FastMig: Leveraging FastFreeze to Establish Robust Service Liquidity in Cloud 2.0

Sorawit Manatura
HPCNC Lab, Department of Computer Engineering
Kasetsart University, Thailand
sorawit.man@ku.th

Thanawat Chanikaphon
HPCC Lab, School of Computing and Informatics
University of Louisiana at Lafayette, LA, USA
thanawat.chanikaphon1@louisiana.edu

Chantana Chantrapornchai*
HPCNC Lab, Department of Computer Engineering
Kasetsart University, Thailand
fengcnc@ku.ac.th

Mohsen Amini Salehi*
HPCC Lab, Computer Science and Engineering Department
University of North Texas, TX, USA
mohsen.aminisalehi@unt.edu

***Abstract*—Service liquidity across edge-to-cloud or multi-cloud will serve as the cornerstone of the next generation of cloud computing systems (Cloud 2.0). Provided that cloud-based services are predominantly containerized, an efficient and robust live container migration solution is required to accomplish service liquidity. In a nod to this growing requirement, in this research, we leverage *FastFreeze*, a popular platform for process checkpoint/restore within a container, and promote it to be a robust solution for end-to-end live migration of containerized services. In particular, we develop a new platform, called FastMig that proactively controls the checkpoint/restore operations of FastFreeze, thereby, allowing for robust live migration of containerized services via standard HTTP interfaces. The proposed platform introduces post-checkpointing and pre-restoration operations to enhance migration robustness. Notably, the pre-restoration operation includes containerized service startup options, enabling warm restoration and reducing the migration downtime. In addition, we develop a method to make FastFreeze robust against failures that commonly happen during the migration and even during the normal operation of a containerized service. Experimental results under real-world settings show that the migration downtime of a containerized service can be reduced by 30X compared to the situation where the original FastFreeze was deployed for the migration. Moreover, we demonstrate that FastMig and warm restoration method together can significantly mitigate the container startup overhead. Importantly, these improvements are achieved without any significant performance reduction and only incurs a small resource usage overhead, compared to the bare (*i.e.,* non-FastFreeze) containerized services.**

***Index Terms*—Containerized Services, Liquid Computing, Robust Live Migration, Fault Tolerance**

I. Introduction

A. Motivation

Cloud 2.0 marks the next generation in cloud computing, integrating cutting-edge technologies like AI, machine learning, and IoT. This phase emphasizes greater efficiency, scalability, and the use of heterogeneous resources, setting the stage for advanced and innovative applications. Services are envisaged to be truly distributed, either across edge-to-cloud—to support low-latency services [18]—or across multi-cloud—to attain high reliability and cost efficiency [14]. The next generation of services in Cloud 2.0 desire *service liquidity* across the resource continuum. That is, the services must be free to execute and relocate seamlessly over distributed and heterogeneous platforms [15]. In addition to the futuristic use cases, service liquidity offers several advantages for day-to-day IT operations, such as facilitating maintenance tasks (*e.g.,* performing hardware upgrades) and improving resource utilization via dynamically reallocating workloads across hosts without disrupting their operations. Establishing service liquidity entails a seamless live service migration solution from one host to another without degrading the quality of experience.

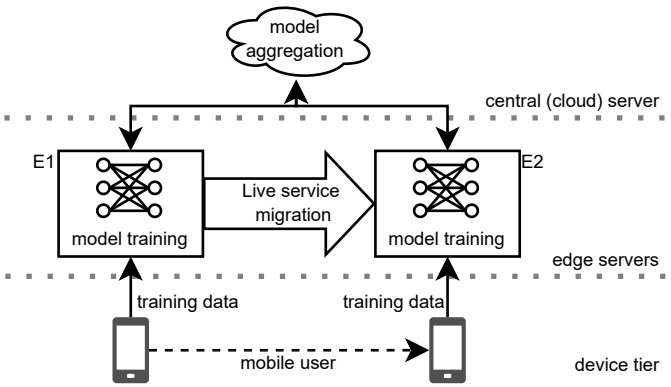


Fig. 1: Service liquidity use case to efficiently achieve federated learning for mobile users. The model training process can seamlessly migrate from a source edge sever (E1) to a destination one (E2).

As an exemplar use case, consider the federated learning scenario, shown in Figure 1, where the model training takes place only on the on-premises edge servers, to preserve the user's data privacy. Then, the trained model is transferred to the central (cloud) server to perform model aggregation [27]. In this scenario, the data collection is often performed on the device (*e.g.,* smartphone) of a mobile user who tends to randomly disconnect from one edge server and connect to another. Such mobility can slow down the training if the model has to be retrained. The ability to live migrate the training service across edge servers, however, enables seamless resumption of the training process at the new edge server,

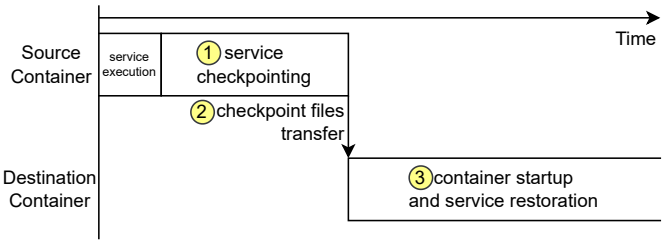
thereby, saving the training time [27].

In modern cloud environments, where services are increasingly containerized for the merit of portability and isolation, various migration approaches are studied, including the native approach that requires the container runtime checkpoint/restore feature [25] and the container nesting approach [6] that nests the containerized service within another container providing the migration ability. A distinct approach that aims not to migrate an entire container but to migrate only the "embedded processes of the service" is desired to minimize the overhead, maximize portability, and cater to multi-cloud benefits. FastFreeze [28] is a turn-key solution that exhibits these features; however, it is purposed for containerized processes checkpointing and restoration. FastFreeze packages up the necessary dependencies into a single library for building a container image, making it easy-to-use through a simple command-line interface and unprivileged to be able to execute inside a container securely. In this work, our hypothesis is that FastFreeze can be an ideal candidate to implement live migration of containerized services and realize the idea of liquid computing.

