entire old container filesystem, the detailed process and the working method of each Rofuse component are shown in Figure 12.

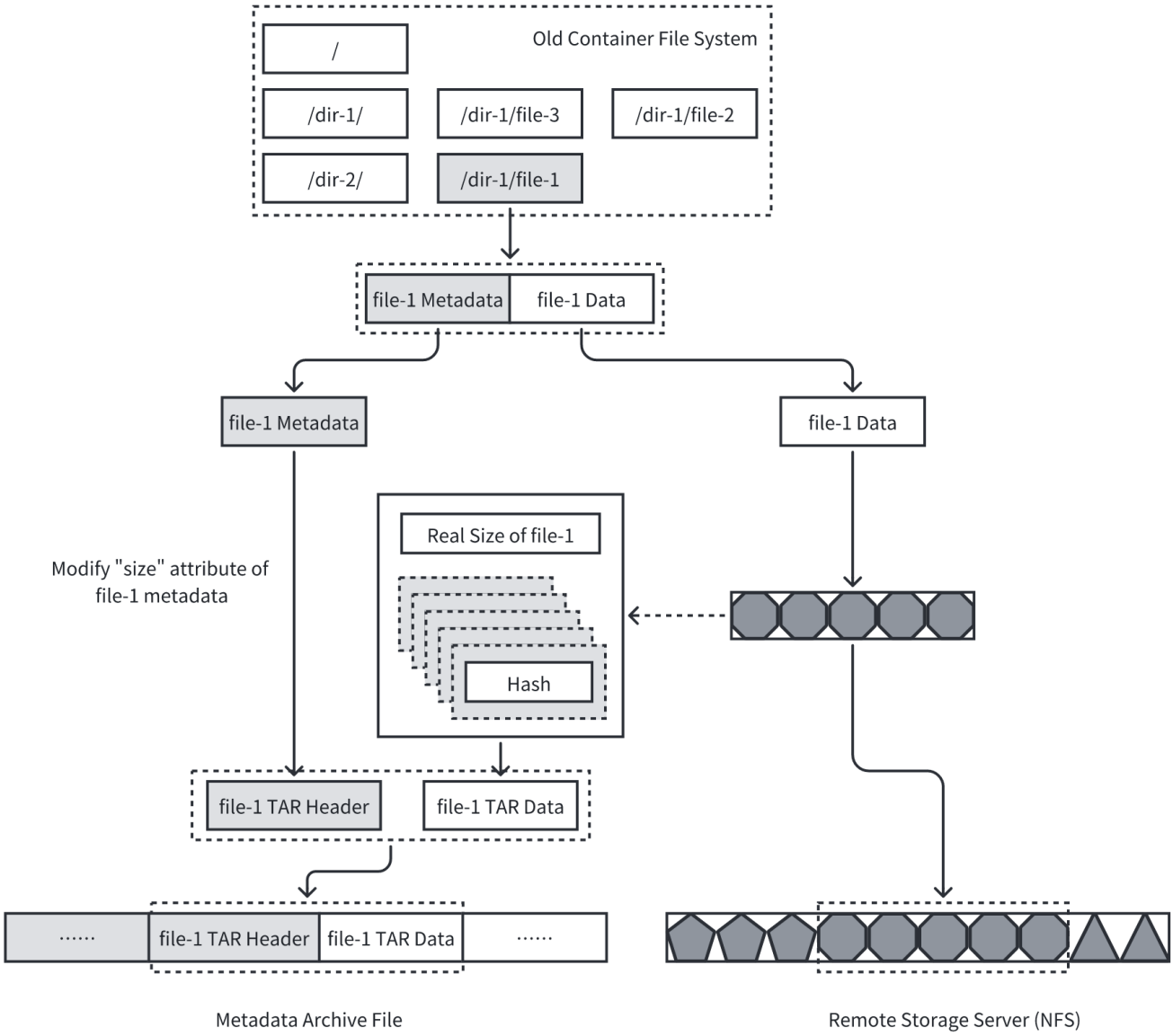

Figure 12. The data block splitting process of the old container filesystem in Rofuse.

In the splitting process, Rofuse traverses the container filesystem directory tree. For non-regular files such as directories and device files, Rofuse directly converts the file metadata into the TAR archive header format and writes it to the final filesystem metadata archive file. For regular files, the following steps are performed, where the currently processed file is assumed as file-1:

1. Split the data portion of the file according to the standard size of data blocks (4 KB is used in this article). The remaining space is less than 4 KB and will be filled with $0x00$ bytes.
2. Calculate the hash value of each data block. Denote the hash value of data block $Chunk_i$ as $Hash_i$.
3. Rofuse writes the mapping relationship between the container and $Chunk_i$ to the data block metadata manager. The data block metadata manager increments the container reference count of $Chunk_i$ by 1.
4. Rofuse concatenates $Hash_i$ to obtain the NFS path $Path_i$ where $Chunk_i$ should be written, and checks whether a data block file has already been written under $Path_i$. If so, it means $Chunk_i$ is duplicate data and can be directly discarded; otherwise, the bytes in $Chunk_i$ are written to NFS as a file with the path $Path_i$.

