Always-On Recording Framework for Serverless Computations: Opportunities and Challenges

Shreyas Kharbanda
Purdue University
West Lafayette, USA
skharban@purdue.edu

Pedro Fonseca
Purdue University
West Lafayette, USA
pfonseca@purdue.edu

Abstract

Serverless computing simplifies cloud programming by managing infrastructure and providing basic primitives for building distributed components. However, these efforts tend to increase complexity as developers create large, interdependent applications. The highly distributed nature of serverless computations makes it challenging for developers to comprehend the granular behavior of their applications. To address this problem, we present our vision for an always-on recording framework for serverless computations. Our goal is to efficiently capture low-level details of serverless computations by refining the ideas of traditional record-replay approaches while exploiting the properties of serverless environments. This envisioned approach is poised to provide greater serverless computation visibility for developers to diagnose faults, understand interactions between components, conduct performance analysis, and ensure security and efficiency. Fully tapping this potential will require exploring several research directions, ranging from low-cost recording, storage, and data management systems to practical replay analysis tools.

ACM Reference Format:
Shreyas Kharbanda and Pedro Fonseca. 2023. Always-On Recording Framework for Serverless Computations: Opportunities and Challenges. In *The 1st Workshop on SErverless Systems, Applications and MEthodologies (SESAME '23), May 8, 2023, Rome, Italy*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3592533.3592810

1 Introduction

Serverless computing is emerging as a popular cloud computing paradigm [35] for building highly distributed cloud applications, such as machine learning [10, 16, 17, 34, 55], video processing [11, 27], data analytics [37, 38], and code compilation [26]. In this model, developers upload self-contained, stateless logical units of their applications known as *serverless functions*, which are triggered by user-defined events and executed in short-lived, stateless sandboxes (e.g., containers [22]).

One of the key promises of serverless computing is to ensure a radically simplified cloud programming experience. Serverless platforms fulfill this promise by managing the underlying infrastructure and providing basic primitives for building distributed components, allowing developers to focus on their application's business logic. However, developers often use these basic primitives to build large, complex applications [32], which span multiple interdependent components across heterogeneous execution environments. These complex interactions between components, the limited visibility into code execution, and the lack of control over the underlying infrastructure jointly hinder the developers' ability to comprehend the granular details of function execution. Thus, these limitations impede developers from diagnosing faults, analyzing performance, auditing security, and conducting other necessary tasks for reliable, secure, and efficient execution of serverless applications.

Encouragingly, there has been notable progress in distributed system analysis techniques in the last decade. In particular, research on distributed tracing [19, 25, 50], failure injection [36], performance monitoring [8], log analysis [28], and profiling [45] has made great strides and shown that even limited information can help developers significantly.

However, existing techniques are often designed for general distributed systems and are not well-suited for the unique challenges and opportunities presented by serverless computing environments. For instance, the serverless functions' short-lived nature implies that even small, constant overheads can easily dominate the end-to-end latency [47]. As a result, techniques that require verbose execution logs (e.g., [28]) may incur prohibitive performance penalties. Similarly, since such ephemeral function instances are created and destroyed on-demand, it can be challenging for distributed tracing systems that rely on persistent identifiers [19, 25, 50] to accurately correlate and trace requests across component boundaries. Furthermore, serverless frameworks often share resources among functions at the level of the OS, libraries, and processes, making it challenging to discern which events correspond to which logical units, i.e., functions. Lastly, techniques such as failure injection may not be feasible due to the limited control over the underlying infrastructure in serverless platforms. As a result, there is a need for systems that can efficiently provide effective and accurate analysis of serverless computations.

Record-replay systems are particularly helpful for providing low-level insights into program execution, diagnosing faults, reproducing intermittent failures, and even performing forensic analysis [14, 23, 39, 43, 44]. Building on record-replay system, developers can implement diagnosis frameworks such as reverse debugging tools allowing practical analysis of fault, for instance. However, as these systems require fine-grained recording of programs, they can incur significant performance overhead. In fact, the traditional high overhead associated with such record-replay systems, which can reach 2× [44], generally precludes their use outside of debugging environments where large performance overheads are tolerable. This overhead can be prohibitive for production runs, where even

SESAME '23, May 8, 2023, Rome, Italy

ACM ISBN 979-8-4007-0185-6/23/05.
https://doi.org/10.1145/3592533.3592810

a small performance penalty per invocation can accumulate, potentially compounding tail latency and degrading overall application performance. Despite these traditional concerns with record-replay, we see in serverless computing unique opportunities that could reduce the typical overheads of record-replay and open the door for a range of use cases.

In this paper, we aim to understand the feasibility of an always-on recording framework for serverless computations, which can provide valuable insights into distributed serverless applications. In Section 2, we begin by discussing the challenges of applying traditional record-replay systems to the serverless context. In Section 3, we explore potential use cases for such a framework. In Section 4, we identify key properties of serverless computations that can address the performance limitations of traditional record-replay system. In Section 5, we discuss the opportunities, challenges, and long-term future directions of an always-on recording framework. In Section 6, we discuss how such a recording framework can be used in a commercial setting. Finally, we conclude our study with a review of related works in Section 7.