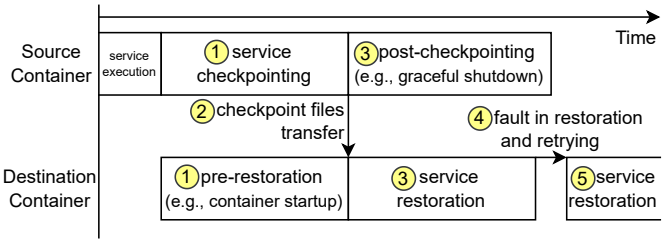
B. Problem Statement

To realize the live containerized service migration, developers may choose to simply import an existing checkpoint/restore solution, like FastFreeze, into the container image. Nevertheless, our preliminary analysis shows that this approach is failure-prone due to the design of a modern container and the uncertainties of the migration process across systems. Figure 2a depicts the necessary steps for live migration: service checkpointing, checkpoint files transfer, and restoration. In the traditional container, the service processes exit after the checkpointing step is completed. Then, the entire container is considered down by the runtime and loses a chance to execute graceful shutdown instructions, which can adversely affect the source system. For instance, the exit behavior of the services may differ from what is expected, potentially disrupting subsequent operations at the source system. Additionally, binding the containerized service management and its execution—or, in other words, binding container startup with the containerized service startup—during the migration limits the containerized service from being *warm*-restored. That is, the container startup instructions must wait unnecessarily for the checkpoint files transfer step to complete. As a result, we realized that the lack of two additional steps—post-checkpointing and pre-restoration, shown in Figure 2b—hinders the establishment of a robust container migration service.

In this paper, we adopt FastFreeze to be the underlying vehicle for a sustainable containerized service migration. We propose the FastMig container that is equipped with the service management layer, as shown in Figure 3, that enables a fast and robust service migration across computing systems. Decoupling the service management layer from the service execution layer enables the pre-restoration operations, that reduce the service migration time, and the post-checkpointing



(a) Traditional container migration



(b) Proposed container migration with post-checkpointing and pre-restoration steps

Fig. 2: Positioning of the post-checkpointing and pre-restoration steps in the live container migration process. The traditional process includes three main steps: ① service checkpointing, ② checkpoint files transfer, and ③ new container startup and service restoration. The proposed solution incorporates the pre-restoration operations that execute simultaneously in step ① and, similarly, post-checkpointing operations in step ③. Decoupling the container startup from the service startup is the key to achieving fault tolerance in steps ④ and ⑤.

operations that pave the way to execute graceful shutdown instructions, thus enhancing the service robustness.

However, it does not fully establish robust service liquidity as FastMig is still prone to faults that occur randomly—both in normal operations and in the container migration process—and cease the running service processes. Traditional fault tolerance mechanisms that offer container recreation are unsuitable for FastMig as the container is reusable for multiple startups/restorations of the same service; hence, they fail to maximize restart efficiency—by performing unnecessary container startup [7]—and traceability—by segregating service execution logs to multiple containers. The proposed FastMig service management layer includes a fault tolerance mechanism that allows restarting the service without recreating the container. As a result, it is capable of handling faults in an efficient manner and satisfies service liquidity robustness.

Lastly, FastMig promotes FastFreeze to be migration-friendly. The problem is that FastFreeze imposes an unusual latency on the migration of a containerized service that forks multiple child processes, particularly, at the restoration time. FastMig, however, improves the performance of the restore operation for multi-process services and reduces the service migration time significantly. FastMig also offers a standard interface to securely serve external migration requests.

C. Contributions

In sum, this paper makes the following contributions:

- Developing FastMig, a live containerized service migration solution that incorporates FastFreeze with the service management layer to realize service liquidity [1]
- Developing FastFreeze Daemon that decouples the containerized service management from its execution. It enables post-checkpointing and pre-restoration operations and incorporates the warm restoration technique that reduces the migration time.
- Developing a configurable fault tolerance mechanism that enhances service liquidity robustness via allowing restarting the service without recreating the container.
- Extending FastFreeze to improve the restoration time of the multi-process services upon migration and develop standard APIs to make FastMig pluggable to other solutions.
- Evaluating and analyzing the impact of FastMig under real-world scenarios and settings.

The rest of this paper is organized as follows: Section II provides background on the live containerized service migration and FastFreeze. Section III presents the design of the robust containerized service using FastFreeze. Section IV describes the evaluation and the results. Section V discusses related studies on service liquidity and live containerized service migration. Finally, Section VI concludes our work.

II. Background

A. Live Containerized Services Migration

Container technology is a virtualization technology that bundles the application and all its dependencies, providing application isolation and making the application migratable while retaining complete functionality [9], [21]. The ability to migrate containerized services between systems is beneficial in service provisioning through load balancing or fault tolerance. On a high level, for live migration, the containerized service migration can be done by checkpointing the processes inside the container, moving the checkpoint files to the destination, and restoring the container to its original state [25]. There are techniques to help reduce the downtime that comes from the transfer process, such as pre/post-copy [23] or page-server [25], but the overall operation remains the same. A main tool that has been used to do the container checkpoint/restore operations is Checkpoint/Restore In Userspace (CRIU) [1]. CRIU achieves this by using `ptrace`, a kernel interface to inspect the current process execution and memories used. CRIU also works with multi-process applications as it can checkpoint and restore an entire process tree.

When using CRIU to restore an application, the PIDs of the application processes have to be the same as before it was checkpointed. With enough privileges or permission (*e.g.,* `root` or `CAP_SYS_ADMIN` capability[2]), CRIU can achieve the desired PID at restoration time. Another solution to control the processes PID is to modify the file `/proc/sys/kernel/ns_last_pid`, which indicates the last PID allocated and determines the next fork PID in the current PID namespace[3], to match its need (*e.g.,* desired PID-1).

B. FastFreeze

FastFreeze is a checkpoint/restore utility built specially for containerized services. It is a wrapper around CRIU; hence, FastFreeze delegates the main processes checkpoint/restore to CRIU. FastFreeze provides management facilities for checkpoint/restore, providing a customized *init* process[4] that makes the containerized service processes its children.

In addition, FastFreeze is designed to be usable within unprivileged containers. It uses a technique called *fork bomb*, rapidly forking and killing child processes until it gets the desired PID to handle the CRIU PID requirement upon restoration. For single-process application restoration, the delay is not significant; however, for multi-process applications [13], the delay from the *fork bomb* procedure can be significant. Chanikaphon *et al.,* [6] show that the migration approach using FastFreeze has increased the migration overhead by ≈40 seconds per 1 additional child process.

