R-Check: A Reactive Checkpointing Approach For Serverless Computing

Rafael Alexandre
IST and INESC-ID
University of Lisbon
Lisbon, Portugal

Abstract

Serverless computing allows developers to not worry about server management by abstracting away the provisioning of computing resources. However, developing applications to run on Serverless platforms is challenging because these platforms only guarantee *at-least-once* semantics. Developers are left with the task of implementing idempotent code, i.e., code that can be restarted without unintended side-effects. Logging and periodic checkpointing have been proposed to alleviate this problem, but these impose noticeable performance overheads. Furthermore, these problems tend to be heightened as cloud providers optimize the use of their infrastructure, e.g., by leveraging harvested resources – cheaper compute nodes that can be evicted if the cloud provider needs them for higher-priority customers. We advocate for a different approach to handling function interruptions in Serverless, through a reactive checkpoint-based mechanism. Our core insight is that evictions are controlled events and should be handled by migrating executions in a structured and reactive way when such faults occur. We discuss the benefits and limitations of our idea, review design alternatives and present the design and implementation of R-Check, a system that follows this approach. Our solution was evaluated on an Apache's OpenWhisk deployment under different benchmarks and conditions, showing that R-Check can be an effective and affordable approach to host Serverless applications on harvested resources.

Keywords: Serverless Computing, Fault Tolerance, Checkpoint/Restore, Reactive

1 Introduction

Serverless computing has emerged as an increasingly compelling cloud programming paradigm, especially in the form of Function-as-a-Service (FaaS). Services of this kind are now provided by all the major cloud providers (Amazon's AWS Lambda [1] or Microsoft's Azure Functions[16]) as well as open-source platforms (Apache OpenWhisk[17]).

FaaS offers an intuitive, event-based interface for developing cloud applications. These services aim to completely hide the management of machines, runtimes, and resources (i.e., everything except the application logic) from the programmer side. For that purpose, it provides an abstraction where developers upload one or more simple functions to the cloud provider. Each such function can then be invoked on demand. These functions boot much faster than a traditional VM, allowing tenants to quickly launch many compute nodes without provisioning a long-running cluster. The cloud provider is then responsible for handling all the underlying infrastructure burden. Serverless was designed to execute short-running and stateless functions. Even today, providers strongly guide developers to write idempotent code. This is so that, in case of a fault (e.g., termination of the underlying container) or for load balancing purposes (e.g., the resources are needed elsewhere), the function can be re-deployed in a different node without unintended consequences to the execution. Consequently, cloud providers can adopt a replay-based approach to Serverless fault tolerance, where the function is re-executed all over again.

A recent trend in cloud computing is for cloud providers to optimize the use of their infrastructure by leveraging spare available resources. In fact, a recent study [20] proposes running Serverless applications on top of harvested VMs i.e., VMs obtained at massive discounts that may be terminated (evicted) at any moment, as part of the cloud provider's resource management. To take advantage of harvested resources, it becomes even more glaring that all functions need to be idempotent, so that they can be restarted upon an eviction. However, this requirement is at odds with the vision of Serverless being the future of cloud computing. It is unreasonable for programmers to adapt their entire cloud code base to become idempotent. In particular, non-deterministic or stateful functions may diverge when replayed (for example, a credit card payment, if a failure happens, may be executed twice). Another trend that reinforces these problems is long-running computations: execution times tend to increase given that Serverless platforms are being used for an increasing variety and complexity of workloads. In these scenarios, not only the chances that a function is terminated (and replayed) increase but also the cost of running a function all over again becomes increasingly expensive.

Recent work proposes solutions that may address this problem, namely by recording the computation through logging or periodic checkpointing. Logging approaches, namely Beldi [18] and Boki [12], save application state when there are any external read/write operations. On the other hand, Kappa [19] uses periodic checkpointing by inserting custom checkpoint code inside functions. These solutions enable the

function to restart from a point close to where it stopped, without duplicating the external outputs. However, both approaches introduce overheads that are present in every execution.

Observing this research opportunity, we aim to deploy Serverless functions that may be terminated before the end of their execution and ensure that operations are executed *exactly-once*, while avoiding any runtime overheads in the vast majority of the executions. To embody this vision, we present R-Check, an efficient and reactive checkpoint-based framework for fault tolerance in Serverless computing. R-Check can leverage not only the eviction grace period granted before harvested resources are fully terminated but also the bounded time limit in Serverless platforms to efficiently snapshot the application state when the system is about to fail and resume from it afterwards.

The main contributions of this thesis are as follows: (1) we proposed a new fault tolerance model for Serverless computations that uses a reactive checkpointing approach to lift the restrictions that planned evictions and function timeouts today pose to cloud functions, while requiring no changes to existing cloud function code; (2) we designed and implemented a prototype based on these principles, and deployed in Apache OpenWhisk [17], a popular open-source Serverless platform; (3) we evaluated its effectiveness with several different cloud functions, and compared it with state-of-the-art fault tolerance solutions for FaaS. Using R-Check, we were able to successfully checkpoint and restore numerous Serverless applications and do so transparently, reducing the number of executions paying that extra cost. When compared to Beldi and Kappa, R-Check achieves lower overall runtime overhead while being applicable to a much higher variety of workloads.

The rest of the document is organized as follows. Section 2 introduces some background concepts and motivates this work with a review of selected work related to fault tolerance in Serverless. In Section 3, we present the architecture of R-Check and explain, in detail, its design choices and how it was implemented. Section 4 is an experimental evaluation, details the scenarios tested, and analyzes a set of experimental results. Finally, Section 5 presents a brief conclusion summarizing the contributions of this thesis and highlight possible directions for future work.