5. All data blocks' $Hash_i$ and the real file size of file-1 are the metadata part $FileMetaData_1$ of the metadata file file-1.
6. $FileMetaData_1$ will be serialized into a byte array. The size of this byte array will be used to replace the recorded file size in the original metadata of file-1, forming a header, $FileMetaHeader_1$, used to represent the metadata of $FileMetaData_1$.
7. $FileMetaHeader_1$ and $FileMetaData_1$ are written to the filesystem metadata archive file as the TAR header and TAR data, respectively.

The metadata archive file obtained after processing the entire filesystem is written as a Blob file into the container snapshot image and stored in the Docker Registry.

5. Results

To verify the feasibility and superiority of the Rofuse optimization mechanism in improving container restart performance in educational practice environments, robustness tests based on actual application scenarios have been conducted.

The experimental framework is built upon Containerd 2.0.0 beta (commit hash 287b4ce) and Kubernetes 1.26. The test environment is a Kubernetes cluster consisting of three Master nodes and five worker nodes. Each node is equipped with 12 CPU cores, 64 GB memory, and runs Ubuntu 22.04 Linux. In addition, there is a container registry built with the official Docker registry image for storing and distributing container snapshot images generated during the experiments, and an NFS server serving as the remote storage for Rofuse.

5.1. Performance

To evaluate the performance of the Rofuse remote filesystem, we first conducted benchmark tests using the standard testing tool FIO. On the one hand, the Rofuse remote filesystem is a read-only filesystem, primarily responsible for handling file read operations. On the other hand, even in scenarios involving file writes, the use case involves mounting through a union filesystem, where write operations to the mounted directory are directly reflected in the read–write layer of the union filesystem. In this case, the Rofuse remote filesystem still only needs to handle read requests, and the write operations are performed using the copy-on-write approach, with the write performance depending on the read performance of the Rofuse remote filesystem. Therefore, we conducted read performance tests on the Rofuse remote filesystem in two scenarios: one with 1G large files and another with 4K small files. In both scenarios, we used direct I/O for sequential reads. The performances of testing targets, including Rofuse with an empty local cache, Rofuse with all reads hitting the local cache, and NFS, are shown in Figure 13.

More specifically, Figure 13 illustrates that when reading requests to Rofuse completely miss the node-local cache, the read performance is worse than NFS because the reading process not only involves the FUSE module but also requires downloading data blocks from NFS. However, when all reading requests to Rofuse hit the node's local cache, the reading performance significantly surpasses that of NFS. This improvement suggests that if the file data used during container restarts can be directly served from the node's local cache, the restart time can be greatly reduced. This demonstrates the feasibility of reducing average container restart latency through filesystem data sharing among containers in the same container group.

Several typical use cases are selected to validate Rofuse's performance in actual educational practice scenarios, as shown in Table 1.

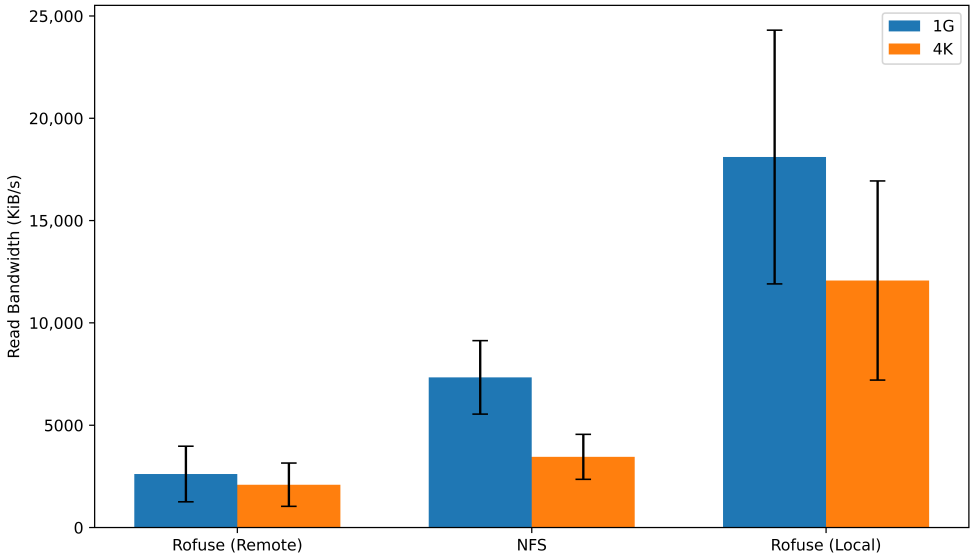

Figure 13. The reading performance comparison between the Rofuse remote filesystem and NFS. Rofuse (Remote) indicates that all read requests miss the node's local cache, and all data blocks need to be fetched from the remote storage server. Rofuse (Local) indicates that all read requests hit the node's local cache, eliminating the need to retrieve data from the remote storage server.