2 Traditional Record-Replay

The record-replay technique [14, 23, 39, 43, 44] involves two stages, the recording and the replay stage, and aims to ensure that users can replay a given program execution at a later time. Record-replay allows users and analysis frameworks to re-execute the exact execution, outside of the production run, where performance is not critical and invasive program analysis techniques (e.g., reverse debugging, data-flow analysis) can be employed, allowing users to diagnose failures and other executions of interest.

During the recording stage, the record-replay system captures in a trace file all sources of non-determinism that can impact program output and state. These sources include user input and other relevant events, such as system calls, signals, and context switches. Using the resulting trace file, the record-replay system can subsequently replay the execution by faithfully reproducing the recorded sequence of events, enabling the developer to observe the program's behavior and pinpoint the exact moment when specific issues occur (e.g., faults, performance bottlenecks, and security attacks).

Mozilla's RR Background. RR [44] is an efficient record-replay framework developed by Mozilla to enable lightweight recording and deterministic debugging of Linux applications. It is known for its maturity, completeness, and low overhead on real-world low-parallelism workloads. Across a range of real-world benchmarks, the record overheads, the most critical overheads in an always-on setting, were measured to range from 1.49× to 7.85× [44]. Although these are prohibitive overheads for deployment settings, they are an order of magnitude lower than prior record-replay techniques and make RR already practical for testing scenarios.

RR is implemented entirely in user space and runs on stock hardware, compilers, runtime, and operating systems, making RR easy to deploy and practical. Although other record-replay systems are available, we chose to focus in this paper on RR as an example because of its emphasis on efficiency and deployability. However, we expect our ideas to be generalizable to other implementations.

RR implements record-replay by combining several techniques. In particular, it employs `ptrace` to record program inputs (e.g., system calls and signals), preemptively schedules one thread at a time to avoid non-deterministic data races, and uses CPU performance hardware counters to measure application progress and deliver asynchronous signals and context switches at the right time during replay. RR also handles corner cases that arise in real-world applications, such as non-deterministic instructions (e.g., `RDSTC` and `RDRAND`), on a per-instruction basis to ensure accurate replay.

Using ptrace typically imposes significant overheads. For instance, when recording system calls with RR, `ptrace` traps induce four context switches between the tracee process (application) and the tracer process (RR). As a result, such context switches can impose high overheads and become system performance bottlenecks. RR mitigates this overhead source by implementing an in-process system-call interception library, which intercepts system calls directly in the tracee process. As a result, RR avoids frequent context switching between the tracee and the tracer process, improving performance and reducing overhead.

3 Real-World Serverless Use Cases

An always-on recording system for serverless computations has the potential to bring forth a wide range of benefits in the serverless ecosystem by enabling developers to understand the behavior of their applications better. To make this vision a reality, we need to build low-cost recording mechanisms that integrate into serverless frameworks and analysis tools that align with the ease of development and use that characterizes serverless computing.

Reverse-Execution Debugging. Reverse-execution debugging is a powerful technique that allows developers to step backward through the execution history of a program to identify the root cause of a bug. With traditional debugging methods, developers must recreate the conditions that caused the error, which can be time-consuming and complex. However, with the always-on execution recording framework, developers can replay the recorded history for a particular function invocation step-by-step and identify the point at which the error occurred. This ability not only reduces debugging time but also improves the accuracy of the debugging process. Moreover, since serverless applications can be complex and have many function invocations, this technique can be particularly beneficial in the serverless computing environment.

Reproducing Non-deterministic Failures. Non-determinism (e.g., inter-lambdas schedules), can lead to intermittent failures, which do not occur predictably or consistently and are consequentially difficult to reproduce and debug. Thus, diagnosing and resolving such failures require significant effort and time on the developer's end. Reproducing such failures is incredibly challenging when dealing with complex distributed systems where reasoning about the underlying behavior is often challenging. However, with an always-on recording system, developers can replay the recording using a debugger (e.g., GDB) and pinpoint the exact cause and any environmental factors that may have contributed to the failure. As a result, such a system can be especially useful to help diagnose and resolve rare and hard-to-reproduce failures. In addition, an always-on recording framework can also aid in integration and end-to-end testing, where non-deterministic factors such as timing and concurrency can lead to flaky results.

Replay Testing on Patches. Much of the program inputs that record-replay systems, like RR, capture during the recording phase (Section 2) are already stored in the serverless platform's data plane. In addition to leveraging this insight to enhance the efficiency of the recording phase, developers can use the data along with the recorded function invocations to check whether the bug can still be reproduced in the patched candidate version. As a result, such a validation technique can enable efficient and effective failure diagnosis and debugging in complex distributed systems.

Diagnosing Hardware Failures. Distributed systems are composed of many interconnected components, each with its respective hardware and software stacks. Hence, it can become challenging to diagnose hardware failures in highly distributed systems running in data centers. Thus, having access to a comprehensive record of applications' execution running on the infrastructure can be crucial in identifying behavior patterns that may indicate hardware failures. Moreover, such recordings can help cloud providers audit the fault tolerance of their systems and ensure the required quality of service to the customers.