2 Background and Related Work

Unlike traditional VMs (IaaS), Serverless users are relieved from dealing with the infrastructure required to run their computation. This responsibility is instead handed over to the cloud provider. As a result, customers benefit from automatic scalability and great elasticity as services scale up and down automatically as requests arrive. Furthermore, Serverless platforms charge only for the resources that are used during function execution, further moving cloud platforms towards a true *pay-as-you-go* billing model.

	Planned Faults[1]	Crash Faults[1]	No Runtime Overhead	No User Involvement
FaaS	✗	✗	✓	✓
R-Check	✓	✗	✓	✓
IaaS	✓	✗	✓	✓
Kappa [19]	✓	✓[2]	✗	✗
Beldi [18]	✓	✓	✗	✗

[1] In the case of non-idempotent functions.
[2] Since Kappa takes periodic checkpoints, some computation might be lost when recovering from crash faults.

Table 1. How evictions and crash faults are handled in different systems and platforms.

The existing service offer of Serverless is based on the Function-as-a-Service (FaaS) programming model. In FaaS, users register functions in the cloud function provider. Then, when a user requests or a pre-defined event occurs, the platform assigns the function to one of its computation nodes. The node, which is a short-lived, stateless execution environment, executes the function and sends the result back to the client. It is also relevant to note that popular Serverless offerings limit the function execution time to a fixed time limit (for example, up to 15 min in 1 s increments on AWS).

Serverless computing is becoming an increasingly compelling option for deploying applications in the cloud. Recent papers have exploited using Serverless for tasks beyond event handling, namely for video processing [10], data analytics [14] and machine learning [11]. With the progression of Serverless to include more general-purpose applications, functions tend to become more long-running and it becomes more difficult to ensure their idempotency.

A model where invocations can be evicted if resources become necessary to host an higher-priority tenant/service has been recently explored in the context of novel Serverless platforms based on harvested resources [20]. Harvested resources such as Harvest VMs [3] or Spot instances [5] are offered by major cloud providers. This type of Virtual Machines leverages temporarily unused resources that one can take advantage of at a significantly discounted (and variable) rate. The core difference between harvested resources and regular virtual machines is that regular VMs enjoy uninterrupted availability. In contrast, harvested resources can get evicted at any time. Even so, customers are given a short eviction notice (generally, a 30-second up to 1-minute grace period) to handle any sensitive ongoing computation. In an ideal world, functions would always be idempotent; therefore, we could simply restart them upon an eviction event (this is the default behaviour in public Serverless platforms). However, this is not always the case, and writing idempotent code can

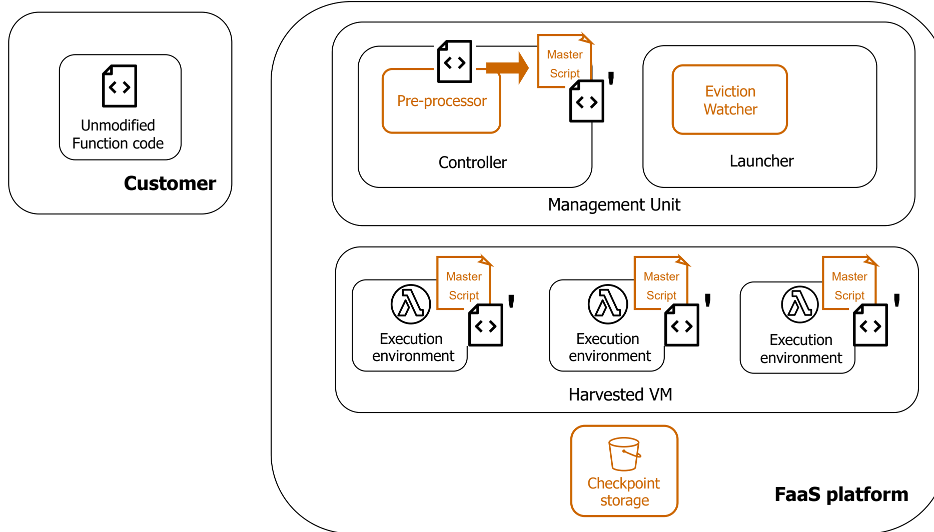


Figure 1. Overview of a generic FaaS platform with R-Check's specific components highlighted in orange.

easily become an entry barrier for developers of Serverless code. In this work, we propose that functions should not be restarted but migrated. Our proposal is grounded on the following insight: *evictions are controlled faults*. In contrast to unexpected crash faults, planned faults such as evictions can be handled by migrating the function before they take place. As a result, the requirement of idempotency can be lifted.

Table 1 presents how different cloud platforms and systems handle two different types of faults: planned and and crash faults. FaaS platforms have no out-of-the-box support for planned or crash faults. R-Check and IaaS support planned faults, while systems that employ periodic checkpointing (Kappa [19]) or logging (Beldi [18] and Boki [12]) can cope with both planned and crash faults. However, these systems handle the latter at the expense of high runtime overheads and some user involvement. Thus, a key observation we make is that, when attempting to lift the restrictions that planned faults in Serverless pose to cloud functions, previous work has gone further than the existing offer of IaaS cloud services (in terms of the classes of faults that are tolerated). Unfortunately, that also comes with the cost of imposing a runtime overhead in every single run to guard against events that only happen in a small subset of the invocations.

In summary, R-Check strikes a balance between FaaS (which does not support planned faults for general code) and systems that support both planned and crash faults (the latter of which are not handled by the system in today's cloud offer), by proposing a fault model that, similarly to IaaS, supports planned faults but not crash faults.

3 R-Check

3.1 Architecture

R-Check[1] extends an existing FaaS platform with the ability to handle the migration of a function from one node to another without incurring any overheads in the typical case where functions run on the same machine until completion. Depicted in Figure 1 is the high-level architecture we propose.