Table 1. Typical use cases of stateful serverless computing in educational practice. The "pre-checkpoint operation of the container" is used to simulate the usage scenario of students after the container is started for the first time. The "container triggering operation" refers to the students' actions that trigger the container to restart after it had shut down due to inactivity.

Number	Experiment Name	Type	Container Image	Pre-Checkpoint Operations on Container	Container Triggering Operation
1	Ubuntu	System Programming	Ubuntu 22.04	Upgrade software library, install and configure OpenSSH server	Connect to container using SSH client
2	Fedora	System Programming	fedora:40	Upgrade software library, install and configure OpenSSH server	Connect to container using SSH client
3	Nginx	Web Server	Nginx:1.19	None	Access Nginx homepage using HTTP client
4	Java	Web Server	openjdk:23-slim-bullseye	Write and run a simple Spring project	Call interface using HTTP client
5	Python	Web Server	python:3.12.2-bullseye	Install Django module and write and run a basic Django service	Access Django service homepage using HTTP client
6	NodeJS	Web Server	node:21-alpine3.18	Write and run a simple HTTP service	Call interface using HTTP client

Table 1. *Cont.*

Number	Experiment Name	Type	Container Image	Pre-Checkpoint Operations on Container	Container Triggering Operation
7	MySQL	Relational Database	MySQL:8.3.0	Create a database and tables with foreign key relationships, insert random data into the created tables	Connect to the database, and execute CRUD (Create, Read, Update, Delete) statements
8	Redis	Middleware	Redis:7.2	Randomly write several key-value pairs	Connect to Redis server and perform query operations

Dmigrate [8] is chosen as the control group to compare the differences in container restart times brought by Rofuse and Dmigrate in various scenarios. We first compare the container restart time of Rofuse and Dmigrate in MySQL and Java experiments with different numbers of startup containers. The results are shown in Figures 14 and 15.

It can be observed from Figure 14 that when the number of restarted containers is 1, the container restart time in Rofuse is slightly higher than that in Dmigrate in the MySQL experiment. However, as the number of restarted containers increases, the average container restart time in Rofuse goes lower than that in Dmigrate, and this trend continues to strengthen as the number of restarted containers increases. Similar results can be observed from the comparison in the Java experiment, as shown in Figure 15. The reason is that the subsequently launched containers share the data in the local cache of the node with the previously launched containers in Rofuse, which reduces the amount of data downloaded from the remote storage server during the restart, thus achieving better performance than Dmigrate.

