TÉCNICO LISBOA

R-Check: A Reactive Checkpointing Approach for Serverless Computing

Rafael Coelho Alexandre

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Rodrigo Seromenho Miragaia Rodrigues
Prof. João Pedro Faria Mendonça Barreto

Examination Committee

Chairperson: Prof. José Alberto Rodrigues Pereira Sardinha
Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues
Member of the Committee: Prof. Luís Manuel Antunes Veiga

November 2022

Acknowledgments

First and foremost, I would like to thank my thesis Supervisors, Prof. Rodrigo Rodrigues and Prof. João Barreto, for sharing their expertise, commitment and flexibility that helped make this thesis possible. I am also thankful to Prof. Rodrigo Bruno and Amândio Faustino for their insightful suggestions and support throughout this whole thesis. I thank all of you for guiding me along the right path and encouraging me to deliver the best work I could.

This work was supported by Fundação para a Ciência e a Tecnologia, under the research grant PTDC/CCIINF/6762/2020 included in the "MS3: New foundations for micro-services and serverless systems" project. I express my gratitude to this project as well as to INESC-ID for supporting my research, and allowing me to perform the empirical evaluations needed for this work.

I am indebted to Instituto Superior Técnico for contributing to my personal and professional growth. Not only for providing me with the technical skill set required to tackle this project but also with the critical thinking to do it in the most competent manner possible. Besides, I have to thank all my friends and colleagues, especially Francisco Cecílio, Afonso, Pedro and Francisco Serralheiro, for their constant support and for allowing me to walk alongside them, during this hard but also rewarding 5-year journey.

I am deeply grateful to my family for all the sacrifices they have made for me and for providing me with emotional and financial support, keeping me motivated and optimistic. To my mom, dad and brother, a very special thank you.

Lastly, a sincere thank you to my dear girlfriend, Mariana, for all her love, friendship and, especially, patience. You were the best thing that happened to me during this period. Thank you for sharing this ride with me and being there when I needed it the most.

Rafael Coelho Alexandre

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.

Resumo

A computação em *Serverless* permite aos programadores não terem de lidar com a gestão de servidores ao fornecer uma abstração sobre o provisionamento desses recursos. No entanto, desenvolver aplicações para Serverless é uma tarefa complexa já que estas plataformas apenas garantem semântica *pelo menos uma vez*. Assim, os programadores ficam responsáveis por implementar código idempotente, isto é, código que possa ser reiniciado sem efeitos indesejados. Técnicas de *logging* e *checkpointing* periódico foram propostas para mitigar estas restrições mas as mesmas impõem *overheads* notórios na performance do sistema. Além disso, estes problemas são acentuados por estratégias de otmização dos recursos utilizadas pelos *cloud providers*, nomeadamente, tirando partido de recursos do tipo *harvested* – nós de computação a preços mais baixos que podem ser terminados precocemente se o *cloud provider* necessitar deles para clientes prioritários. Nesta tese, sugerimos uma abordagem diferente para lidar com interrupções de funções em *Serverless*, através de um mecanismo de *checkpointing* reativo. A nossa visão principal é que interrupções de recursos *harvested* são eventos controlados e, por isso, devem ser tratados através da migração das execuções de forma estruturada e reativa quando estas falhas acontecem. Neste trabalho, debatemos as vantagens e limitações da nossa ideia, discutimos alternativas de design e apresentamos o design do R-Check, um sistema que segue esta abordagem. A nossa solução foi avaliada num ambiente *OpenWhisk* sob vários *benchmarks*, de forma a mostrar que o R-Check pode ser uma solução eficaz e com baixo custo de performance para correr funções Serverless em recursos *harvested*.

Palavras Chave

Computação Serverless, Tolerância a Faltas, *Checkpoint/Restore*, Reativo.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Acronyms

AES	Advanced Encryption Standard
BFS	Breadth-First Search
CBC	Cipher Block Chaining
EC2	Elastic Compute Cloud
FaaS	Function-as-a-Service
HPC	High-Performance Computing
IaaS	Infrastructure-as-a-Service
PaaS	Platform-as-a-Service
PID	Process Identifier
S3	Simple Storage Service
SaaS	Software-as-a-Service
SLA	Service Level Agreement
TTY	Teletypewriter
VM	Virtual Machine
VPC	Virtual Private Cloud

1

Introduction

Contents

Cloud computing came into existence in the form of IaaS (e.g., EC2 from AWS) and then, PaaS (e.g., Google App Engine) as businesses started shifting from on-premise infrastructures to third-party low-level abstractions that provide the same functionality while being more cost-effective. More recently, moving up the abstraction ladder, Serverless computing surfaces as an increasingly compelling cloud programming paradigm, especially in the form of FaaS. Services of this kind are now provided by all the major cloud providers (Amazon's AWS Lambda [2], Microsoft's Azure Functions [3], Google's [4] and IBM's Cloud Functions [5]) as well as popular open-source platforms (Apache OpenWhisk [6] and OpenFaas [7]).

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 Virtual Machine (VM), allowing tenants to launch many compute nodes quickly without provisioning a long-running cluster. (For example, AWS Lambda functions launch at least 30$\times$ faster than Amazon Elastic Compute Cloud (EC2) c4.4x large VMs with a similar compute capacity [8]). The cloud provider is responsible for handling all the underlying infrastructure burden – allocating VMs or containers, deploying the user code, scaling the resources up and down – and doing so transparently to users. It ultimately allows backend infrastructure maintenance to become increasingly decoupled from application development.

Serverless was originally designed to execute short-running and stateless functions [9]. Even today, providers firmly guide developers to write idempotent code [10]. 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, possibly at a different container or VM, whenever the execution is interrupted by the cloud provider.

A recent trend in cloud computing is for cloud providers to optimize the use of their infrastructure by leveraging spare available resources. This trend can be complementary to Serverless computing: a recent study [11] shed light on the possibility of 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. Another work [12] proposes the usage of idle spots inside High-Performance Computing (HPC) clusters that are too small for any HPC job but a perfect fit for a Serverless computation. Using idle assets for Serverless provides a fine-grained, optimized management of the infrastructure for the provider side and the possibility to run functions at even lower rates for the customer side.