Any generic FaaS platform comprises a controller unit that handles communication with the customer and function creation. Besides, a launcher unit is responsible for launching isolated execution environments where functions will run, deploying functions on them and monitoring their execution. With R-Check, the customer is only required to provide the application code as they would in any other platform to create functions and trigger requests to execute the said function. We extend the existing controller unit with a new pre-processor component responsible for repackaging the function with a master script and transforming the user code. The master script acts as a function runner, monitors the execution and provides an endpoint to where termination and restore requests can be sent. Regarding the user code, the pre-processor inserts a plug-and-play trigger to run the function through the master script. In the launcher unit, we introduce a separate module called Eviction Watcher to track the eviction notices of the underlying resources that power the function execution environments of the platform, that are now running on top of harvested VMs. It periodically queries for evictions and spins up new VMs to maintain a minimum pool of available resources if they fall below a pre-configured threshold.

[1] Short for Reactive Checkpointing.

Algorithm 1: R-Check master script

```
// Checkpointing segment of the master script
Input  : fn, params
Output: r_check_result
begin
    exec_id ⟵ run_fn(fn, params)
    fault ⟵ FALSE
    fn_result ⟵ get_execution_result(exec_id)
    while not fault and fn_result == null do
        fault = fetch_termination_notice()
        fn_result ⟵
          get_execution_result(exec_id)
    if fault then
        chkpt_result ⟵ checkpoint_fn(exec_id)
        r_check_result ⟵
          chkpt_result, CHKPT_OK
    else
        if success(fn_result) then
            r_check_result ⟵ fn_result, SUCCESS
        else
            r_check_result ⟵ fn_result, ERROR
---------------------------------
// Restoring segment of the master script
Input  : exec_id
Output: r_check_result
begin
    restore_function(exec_id)
    ...    // checkpointing may be triggered
    ...    // until the function returns
    fn_result ⟵ get_execution_result(exec_id)
    if success(fn_result) then
        r_check_result ⟵ fn_result, SUCCESS
    else
        r_check_result ⟵ fn_result, ERROR
```

A typical function execution in R-Check works as follows. The master script, whose logic is described in Algorithm 1, runs the user functions with the user's desired parameters and waits until its completion. If a termination notice is detected by the eviction watcher, the launcher relays a termination request to the corresponding execution environments (all of the execution environments in a VM if an eviction is occurring or to a specific function environment if a function timeout is approaching). This request is captured by the function's master script. The master script triggers the checkpointing mechanism and checkpoints the function invocation immediately. The checkpoint object associated with that execution ID is then stored in checkpoint storage. The function is re-launched when the launcher receives confirmation that all the checkpoints have been taken. Then. the master script is requested to retrieve the checkpoint object with the previous ID from storage and re-establishes the context and state of the function execution. If the execution finishes successfully, the result of the user function is retrieved by the master script that sends it back to the platform, which, in turn, sends it to the user.

3.2 Checkpoint and Restore in Userspace (CRIU)

Many of the popular system-level approaches for checkpointing are not entirely suitable for Serverless computations because they either require modifications to the kernel (BLCR [9]), which the cloud provider would have to implement or introduce additional overheads on system calls (DMTCP [4]). On the other hand, CRIU [8] can checkpoint/restore a broader range of applications and features while not imposing extra overheads on wrapped calls, so it stands out as the best tool available for our use case. We now describe a typical CRIU checkpont/restore process.

When **checkpointing** a process with a PID `p`, the entire tree of processes associated with PID `p` as the root process must also be checkpointed. CRIU starts by recursively freezing all of these processes and their respective threads. While doing so, CRIU *attaches* to these processes through the `Ptrace` system call. Once CRIU has frozen and controls all relevant processes, it begins reading those processes' state. Much of this information can be read from the filesystem directly, particularly from the `/proc/{PID}` directory. The rest of the information needed is read from the process itself. For that purpose, CRIU injects *parasite code* into each process, saving memory contents and UNIX credentials. All this information is saved in a dump file, the parasite code is removed, and the processes can resume or be killed.

Upon **restoring**, CRIU first looks for the path `/proc/sys/kernel/ns_last_pid`, where the last assigned PID is stored. To restore the desired PID `P` from the root process of the application, CRIU locks `ns_last_pid`, writes `P-1` and calls *clone*(). It then proceeds to open files, creates maps and sockets and restores other basic process state. CRIU eventually transforms itself into the target process and, in order to do that, some code must exist that will unmap CRIU's memory, replacing it with the target processes' memory. For this purpose, a "restorer blob" is created. All memory mappings, timers, credentials and threads are finally restored by the "restorer blob", before restarting the process.

3.3 Employing CRIU in R-Check

CRIU was originally built to snapshot processes in serverful computing. As such, it is not equipped with out-of-the-box support for our use case. We now expand on some improvements made to CRIU so that it fits more closely our requirements.

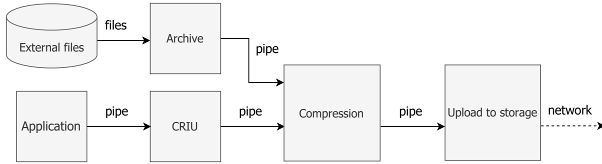


Figure 2. Modified CRIU data flow.

Checkpoints directly in remote storage. The primary issue in this approach is that CRIU hits local disk, both on checkpointing and then on restore, which is a significant performance bottleneck. In addition, Serverless environments are generally memory-limited, which implies that checkpoint images, depending on the function's footprint, may exhaust the resources allocated to said function environment. To address these concerns, we use a modified version of CRIU (depicted in Figure 2) that streams their content directly to a UNIX pipe instead of buffering checkpoint files in local storage. This pipe is then plugged into the compression stage, and its output is, in turn, uploaded into the storage unit.