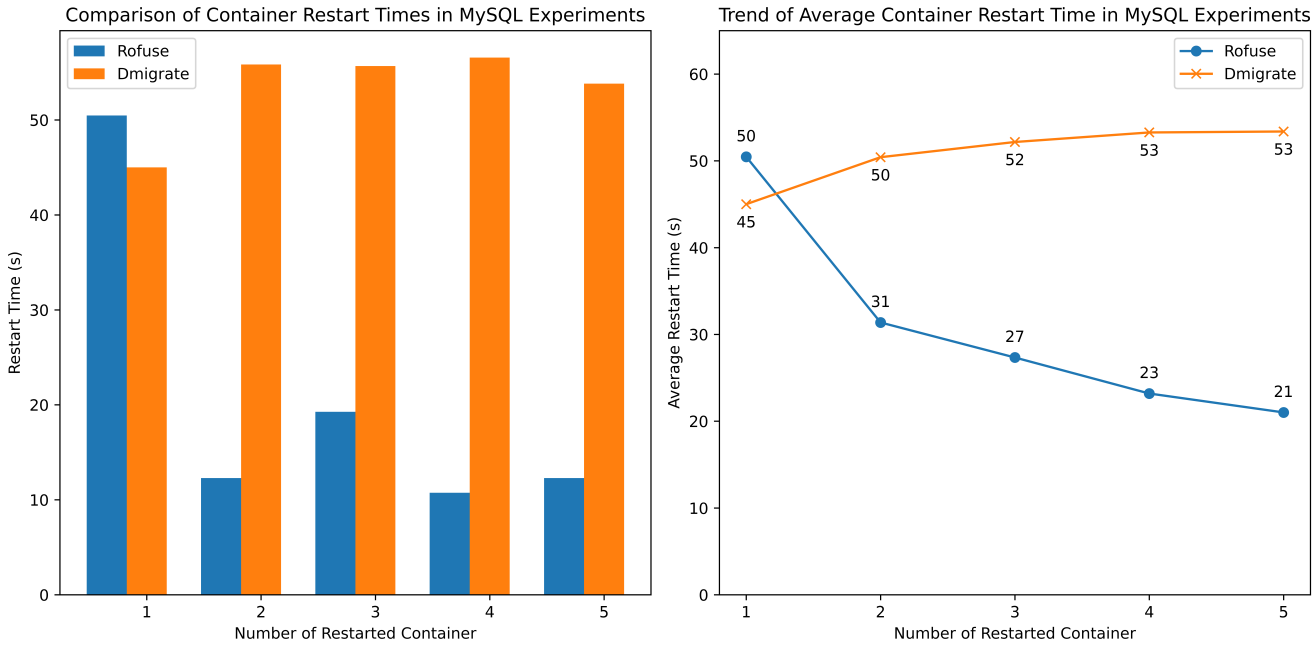

Figure 14. The difference in restart time in the container between Rofuse and Dmigrate in Java experiments.

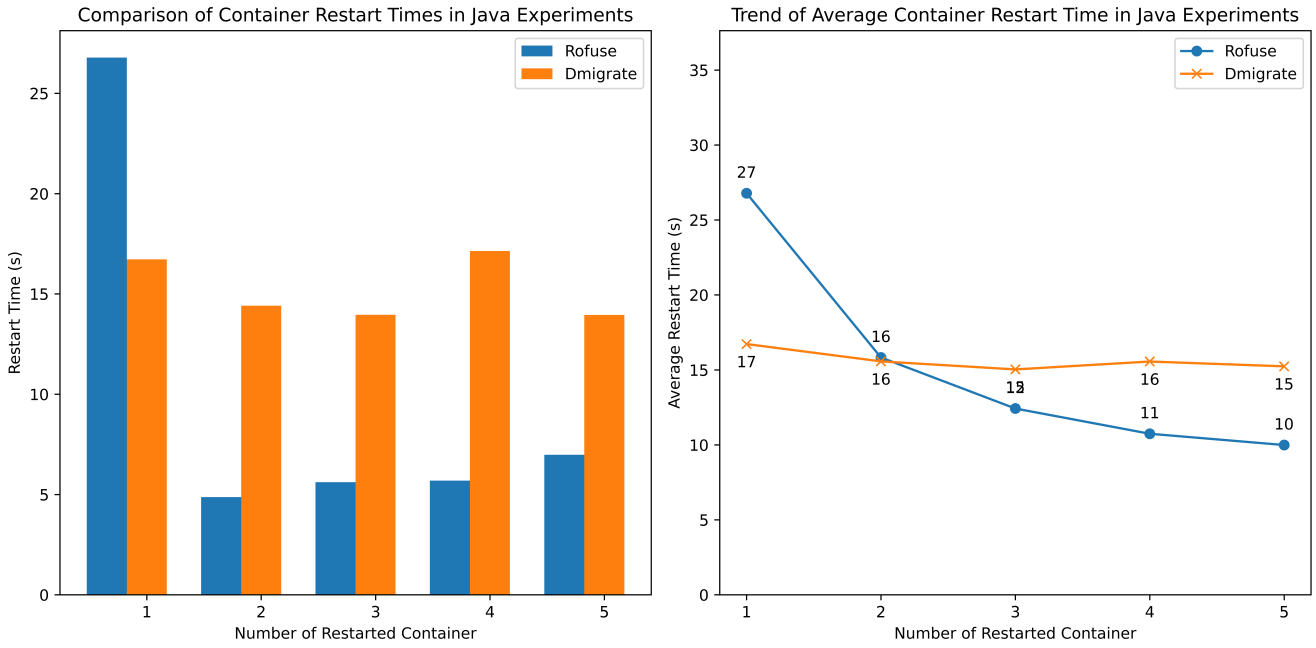

Figure 15. The difference in restart time in the container between Rofuse and Dmigrate in Java experiments.

To further validate the above conclusions, we test the average restart time of containers in all application scenarios, where the number of container startups in each experimental scenario is set up as 5, listed in Table 1. The experimental results are depicted in Figure 16.