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, as functions with side-effects are ubiquitous. In particular, non-deterministic or stateful functions may diverge when replayed, producing undesired behaviours (for example, a credit card payment, if a failure happens, may be executed twice). For this kind of functions, guaranteeing *exactly-once* semantics in the presence of faults is challenging. Another use case 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 fact, AWS Lambda recently had to increase its execution time limit [13]. In these scenarios, the chances that a function is terminated (and replayed) increase and the cost of running a function all over again can be expensive, duplicating resources and the work done up to the failure point. A common workaround to this feature is manually partitioning the application logic into several sub-functions and stitching them together, which adds to the programmer's burden.

In summary, the current replay-based fault tolerance mechanism from cloud offerings and, at the same time, the challenge of dealing with arbitrary faults are preventing the generalization of Serverless workloads and making cloud providers recommend very strict semantics for their functions. Thus, recent work proposes solutions that may address this problem by recording the computation through logging or periodic checkpointing. Logging approaches, namely Beldi [14] and Boki [15], implement new robust data structures that save application state when external read/write operations exist. On the other hand, Kappa [8] uses periodic checkpointing by inserting custom code into functions to snapshot the application state. However, both types of approaches introduce overheads present in every execution of a cloud function to proactively protect against occurrences that affect only a small subset of executions. Furthermore, both methods limit the types of workloads for which they can achieve correct semantics – for logging, external read/write operations need to be executed during the function, while for checkpointing, there have to exist execution points where checkpoint code may be inserted.

1.1 Objectives

This thesis aims to deploy Serverless functions that may be terminated before the end of their execution and ensure that operations are executed *exactly-once*. Plus, accomplish this goal while avoiding any runtime overheads in the vast majority of the executions, leading to a convergence between the fault-tolerance and performance characteristics of the environment of different cloud paradigms, namely IaaS and FaaS.

Towards that goal, in this thesis, we advocate that Serverless, and FaaS in particular, should support

a reactive checkpointing model instead of forcing existing code to be idempotent or to apply complex logging mechanisms. 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. This way, R-Check's functions can be non-deterministic, have arbitrary side effects, and still be interrupted at any controlled point during their execution.

Overall, this new paradigm brings the advantages of (1) eliminating the need for Serverless functions to be idempotent, (2) bypassing rigid execution duration limits imposed by cloud providers, (3) avoiding unnecessary runtime overheads due to logging or periodic checkpointing, while (4) requiring no changes to existing cloud function code.

1.2 Contributions

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 it in Apache OpenWhisk [6], a popular open-source Serverless platform;

3. We evaluated its effectiveness with various 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. Compared to Beldi and Kappa, R-Check achieves lower overall runtime overhead while being applicable to a much greater variety of workloads.

1.3 Thesis Outline

The rest of the document is organized as follows. In Chapter 2, we introduce some background concepts used throughout the rest of the document. It includes the basics of how Serverless computing works and how it handles failures by default. We proceed to Chapter 3 with an overview of existing work related to fault tolerance in Serverless and other relevant research topics in this domain. In Chapter 4, we present

the architecture of R-Check and explain, in detail, its design choices and how it was implemented. Chapter 5 is an experimental evaluation, which presents the methodology used, details the applications and scenarios tested and analyzes experimental results. Finally, in Chapter 6, we present a brief conclusion summarizing the contributions of this thesis and highlight limitations and possible directions for future work.

2

Background

Contents

2.1 Overview

In this chapter, we introduce some key notions required to understand the rest of this document. In Section 2.2, we look at the main differences between serverful computing, particularly the current most popular cloud paradigm, IaaS, and Serverless computing, viewed by many as the next default computing paradigm of the Cloud Era, largely replacing serverful computing. After that, we briefly describe the overall logic and architecture behind the Serverless paradigm with its traditional workloads (Section 2.3) and default fault tolerance approaches (Section 2.4). Then, we explore checkpointing as a fault tolerance approach and compare different types of checkpointing schemes (Section 2.5). Finally, in Section 2.6, we characterize harvested resources, their nuances and challenges that arise in the context of Serverless computing.

2.2 From Serverful to Serverless computing

The hi-tech area has significantly changed regarding underlying computing platforms in recent years. One such prominent technological shift has been the upgrade from on-premise infrastructure to cloud environments. Cloud has become pivotal for most organizations' technology strategies: cloud setup is more straightforward, more scalable and brings much fewer maintenance costs than on-premise clusters.

The initial cloud computing offer from vendors consisted of very low-level abstractions in the form of Infrastructure-as-a-Service (IaaS). In IaaS, the service provides the infrastructure, like hardware-level virtualization and storage, and customers are responsible for managing the Operating System and any data, middleware, runtimes and applications. Platform-as-a-Service (PaaS) is a step further in abstraction, where the cloud provider hosts the hardware and software while the client only deals with the application deployment using the platforms provided. Contrary to IaaS customers, PaaS customers enjoy transparent scaling and high availability guarantees of their systems. These services were combined with very high-level abstractions in the form of SaaS. In this solution, the provider offers the entire application stack, and the customers only serve as users of such applications.

More recently, Serverless has emerged as a promising cloud paradigm, especially in the form of Function-as-a-Service (FaaS), standing in the middle of the paradigms presented before in terms of abstraction level. To better understand this paradigm, Table 2.1 compares different infrastructure options to deploy services and applications.

Unlike the traditional cloud interface (IaaS), in FaaS, users are relieved from dealing with the infrastructure itself, meaning they do not explicitly provision or configure the resources, such as Virtual Machines (VMs) or containers, required to run their computation. Instead, this responsibility is handed over to the provider, who holds more control over optimizing such resources, allowing for a better uti-

	On-premise server	Cloud VM (IaaS)	Serverless (FaaS)
Time to provision	Weeks / Months	Minutes	Milliseconds
Utilization	Low	High	Highest
Billing granularity	Capital Expenditures[1]	Minutes	Milliseconds
Management	Hardware, Networking, OS, Runtime, Functions	OS, Runtime, Functions	Functions

Table 2.1: Comparison between different infrastructure options.

lization of their infrastructure. As a result, customers only have to focus on the application logic and, at the same time, can benefit from a great elasticity in allowing the service to rapidly scale up and down automatically to handle request surges [16].

When it comes to a pure price-by-price comparison, if we look at the per-minute cost of running an AWS Lambda cloud function with the cost of an AWS t3.nano VM with the equivalent 0.5 GB memory, it might seem like Serverless is 7.5× as expensive. However, such a naive comparison is misleading. FaaS lowers the cost of deploying cloud code by charging only for the resources effectively used during function execution, further approximating to an actual *pay-as-you-go* billing model – the original promise of cloud computing. This model is different from IaaS in the sense that, in the latter, customers pay for long-term reserved VMs, even if they are not utilized up to their total capacity. In practice, cloud providers claim that tenants can observe cost savings between 4× and 10× when transitioning to Serverless [17].