Lastly, FastFreeze provided only a command-line interface to activate all its operations; this leads to obstacles in the migration requirements in the aspects of container access permission, synchronization, and monitoring. Moreover, the lack of easy-to-use/standard interfaces obstructs the integration with web services and leads to incompatibility with service-oriented architecture (SOA).

III. FastFreeze for Robust Service Liquidity across Computing Systems

A. Overview

Figure 3 illustrates the overview of FastMig, the system that utilized the adapted FastFreeze for robust service liquidity across computing systems. The system consists of 5 main modules: (A) *the FastMig Interface* allowing requests from outside, (B) *FastFreeze Daemon*, providing containerized service management, (C) *FastFreeze*, utilizing CRIU for checkpoint/restore operations and acting as parent process of the containerized service, (D) *the containerized service* and, (E) *the Fault Handling Module*, which is the configurable logic used to handle the fault from logs. Figure 3 also shows another

[1]FastMig and all the experimental data are publicly available for reproducibility purposes in the following addresses: https://github.com/hpcclab/fastfreeze4service_migration

[2]Capability is a privilege for specific functionality in Linux, *e.g.,* the ability to change file owner. In analogy, it is a subset of root privileges that can be granted to unprivileged processes. To minimize the attack surface, it is a practice to grant only the capabilities the process requires to function.

[3]PID namespace is a Linux kernel feature that isolates process ID number space. A process in an isolated namespace can have an arbitrary PID independent from existing ones in other namespaces. Processes are not visible across namespaces except in a number of cases. PID namespace is one of the important foundations of container technology.

[4]The container *init* process is the root of the container process tree. Its status represents the container status; for example, the container is considered down if its *init* process exits. The *init* process is also responsible for other duties, such as forwarding signals to child processes and reaping zombie processes.

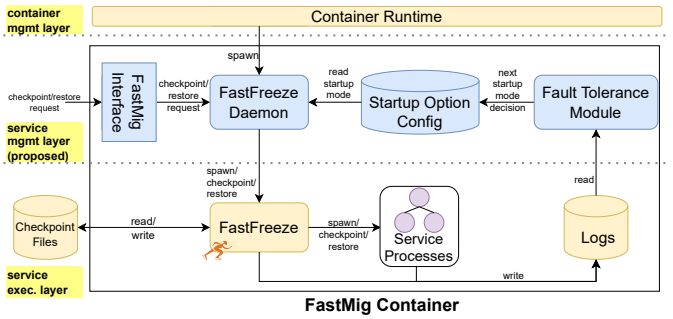


Fig. 3: Overview of the FastMig within a container. We propose adding the "service management layer" (components with the blue color) to enable fast and robust live migration of containerized services.

component called *the Start Option Config*, which is the file that indicates how the service will be started.

B. Adapting FastFreeze to Establish Robustness

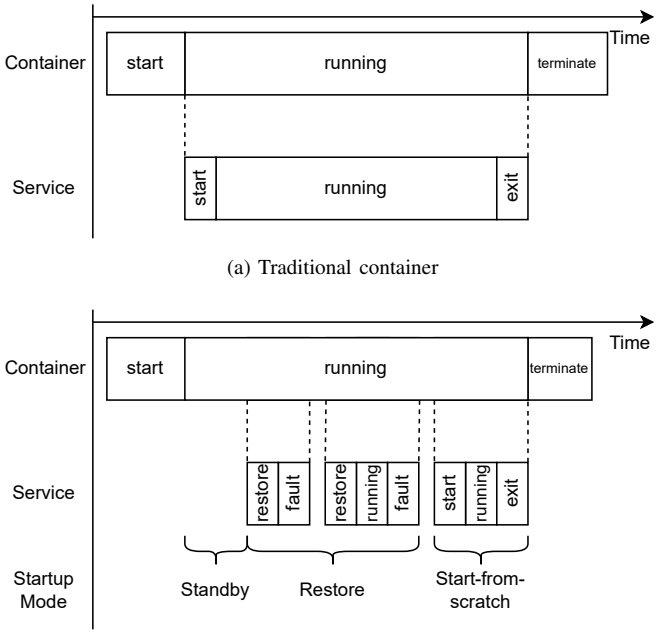
To enhance robustness, we address three key aspects. Firstly, we separate containerized service management and its execution and examine startup options, enabling finer service control during migration. Secondly, we establish a fault tolerance mechanism to bolster the resilience of containerized services during both regular operation and migration. Lastly, we offer the FastMig Interface to facilitate seamless integration with migration systems, enhancing system security.

1) Service Management Decoupling: For modern containerized services, the container and the containerized service execution are bound together. For instance, when the service exits, the container also exits, and the service is started upon the container start. This factor limits the ability to manage the containerized service lifetime during the migration, as discussed in Section I-B. To establish the robustness of service liquidity, we build a solution that allows us to separate the containerized service management (*e.g.,* states) and its execution. FastFreeze acts as a parent process of the containerized service, covering the service execution layer; therefore, we developed a component for the service management layer called FastFreeze Daemon.

FastFreeze Daemon provides a standby behavior for the containerized services. The container can be in a running state while the internal service is not running. As FastFreeze Daemon is the first and always running process in the container, in standby mode, it listens for the incoming request to run the service up, either restore or start it from scratch. The containerized services equipped with FastFreeze Daemon can start in three different ways, depicted in Figure 4: (i) start the service from scratch disregarding the checkpoint; (ii) restore the service from the given checkpoint; or (iii) remain on standby—not running the service. The startup behavior for each circumstance is configurable by providing FastFreeze with the *Startup Option Config* indicating the startup mode and relevant information. If no configuration is provided, FastFreeze Daemon remains in the standby mode by default.

A notable use case for the standby mode is that in the migration process, one can start the container of a service at the destination using the standby mode. When the checkpoint files are transferred, the service can then restore with the container already warm-up. We named this technique *warm restoration*, inspired by a serverless function warm start [4], [8].

The separation of the service management layer and the service execution layer, together with startup modes, enhances the service migration robustness by allowing post-checkpointing/pre-restoration operations such as graceful shutdown and warm restoration. Moreover, it enables restarting the service when faults occur without creating a new container via the fault tolerance mechanism. Not only does it eliminate unnecessary startup overhead for a new container, but it also enhances service traceability by eliminating the need to aggregate logs across old and new containers.