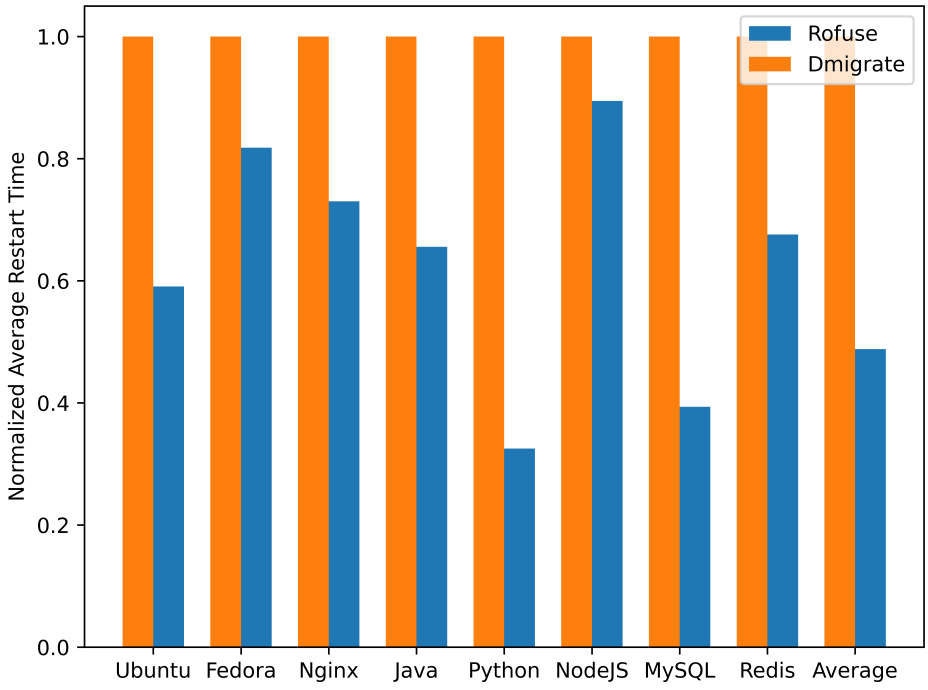

Figure 16. The performances of Rofuse and Dmigrate optimizing the time consumption of container restarts in various stateful serverless application scenarios for educational practices. In each scenario, the average restart time taken by restarting 5 containers on a single node is measured.

As the specific restart time varies in different application scenarios, the time consumed on the container of the Dmigrate benchmark group is normalized to 1 to compare the

differences between Rofuse and Dmigrate after optimization. As shown in Figure 16, the average restart time of Rofuse consumes less than Dmigrate in different application scenarios, reducing the average container restart time by 51.19%.

Due to the separation of metadata and data in the Rofuse mechanism, Rofuse needs to store additional hash information generated by data block partitioning and metadata of data block files compared to native and NFS storage of Dmigrate. To measure the cost of this extra storage, we examine the ratio of extra storage data to the total size of the container filesystem in various scenarios, as mentioned in Table 1. The results are shown in Table 2, demonstrating that the extra storage accounts for around 7% to 8% of the total size of the container filesystem in all scenarios, bringing very little additional storage pressure.

Table 2. The extra storage overhead introduced by the Rofuse.

Experiment Name	Container Filesystem Size (MB)	Extra Storage (MB)	Ratio
Ubuntu	225.80	17.26	7.64%
Fedora	243.03	18.03	7.42%
Nginx	124.38	9.44	7.59%
Java	483.63	35.02	7.24%
Python	919.56	69.58	7.57%
NodeJS	133.46	9.84	7.37%
MySQL	783.63	57.71	7.36%
Redis	128.03	9.88	7.71%

5.2. Robustness

To compare the performance differences between Rofuse and Dmigrate under various network conditions, we examined the impact of different NFS downstream bandwidths on the time required for container restarts in both systems. The experimental results are depicted in Figure 17, which shows the average time taken to restart five containers on the same node.