2.3 Serverless Architecture

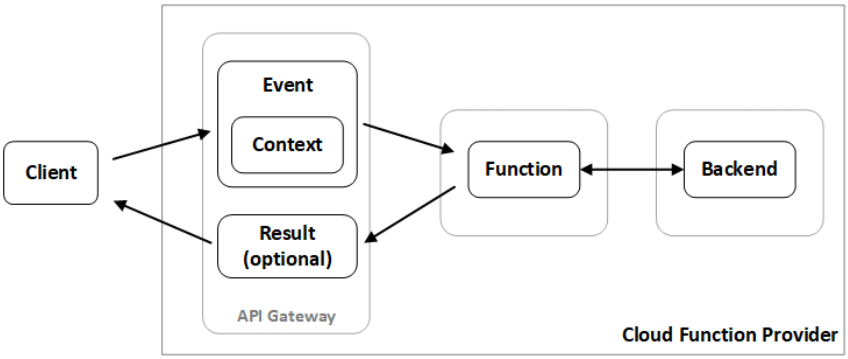


Figure 2.1: Network Diagram

Figure 2.1 depicts the overview of a generic Serverless architecture. First, tenants register their

[1] The funds available to acquire, upgrade, and maintain physical assets.

functions (also known as *event handlers*), like the one presented in Listing 2.1, in the cloud function provider and configure how they want them to be invoked, such as through API calls, timers going off or other specific triggers. The cloud provider stores the function code in an object storage service until an invocation is requested. Then, when such a request arrives, the platform assigns the function to one of its computation nodes. The node, which is a short-lived, stateless execution environment (e.g., a container, micro-container or any other lightweight sandbox execution environment), executes the function. Once it finishes, the result is sent back to the client, and an execution log is sent to the provider. These logs can then be used to detect bugs and execution failures, improving further executions of the function.

A Serverless function is served by one or multiple sandbox execution environments. When a request arrives, the provider checks whether a container is already running to serve the invocation. The function can be executed immediately when an idle container is already available (a "warm" start). If a container is not readily available, the provider will spin up a new one (a "cold" start). When a function suffers from a "cold" start, the request will take additional time to complete because there is a latency associated with downloading the function code and booting up a new container. To mitigate the problem of cold starts, providers keep recently used containers active for a set keep-alive time in anticipation of subsequent invocation being triggered. This keep-alive period is a trade-off between memory usage and performance, and it is highly dependent on the function invocation frequency.

```
import json

 def event_handler(event, context):
    return {
        "statusCode": 200,
        "body": json.dumps({
                "message": "hello world"
        })
}
```

Listing 2.1: An example Serverless function.

Also note that, on current Serverless offerings, a single function can execute for a bounded duration before being forcibly terminated. This time limit is configurable and different for the multiple cloud providers (up to 15 min in 1 s increments on AWS, up to 9 min in 1 ms increments on Google Cloud, and up to 60 minutes in 1 s increments on Azure).

2.3.1 Common use cases and workloads

From an infrastructure perspective, Serverless and more traditional cloud paradigms may be used interchangeably or in combination. Choosing between these options is likely influenced by some other requirements from developers, such as the amount of control required over the operations, cost budgets and intrinsic application workload characteristics.

From a cost perspective, a Serverless architecture favors bursty and compute-intensive workloads. Bursty workloads prosper in these environments because function resources can scale up as the demand increases. Just as important, they can scale down to zero, so there is no cost to the consumer when the system is idle. Compute-intensive workloads are also appropriate since the price of a function invocation is proportional to the running time of the function. On the other hand, I/O bound functions might not be as cost-effective since the CPU is largely unused, thus the programmer is paying for computing resources they are not using to their full potential.

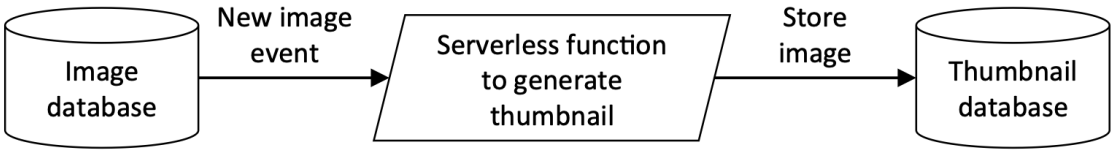


Figure 2.2: Example of Serverless use case: Image-processing.

In the programming model landscape, the nature of Serverless platforms makes it an excellent fit for applications that resemble those of functional programming [18]. This model includes applications that exhibit event-driven patterns, such as image-processing functions. Figure 2.2 shows an example of this type of workload. Each time a new image file is uploaded to a folder in an Amazon Simple Storage Service (S3) [19] bucket, an event is generated and forwarded to the event handler. It then extracts a thumbnail from the input image file, which is stored in another S3 folder.

2.4 Default Failure-Handling in Serverless

When a Serverless function fails during its execution,e.g., due to the need to free up the resources, the default behaviour adopted by cloud providers can be one of the following:

1. Ignore the error and do nothing.

2. Retry the whole function a configurable number of times (depending on the cloud host), after which an exception will be thrown and the execution aborted.

The first option represents the complete absence of fault tolerance. The second behaviour attempts to mask or alleviate the effects of faults. However, it presents a significant drawback: some functions may produce side-effects when replayed, for example, potentially incrementing a counter twice or making a database state inconsistent. In order to avoid this unintended behaviour, providers recommend that functions are built in an idempotent fashion to ensure that re-execution is safe [10]. In this regard, AWS recently launched Lambda Powertools [20], a suite of utilities to help developers make their code idempotent. While helpful, these recommendations and tools place the burden entirely on developers, and limit the class of application logic that may be deployed on Serverless platforms. We aim to expand that class of applications to more complex (possibly non-idempotent) use cases in this thesis, namely through the solution we present in Chapter 4.

2.5 Checkpointing

Checkpointing is a fault tolerance approach that records and persists the program state in stable storage and fetches it when there is a failure so that program execution is not forced to restart from the beginning. This approach is beneficial for applications running for a long time and those executed in failure-prone computing systems. This technique enables debugging and performance tuning: creating checkpoints simplifies reproducing a program error or analyzing a performance hotspot from a relevant code region. This section goes deeper into different settings that might be considered in a checkpointing approach and the corresponding comparison between those alternatives since this is the approach we take in our work.