Uncovering Performance Bottlenecks. Performance analysis is a crucial task for the proper functioning of distributed systems. With the rise of serverless computing, identifying and diagnosing such bottlenecks has become more challenging due to the lack of visibility into the underlying infrastructure. Always-on recording can help assess performance and address bottlenecks by enabling more fine-grained system analysis. Furthermore, recording at different levels of granularity can assist developers in pinpointing the exact location of a problem and diagnosing its root cause.

Comprehensive Security Audit. Security is paramount, especially in today's multi-tenant environments, and with the increasing use of third-party components [12, 53]. Always-on recording can enable developers to identify potential application vulnerabilities and weaknesses. First, developers can replay particularly malicious-looking invocations for behavior patterns that indicate an attack or an attempt at a security breach. Moreover, the recorded execution can also be used to reconstruct the sequence of events leading up to the incident, which can help identify the root cause and prevent similar incidents from happening.

4 Deploying Record-Replay in Serverless Environments

In this section, we explore the feasibility of using traditional record-replay systems in serverless computing environments. Specifically, we explore different approaches to build a serverless record-replay framework by integrating core components of RR (Section 2) with OpenFaaS, an open-source serverless platform, measuring its performance and identifying associated costs.

4.1 Straw Man Design and Experimental Setup

As a straw man design, we integrated RR with a modified version of OpenFaaS [5] as shown in shown Figure 1. Our modifications enabled OpenFaaS to deploy containers with the necessary `ptrace` capability and unconfined `seccomp` profile for RR to function effectively. Within each container, we employed a dispatcher process written in Python using the Flask [2] framework.

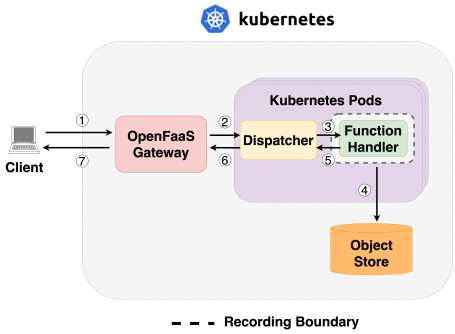

Figure 1: Straw man system design.

As Figure 5 shows, the client invokes the function (①), which passes through OpenFaaS's gateway (②), and is intercepted by the dispatcher. The dispatcher is configured to spawn a new RR process (③) for each function invocation, and the function handler runs as the child process of the RR process. Once the execution of the function handler completes, the recording of the execution is saved to an object store (e.g., MinIO [4]) (④), and the response is returned to the dispatcher (⑤). The dispatcher subsequently returns the response to the gateway (⑥), which returns it to the client (⑦). Our modified version of OpenFaaS is deployed on a Kubernetes cluster using FaaS-Netes [6], OpenFaaS's provider for Kubernetes. Finally, our Kubernetes cluster runs in a docker container using the `kind` (Kubernetes in Docker) project.

We conducted all system measurements on a virtual machine (VM) instance of Ubuntu 20.04.5 LTS (focal) that had 4 vCPUs, 8192 MiB of memory, and 50 GiB of disk space with the perf setting of `perf_event_paranoid` set to 1. The VM was hosted on a machine running Ubuntu 22.04 (jammy) LTS with 128 GiB of RAM on a 2.8 GHz AMD EPYC 7402P 24-Core processor.

4.2 Latency Overhead

Our experiments show that significant latency overhead arises when recording function invocations for deployed serverless functions using the straw man design (Figure 3). Specifically, we measure the client-side latency of two functions that are representative of compute-bound and IO-bound workloads: (1) a compute-bound Python function that multiplies a matrix (MatrixMultiplication), which was ported from the FaaSDOM benchmark suite [42], and (2) an IO-bound Python function (Uploader) that uploads a file from a provided URL to an object store, which was ported from the SeBS benchmark suite [20]. We use RR to perform the fine-grained recording of the function handler to measure the overhead induced in serverless computations accurately.

We ran both functions in two modes: with recording (represented by the orange line graph) and without recording (represented by the blue line graph) for 100 requests each. We observe that for the matrix multiplication workload, the recording enabled function invocation takes 328.20 ms on average with a standard deviation of 77.36 ms. In comparison, the recording disabled function invocation takes 169.86 ms on average, with a standard deviation of 5.63 ms. As a result, we can compute that the function invocation incurs 1.93× latency overhead when recording is enabled. Similarly, we observe that for the uploader workload, the recording enabled function invocation takes 838.80 ms on average with a standard deviation of

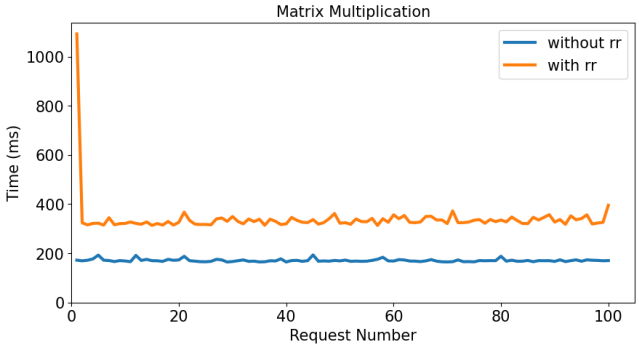

Figure 2: Client-side latency comparison between recording-enabled (using RR) and recording-disabled execution of the function handler for a matrix multiplication workload. The initial request experiences intermittently high overheads. Despite this initial instability, the system stabilizes quickly after the first request.