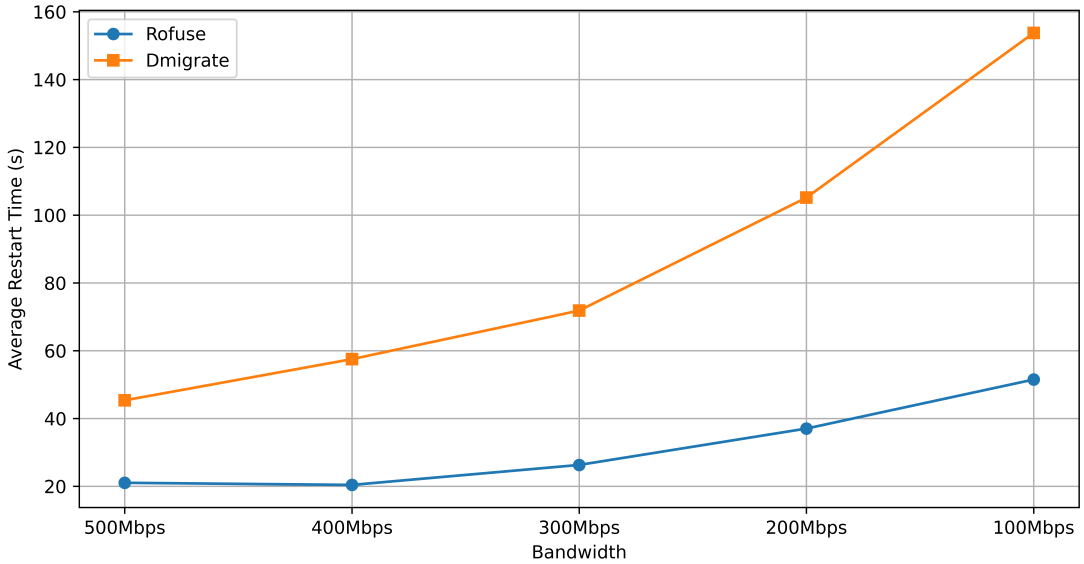

Figure 17. The comparison of the average restart time of the container between Rofuse and Dmigrate under different NFS bandwidths in MySQL experiments.

From Figure 17, it can be observed that the average restart time of the container in both Rofuse and Dmigrate increases as the NFS downstream bandwidth decreases. However, the trend in Rofuse increases significantly less than that in Dmigrate, which indicates that Rofuse is less sensitive to changes in network bandwidth and can exhibit better robustness in poor network environments.

To verify the scalability performance of Rofuse under different container scales, we tested the change in average container restart times when restarting different container numbers in the same node (up to 90 containers, slightly lower than the maximum number of Pods per node recommended by Kubernetes), and compared it with Dmigrate. The results are shown in Figure 18.

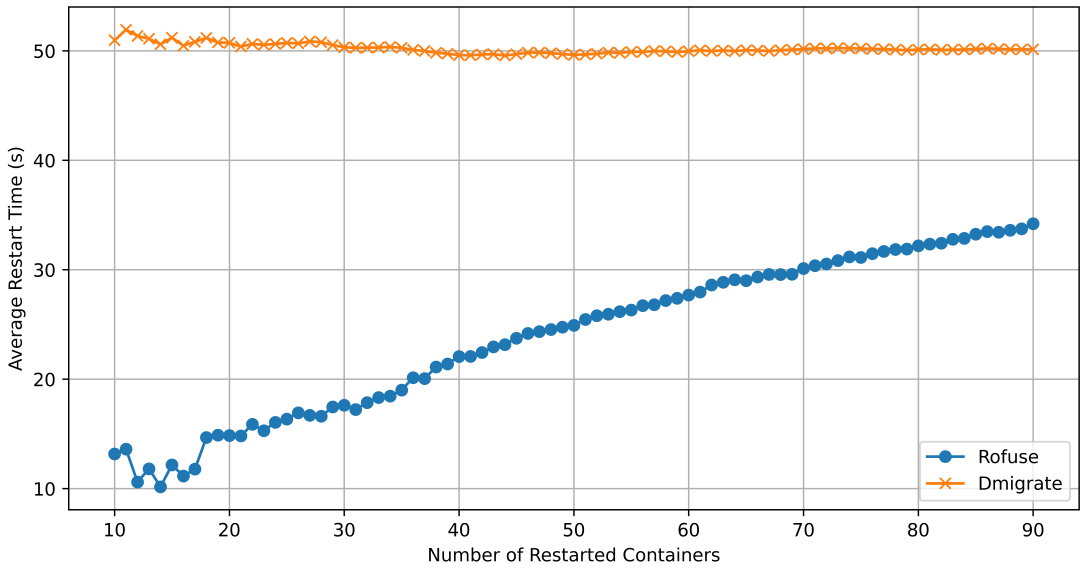

Figure 18. Comparison of the average restart time for containers on the same node as the number of containers varies between Rofuse and Dmigrate in MySQL experiments.

From the test results in Figure 18, it can be seen that as the number of restarted containers in the same node increases, the average container restart time gradually increases. This is because as the number of containers increases, the node-local cache data increase, the priority of the multi-level LRU cache gradually increases, the capacity of the LRU queue where the shared data within the container group is located gradually decreases, and the probability of the Rofuse FUSE server hitting the local cache gradually decreases. Even so, the average container restart time in Rofuse is still lower than that in Dmigrate.