2.5.1 Checkpointing timing

A crucial axis of checkpointing is the decision of when and how often to store the application state, for which two strategies may be employed.

A **Preventive** approach (see Figure 2.3) is when the system, periodically or in predefined moments (e.g., lines in the code specified by the user or dependent on some more specific heuristic), takes a snapshot of the execution in anticipation of a fault occurring. The latest snapshot is read and restored upon failure on a new node, ensuring function progress. Given its periodic nature, this strategy incurs recurring overheads in every execution. However, optimization mechanisms may be developed to reduce the overall overhead and memory required for the operation (e.g., incremental checkpoints in which only modified state since the last checkpoint is recorded).

A **Reactive** approach (see Figure 2.4) is defined by the system producing a checkpoint based on a specific event that would disrupt the program's intended behaviour (e.g., a running program suffering from a planned eviction of the underlying infrastructure). In this case, contrary to the previous ap-

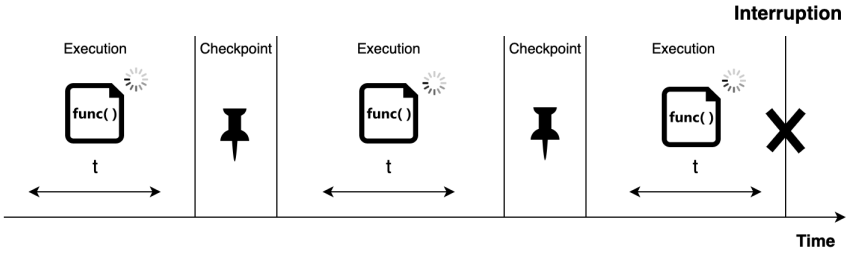


Figure 2.3: Preventive checkpoint approach.

proach, the checkpoint must record the entire relevant state to be resumed. On the positive side, since checkpoints are taken precisely before the program is about to be killed, no progress is lost, providing *exactly-once* semantics. On the downside, this approach only works for controlled failures, those that are given a termination grace period to operate with, whereas for crash faults, this approach has no time to record the application's state.

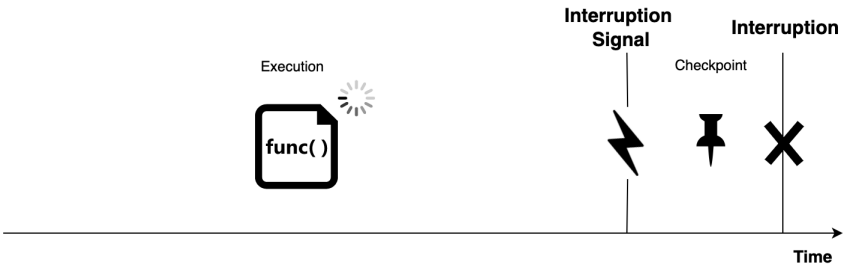


Figure 2.4: Reactive checkpoint approach.

2.5.2 Checkpointing scope

Checkpoints can be defined at different scopes, where the notion of *scope* entails who is responsible for performing the checkpoint and what information is stored by that process.

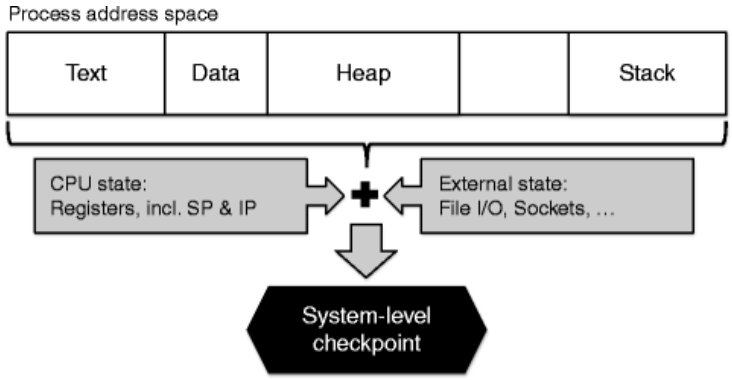


Figure 2.5: System-level checkpointing.

System-level checkpoints are a user-transparent way of implementing checkpointing. This strategy stands behind the rationale that leaving developers with the task of implementing failure-handling mechanisms for their applications is a cumbersome and error-prone task. In this approach, the user typically needs to specify only the checkpointing interval, with no additional programming effort; other details, such as checkpoint contents, are handled by the operating system [21]. System-level checkpointing treats applications as "black boxes" and involves saving the entire state of the application, including all processes and temporary data, at the checkpoint time, as depicted in Figure 2.5. Since this checkpointing method does not consider the application's internal characteristics and semantics, the total checkpoint size and overhead of checkpointing can dramatically increase with the size allocated to the application.

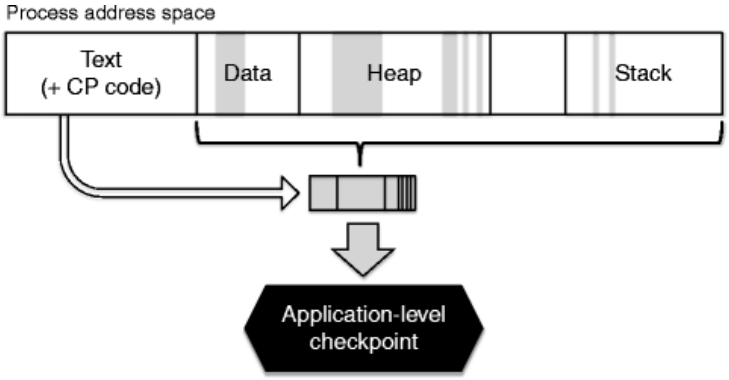


Figure 2.6: Application-level checkpointing.