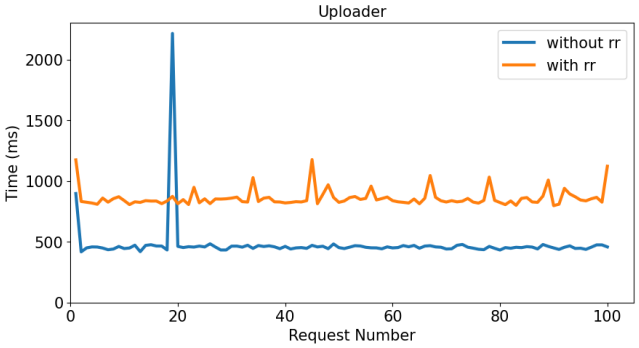

Figure 3: Client-side latency comparison between recording-enabled (using RR) and recording-disabled execution of the function handler for a file upload workload.

70.70 ms, while the recording disabled function invocation takes 456.00 ms on average with a standard deviation of 181.58 ms. As a result, we can compute that the function invocation incurs 1.84× latency overhead when recording is enabled. Thus, our findings confirm that RR, in its current form, imposes a high overhead that is prohibitive for always-on recording serverless computations of deployed functions.

We note that we focus on the recording stage cost because that is the main performance consideration for always-on production settings. This observation is made under the assumption that recording would happen more often than replaying, and recording would potentially operate on client-facing applications and workloads, which are generally performance critical. Furthermore, as RR is not multi-threaded, the latency increase also results in decreased throughput and corresponding increased CPU usage.

4.3 Overhead Sources

Interestingly, there are only a few primary overhead sources for recording computations with RR. However, some sources play a significant role in the overall system performance.

Instrumentation Mechanism. Because RR uses the `ptrace` kernel mechanism to intercept program events, in particularly system call IO, this process involves significant context switching between the application process (tracee) and the RR process (tracer) caused by the `ptrace` traps. The context switch overhead is especially detrimental to system calls that execute faster than the typical cost of a context switch, some of which are common (e.g., `gettimeofday`). RR tries to mitigate this overhead using an in-process system-call interception library, which intercepts system calls directly in the tracee process and shares the I/O of the system call with RR via a shared buffer. However, the short-lived nature of serverless computations undermines most of the performance gains of this optimization because short-lived processes pay the cost of library initialization but terminate before patching becomes mainstream.

Capturing Program Inputs. In order to accurately capture all the relevant program inputs, the system needs to record the inputs provided to the program through the system call interface.

During the system call traps, RR usually duplicates the system call output (i.e., inputs to the program) buffers to per-thread scratch buffers in order to prevent race conditions between the kernel's write to the output buffers and the running thread. After the completion of the system call, RR copies the contents of the scratch memory to the real user-space destinations, thereby keeping a copy of it. As the RR process blocks the running thread when copying the contents of the buffer to scratch memory, and subsequently to the actual user-space destination, this mechanism adds overhead to program recording, which can be particularly pronounced for IO-bound applications.

Uniprocessor Scheduling. Concurrency is typically a major contributor to the overhead incurred by RR, especially on highly concurrent programs. Since RR preemptively leverages uni-processor scheduling (i.e., it only allows a single thread to execute at a time) to resolve data races during recording, forcing a deterministic schedule [44], applications with high parallelism incur high overheads.

Other Overheads. Special instructions (e.g. `RDTSC`) present another source of program non-deterministic input that must be recorded for accurate replay. RR records such instructions by patching them explicitly. For instance, RR configures the CPU to trap `RDTSC` using Linux's `prctl` API. While dealing with special instructions requires custom patches to be written and maintained by RR developers, these patches do not significantly increase the recording overhead.

To better assess the overhead of a record-replay technique, we compared the overhead of RR with the overhead of only using strace on the function. Unlike RR, strace records both the input and output and relies exclusively on ptrace. As Table 1 shows, recording system results through ptrace, the traditional interposition mechanism, is very expensive for a default-on technique that is aimed to be deployed at scale. However, this experiment also shows that the other sources of overhead used by RR further and drastically increase the overheads when compared with the strace approach for ephemeral functions.

5 Opportunities and Challenges for Always-On Recording

As discussed in Section 4, RR in its current form has a high overhead, rendering it impractical to use outside debugging environments.

Function	Baseline	With RR	With `strace`
Matrix	169.9 ms (1.00×)	328.2 ms (1.93×)	206.2 ms (1.21×)
Uploader	455.9 ms (1.00×)	838.8 ms (1.84×)	540.2 ms (1.18×)

Table 1: Overhead comparison for two functions, Matrix and Uploader. Table compares the median per-request overhead caused by RR and `strace` over the baseline (i.e., without RR).

Hence, it is imperative to re-think the record-replay approach to make it cost-effective, given the unique properties of serverless computing environments.

5.1 Towards Always-On Serverless Record-Replay

This section explores the unique characteristics of serverless computing that allow for cost-effective insights into application behavior in production settings. Furthermore, it examines the challenges of implementing an always-on approach.