From the restore side, the process is relatively similar to its checkpointing counterpart, except we reverse the data flow that we just presented. This restore process, however, presents a problem: CRIU reads the checkpoint files out-of-order. As a solution, we buffer the entire CRIU checkpoint image in memory so that CRIU can access the files in whatever order it desires. Consequently, this may create a "2× memory" problem where we can possibly have the full checkpoint image in memory and, simultaneously, the application restored next to it with similar contents to one another. To deal with this potential issue, during restore, as CRIU reads data, that memory is deallocated immediately after.

Local files of a function. To execute a successful migration of the computation from one container to another, it is necessary to make sure that the files accessed by the applications are available on both ends of the checkpoint/restore process. However, CRIU does not provide any support for cloning relevant files from one side to the other. We mitigate this limitation by taking advantage of the fact that Serverless environments generally provide a small writable folder which developers can use (usually `/tmp`). With R-Check, we bundle all the contents inside this writable folder and plug the archive into the same compression process we use for CRIU's contents. The final checkpoint file uploaded to storage can then be used alone to re-deploy the function on any Serverless environment.

4 Experimental Evaluation

In this section, we evaluate R-Check's overall performance. We measure the overhead introduced by checkpointing and restoring using a microbenchmark and several real application benchmarks to demonstrate the generality of R-Check. After that, we compare it with an application-specific method of checkpointing and with other state-of-the-art periodic approaches in this space. Finally, we conduct an experiment to determine the scalability of R-Check as the number of functions co-located on the same machine increases.

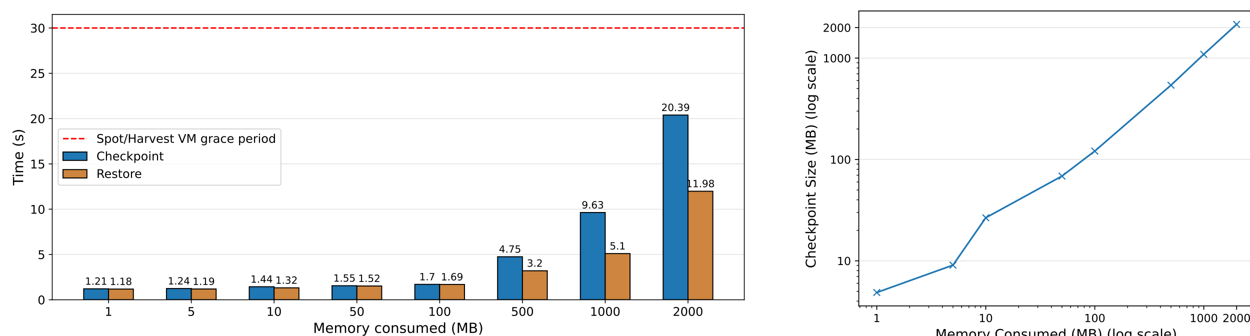
For these experiments, we ran our implementation based on a modified Apache OpenWhisk deployment. This deployment consists of a Kubernetes cluster and function instances are deployed in lightweight Docker containers with a custom OpenWhisk runtime containing the bare minimum required dependencies (CRIU binaries) to run each function. We use one controller and one invoker instance. We configure functions to use the maximum 2048 MB of memory for OpenWhisk containers. Unless otherwise noted, the cluster runs inside an EC2 t3a.large machine in the us-east-1 availability zone. The EC2 machine has 2 virtual CPUs, 4 GiB of RAM and a bandwidth limit of 5 Gbit/s, running Amazon Linux 2 kernel 5.10. Regarding the network, our EC2 machine connects to S3 through an internal VPC gateway endpoint.

Each experiment performed comprises an original execution of a Serverless function, followed by a simulation of a Serverless container eviction in a pre-defined instant, which will trigger the checkpoint operation and, subsequently, restore the function execution in a new container. Every function's input, output, and checkpoint objects were stored in their own S3 bucket. For compression, we use LZ4 [15], a popular compression algorithm that achieves a ≈ 3× compression factor for a typical Python application. Unless otherwise stated, each one of the individual experiments we describe next was executed 50 times.

4.1 R-Check Overheads

In order to assess how R-Check performs, we need to look at the limiting factors for our checkpointing tool. CRIU saves memory pages directly from the processes, so the most prevalent limiting factor is the number of memory pages allocated to a given process. Thus, we study the impact of memory consumed during a function execution on the overhead introduced by R-Check. For this experiment, we wrote a microbenchmark in Python using `bytearray(size_of_array)` that simply allocates the amount of memory indicated by `size_of_array` and sleeps for a fixed 30 seconds. At 15 seconds, the function is interrupted, and the checkpointing/restore process is triggered.

Figure 3a presents the median checkpoint and restore times for allocated byte array sizes ranging from 1 MB to a maximum 2000 MB. Checkpointing overhead accounts for less than 2 seconds for up to 100 MB and grows linearly as the amount of allocated memory reaches 2000 MB. Even when exhausting all container memory, checkpoint times are kept within safe boundaries (≈ 20.4 seconds), namely within the time allowed by the grace periods granted in the current cloud offer (typically 30 seconds). We can also depict that restore operations take less time to complete



(a) Checkpoint/restore times for various sizes of memory. Each bar represents the median (P50) over 50 executions.

(b) Checkpoint sizes for various sizes of memory. Each point represents the median (P50) over 50 executions.

Figure 3. Evaluation of checkpointing performance for a microbenchmark that allocates various memory sizes.

than checkpoint operations for all values measured. This behaviour is explained by the fact that, from the various steps of our checkpoint/restore process, the network is, by far, the stage with the lowest throughput, dominating all executions. Moreover, we observed that download rates were higher than the upload rates, contributing to the higher overhead of checkpointing compared to the restore procedure.