The **application-level** approach sits on the other extreme. This approach relies on the application to provide its own fault-tolerant capabilities. Here, checkpoints are written by the application developers based on data structures and variables that they define as relevant to include in the checkpoint contents. The primary notion is that users can semantically understand the nuances of their applications and place checkpoints, allowing a significant optimization in the size of the checkpoint objects and checkpointing time. Figure 2.6 illustrates the overall idea behind this technique. This approach requires more programmer effort, but it is more portable since checkpoints can be saved in a machine-independent format and can be migrated between heterogeneous nodes. This flexibility is a relevant advantage compared to system-level checkpoints that are limited to being migrated between machines with the same hardware and software characteristics [21]. This approach's usage should reflect limited changes to the original application code, usually through the addition of library calls, as exemplified in Listing 2.2. Generally, three calls should exist: one to indicate a point of checkpoint, another to indicate a point of restoration, and a third to mention what data should be stored. The applications are generally scanned by a preprocessor that identifies the library calls and instruments the code accordingly. In sum, this strategy allows to reason about the essential data to store when a checkpoint is taken, which can prove to be a

substantial upside considering the checkpointing overhead and memory footprint caused.

```
import checkpoint_library

def event_handler(x,y):
    if x < y:
        # checkpoint call inserted by the developer
        checkpoint_library.checkpoint()
        z = x + 1
    else:
        z = y + 2
    return z * x
```

Listing 2.2: An example usage of a checkpointing Library.

2.6 Harvested Resources

In this section, we provide the overall idea behind harvested resources[2], the benefits that make them suitable for running Serverless functions, and possible challenges that arise from this usage.

Even with high customer demand, cloud resources are not always being used up to their full capacity. To minimize resource idleness, cloud providers started providing some form of harvested resources (e,g., Spot [22] or Harvest VMs [23]). These resources are temporarily unused, and one can take advantage of that spare compute capacity at a significantly discounted (and variable) rate. In fact, cloud providers state that these instances are up to 90% cheaper than on-demand instances, as exemplified in Table 2.2. This type of resource is also highly beneficial to providers since that compute capacity would, otherwise, remain idle, not generating any profit. The amount of available spare resources can vary based on various factors like region for VM deployment, size of VM and time of day. Whenever idle capacity is available, the provider will allocate more spot Virtual Machines that customers can use.

2.6.1 Evictions

The core difference between harvested resources and regular virtual machines is that regular VMs enjoy uninterrupted availability according to the Service Level Agreement (SLA) defined between customers and cloud providers. In contrast, harvested resources can get evicted at any time if resources become necessary to host another, higher-priority tenant/service. Even so, customers are given a short eviction

[2]Throughout this document, we use the terms “evictable VM” and “harvested VM” interchangeably to refer to these resources.

Instance Type	On-demand price / hour [1]	Spot price / hour [1]	Savings over On-demand
t2.micro	$0.0116	$0.0035	70%
m5d.xlarge	$0.226	$0.0399	82%
r5n.4xlarge	$1.192	$0.1672	86%
m5zn.12xlarge	$3.9641	$0.4789	89%
m6id.metal	$7.5936	$1.3409	82%
x1.32xlarge	$13.338	$4.0014	70%

[1] Prices were retrieved from https://aws.amazon.com/ec2/spot/pricing/ in September 2022.

Table 2.2: Comparison between on-demand and spot hourly prices of various examples of AWS EC2 instances. Spot instance prices are the average over a 30-day period and represent savings between 70% and 89% over on-demand instances.

notice (generally, a 30-second up to 1-minute grace period) to handle any sensitive ongoing computation before being fully terminated.

Next, we briefly describe these instances' two major eviction policies.

Capacity policy: the cloud provider will evict the VM when it is experiencing a surge in demand and needs that capacity back for regular *pay-as-you-go* workloads.

Price policy: in this type of eviction, the customer has the flexibility to set the maximum price they are willing to pay for the VM. If the spot price exceeds the defined price threshold, the VM evicts.

Because the resources can be terminated at any moment, harvested resources are traditionally a great solution for low-priority workloads that can tolerate function interruptions without unintended consequences like data analysis, batch jobs, background processing and optional tasks. In order to apply these resources to Serverless consistently, we need a mechanism capable of coping with those evictions.

3

Related Work

Contents

3.1 Overview

In this chapter, we describe some relevant work that has been developed to generalize Serverless computing (Section 3.2) and fault-tolerant systems to overcome the challenge of deploying complex Serverless applications, namely those that incorporate state/side-effects (Section 3.3). Furthermore, we present some transparent optimizations in Serverless infrastructures that can benefit our approach (Section 3.4). Finally, we discuss what we want to achieve with this thesis and compare it with existing solutions in this space (Section 3.5).

3.2 Serverless generalization

Serverless computing is becoming an increasingly compelling option for deploying applications in the cloud [16], given its ability to launch and scale fast, with the developer being only responsible for developing the application logic. Serverless functions were originally stateless and event-oriented. However, recent papers have exploited using the elasticity of Serverless for tasks beyond event handling, namely for video processing [24, 25], numerical computation [26–28], data analytics [29–33], machine learning [34–36] and parallel compilation [37]. With the progression of Serverless to include more general-purpose applications as the ones mentioned above, functions tend to naturally become more long-running, and it becomes increasingly more challenging to ensure their idempotency.

3.3 Fault tolerance in Serverless

Since a function's context can be terminated as part of dynamic resource management or load balancing, the state is not guaranteed to be preserved across function invocations. In current commercial and open-source FaaS platforms, the default behaviour is to retry the failed function. This strategy is acceptable for short-running stateless functions that can be replayed without much extra cost and provide the same result. Unfortunately this approach is not suitable for cases that are either (1) non-idempotent or (2) long-running. Consequently, a few recent research proposals attempted to address this major shortcoming.

Because out-of-the-box support for stateful executions is not provided in FaaS platforms, some systems have presented workarounds to ensure that state is preserved across re-executions. One general idea for building fault-tolerant frameworks is to avoid re-doing all the workload from scratch when functions fail and need to be placed in another node for execution. We now look at the two major schemes for fault tolerance in Serverless.

3.3.1 Log-based

Some systems adopt log-based mechanisms to work around the undesired effects of restarting the function from the beginning. In this class of systems, a failed component restarts from the beginning and skips any completed tasks by consulting a history log table kept in stable storage. This is the approach taken by Fouladi *et al.* in their framework gg [37] and in Azure's Durable Functions [38]. gg is an orchestration framework focused on building and executing burst-parallel applications on top of cloud functions. gg introduces an intermediate representation that abstracts the compute and storage platform and provides dependency management and straggler mitigation. Plus, gg relies on an external coordinator component and an external queue for submitting jobs (gg's *thunks*) to the execution engine. In case of coordinator failure, the job can be resumed from where it failed, as the coordinator program uses an on-disk cache to avoid re-doing the work that has already been finished.

Durable Functions are an extension to Azure's Functions that allow the execution of long-running functions that would otherwise not fit into the platform execution time limits. In this system, the programmer creates Serverless workflows composed of smaller functions managed by orchestrator functions. Durable Functions address failures and provide a model similar to gg that stores orchestration execution states as histories of events. The runtime can then replay the history, loading the state of the orchestration in memory [39].