Limited Concurrency. The design of serverless platforms naturally encourages single-threaded execution by allowing developers to simply use more lambdas for parallelism [30]. As a result, serverless functions typically have limited (or no!) concurrency within the execution pod (e.g., a single thread runs in a container). Moreover, serverless functions are designed to be event-driven and invoked on-demand for a specific task rather than running continuously in the background.

Opportunity: *The lack of intra-lambda concurrency significantly (or entirely) reduces the need for schedule recording, a major source of runtime overhead in traditional record-replay [9, 41, 44]. When there is only a single thread of execution within a serverless function, we obviate the need to record shared memory accesses or enforce deterministic schedules during recording.*

Although serverless functions typically have low-to-no concurrency within a single lambda, the serverless model allows developers to scale out computations by deploying multiple instances of the same function concurrently (i.e., launching more lambdas in parallel). This means that an effective record-replay technique still needs to address inter-lambda concurrency, but this can be done through data-plane recording, as we discuss next.

Dataflow-Aware Platforms. In the serverless model, I/O operations are prevalent in the data-plane of serverless platforms. This means that large amounts of I/O data related to the serverless computations is stored within the data-plane (e.g., key-value stores, object stores). However, record-replay systems conduct expensive operations to capture I/O data associated with system calls, which negatively impact the performance of serverless applications, particularly in I/O-bound workloads.

Opportunity: *Since much of the inter-lambda IO is already tracked by the serverless infrastructure, we do not need special instrumentation techniques to intercept this program input. Importantly, not just we avoid instrumentation costs, but it may also be possible to leverage data already stored by serverless infrastructure, in particular, data stores, by exploiting the immutability of data in key-value stores or through store versioning techniques (e.g., [21]).*

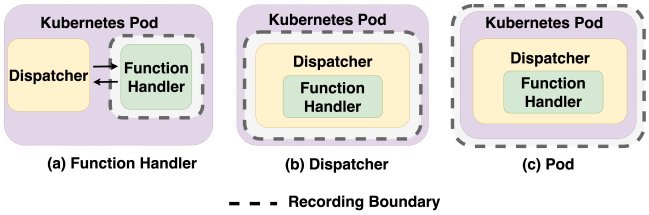

Figure 4: Levels of granularity for always-on recording.

Although the high-level IO arising from the interaction of the lambdas with stores or other lambdas can be addressed with this approach, instrumentation may still be required for low-level program IO, such as system calls made by libraries (e.g., getting the time of day, getting process ID). We expect to be possible to avoid the low-level IO overhead through dedicated runtimes that constrain the interactions with the kernel, such as minimal library OSes [48] or kernel extensions [13], or through optimized instrumentation mechanisms.

Extensive Code Reuse and Execution Redundancy. In serverless platforms, it is typical for developers to run multiple instances of the same function. As these instances execute the same code across similar inputs, there can be significant redundancy in the function computations. In recent times, distributed systems have employed techniques to aggregate knowledge distributed across nodes, thereby improving system efficiency. For instance, prior work leverages JIT aggregated runtime profiles across nodes to start functions faster [18] and proposes optimized fork system calls that generalize copy-on-write to the page tables [56].

Opportunity: *The computation patterns of serverless computing present an opportunity to enhance system-wide efficiency by aggregating knowledge about unique sub-parts of the execution traces and avoiding redundant analysis or binary patching.*

Transient Function Executions. In production, serverless functions typically have short-live executions [47], posing a challenge for record-replay systems, which are optimized for long-running computations. For instance, RR employs an in-process system-call interception library to bypass expensive context switches when recording system calls. However, this technique is ineffective in serverless environments due to the short lifespan of serverless functions, which prevent amortization of the library initialization cost, as shown in Figure 3.

Opportunity: *Reducing overheads by caching the in-process system-call interception library. In the serverless model, the checkpoint-restore technique has become prevalent. Various systems [51] use checkpoint-restore tools like* `CRIU` *to obtain the snapshot of the sandbox after the initialization is completed and subsequently use this snapshot to restore the state for future invocations, effectively bypassing the cold start. Leveraging this approach, we can obtain warm-starts and also cache record-replay system optimizations that have a high initialization cost [18].*

5.2 Recording Granularity

Serverless computations can be recorded at various levels of granularity, each with their own advantages and disadvantages. Figure 4

```
@app.route("/", defaults={"path": ""}, methods=["POST", "GET"])
def main_route(path):
    raw_body = os.getenv("RAW_BODY", "false")
    as_text = True
    if is_true(raw_body):
        as_text = False
    # Invoke the function handler by creating a child process per
    ↪  invocation with recording enabled.
    result = subprocess.run("rr record python3
    ↪  /home/app/function/handler.py", shell=True,
    ↪  stdout=subprocess.PIPE)
    output = result.stdout.decode()
    return {"response": output}
```

Figure 5: Code snippet for dispatcher's handling of a function invocation. For each invocation, the dispatcher (implemented using Flask) creates a new child process.

shows three levels of granularity, which range from fine-grained (recording each function handler), to slightly more coarse-grained (recording the dispatcher), and finally, coarse-grained (recording the entire sandbox, a Kubernetes pod in our case).