Among the various components of Rofuse, the data block metadata manager and remote storage server work at the cluster level. Therefore, at the cluster level, the key components affecting the scalability of Rofuse are the data block metadata manager and remote storage server. Among them, the data block metadata manager is only called when the container is started and closed; usually, the number of containers in a Kubernetes cluster is less than 500,000, so the data block metadata manager needs to bear a very small amount of concurrency and will not become a bottleneck for Rofuse's scalability. The main bottleneck of Rofuse's scalability at the cluster level comes from the remote storage server, i.e., the NFS server. When containers in different nodes of the cluster restart, they share the network bandwidth and disk IO bandwidth of the NFS server. Figure 17 shows the changes in the average container restart times under different NFS bandwidths. Rofuse's performance under a low NFS bandwidth is better than Dmigrate, which indicates that even as the cluster size expands and the average NFS bandwidth allocated to each node decreases, Rofuse's performance is still better than Dmigrate, reflecting Rofuse's good scalability at the cluster level.

The experiments in this section collectively demonstrate that cheaper network devices and storage devices can be used as remote storage servers in Rofuse while maintaining system stability and reliability. This has important application value for educational practice environments that lack expensive high-performance equipment.

6. Conclusions

This paper focuses on resource utilization efficiency in stateful serverless services within educational practice environments. Traditional container orchestration platforms encounter resource wastage in educational settings, whereas stateless serverless services face challenges in maintaining container state persistence during the teaching process. To address these issues, we propose a stateful serverless mechanism based on Containerd and Kubernetes, with an emphasis on optimizing the startup process for container groups.

A checkpoint/restore framework (we call it "Base Framework" in this paper) provides fundamental support for stateful container management. Compared to other existing stateful serverless solutions [9–14], the base framework does not rely on specific programming languages or runtimes, nor does it cause invasiveness to the hosted applications. Based on this foundation, we first introduce the concept of container groups to address the characteristics of numerous similar containers on the same node in an educational practice. Then, we propose the Rofuse optimization mechanism, which employs delayed loading and block-level deduplication techniques that enable containers within the same group to reuse locally cached filesystem data at the block level, thus reducing container restart latency.

Experimental results demonstrate that our stateful serverless mechanism can run smoothly in typical educational practice scenarios. Compared to existing solutions like Dmigrate, Rofuse reduces the container restart time by approximately 50% on average, crossing various application scenarios. Furthermore, Rofuse exhibits better robustness under limited network bandwidth conditions, making it more adaptable to educational environments with resource constraints.

This research provides valuable insights into serverless practices in the education domain, contributing new perspectives and methods to enhance resource utilization efficiency and flexibility in teaching environments. However, there are still some limitations and areas for future work:

- The current Rofuse prototype is implemented based on a specific version of Containerd and Kubernetes. Further research is needed to adapt it to the latest versions and other container runtimes.
- The block-level deduplication mechanism in Rofuse currently uses a fixed block size. The block size based on workload characteristics should be dynamically adjusted, based on which the performances should be further optimized [22].
- Rofuse focuses on optimizing the restart time in the container in this work. How to accelerate the initial container creation process is a potential research direction.

In summary, this paper proposes a stateful serverless mechanism for educational practice environments and designs the Rofuse optimization scheme to improve container restart performance. The experimental results validate the feasibility and effectiveness of our approach. This research provides new insights into the application of serverless computing in education and promotes the development of cloud-native technologies in teaching environments.