These log-based approaches, despite being efficient and transparent to the end user, present two considerable limitations:

- First, the orchestration code must be replayable as well. Otherwise, execution may diverge if the orchestrator functions fail and need to be replayed;
- Second, history size can become an issue because history tables may grow significantly. This is a problem even if the program state has a bounded size (e.g., a long-running orchestrator function on Azure must be restarted manually once in a while to avoid memory exhaustion).

To avoid the problem of orchestration code, systems like Beldi [14] and Boki [15] express orchestrations through regular code but execute it in a way that logs progress and can resume intermediate states via replay. Both systems let developers write complex stateful applications (composed of multiple Serverless functions) that require fault tolerance. Beldi extends Olive [40], a log-based fault tolerance protocol, and builds transactional workflows using locks. Beldi keeps an operation log that ensures that functions are executed *at-most-once* and periodically restarts failed functions to ensure *at-least-once* semantics. Boki presents another log-based approach to storing function state in serverless workflows. The authors introduce the concept of "LogBooks", an abstraction allowing different functions from the same application to access state kept in a shared log object. These systems, however, increase an already high overhead of remote storage access (median request completion time increase 2.4–3.3×

in the case of Beldi and 1.8-3.0× in the case of Boki). They also try to minimize the logging size by garbage collecting the history kept in storage. Nevertheless, their garbage collection introduces a new source of overhead, adding to the already significant overhead caused by the logging process

AFT [41] builds an interposition between a storage engine and a common Serverless platform by providing an atomic fault tolerance shim and ensuring *exactly-once* semantics in a replay-based framework. Each logical request at the compute layer (an aggregation of multiple functions) is treated as a transaction. AFT guarantees that all of the updates made by a transaction are atomically installed at the storage layer. However, when an application function fails, none of its updates will be persisted, and its transaction will be aborted after a timeout. After that timeout, the whole transaction must be retried.

3.3.2 Checkpoint-based

As we previously explained, checkpointing techniques include two main phases: (1) periodically record the program state in storage, needed to recover from the failure and (2) restore the system state and memory from the checkpoint and resume the computation from there. This approach handles non-determinism by restarting from the point where the last checkpoint was taken and avoids a blowup in memory by saving the most recent application state rather than an arbitrarily long history.

There is still little research on checkpointing for fault tolerance in the Serverless domain. Kappa [8] stands out as the state-of-the-art work that addresses a problem similar to ours in a checkpointing fashion. Kappa is a framework that aims to deploy general-purpose applications using Serverless infrastructure, focusing on handling function timeouts and providing concurrency primitives that those platforms lack. To address function timeouts, Kappa presents an application-level checkpointing mechanism based on continuations (an abstraction that represents program state), periodically checkpointing function state and saving it to remote stable storage. The function can then be resume from previously stored checkpoints when it times out. First, the developer inserts checkpoint points through Kappa's library. Then, a compiler transforms the application code, generating checkpointing code, and re-packages the instrumented function along with Kappa's library, sending it to the FaaS platform. A coordinator is responsible for launching and resuming Kappa's functions that communicate back to the coordinator through RPC for checkpointing and synchronization reasons. Regarding performance, Kappa's overhead is around 100 ms for up to 1000 KB of checkpoint size and scales linearly from there. It is also shown that the checkpointing mechanism is scalable by running up to 1000 concurrent functions checkpointing every 100 ms while observing no additional overhead.

While each checkpoint may be efficient, being a periodic approach, Kappa's overhead is present in every single run to protect against faults that occur in a limited set of invocations. Besides, Kappa is an application-level approach for Python, which means that support for other programming languages would have to be added to Kappa's compiler, limiting which applications it can checkpoint. Even in Python,

there are several limitations to what can be checkpointed, e.g., exception handling is not supported.

Karhula *et al.* [42] also show a use case of checkpointing on an IoT edge FaaS platform. They argue that IoT devices that run Linux containers could be used as FaaS platforms within an IoT network. However, IoT devices are severely resource-constrained, and applications can be long-running and go beyond FaaS execution time limits. To tackle these issues, the authors propose a checkpointing approach that can pause idle containers to decrease resource waste and migrate long-running computations to platform nodes with lower load.

Checkpointing techniques have been developed in Serverless but for a completely orthogonal purpose: fast function startup. Catalyzer [43] demonstrates snapshot and restore in VMs to achieve millisecond Serverless cold starts. Instead of booting from scratch, Catalyzer restores well-formed checkpoint images or forks from running template sandbox instances. Fundamentally, it removes the initialization cost by reusing state, which enables general optimizations for diverse Serverless functions, obtaining a 1000$\times$ speedup over approaches with no checkpointing. REAP [44] leverages that functions generally access the same memory pages across multiple invocations. It records frequently accessed set of pages to proactively pre-fetch them into memory for subsequent invocations. This approach achieves a 3.7$\times$ reduction in cold start times compared to other checkpointing approaches. SEUSS [45] uses a custom Operating System that packages the application and other components in the same address space through Unikernel Contexts (a special Unikernel that consists of a high-level language runtime). When a function is invoked for the first time, a runtime snapshot is retrieved, and the function is initialized, having its Unikernel context cached. For subsequent invocations of that same function, when there are no warm nodes, the cached runtime is fetched, and the function is initialized from there. This approach reduces the total initialization time from 100 ms to 10 ms.

Checkpointing in other contexts. Although still not widely adopted in Serverless, checkpointing is a pervasive technique in systems literature to support fault tolerance, mainly in the form of system-level schemes (see Section 2.5.2). Some of these approaches may apply to our use case.

The concept of stopping, checkpointing and restoring processes and computations at system-level has been made popular by Berkeley Lab's Checkpoint/Restore (BLCR) project, which started in 2002 [46]. BLCR relies on kernel support to suspend and checkpoint a process, which has the advantage of not wrapping system calls, avoiding those overheads. This kernel approach requires an administrator to load a special module to operate and does not support checkpointing network communications like sockets, so the application must be aware of those limitations to handle them correctly. Plus, this approach may not be a good fit for serverless environments because they rely on kernel modifications, which the provider is required implement in existing Serverless environments.