(a) Traditional container

(b) FastMig container

Fig. 4: Container and containerized service lifespan in traditional and FastMig containers. FastMig container enhances robustness by allowing the service to restart without recreating a new container. For each restart, the fault tolerance mechanism determines how the containerized service starts: restore from checkpoint files, start from scratch, or standby (not starting).

2) Fault Tolerance Mechanism: Traditional FastFreeze already implemented a feature for users (*i.e.,* service developers) to plug in the metric recorder program, which consumes FastFreeze JSON-structured logs as its argument. We developed the *Fault Handling Module* that analyzes logs of how the service exits and decides how it should restart. After that, it outputs its decision into the *Startup Option Config*, and the FastFreeze Daemon then reads it in the next service startup. In summary, we use an analogy to a self-feedback method.

We defined a default logic that can handle simple faults

based on the service exit code as a demonstration. The pseudo-code of the mentioned logic is shown in Listing 1. The logic indicates that when the service exits as expected with code 0, FastFreeze Daemon will try to restore the service from the checkpoint files in the next startup, and if the service exits by receiving signals, including the signal from FastFreeze checkpoint operation, the startup option will be standby; otherwise, FastFreeze Daemon will force the service to start from scratch.

```
if exit_code == 0:
    start_option = restore
elif exit_code >= 128 and exit_code <= 159:
    start_option = standby
else:
    start_option = from_scratch
```

Listing 1: Default Fault Handling Module logic written in Python language

In other use cases, with different requirements, users can specify their logic for how their containerized services should restart after specific faults and situations happen.

3) The FastMig Interface: The FastMig Interface exposes the FastFreeze operation to be usable from outside of the container. In detail, the HTTP interface is implemented within the FastFreeze Daemon. With provided HTTP APIs, external entities can call the migration commands by sending requests, and as distinct from command-line execution, the HTTP request does not require container access permission. As a result, this enhances FastFreeze integration with other systems and enhances the migration operation security. We achieved this by modifying the FastFreeze to communicate through the Unix domain socket after each operation. FastMig keeps listening to the socket, waits for the operation finishing sign, and then responds to the client. The FastMig Interface mimics FastFreeze commands for its APIs, so it has two main APIs, namely `run` API and `checkpoint` API. Both APIs accept a JSON request body that aligns with each FastFreeze command. After FastMig Interface receives a request, it processes the body and the Startup Option Config; then it spawns FastFreeze with extracted arguments.

Another significant behavior difference between the HTTP Interface and FastFreeze command-line interface is that the APIs are synchronous, while the prior command-line is not. The FastMig Interface responds to the client only after the called operation is ended, either finished or failed (*e.g.,* the application started successfully). This characteristic makes the checkpoint/restore and migration operations more traceable and eases the evaluation of measuring their duration.

C. Adapting FastFreeze for multi-process Services

As mentioned in II-B, there can be a significant delay when using FastFreeze to restore multi-process services in unprivileged scenarios. This is usually normal and practical cases arising from FastFreeze using the *fork bomb* workaround. Our examination reveals that the current FastFreeze solution tries to edit the `/proc/sys/kernel/ns_last_pid` first to set the desired process ID, and if it cannot, it will use *fork bomb* until it gets the desired `ns_last_pid`, as shown in Listing 2.

```
set_next_pid(pid):
    if /proc/sys/kernel/ns_last_pid is writable:
        write the desired last_pid(pid-1)
    else:
        use fork_bomb(pid-1)

fork_bomb(pid):
    While ns_last_pid != pid:
        fork_process
        kill_process
```

Listing 2: The pseudo-code of Fastfreeze solution to obtain the desired PID during the service restoration

To avoid the delay, it must prevent the *fork bomb* by allowing FastFreeze to write to `/proc/sys/kernel/ns_last_pid` while still not over-giving the privilege to the container and FastFreeze itself. To do that, two limitations prevent FastFreeze from editing `/proc/sys/kernel/ns_last_pid`: (i) without root privilege, FastFreeze has no permission to edit `/proc/sys/kernel/ns_last_pid`, which is a system file, (ii) in the typical container environment(*e.g.,* Kubernetes [3], Docker [2]), the container mount `/proc` directory as a read-only directory.

Since Linux kernel version 5.9, it introduced a new capability named `CAP_CHECKPOINT_RESTORE` [20], which provides the ability to control PID for corresponding PID namespace via editing `ns_last_pid` and `clone3` system call. With this capability, the first limitation can be overcome. To tackle the second limitation, we study the case of running FastFreeze in a Docker container and adapt the Docker security options `systempath=unconfined` and `apparmor=unconfined` to allow access to system directories. However, disabling AppArmor allows writing access to all system files and may lead to security issues.[5] Thus, it must be set together with a custom AppArmor profile only to permit modifying `ns_last_pid` but not for others in `/proc` file system.

An example of a Docker configuration avoiding the *fork bomb* delay is shown in Listing 3 together with an AppArmor profile entry in Listing 4. The performance improvement is reported in Section IV-C.

```
Docker run --cap-add=cap_sys_ptrace
           --cap-add=cap_checkpoint_restore
           --security-opt systempaths=unconfined
           --security-opt apparmor=
    apparmor_config
           ff_container
           fastfreeze run app
```

Listing 3: Example of Docker run parameters avoiding FastFreeze fork bomb

[5]AppArmor is a mandatory access control (MAC) system for Linux. It interfaces with the Docker container as another layer of security. The Docker container applies a default AppArmor profile, restricting actions and accesses, *e.g.,* avoid editing `/proc` file system. Instead of just disabling AppArmor, users can provide a secure custom profile for hardening purposes.

```
1 deny @{PROC}/sys/kernel/{?,??,[^s][^h][^m]**}-@{
      PROC}/sys/kernel/ns_last_pid  w,
```

Listing 4: Example of secured AppArmor profile entry

IV. Evaluation

To evaluate that FastMig is feasible for live container migration with minimal overheads and downtime while still being robust with the fault tolerance mechanism, we summarized evaluation in a number of aspects in the following metrics:

1) The increased overhead on service performance caused by incorporating the FastFreeze and FastMig into the container
2) The improvement of the service restoration time when granted appropriate privileges
3) The improvement of migration time when using FastMig and warm restoration
4) The impact of the fault tolerance mechanism on migration performance and its coverage for various types of faults

A. Experimental Setup

We created Ubuntu 22.04 LTS VMs to represent each as a physical computing node with 4 vCPU, 16 GiB memory, and 100 GiB storage. All VMs are connected with 1 GiB Ethernet. We used Docker version 25.0.1 as the container engine and FastFreeze version 1.3.0.