Function Level. The finest-level of granularity consists of recording at the function handler level. To achieve this level of granularity, the dispatcher spawns a new function handler child process with recording enabled per invocation. This model allows for a one-to-one relation between recording and function invocation, making it suitable for debugging and profiling. As each function invocation incurs the cost of launching a recorder process, recording at such granularity can incur significant overhead, which may be prohibitive for production environments.

Dispatcher Level. The slightly coarser level of granularity is where we record the dispatcher, which is responsible for handling requests for a particular serverless function. At this granularity, we launch only one recorder process for all the invocations that flow through the dispatcher. Figure 6 shows the latency overhead that arises when recording the dispatcher for deployed serverless functions. We use the same setup and methodology as the experiment outlined in Section 4. Compared to unrecorded invocations, we observe that the compute-bound function incurs an overhead of 8% and the IO-bound function incurs an overhead of 17%, which are both much lower than the overhead of recording with fine-grained granularity. Furthermore, recording at this granularity may enable interesting applications such as differential request analysis through request annotation. However, as the scope of recording increases, we may also have higher storage requirements and incur slowdowns caused by multi-threading in the dispatcher (but not lambda functions).

Sandbox Level. At the coarsest level of granularity, the recording is done at the sandbox level. This involves launching a single recorder process for the entire lifecycle of the serverless function. While it provides a comprehensive view of the environment and is useful for identifying infrastructure and network issues, recording the entire pod for the function's entire lifecycle can add considerable overhead. Moreover, the record-replay system may experience slowdowns due to multi-threaded executions, making it challenging for production deployments.

5.3 Long-term Research Directions

Besides the opportunities that we identify in the previous section, suggesting the viability of an always-on record framework for serverless, we further elaborate on potential long-term research directions that address open problems in this setting.

Prevalence of Managed Runtimes. Most serverless functions are written in high-level languages, which run on managed runtimes within the sandboxes. For example, in AWS Lambda [31], 89% of all serverless functions are written in Node.js and Python [49]. One challenge of managed runtimes is that they add significant complexity, which may hinder analysis during replay. This complexity arises from their advanced features and complex language requirements and includes notable examples such as garbage collection, just-in-time compilation, and dynamic class loading. Hence, it is important to build techniques that can provide actionable data to developers when such executions are replayed, especially if failures arise from subtle interactions between the language and the runtimes (e.g., failures dependent on the memory layout or JIT optimizations).

Disparate Traffic Patterns. Due to the event-driven nature of serverless computing, there is a high variance in traffic patterns of different workloads. This variance can be particularly challenging in burst-parallel applications, such as data analytics, video encoding, and code compilation [54], which trigger highly parallel tasks with thousands of serverless functions. Such workloads can generate a massive amount of data to record, which can result in large recording volumes of data, making it challenging to store and manage the data. As a result, it is crucial for an always-on recording framework at the data center level to be sufficiently elastic to accommodate workloads bursts of serverless applications and efficiently manage recorded data.

Pervasive Heterogeneity. Serverless computing environments can be heterogeneous with various combinations of hardware and software, including operating systems and runtimes. This heterogeneity in serverless platforms, such as the one found in AWS Lambda [31], Azure Functions [1], and Google Cloud Functions [3], can present a hindrance in enabling efficient and reliable recording of serverless computations. As a result, a practical always-on recording framework needs to support hardware and software heterogeneity.

Sensitive Data and Code Execution. Serverless functions run in highly consolidated multi-tenant environments [7], and therefore have rigorous security requirements. Incorporating always-on recording adds another layer of complexity to the security landscape. By nature, recording the serverless computations captures a wealth of data, including potentially sensitive user I/O, environment variables, and secrets. As a result, such recordings increase the attack surface of the system and make it vulnerable to data breaches. Furthermore, as replaying the code requires execution of instructions, attackers can potentially tamper with the recordings or falsify them to execute malicious code, potentially compromising the integrity and security of the system. As a result, it is crucial implement appropriate security measures, such as encryption and access control mechanisms, to mitigate potential risks.

Trace Data Management. Serverless computations often involve multiple small functions that interact with each other to accomplish

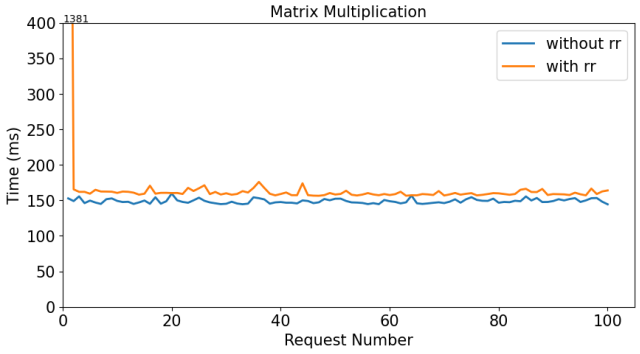

Figure 6: Client-side latency comparison between recording-enabled (using RR) and recording-disabled execution for a matrix multiplication workload with the granularity set to the dispatcher level.

a task, which can result in a large number of function invocations and complex control flows. As a result, the always-on recording framework can generate a significant amount of trace data that needs to be processed and analyzed, which can be computationally expensive and may become a bottleneck in the analysis pipeline. As a result, it is crucial to enhance the efficiency of trace data storage and processing in order to avoid incurring prohibitive costs.