Regarding checkpoint sizes, Figure 3b shows not only a near-linear growth of checkpoint sizes with the increase in memory allocated, but also similar memory sizes from the application and the checkpoint object. The extra size of the checkpoint images comes from extra space on the heap, stack, the application code itself and other imported libraries.

4.2 Real Applications

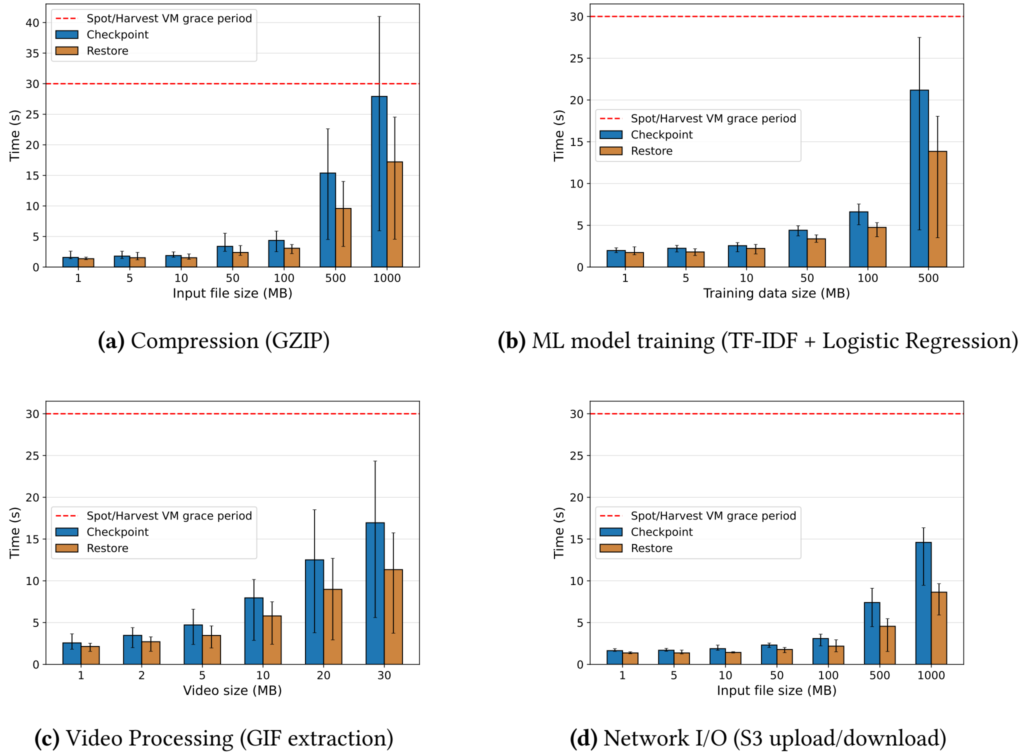
In this section, we present an evaluation of using our system for multiple real applications. We tested R-Check on diverse workloads, focusing on functions that are either long-running or have a non-deterministic/stateful nature. For these experiments, we used several Python benchmark applications from SeBS [7] and FunctionBench [13]. The functions used are described next. **(1) Compression (FunctionBench)**, a benchmark that takes a file as input and compresses it through GZIP. The primary objective of this application is to represent realistic disk I/O-heavy operations; **(2) Machine learning model training (FunctionBench)**, that uses Python's `scikit-learn` package and reads training sets from Amazon Fine Food Reviews [2], transforming each review into a TF-IDF vector. The outcome of this process is applied to a Logistic Regression classifier to build a model that predicts reviews' sentiment scores; **(3) Video Processing (SeBS)**, an application receives a video file and converts it into 1-second GIF files with 10 fps using `FFmpeg`, a software for handling video, audio, and other multimedia files; **(4) Network I/O (SeBS)**, a network I/O-intensive benchmark that receives an input and output S3 bucket names and an object key, downloads that object and uploads it right after to another S3 bucket. The idea behind this benchmark is to assess the viability of R-Check to checkpoint and restore applications that contain network connections.

We calculate beforehand the average duration of each function for each input object size tested. This average duration serves as an upper bound for a randomly selected instant within the interval $[0, function_duration]$ for which the function fails and a checkpoint is taken. Figure 4 shows the 5th, 50th and 95th percentile checkpoint/restore latency for each one of those applications and multiple input object sizes. Median checkpoint times range from 1.2 seconds up to around 27.9 seconds, when exhausting the memory limits of the containers, all of them still under a typical 30-second eviction grace period. On the other hand, median restore times go from 1 second to 17.2 seconds. We also notice that some results vary slightly from the ones inferred from the experiment in Section 4.1: for example, Figure 4a shows that, in the 95th percentile, the compression benchmark takes 42 seconds to checkpoint. This is likely due to lower compression ratios for such functions and, given that our main bottleneck is network latency, checkpoints could take more to upload/download.

Another observation is that, as the input sizes grow, the disparity between the 5th and 95th percentiles also grows accordingly. This is because we use random timeouts to decide when to checkpoint, which leads to some executions being terminated at an early stage when little memory has been allocated and others being evicted at later points when the applications are using much more memory.

4.3 R-Check vs. Application-specific Checkpointing

Next, we will answer whether resorting to application-specific code to produce checkpoints leads to better results than our proposed transparent, application-agnostic approach.