The live migration performed in the experiments applied the stop-and-copy method by checkpointing the container, dumping checkpoint files to the shared filesystem (NFS), and restoring the container at the destination using the shared checkpoint files.

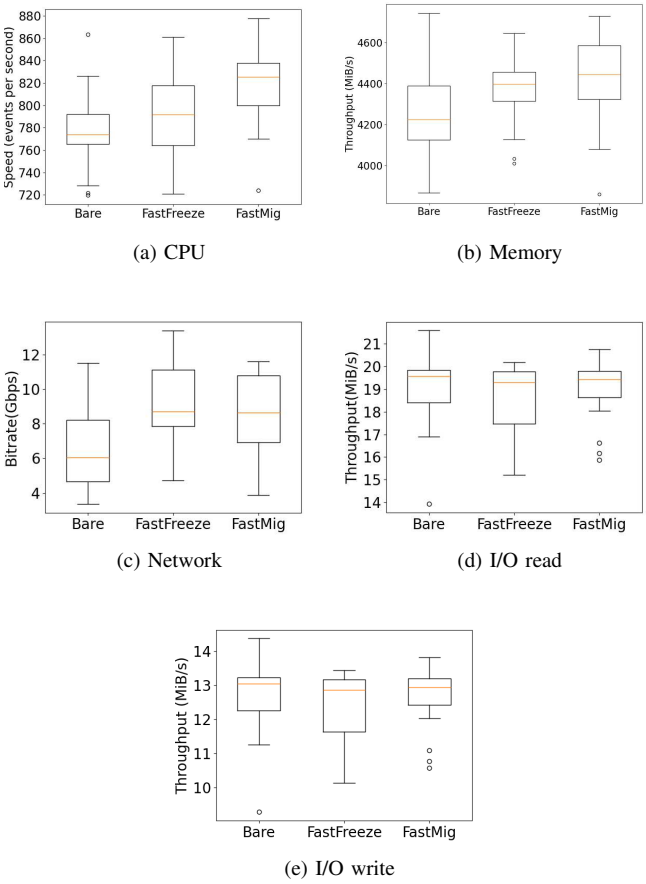
In most experiments, we deployed 2 distinct applications with different memory footprint behaviors, as it is a critical factor in the migration performance. First, we deployed a popular benchmarking application called `memhog` [25]. We configured `memhog` to write random data to the allocated memory and print a counter number every second, representing a static memory footprint application. Second, we configured YOLOv3-tiny [22], a popular object detection application, feeding it with an input image (160KB) from their repository. As opposed to `memhog`, `YOLOv3-tiny` has a dynamic memory footprint.

Container Types	%CPU	Mem(MiB)
Bare (non-FastFreeze)	0.0183	128.7
FastFreeze	0.0193	129.4
FastMig	0.0193	130.4

TABLE I: Resource Usage

B. Resource Usage and Performance Overhead (No Migration)

This experiment aims to evaluate the impact of incorporating FastFreeze and FastMig into the container by measuring service performance during normal operation (*i.e.,* , no migration). For that, we configured three kinds of containers, including (A) regular Ubuntu container, (B) Ubuntu container with FastFreeze, and (C) Ubuntu container with FastMig running, comparing with each other on the container resource usage and performance.



(a) CPU (b) Memory

(c) Network (d) I/O read

(e) I/O write

Fig. 5: Performance metrics of regular, FastFreeze-enabled, and FastMig-enabled service during the normal operation (no migration).

First, we used the `memhog` benchmark, which has a static memory footprint of 128 MiB on each container, to inspect the CPU usage percentage and memory usage with `docker stats` 30 times and calculate the average. As seen in Table I, CPU usage had no significant differences. For memory usage, the container with FastFreeze used $<$1 MiB more than the regular container since FastFreeze will spawn a process (mentioned as customized *init* process in II-B) and act as the service parent. The memory usage was increased by 1 MiB for the container with FastMig. Since the average modern container memory footprint is between 50 MB to 300 MB per container [13], we conclude that the resource usage overhead is very negligible.

Secondly, we utilized the iperf3 [10] benchmark to measure network bitrates and Sysbench [16] to measure the performance metrics, including CPU speed (in the number of

events processed per second), memory access throughput, and I/O read/write throughput. We ran each benchmark on each container for 30 times. The results in Figure 5 show that there is no significant performance degradation for all test cases.

Takeaway: *Enhancing service liquidity by incorporating FastFreeze or FastMig into the container incurs little to no additional resource and performance overheads.*

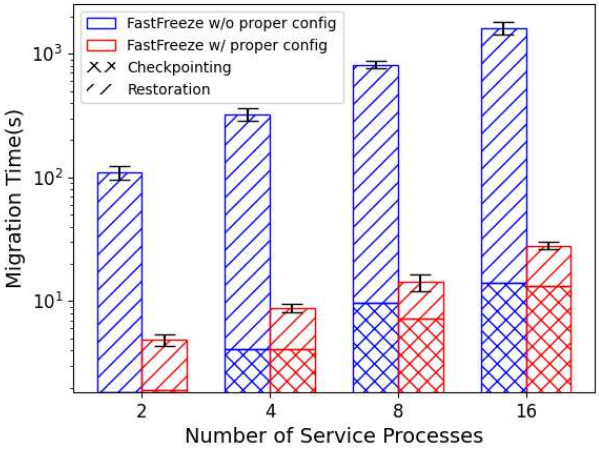


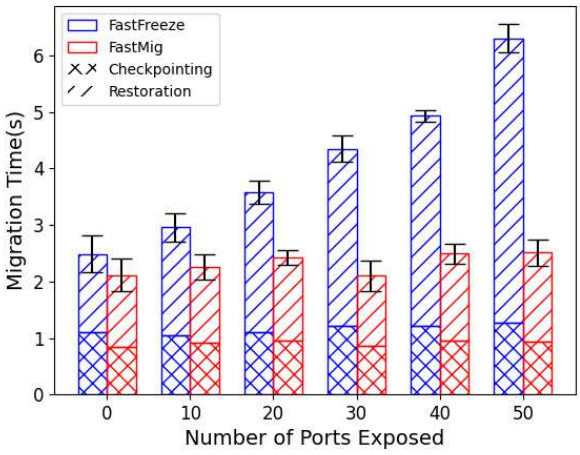
Fig. 6: Comparison of migration time before and after FastFreeze configuration fix. Error bars show 95% confidence intervals.