Serverless Testing. Despite extensive work on testing techniques [24, 29, 40], serverless testing poses unique challenges due to the distribution, heterogeneity, and scale of serverless workloads and environments. A record-replay system could alleviate some concerns with the testing space size by allowing developers to direct testing toward faults observed in specific deployments [15].

6 Discussion

Always-on, selective-on, and pay-to-be-on in commercial clouds. Our work explores the technical feasibility of building a record-replay technique with sufficiently low overhead to be considered an always-on feature made available to all serverless functions. A low-cost approach aims to ensure that the overall computational cost of the technique is low and that the impact on end-users is low for end-user-facing services. Despite our preliminary analysis showing encouraging results, we still expect some overheads and hence costs. A non-zero cost means that users are still incentivized to selectively enable such a feature when the value outweighs the costs. When enabled, the users would ultimately pay these costs according to the provider's business model in a commercial setting. Furthermore, the record-replay mechanism we envision would add value to the service, representing an added-value opportunity that could be charged according to its utility. Thus, whether a low-cost record replay mechanism becomes an always-on service or a pay-to-be-on service will ultimately depend on the cloud provider and developer choices.

7 Related Work

There is a growing body of work dedicated to understanding and diagnosing the complex behaviors of distributed systems. One such technique is distributed tracing, which is used to understand the high-level path of a request as it flows through the different components of the system. Black-box systems [8, 46] conduct tracing transparently to the application, however, may have imprecision and higher overheads due to their reliance on statistical inference for determining causality. On the other hand, some approaches [19, 25, 50] modify libraries or other system layers to determine causality. Distributed tracing systems often rely on persistent identifiers to accurately trace requests across component boundaries in a distributed environment. Fault injection is another well-known technique, which is used to enable more effective testing strategies for distributed systems to ensure their robustness and dependability. Fault injection systems [36] are used which deliberately introduce faults into the system, both hardware and software, ranging from simple and deterministic to more complex and non-deterministic.

Record-replay systems have been used for quite some time to provide accurate low-level insights into the execution of a program. These systems record all the sources of non-determinism such as system calls, signals, and context switches during execution and then replay these events to deterministically reproduce the behavior of program. Several approaches [14, 23, 39] have limited deployability due to their coarse granularity and required modifications to underlying hardware or software. RR [44] is a record-replay system that is particularly well suited for mass deployment because of its comprehensive support for real-world program features and reduced overhead, especially for low-parallelism workloads.

Additionally, systems like Boki [33] and AFT [52] provide novel mechanisms to manage the state of serverless functions and enhance the reliability of serverless applications. Some of the techniques rely on intercepting environment interactions to improve the guarantees of serverless functions. Unlike these approaches, our goal is to record the entire function execution such that the effects and state of the functions can be faithfully replayed.

8 Conclusion

To address the programming, debugging, and analysis challenges of serverless computing, we have presented our vision for an always-on recording framework that efficiently captures low-level details of serverless computations, building upon traditional record-replay approaches. In this paper, we have examined the design considerations for such a framework and discussed the potential opportunities and challenges for building it. Additionally, we have outlined promising future research directions for improving the system. We believe that such a framework has the potential to provide greater visibility for developers, enabling them to diagnose faults, understand interactions between components, conduct performance analysis, and ensure security and efficiency.

References

[1] 2023. Azure functions. https://azure.microsoft.com/en-gb/products/functions/
[2] 2023. Flask. https://flask.palletsprojects.com/en/2.2.x/
[3] 2023. Google Cloud. https://cloud.google.com/
[4] 2023. MinIO Object Storage. https://min.io/docs/minio/kubernetes/upstream/index.html
[5] 2023. OpenFaaS. https://www.openfaas.com/
[6] 2023. OpenFaas FaaS-Netes: Serverless functions for Kubernetes. https://github.com/openfaas/faas-netes
[7] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings*

of the 17th Usenix Conference on Networked Systems Design and Implementation (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 419–434.

[8] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP '03)*. Association for Computing Machinery, New York, NY, USA, 74–89. https://doi.org/10.1145/945445.945454

[9] Adil Ahmad, Sangho Lee, Pedro Fonseca, and Byoungyoung Lee. 2021. Kard: Lightweight Data Race Detection with per-Thread Memory Protection. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3445814.3446727

[10] Ahsan Ali, Syed Zawad, Paarijaat Aditya, Istemi Ekin Akkus, Ruichuan Chen, and Feng Yan. 2022. SMLT: A Serverless Framework for Scalable and Adaptive Machine Learning Design and Training. https://doi.org/10.48550/ARXIV.2205.01853

[11] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[12] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016).

[13] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. 2013. Composing OS Extensions Safely and Efficiently with Bascule. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/2465351.2465375

[14] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. 2010. Deterministic Process Groups in dOS. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/osdi10/deterministic-process-groups-dos

[15] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2329–2344. https://doi.org/10.1145/3133956.3134020

[16] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2018. A Case for Serverless Machine Learning. In *Proceedings of Workshop on Systems for ML at NeurIPS (MLSys)*.