(a) Compression (GZIP)

(b) ML model training (TF-IDF + Logistic Regression)

(c) Video Processing (GIF extraction)

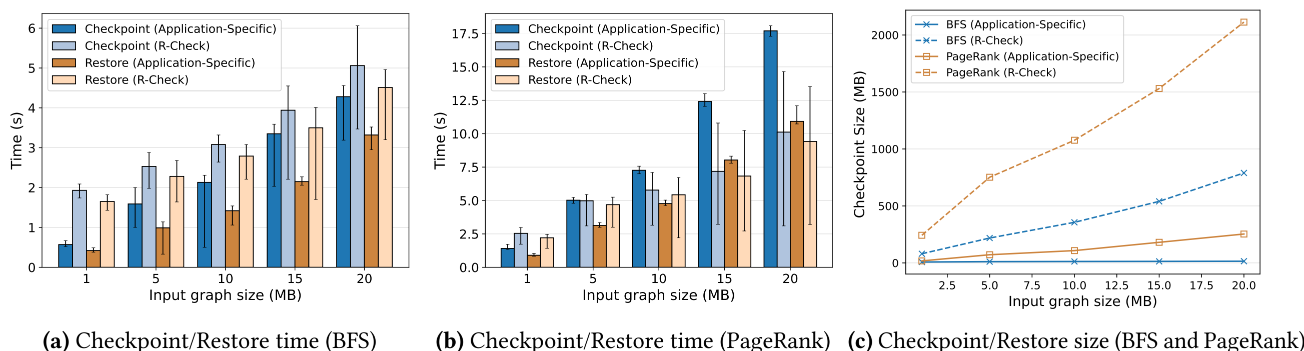
(d) Network I/O (S3 upload/download)

Figure 4. Evaluation of R-Check on real applications from SeBS [7] and FunctionBench [13] benchmark suites. Each bar represents the median (P50), 5th percentile (P5) and 95th percentile (P95) of 50 executions.

To conduct this part of the evaluation, we focused on benchmarks for which taking checkpoints in an application-specific manner would be realistic. Functions that spend most of their compute time inside third-party library calls or communicating over the network could possibly miss the checkpointing window. In light of these circumstances, we chose two Python benchmarks that present the most suitable characteristics for application-level benchmarking: (1) a **Breadth-First Search (BFS)** algorithm which takes a graph as input and traverses it and (2) an instance of **PageRank**, which performs link analysis in a given graph. We wrote checkpoint/restore code specific to that function's logic for each benchmark. In each case, only relevant variables to restore the application state are saved and then serialized to a checkpoint file using `pickle`[2], a widely-used module used to serialize and de-serialize objects in Python. Those objects are then compressed with the same lZ4 algorithm as used for R-Check, and uploaded to an S3 bucket. When restoring, the checkpoint object is retrieved and loaded into the function and execution resumes. Once again, we calculate the average duration of a fault-free execution and then randomly choose an instant in which the function will be checkpointed. These hand-developed "checkpoint/restore handlers" do not cover all the application code but only the sections where checkpointing is feasible. Thus, the application might not be snapshotted precisely at the selected instant but at the next checkpoint section after the time has elapsed.

The results obtained are depicted in Figure 5. We see that, for small input sizes (1 MB and 5 MB in the BFS function and 1 MB in the PageRank function), the application-specific approach performs much better than R-Check does, both for checkpoint and restore. However, as input sizes increase, the application-specific algorithms start performing worse and the metrics grow faster than for R-Check. In fact, for the PageRank function, R-Check outperforms the application-specific approach from 10 MB of input size and provides slightly worse performance from 15 MB of input size in the BFS function. These results are largely unexpected since we are saving the entire process address space in R-Check and only storing essential variables in the application-specific approach. To understand these numbers, let us consider, for example, the 15 MB BFS experiment. According to Figure 5c, using the application-specific approach, this benchmark corresponds, at the median, to a checkpoint time of 3.35 seconds and checkpoint object size of 49.45 MB (16.48 MB in storage, given LZ4's 3× compression ratio). At a ≈ 50 MB/s upload rate to S3, this translates into only ≈ 10% of the total time uploading the checkpoint object to storage and ≈ 90% of the

[2]From the modules available for Python, `pickle` provides the best compromise between performance and data structures that it can serialize.



(a) Checkpoint/Restore time (BFS) **(b)** Checkpoint/Restore time (PageRank) **(c)** Checkpoint/Restore size (BFS and PageRank)

Figure 5. Comparison of R-Check's checkpoint/restore times and sizes against an application-specific approach that saves only relevant variables. In Figure 5a and Figure 5b, each bar represents the median (P50), 5th percentile (P5) and 95th percentile (P95) of 100 executions. In Figure 5c, each line represents the median (P50) of 50 executions.

total time to serialize the variables with `pickle`, with negligible LZ4 compression time. This led us to conclude that serialization costs are considerably higher in this method, dominating the checkpointing process duration. These values remain fairly consistent across all experiments and contrast with R-Check, where the upload process is the limiting factor. On the other hand, as shown in Figure 5c, checkpoint sizes in R-Check grow more rapidly than for the application-specific method, as expected. We argue that this does not represent a high cost for the cloud provider because checkpoint objects are kept for a short time until the execution is restored and can be deleted immediately.