C. Multi-process Application Restoration Time

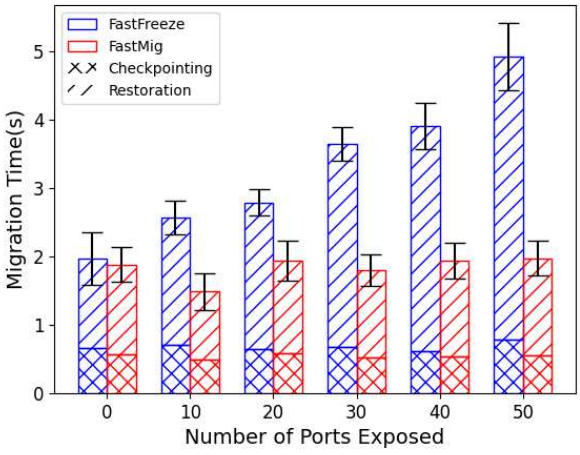
As mentioned in Section II-B, FastFreeze restoration time is irregularly high when used with multi-process applications and affects service downtime during migration. The cause examination and fixing were explained in Section III-C. We conducted experiments that compared the migration time between using FastFreeze without additional privileges and after allowing it to modify the `/proc/sys/kernel/ns_last_pid`. We deployed `memhog` in a similar manner to the previous experiments in Section IV-B. The number of `memhog` processes run in a single container was 2-16, with a scaling factor of 2. Thirty rounds were performed for each case with a different number of processes and reported in Figure 6.

When allowing FastFreeze to modify the `ns_last_pid` file, the migration time was dramatically reduced compared to using unprivileged FastFreeze with the same number of processes. For two processes, the migration time was reduced by ≈30 seconds, and for 16 processes, the reduction was greater, *i.e.,* ≈450 seconds. The differences are mainly influenced by the restoration time differences, as the checkpointing time is mostly identical between cases with the same number of service processes.

Takeaway: *To have FastFreeze-based live migration solution operate efficiently with multi-process services, certain privileges must be granted to the container.*



(a) `memhog`



(b) `YOLOv3-tiny`

Fig. 7: Migration time of `memhog` and `YOLOv3-tiny` when disabling and enabling warm-restoration. Error bars show 95% confidence intervals.

D. Measuring the Overhead of Live Migration

This experiment aims to measure the migration time overhead of live migration using FastFreeze and, second, with FastMig that allows migration with a warm restoration at the destination. Both applications were configured for this purpose, `memhog` and `YOLOv3-tiny`. To emphasize the impact of the warm restoration, which improves the container migration time by omitting the container startup time at the destination, we measured the migration time on the different numbers of ports exposed on the container since the number of ports exposed is a factor that affects the container startup time [26]. Each test was conducted 30 times, measuring the total time from the checkpoint phase until the completion of restoration at the destination. The average time was reported in Figure 7.

From the results, when not using FastMig and not allowing a warm restoration at the destination, the migration time

increased linearly as the number of ports increased. Unlike using FastMig and warm restoration, the migration time was not increased significantly, *i.e.,* ≈2 seconds. The checkpointing duration among these cases is less affected than the restoration time. This shows the benefit of warm restoration. The same cases also happened when using `YOLOv3-tiny` as the evaluated application. Although exposing 50 ports is an uncommon use case, it emphasizes the impact of hiding the container startup time. Furthermore, even with only 10 ports exposed, FastMig demonstrated an approximate 1-second improvement which can already be significant for live migration use cases. It is important to note that not only does exposing ports affect container startup time, but other factors such as the image size and the number of image layers also contribute to the overall startup duration [26].

Additionally, we observed that the migration time when equipped with FastMig is inconsistent. We also observed that the checkpointing time is even between each case, but the restoration time is not. As the restoration operation relies on networking between nodes (*e.g.,* operation request across nodes, reading checkpoint from network file system), we surmise that the migration time inconsistency happens due to the network consistency.

Takeaway: *FastMig, which enables warm restoration, significantly reduces migration time during live migration by mitigating container startup time.*

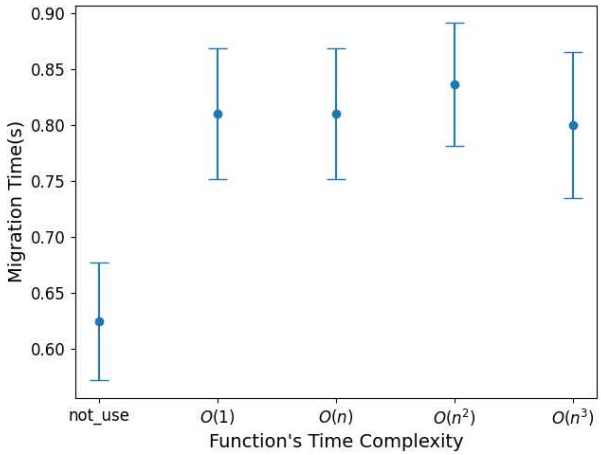


Fig. 8: The impact of embedding the fault tolerance mechanism into the container on the service migration time. Error bars show 95% confidence intervals.

E. Impact of the Fault Tolerance Mechanism

Our proposed solution relies on the logging mechanism and we built a configurable function (used in the Fault Handling Module) to decide on how the container should restart. This experiment focuses on evaluating the overhead of this mechanism by using different string processing logic (function) with different time complexity consisting of $O(1)$, $O(n)$, $O(n^2)$, $O(n^3)$ and also a case that did not use the fault tolerance mechanism. We define the input size (n) as the number of lines from the service log that we feed in. We utilize a 2000-line Apache Web Server log as a simulated input log [29]. The experiments were done with 30 repetitions for each function time complexity.

As shown in Figure 8, the fault tolerance mechanism introduces a negligible additional migration overhead of ≈0.2 seconds. The overheads are nearly constant across function time complexity as the size of n is small; however, thousand-of-line logs generally suffice to determine the fault cause and the appropriate Start Option Config.