Author Contributions: Conceptualization, X.L.; methodology, X.L. and N.L.; software, N.L. and L.Y.; validation, N.L. and L.Y.; investigation, X.L. and N.L.; writing—original draft preparation, N.L., X.L., and J.Z.; writing—review and editing, J.Z.; visualization, N.L. and L.Y.; supervision, X.L.; project administration, X.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The code of this paper can be found at https://github.com/loheagn/ccr (accessed on 28 April 2024).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Alam, A. Cloud-based e-learning: Scaffolding the environment for adaptive e-learning ecosystem based on cloud computing infrastructure. In *Computer Communication, Networking and IoT: Proceedings of 5th ICICC 2021, Volume 2*; Springer: Berlin/Heidelberg, Germany, 2022; pp. 1–9.
2. Li, Y.; Li, W.; Jiang, C. A survey of virtual machine system: Current technology and future trends. In Proceedings of the 2010 Third International Symposium on Electronic Commerce and Security, Nanchang, China, 29–31 July 2010; pp. 332–336.
3. Zhang, F.; Liu, G.; Fu, X.; Yahyapour, R. A survey on virtual machine migration: Challenges, techniques, and open issues. *IEEE Commun. Surv. Tutor.* **2018**, *20*, 1206–1243. [CrossRef]
4. Gupta, A.; Mazumdar, B.D.; Mishra, M.; Shinde, P.P.; Srivastava, S.; Deepak, A. Role of cloud computing in management and education. *Mater. Today Proc.* **2023**, *80*, 3726–3729. [CrossRef]
5. Lloyd, W.; Ramesh, S.; Chinthalapati, S.; Ly, L.; Pallickara, S. Serverless computing: An investigation of factors influencing microservice performance. In Proceedings of the 2018 IEEE International Conference on Cloud Engineering (IC2E), Orlando, FL, USA, 17–20 April 2018; pp. 159–169.
6. Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; Wilkes, J. Borg, omega, and kubernetes. *Commun. ACM* **2016**, *59*, 50–57. [CrossRef]
7. Li, Y.; Lin, Y.; Wang, Y.; Ye, K.; Xu, C. Serverless computing: State-of-the-art, challenges and opportunities. *IEEE Trans. Serv. Comput.* **2022**, *16*, 1522–1539. [CrossRef]
8. Wang, Z.H.; Zhou, Z. Research on Optimization Technology for Container Live Migration. *Comput. Syst. Appl.* **2023**, *32*, 86–93. [CrossRef]
9. Jia, Z.; Witchel, E. Boki: Stateful serverless computing with shared logs. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual, 26–29 October 2021; pp. 691–707.
10. Zhang, T.; Xie, D.; Li, F.; Stutsman, R. Narrowing the gap between serverless and its state with storage functions. In Proceedings of the ACM Symposium on Cloud Computing, Santa Cruz, CA, USA, 20–23 November 2019; pp. 1–12.
11. Barcelona-Pons, D.; Sutra, P.; Sánchez-Artigas, M.; París, G.; García-López, P. Stateful serverless computing with crucial. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **2022**, *31*, 1–38. [CrossRef]
12. Sreekanti, V.; Wu, C.; Lin, X.C.; Schleier-Smith, J.; Faleiro, J.M.; Gonzalez, J.E.; Hellerstein, J.M.; Tumanov, A. Cloudburst: Stateful functions-as-a-service. *arXiv* **2020**, arXiv:2001.04592.
13. Shillaker, S.; Pietzuch, P. Faasm: Lightweight isolation for efficient stateful serverless computing. In Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20), Boston, MA, USA, 15–17 July 2020; pp. 419–433.
14. Jonas, E.; Pu, Q.; Venkataraman, S.; Stoica, I.; Recht, B. Occupy the cloud: Distributed computing for the 99%. In Proceedings of the 2017 Symposium on Cloud Computing, Santa Clara, CA, USA, 24–27 September 2017; pp. 445–451.
15. Zhang, L.; Litton, J.; Cangialosi, F.; Benson, T.; Levin, D.; Mislove, A. Picocenter: Supporting long-lived, mostly-idle applications in cloud environments. In Proceedings of the Eleventh European Conference on Computer Systems, London, UK, 18–21 April 2016; pp. 1–16.
16. Dua, R.; Kohli, V.; Patil, S.; Patil, S. Performance analysis of union and cow file systems with docker. In Proceedings of the 2016 International Conference on Computing, Analytics and Security Trends (CAST), Pune, India, 19–21 December 2016; pp. 550–555.
17. Imran, M.; Kuznetsov, V.; Paparrigopoulos, P.; Trigazis, S.; Pfeiffer, A. Evaluation and Implementation of Various Persistent Storage Options for CMSWEB Services in Kubernetes Infrastructure at CERN. In *Proceedings of the Journal of Physics: Conference Series*; IOP Publishing: Bristol, UK, 2023; Volume 2438, p. 012035.
18. Puliafito, C.; Cicconetti, C.; Conti, M.; Mingozzi, E.; Passarella, A. Stateless or stateful FaaS? I'll take both! In Proceedings of the 2022 IEEE International Conference on Smart Computing (SMARTCOMP), Helsinki, Finland, 20–24 June 2022; pp. 62–69.
19. Mizusawa, N.; Kon, J.; Seki, Y.; Tao, J.; Yamaguchi, S. Performance improvement of file operations on overlayfs for containers. In Proceedings of the 2018 IEEE International Conference on Smart Computing (SMARTCOMP), Taormina, Italy, 18–20 June 2018; pp. 297–302.
20. Nadgowda, S.; Suneja, S.; Bila, N.; Isci, C. Voyager: Complete container state migration. In Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; pp. 2137–2142.
21. Harter, T.; Salmon, B.; Liu, R.; Arpaci-Dusseau, A.C.; Arpaci-Dusseau, R.H. Slacker: Fast distribution with lazy docker containers. In Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16), Santa Clara, CA, USA, 22–25 February 2016; pp. 181–195.
22. Mitropoulou, K.; Kokkinos, P.; Soumplis, P.; Varvarigos, E. Anomaly Detection in Cloud Computing using Knowledge Graph Embedding and Machine Learning Mechanisms. *J. Grid Comput.* **2024**, *22*, 6. [CrossRef]