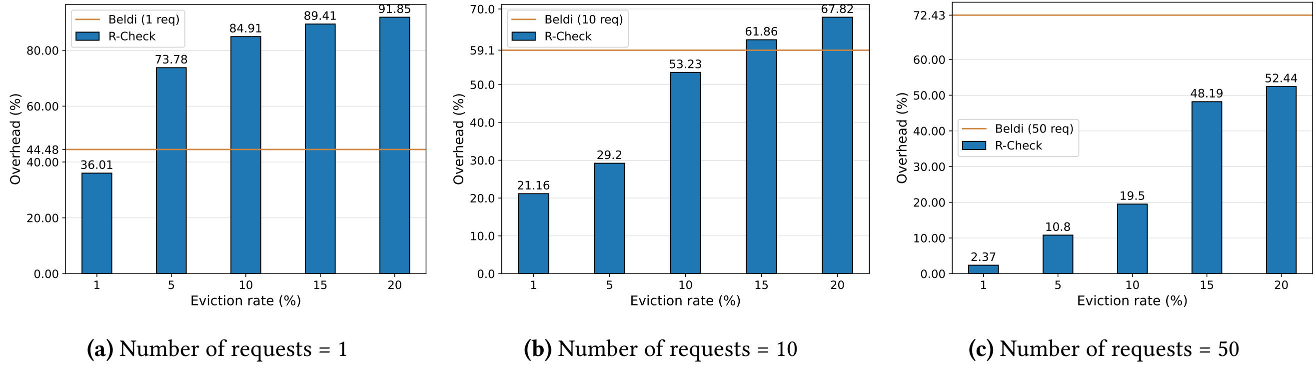
Through this analysis, we can conclude that we risk obtaining worse performance than anticipated if we employed an application-specific method. This factor, coupled with the challenges faced when writing custom checkpointing code, leads us to argue that employing a system-level approach through CRIU is a sensible approach.

4.4 Comparison with the State-of-the-Art

We compared R-Check with the state-of-the-art in Serverless fault tolerance, namely solutions that employ preventive approaches: Beldi [18] and Kappa [19]. Our key premise is that, by checkpoint only when strictly necessary, our reactive approach introduces a much smaller overhead than preventive approaches, that incur an extra cost for all executions. To test this premise, for each one of the approaches mentioned above, we run a set of 1000 function invocations and calculate the relative overhead in the total execution time. Then, we calculate the relative overhead for R-Check with common eviction rates in harvested instances (1%, 5%, 10%, 15%, 20%), for which R-Check checkpoints only when evictions happen.

In **Beldi**, the fault tolerance mechanism is activated once an external read/write is performed through Beldi's API (for example, an update in a DynamoDB table). As Beldi is written in Go, we developed a Golang I/O-intensive benchmark that writes and reads entries from a DynamoDB table on a 50%-get-50%-put workload. In R-Check, we deployed the benchmark unmodified and performed checkpoints on a random instant during the execution. For Beldi, we used its library API for external operations to perform the write/read calls. We measured the total overhead introduced by such calls and compared it with the overhead observed using R-Check for multiple eviction rates. Figure 6 shows the overhead introduced by both approaches relative to the whole execution time. Beldi performs better than R-Check for functions that contain small number of requests, in this case, 1 request, , where R-Check's overhead is lower only at a 1% eviction rate. However, However, Beldi's performance degrades as the number of requests increases. At 10 requests, R-Check achieves lower overheads for up to a 15% eviction rate and, at 50 requests, it achieves lower overheads for all eviction rates tested. Furthermore, Beldi guarantees idempotency by preventing duplicate external calls that cause side effects but it replays all the computation inside a function on restore, a behaviour we want to avoid. Plus, the fact that it can only secure applications with external stateful calls makes it a relatively limited approach compared to R-Check that can checkpoint and restore a more vast range of functions.

For **Kappa**, checkpointing occurs by calling Kappa's library. In this evaluation, we faced some of the same challenges as we did in Section 4.3, because, in order to successfully checkpoint executions, Kappa's functions need to be structured in a way that it is possible to signal where checkpoints should be taken, otherwise checkpoints cannot be performed. In light of this restriction, we again test a Python BFS benchmark with graph input sizes ranging from 1 MB to 30 MB. To evaluate both approaches in a realistic scenario, we used a public 24-hour Azure Functions trace [6]. From the trace, we grouped functions based on their execution duration to come up with an approximate distribution that



(a) Number of requests = 1 **(b)** Number of requests = 10 **(c)** Number of requests = 50

Figure 6. Comparison of relative R-Check's overhead against Beldi for different evictions rates on a set of 1000 DynamoDB put/get function executions. We test functions that include different numbers of requests: 1, 10 and 50.

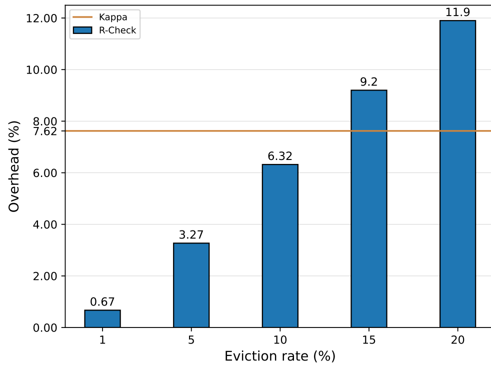


Figure 7. Comparison of relative R-Check's overhead against Kappa for different evictions rates on a set of 1000 BFS function executions.

we could use to model our experiment. Then, we ran a set of 1000 of our functions that would match that distribution.

To set up Kappa, we injected `checkpoint()` calls and configured checkpoints to be stored in S3 and the checkpoint period to 25 seconds since this is the optimal interval to produce a checkpoint in the given a 30-second termination grace period. For R-Check, we measured the checkpointing time for each input size and calculated how many executions from that set would need to be checkpointed based on the eviction rate being tested. Figure 7 shows the percentage of overhead in the total execution time for the experiment presented above. R-Check outperforms Kappa for eviction rates under 15%, representing the great majority of evictable VMs on the market. When the eviction rate is around 1%, Kappa's overhead is $\approx 11.3\times$ higher than R-Check's and, when at 5%, it is still $\approx 2.3\times$ higher. Hence, we can be confident that, for most evictable resources, R-Check performs better than Kappa while not adding extra user programming burden.