Takeaway: *FastMig fault tolerance mechanism with basic log processing introduces little to no impact on migration time.*

F. Coverage of the Fault Tolerance Mechanism

To determine how the fault tolerance mechanism reacts to various types of faults, we set up a fault injection experiment that intentionally causes common faults during each phase of container migration and inspected its actions. To provide generality over various specific use cases, we use the default logic, shown in Listing 1, that determines FastFreeze restart state from FastFreeze exit code in this experiment. Each fault was injected ten times in each situation where possible.

If the application and FastFreeze are restarted and still function properly (*e.g.,* can be checkpointed/restored or migrated after the incident is fixed), this will be indicated as a valid result. The setup faults are reported in Table II. Some faults are not included in the experiment since they are impossible to occur during that phase.

Most of the results are valid, as shown in II, but only the network failure during application restoration is an exception. This happened because FastFreeze waits to read the checkpoint files from the NFS service, which cannot reach its peer due to network disconnection. NFS blocks FastFreeze process indefinitely and waits for reconnection without a timeout. In this situation, users must either manually stop the container or wait until the network reconnects, which causes the NFS service to unblock the process and resume the restoration.

Additionally, we observed two limitations of the fault tolerance mechanism, though the results are valid. Firstly, if a fault occurs during service checkpointing or restoration, the progress of the interrupted operation is discarded. Then, the next attempt of the same operation, *e.g.,* a second restoration from the same checkpoint files, is started from the beginning as if it has never been run before. Secondly, For system/hardware failure, the behavior may depend on how the system (*i.e.,* operating system) acts on the container. For instance, when the system is shut down gracefully and sends termination signals to the processes in the container, the Fault Handling Module will control FastFreeze restart mode to be a standby mode, but, in another case, when there is no signal sent to the container (*i.e.,* hard shutdown), the logic does not have any chance to evaluate the fault. As a result, the restart mode will be standby

Faults	Normal operation	Application checkpointing	Application restoration
Memory exceed	Restart in standby mode	Checkpoint stop, service continue	Restoration succeed
Storage exceed	Restart in from-scratch mode	Checkpoint stop, service continue	Not included
Signals (*e.g.,* SIGINT, SIGKILL)	Restart in standby mode	Checkpoint stop, restart in standby mode	Restart in from-scratch mode
Application unexpected exit (*i.e.,* exit code >0)	Restart in from-scratch mode	Not included	Not included
Corrupted checkpoint files	Not included	Not included	Restart in from-scratch mode
Network failure (cannot reach NFS)	Not included	Checkpoint stop, service continue	Restoration freeze until network reconnects
Underline system/hardware failure	Restart from previous config or standby mode	Restart from previous config or standby mode	Restart from previous config or standby mode

TABLE II: Types of faults that are injected into each situation. Some faults are not included in the experiment since they are impossible to occur during that phase.

as the default mode or will be in the state indicated in the Startup Option Config from the previous function trigger.

Takeaway: *The fault tolerance mechanism enhances the robustness by making the failed container restart at the desired state on the host where the container resides and the failure happens.*

V. Related Works

Several studies point out the essential of service liquidity in modern computing architecture; that is, services should be able to run and move freely across underlying platforms. Iorio *et al.,* [15] introduced the concept of Liquid Computing, a computing paradigm that abstracts services from the underlying computing continuum. Gallidabino *et al.,* [12] proposed the Liquid Software paradigm that offers a seamless experience to users while migrating across devices. Galantino *et al.,* [11] addresses the advantages of distributing a fluid workload across diverse computing devices, with a specific focus on the power consumption within the computing continuum.

Live container migration has been increasingly studied as an alternative solution to VM migration. Nadgowda *et al.,* [19] presented Voyager, a CRIU-based container migration service that utilizes post-copy filesystem replication. Voyager lazy replication transfers the filesystem of a container in the background when needed while allowing the container to resume operation instantly on the target host, providing zero-downtime data migration and reducing network overhead. Benjaponpitak *et al.,* [5] proposed CloudHopper, an automated live migration solution across multi-cloud while maintaining connectivity to the client service. CloudHopper allows live migration efficiently through pre-copy techniques and redirects the traffic between the cloud using HAProxy with the cloud provider's VPN. Among others, CloudHopper is the only live migration solution that takes connectivity between computing systems into account. Ma *et al.,* [17] proposed a framework for offloading services across the edge servers by leveraging the pre-copy live migration technique and eliminating the transfer of redundant container storage layers. The aforementioned works mainly relied on checkpoint/restore features supported by the container runtime. On the contrary, Souza *et al.,* [24] presented MyceDrive, a solution to migrate containers within a Kubernetes cluster by embedding checkpoint/restore libraries into the container. It allows only migration of the containerized service memory. FastMig provides an analogous live migration solution while simultaneously offering robustness enhancement to the container. FastMig is also open to integration with other migration solutions components through standardized interfaces.

VI. Conclusion

In this research, we leverage FastFreeze to establish a robust service liquidity solution, called FastMig, for the next generation of Cloud computing systems. The main idea behind FastMig is to separate the service management from its execution. It allows restarting the service without recreating the container and allows post-checkpointing operations, such as those for graceful shutdown instructions, thereby making it robust against failures that occur during the container migration. The decoupled pre-restoration operations spur a warm restoration technique that significantly reduces the overall live migration time. In addition, FastMig provides an HTTP-based interface; thus, the migration can be requested from the outer component without requiring the privilege to access the container command-line interface, and the solution can be easily integrated with existing systems. FastMig includes

a self-feedback fault tolerance mechanism that executes a configurable function to decide how the service should be restarted upon failure occurrence to enhance the robustness. Last but not least, FastMig is able to efficiently perform migration for multi-process services, commonly needed for service migration, via enhancing the restoration process of such services. The evaluations show that, firstly, incorporating FastFreeze or FastMig into the container imposes little to no additional resource and performance overheads in normal service operations(no migration). Secondly, FastFreeze-based live migration solution can operate efficiently with multi-process services when certain privileges are granted to the container through the appropriate container configuration. Thirdly, the container startup time can be overlapped during the migration when applying the warm restoration technique introduced by FastMig, which significantly reduces migration time during live migration. Lastly, we investigated the impact of the self-feedback fault tolerance mechanism and noticed that it introduces a negligible overhead while allowing the handling of flexible types of faults via a simple configuration.