[17] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. 2019. Cirrus: A Serverless Framework for End-to-End ML Workflows *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3357223.3362711

[18] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 58–64. https://doi.org/10.1145/3458336.3465305

[19] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services. In *Proceedings International Conference on Dependable Systems and Networks*. 595–604. https://doi.org/10.1109/DSN.2002.1029005

[20] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 64–78. https://doi.org/10.1145/3464298.3476133

[21] Giuseppe deCandia, Deniz Hastorun, Madan Mohan Rao Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Symposium on Operating Systems Principles*. https://doi.org/10.1145/1294261.1294281

[22] Docker. 2023. https://www.docker.com/

[23] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2002. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. In *5th Symposium on Operating Systems Design and Implementation (OSDI 02)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/osdi-02/revirt-enabling-intrusion-analysis-through-virtual-machine-logging-and-replay

[24] Pedro Fonseca, Xi Wang, and Arvind Krishnamurthy. 2018. MultiNyx: A Multi-Level Abstraction Framework for Systematic Analysis of Hypervisors. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys)* (Porto, Portugal). Article 23, 12 pages. https://doi.org/10.1145/3190508.3190529

[25] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[26] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[27] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[28] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *2009 Ninth IEEE International Conference on Data Mining*. 149–158. https://doi.org/10.1109/ICDM.2009.60

[29] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. 2021. Snowboard: Finding Kernel Concurrency Bugs through Systematic Inter-thread Communication Analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 66–83. https://doi.org/10.1145/3477132.3483549

[30] Joe Hattori and Shinpei Kato. 2022. Sentinel: A Fast and Memory-Efficient Serverless Architecture for Lightweight Applications. In *Proceedings of the Eighth International Workshop on Serverless Computing* (Quebec, Quebec City, Canada) *(WoSC '22)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3565382.3565880

[31] Roger W. Hendrix. 1983. AWS Lambda. https://aws.amazon.com/lambda/

[32] Baskaran Jambunathan and Kalpana Yoganathan. 2018. Architecture Decision on using Microservices or Serverless Functions with Containers. In *2018 International Conference on Current Trends towards Converging Technologies (ICCTCT)*. 1–7. https://doi.org/10.1109/ICCTCT.2018.8551035

[33] Zhipeng Jia and Emmett Witchel. 2021. Boki: Stateful Serverless Computing with Shared Logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[34] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 857–871. https://doi.org/10.1145/3448016.3459240

[35] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. https://doi.org/10.48550/ARXIV.1902.03383

[36] Wei-Lun Kao and R.K. Iyer. 1994. DEFINE: a distributed fault injection and monitoring environment. In *Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*. 252–259. https://doi.org/10.1109/FTPDS.1994.494497

[37] Youngbin Kim and Jimmy Lin. 2018. Serverless Data Analytics with Flint. https://doi.org/10.48550/ARXIV.1803.06354

[38] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic

[39] Oren Laadan, Nicolas Viennot, and Jason Nieh. 2010. Transparent, Lightweight Application Execution Replay on Commodity Multiprocessor Operating Systems. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (New York, New York, USA) *(SIGMETRICS '10)*. Association for Computing Machinery, New York, NY, USA, 155–166. https://doi.org/10.1145/1811039.1811057

[40] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-level Virtualization for Functional Interference Bugs. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3575693.3575731

[41] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 327–336. https://doi.org/10.1145/2043556.2043587

[42] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. 2020. FaaSdom: A Benchmark Suite for Serverless Computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems* (Montreal, Quebec, Canada) *(DEBS '20)*. Association for Computing Machinery, New York, NY, USA, 73–84. https://doi.org/10.1145/3401025.3401738

[43] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazières, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (2017).

[44] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability: Extended technical report. https://arxiv.org/abs/1705.05937

[45] Aidi Pi, Wei Chen, Xiaobo Zhou, and Mike Ji. 2018. Profiling Distributed Systems in Lightweight Virtualized Environments with Logs and Resource Metrics. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing* (Tempe, Arizona) *(HPDC '18)*. Association for Computing Machinery, New York, NY, USA, 168–179. https://doi.org/10.1145/3208040.3208044

[46] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. 2006. WAP5: Black-Box Performance Debugging for Wide-Area Systems. In *Proceedings of the 15th International Conference on World Wide Web* (Edinburgh, Scotland) *(WWW '06)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/1135777.1135830

[47] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[48] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, Lausanne, Switzerland, 955–970. https://doi.org/10.1145/3373376.3378469

[49] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. 2022. Fireworks: A Fast, Efficient, and Safe Serverless Framework Using VM-Level Post-JIT Snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 663–677. https://doi.org/10.1145/3492321.3519581

[50] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[51] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3423211.3425682

[52] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. 2020. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/3342195.3387535

[53] Hassan Takabi, James B. D. Joshi, and Gail-Joon Ahn. 2010. Security and Privacy Challenges in Cloud Computing Environments. *IEEE Security & Privacy* 8 (2010), 24–31. https://doi.org/10.1109/MSP.2010.186

[54] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. 2020. Particle: Ephemeral Endpoints for Serverless Networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 16–29. https://doi.org/10.1145/3419111.3421275

[55] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: A Native Serverless System for Low-Latency, High-Throughput Inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[56] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-Demand-Fork: A Microsecond Fork for Memory-Intensive and Latency-Sensitive Applications. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3447786.3456258