4.5 Concurrent checkpoints

With limited Openwhisk invoker nodes in our deployment (and in an actual Serverless deployment alike), it is safe to assume that multiple functions will be running concurrently in containers on top of the same harvested VM. In this scenario, if an eviction occurs, several functions must be checkpointed at once. Therefore, we evaluate the scalability of checkpointing by measuring the latency of concurrent functions being checkpointed simultaneously. To be able to handle more requests concurrently, we scale up our infrastructure to an EC2 c5a.8xlarge VM with 64 GiB of Memory, 32 vCPUs and a 10 Gbit/s network bandwidth. We run a workload composed of a previously described application in Section 4.2 (Compression with GZIP, input size = 50 MB), simultaneously checkpoint every execution when 15 seconds have elapsed from the launching of the first function and measure the end-to-end duration of that process. Figure 8 shows that the median checkpoint latency remains slightly over 2 seconds and median restore latency under 1.3 seconds even for 16 concurrent functions. Although R-Check is network-bound for a single connection, it can scale horizontally and take advantage of its larger bandwidth to checkpoint more functions simultaneously (on multiple S3 connections). Besides, we believe that, even though larger sets of concurrent functions were not tested, R-Check would scale well. This is because we would also need to scale up the underlying infrastructure to deploy more functions simultaneously, increasing both CPU power and network bandwidth.

5 Conclusion

Serverless computing is becoming a viable cloud paradigm to an increasing number of use cases. However, since functions can fail through the execution, a re-execution might unnecessarily duplicate resources or cause an unintended behaviour due to visible side-effects. This claim is even more glaring with the recent trend of deploying Serverless environments on top of harvested resources that may be terminated at

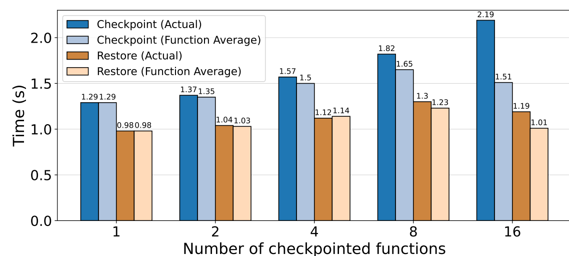


Figure 8. Evaluation of checkpoint/restore overhead when scaling concurrent checkpoints. "Actual" values represent the median latency (P50) over 50 measurements and "Function Average" represent the average overhead of a function for each set of checkpointed functions.

any given moment. We envision that the future of serverless computing should include a fault tolerant mechanism to deal with controlled system failures from underlying harvested resources and function timeouts. In this thesis, we designed and implemented R-Check, a framework that simplifies the development of applications in a Serverless context with a reactive checkpointing scheme for fault tolerance. We evaluated R-Check on various real applications and compared it to state-of-the-art systems, namely Kappa [19]. This study shows that, while employing a reactive approach, a diminished set of invocations will be affected by checkpointing, introducing no runtime overhead to most executions.

While the results from this work are promising, there are a few limitations and plenty of ideas for future work. For instance, as mentioned, R-Check can only checkpoint functions in the case of controlled evictions or function timeouts. We highlight that this is already the case with the existing offer of IaaS and similar cloud services. However, other techniques are required for handling unpredictable failures in critical workloads. An avenue for future work could be studying other complementary techniques for a system that strives to have high availability in the presence of unexpected crashes.

Also, with our current approach, when a VM is evicted, all functions are required to be checkpointed. However, many of those functions could finish within the termination grace period, so checkpointing those functions is a waste of resources for the provider and an unnecessary increase in wait time for the customers. Following a similar approach to the one presented in [20], it would be possible to characterize function invocations and predict which workloads would likely finish within the grace period and do not need to be checkpointed.

Finally, as seen in Section 4, uploading and downloading checkpoint objects is currently the major bottleneck of R-Check. However, observed throughputs are much lower than the actual per-instance advertised bandwidth. One possible improvement is to leverage checkpoint image sharding to maximize horizontal upload/download throughput.

References

[1] Amazon. 2022. AWS Lambda. https://aws.amazon.com/lambda/

[2] Amazon Fine Food Reviews 2022. Amazon Fine Food Reviews. https://snap.stanford.edu/data/web-FineFoods.html

[3] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, et al. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 735–751. https://www.usenix.org/conference/osdi20/presentation/ambati

[4] Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel Distributed Processing*. 1–12. https://doi.org/10.1109/IPDPS.2009.5161063

[5] AWS Spot 2022. AWS Spot. https://aws.amazon.com/ec2/spot/

[6] Azure Traces 2022. Microsoft Azure Traces. https://github.com/Azure/AzurePublicDataset

[7] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, et al. 2021. SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing. arXiv:cs.DC/2012.14132

[8] CRIU 2022. Checkpoint/Restore in Userspace. https://criu.org/

[9] Jason Duell. 2005. The design and implementation of berkeley lab's linux checkpoint/restart. (2005).

[10] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, et al. 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

[11] Vatche Isahagian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving Deep Learning Models in a Serverless Platform. 257–262. https://doi.org/10.1109/IC2E.2018.00052

[12] 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 (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 691–707. https://doi.org/10.1145/3477132.3483541

[13] Jeongchul Kim and Kyungyong Lee. 2019. FunctionBench: A Suite of Workloads for Serverless Cloud Function Service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 502–504. https://doi.org/10.1109/CLOUD.2019.00091

[14] Ana Klimovic, Yawen Wang, Patrick Stuedi, et al. 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

[15] LZ4 2022. LZ4 compression. https://github.com/lz4/lz4

[16] Microsoft Azure 2022. Azure functions. https://microsoft.com/en-us/services/functions/

[17] OpenWhisk 2022. OpenWhisk. https://openwhisk.apache.org/

[18] Haoran Zhang, Adney Cardoza, Peter Baile Chen, et al. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[19] Wen Zhang, Vivian Fang, Aurojit Panda, et al. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277

[20] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, et al. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739. https://doi.org/10.1145/3477132.3483580