DMTCP [47] is another well-known tool for checkpointing purposes. DMTCP checkpoints applica-

tions at user-level by injecting a pre-loaded shared library when the program starts, which wraps system calls as they happen. That library runs before the `main()` routine, creating a checkpoint thread. The checkpoint thread then creates a socket to a separate DMTCP coordinator process and registers itself. Once a checkpoint happens, the checkpoint thread creates an image of the applications and communicates it to the coordinator, which saves those images to disk. This approach has the advantage of not needing any recent kernel features nor administrative rights to checkpoint features but the fact that system calls are wrapped introduces a non-negligible performance overhead.

More recently, CRIU [48] has appeared as a popular tool among checkpointing solutions, as it performs fully transparent checkpoints with no need to pre-load special libraries. Contrary to DMTCP, CRIU does not wrap system calls, eliminating potential performance overheads that arise from that procedure. CRIU is also capable of checkpointing TCP, UDP sockets and established TCP connections, proving useful for a broader range of applications. However, it relies on more recent kernels that allow it to retrieve information from the operating system. This limitation may not be an issue for current FaaS services that are built on top of up-to-date kernels and, therefore, compatible with CRIU.

3.4 Optimizations to the Serverless infrastructure

Fault-tolerant approaches like the one we present in this thesis may benefit substantially from transparent optimizations to the underlying Serverless infrastructure and can exploit them to provide lower costs to the user and better end-to-end performance.

3.4.1 Harvested resources

In the area of harvested resources (explored in Section 2.6), prior work [11] suggests deploying Serverless computations on top of Harvest VMs [23], a special kind of evictable VMs that have a variable number of CPU cores. They argue that eviction-based failures are still rare because they require two low-probability events to happen simultaneously: a Harvest VM getting evicted while running a long invocation (an invocation that takes more time than a pre-determined grace period). In this paper, it is estimated that, in these VMs, only 0.0015% of the total number of invocations will be affected by evictions while achieving a cost reduction of up to 80% over regular VMs. However, for those invocations affected, the solution presented is to run them on regular VMs. When choosing where to run a specific function, if a conservative strategy is employed, much of the computation will have to be run on regular VMs. If a loose strategy is employed, some computations will be lost. Although this study motivates the usage of evictable resources for Serverless computing, it fails to provide a definitive solution to deal with invocations that are evicted, something we address with our work.

	Fault-tolerant Strategy	Supported Faults		Other Features	
		Planned Faults[1]	Crash Faults[1]	No User Involvement	No Runtime Overhead
FaaS	Replay	✗	✗	✗	✗
R-Check	Execution Migration	✓	✗	✓	✓
IaaS	Execution Migration	✓	✗	✓	✓
Kappa [8]	Periodic Checkpointing	✓	✓[2]	✗	✗
Beldi [14]	Logging	✓	✓	✗	✗

[1] In the case of non-idempotent workloads. For idempotent workloads, all the approaches can recover by just replaying the whole execution.

[2] Since Kappa takes periodic checkpoints, some computation might be lost when recovering from crash faults.

Table 3.1: Comparison of how different platforms and systems handle faults.

3.4.2 Other transparent improvements

As mentioned in Section 3.3.2, several papers [43–45, 49] also address the cold start problem through checkpointing. Towards that goal, they save a generic image for a specific function (with all the dependencies and libraries needed to run it). When a warm instance is unavailable, these generic pre-initialized images are retrieved instead of booting up from scratch, which reduces the total initialization time.

Moreover, other works presented storage and caching solutions optimized for Serverless. Along those lines, Cloudburst [50] is a custom runtime for running Serverless workflows that deploys caches on the same nodes as the compute workers, allowing for low-latency data access. These caches transparently write updates to a high-performance key-value store, Anna. Faa$T [51] presents an auto-scaling distributed cache solution that pre-loads data frequently used by the application to speed up read and write operations. OFC [52] exploits the resource waste of each Serverless container to generate capacity for an in-memory caching system in FaaS platforms.

3.5 Discussion

In an ideal world, functions would always be idempotent; therefore, we could simply restart them upon an eviction event. However, this is not always the case, and writing idempotent code can easily become an entry barrier for developers of Serverless code. This essentially precludes any interaction with external resources such as reading or writing a value to/from external storage without proper idempotency checks. For example, if a function is re-executed, the second execution has to guarantee that it reads the same value as the previous execution. Otherwise, if the value that results from the read is used to

produce the function output, the function may not result in the same value, and therefore, idempotency would not be achieved.

Idempotency is crucial when evictions are handled by simply replaying a function elsewhere (this is the default behaviour in public Serverless platforms). In this work, we advocate that functions should not be restarted, but instead 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 3.1 presents how different cloud platforms and systems handle two different types of faults: planned and crash faults. FaaS platforms have no out-of-the-box support for planned faults. To improve that, we intend that R-Check supports migration of computations when planned faults occur in the same way that IaaS already does, particularly for maintenance purposes. Differently, systems that employ periodic checkpointing (Kappa [8]) or logging (Beldi [14]) can cope with both planned and crash faults. However, these systems handle the latter at the expense of some user involvement and high runtime overheads (as we evaluate in Section 5.6). 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, we aim that R-Check can strike a balance between FaaS (which does not support planned faults for general code) and systems that support both types of faults (the latter of which are not handled by the system in today's cloud offer), by proposing a fault model similar to IaaS, supporting planned evictions but not crash faults. By doing so, we want to enable users to choose a sensible balance between performance overhead, user involvement and supported faults. Next, we present R-Check and explain how we accomplished these features.

4

R-Check Design

Contents

4.1 Overview

In this chapter, we propose our solution for the problem of failure handling in Serverless computing. To this end, we present R-Check, a Serverless framework that is capable of handling Serverless functions that may suffer from failures due to the resource optimization conducted by the cloud provider, namely a possible use of harvested resources, while taking advantage of a model that introduces minimal overhead. This framework can be easily deployed on top of existing Serverless infrastructures to provide out-of-the-box handling of controlled faults. We start this chapter with our core design choices (Section 4.2). Then we proceed to the overall architecture of R-Check (Section 4.4) and detail how we checkpoint and restore applications (Section 4.5). Finally, we present some implementation considerations (Section 4.7.)

4.2 Key design choices

During the design phase of our solution, we explored multiple architectural directions representing different trade-offs of two main axes. The first axis was deciding if checkpointing should follow a periodic policy or a reactive one and the other axis was concerned with deciding the scope of our checkpointing approach (application-level or system-level). In the following sections, we will dive into each of these axes and choices in more detail while clearly defining the expectations concerning the types of events we intend to withstand, encapsulated in a fault model that influenced those decisions.