There are several avenues to extend this research in the future. Firstly, machine learning techniques can be added to enable the flexible fault tolerance mechanism for the undefined faults. Currently, FastMig reacts to the undefined faults with the default behavior—restart in standby mode. As such, the second avenue for future research can be to enhance the robustness of service liquidity at the inter-system coordination level. For instance, if the hardware that runs the service fails permanently, the service unavailability should be detected, and a new instance of the service should be started/restored at the last location it has traveled to. The third avenue for future research can be on the security and privacy aspects of container migration, dealing with challenges such as secure container migration through a third-party network provider or to an untrusted destination system, and also dealing with the authorization and encryption challenges of the migrating container.

REFERENCES

[1] CRIU Main Page. https://criu.org/Main_Page. Accessed on 2023-12-4.

[2] Docker. https://www.docker.com/. Accessed on 2024-3-21.

[3] Kubernetes. https://kubernetes.io/. Accessed on 2024-3-21.

[4] Operating Lambda: Performance optimization. https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/. Accessed on 2024-3-30.

[5] Thad Benjaponpitak, Meatasit Karakate, and Kunwadee Sripanidkulchai. Enabling live migration of containerized applications across clouds. In *Proceedings of the 2020-IEEE Conference on Computer Communications*, pages 2529–2538. IEEE, 2020.

[6] Thanawat Chanikaphon and Mohsen Amini Salehi. Ums: Live migration of containerized services across autonomous computing systems. In *Proceedings of the IEEE Global Communications Conference*, pages 467–472. IEEE, 2023.

[7] Chavit Denninnart and Mohsen Amini Salehi. Smse: A serverless platform for multimedia cloud systems. *Concurrency and Computation: Practice and Experience*, 36(4):e7922, 2024.

[8] Chavit Denninnart, Thanawat Chanikaphon, and Mohsen Amini Salehi. Efficiency in the serverless cloud paradigm: A survey on the reusing and approximation aspects. *Software: Practice and Experience*, 53(10):1853–1886, 2023.

[9] Docker. What is a container? https://www.docker.com/resources/what-container/. Accessed on 2024-3-28.

[10] Daniil Ermolenko, Claudia Kilicheva, Ammar Muthanna, and Abdukodir Khakimov. Internet of Things Services Orchestration Framework Based on Kubernetes and Edge Computing. In *Proceedings of the IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, pages 12–17, 2021.

[11] Stefano Galantino, Fulvio Risso, Vlad C Coroamă, and Antonio Manzalini. Assessing the Potential Energy Savings of a Fluidified Infrastructure. *Computer*, 56(6):26–34, 2023.

[12] Andrea Gallidabino, Cesare Pautasso, Tommi Mikkonen, Kari Systä, Jari-Pekka Voutilainen, and Antero Taivalsaari. Architecting Liquid Software. *J. Web Eng.*, 16(5&6):433–470, 2017.

[13] Davood Ghatrehsamani, Chavit Denninnart, Josef Bacik, and Mohsen Amini Salehi. The art of cpu-pinning: Evaluating and improving the performance of virtualization and containerization platforms. In *Proceedings of the 49th International conference on parallel processing*, pages 1–11, 2020.

[14] Hamza Ali Imran, Usama Latif, Ataul Aziz Ikram, Maryam Ehsan, Ahmed Jamal Ikram, Waleed Ahmad Khan, and Saad Wazir. Multi-cloud: a comprehensive review. In *Proceedings of the 23rd International Multitopic Conference (INMIC 2020)*, pages 1–5. IEEE, 2020.

[15] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. Computing Without Borders: The Way Towards Liquid Computing. *IEEE Transactions on Cloud Computing*, 11(3):2820–2838, 2023.

[16] Eunsook Kim, Kyungwoon Lee, and Chuck Yoo. On the Resource Management of Kubernetes. In *Proceedings of the International Conference on Information Networking (ICOIN)*, pages 154–158, 2021.

[17] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. Efficient live migration of edge services leveraging container layered storage. *IEEE Transactions on Mobile Computing*, 18(9):2020–2033, 2018.

[18] Vincenzo Mancuso, Leonardo Badia, Paolo Castagno, Matteo Sereno, and Marco Ajmone Marsan. Efficiency of distributed selection of edge or cloud servers under latency constraints. In *Proceedings of the 21st Mediterranean Communication and Computer Networking Conference (MedComNet 2023)*, pages 158–166. IEEE, 2023.

[19] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. Voyager: Complete container state migration. In *Proceedings of the 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2137–2142. IEEE, 2017.

[20] Adrian Reber. capabilities: Introduce CAP_CHECKPOINT_RESTORE. https://patchwork.kernel.org/project/linux-security-module/patch/20200715144954.1387760-2-areber@redhat.com/, 2020. Accessed on 2023-12-4.

[21] Red Hat. Understanding containers. https://www.redhat.com/en/topics/containers. Accessed on 2024-3-28.

[22] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[23] Gursharan Singh and Parminder Singh. A taxonomy and survey on container migration techniques in cloud computing. *Sustainable Development Through Engineering Innovations: Select Proceedings of SDEI 2020*, pages 419–429, 2021.

[24] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes. In *Proceedings of the 6th IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 26–33. IEEE, 2022.

[25] Radostin Stoyanov and Martin J Kollingbaum. Efficient live migration of linux containers. In *Proceedings of the ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, June 28, 2018, Revised Selected Papers 33*, pages 184–193. Springer, 2018.

[26] Martin Straesser, André Bauer, Robert Leppich, Nikolas Herbst, Kyle Chard, Ian Foster, and Samuel Kounev. An empirical study of container image configurations and their impact on start times. In *Proceedings of the 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 94–105. IEEE, 2023.

[27] Ullah, Rehmat, Di Wu, Paul Harvey, Peter Kilpatrick, Ivor Spence, and Blesson Varghese. FedFly: Toward migration in edge-based distributed federated learning. *IEEE Communications Magazine*, 60(11):42–48, 2022.

[28] Nicolas Viennot. FastFreeze. https://github.com/twosigma/fastfreeze. Accessed on 2023-11-20.

[29] Jieming Zhu, Shilin He, Pinjia He, Jinyang Liu, and Michael R Lyu. Loghub: A large collection of system log datasets for ai-driven log analytics. In *Proceedings of the 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 355–366. IEEE, 2023.