4.2.1 Be reactive, not proactive

We believe that Serverless computing will become increasingly adopted since it is the paradigm that most faithfully embodies the principles underlying cloud computing. With this increased adoption, we will see more diversity in function logic (with increased pressure to go beyond stateless, short-running, and idempotent code).

At the same time, many of the current techniques that cloud providers use for optimizing their infrastructure usage (namely evictable VMs) are also being adopted in the Serverless context. The critical insight behind evictable VMs is that they can be purchased at much lower costs since the corresponding resources can be terminated whenever the cloud provider requires them elsewhere. This behaviour means that evictions must be properly handled to avoid risking computations from paying customers. An important point is that evictions are controlled events, so when a VM is requested for termination, it receives a grace period to handle ongoing tasks. VM evictions are rare; for example, around 65% of AWS Spot instances types have an eviction rate lower than 5%, and around 78% do not go beyond a 15% eviction rate [53]. [11] shares even more encouraging results on Harvest VMs, estimating that

evictions only cause 0.0015% of total invocations to fail.

Another challenge we explore is function timeouts. For long-running functions, executions might time out, and all computation done up to that point is lost. The most common way to ensure safe execution within established duration limits is partitioning function logic into smaller functions. This is a burdensome task for the developer and requires additional effort to write Serverless code. Moreover, this workaround does not work if long execution are bottlenecked on a single specific operation, in which case, partitioning is not a viable solution.

These characteristics allow a window of opportunity to handle evictions and function timeouts gracefully. It allows for intervention to ensure fault tolerance only in the rare occasions when needed instead of always assuming these failures may occur. Therefore the price to handle them must be paid in every single invocation. To this end, we support the idea that Serverless platforms should migrate executions by reactively checkpointing and restoring applications if (and only if) faults happen. This is in contrast to proactive approaches where checkpoints are periodically taken, introducing avoidable overheads in the specific case where these failures are scheduled events and only affect a minority of the executions.

4.2.2 Application-level Checkpointing vs. System-level

Application-level checkpointing approaches allow one to reason about the important data to restore the function context during a restore operation, reducing its memory footprint to the bare minimum. Although largely optimized in terms of checkpointed data, an application-level approach requires significant programming effort to write checkpoint/restore code or to design a compiler that inserts custom checkpoint/restore code. Plus, application-level checkpointing essentially limits the regions of the application code where a snapshot can be produced. Consequently, when employing an application-level checkpointing scheme, it is convenient to write functions in a more unfolded way using fewer third-party libraries when possible to achieve a fine-grained control over checkpointing. This is not only more time-consuming but more error-prone. Listing 4.1 shows an Advanced Encryption Standard (AES) encryption algorithm with Cipher Block Chaining (CBC) mode of operation for a regular execution with third-party library calls (“Implementation with library calls”) and a modified version more suitable for application-level checkpointing (“Implementation without library calls”). If the “regular algorithm” function were used with an application-level approach, checkpointing locations would be much more scarce. Also, if any library calls take longer to return, checkpointing would have to wait for the end of such call, possibly missing the grace period window and terminating without checkpointing.

```
#    Implementation
# without library calls

import os

s_box = (...)
r_con = (...)
def sub_bytes(s):
    ...     # collapsed code
def shift_rows(s):
    ...     # collapsed code
def mix_columns(s):
    ...     # collapsed code
def add_round_key(s, k):
    ...     # collapsed code
def bytes2matrix(text):
    ...     # collapsed code
def matrix2bytes(matrix):
    ...     # collapsed code
def xor_bytes(a, b):
    ...     # collapsed code
def inc_bytes(a):
    ...     # collapsed code
def pad(plaintxt):
    ...     # collapsed code
def split_bks(msg, size, require_pad):
    ...     # collapsed code
def get_key_iv(passwd, salt, workld):
    ...     # collapsed code

class AES:
    rounds_by_key_size = {...}
    def __init__(self, master_key):
        ...     # collapsed code

    def _expand_key(self, master_key):
        ...     # collapsed code

    def encrypt_blk(self, plaintxt):
        p_state = bytes2matrix(plaintxt)
        add_round_key(p_state,
                self._key_matrices[0])
        for i in range(1, self.n_rounds):
            sub_bytes(p_state)
            shift_rows(p_state)
            mix_columns(p_state)
            add_round_key(p_state,
                self._key_matrices[i])
        sub_bytes(p_state)
        shift_rows(p_state)
        add_round_key(p_state,
                self._key_matrices[-1])
        return matrix2bytes(p_state)

    def encrypt_cbc(self, plaintxt, iv):
        plaintxt = pad(plaintxt)
        blocks = []
        previous = iv
        for p_text_b in split_bks(plaintxt):
            block = self.encrypt_blk(
                xor_bytes(plaintext_b, prev))
            blocks.append(block)
            prev = block
        return b''.join(blocks)

def encrypt(key, plaintxt, workld):
    salt = os.urandom(SALT_SIZE)
    iv = get_key_iv(key, salt, workload)
    ciphertext = AES(key).encrypt_cbc(plaintxt, iv)
    return ciphertext
```

```
#         Implementation
#       with library calls

from Crypto.Cipher import AES
from Crypto.Util.Padding import pad

def encrypt(key, plaintxt):
    iv = get_random_bytes(AES.block_size)
    cipher = AES.new(key,
            AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(
          pad(plaintxt, AES.block_size))
    return ciphertext
```

Listing 4.1: Comparison of Python AES encryption algorithms with CBC mode, with and without library support.

Contrarily, system-level checkpointing is a fully user-transparent way of implementing checkpoints. This approach involves saving the entire state of the application, including all processes and temporary data, at checkpoint time, which produces a large memory footprint. With a system-level technique, however, there is no additional programming effort apart from the application code itself, so users can submit their functions completely unmodified. In this approach, there is usually a deep concern about the portability of the checkpoint objects. System-level checkpointing schemes are hardly portable when migrating between machines but, in Serverless checkpointing, this should not be an issue since the source and destination nodes will most likely present the same architecture and operating system configurations.

Furthermore, with our evaluation in Section 4.2.2, we will show that a system-level checkpointing approach provides an attractive compromise between transparency and performance and is, in turn, much more comparable in terms of overhead to application-level checkpointing than one might think beforehand. For this reason, combined with the multiple advantages of running a system-level checkpoint mechanism, we opted for a system-level approach.

4.3 Fault Model

As introduced in Section 3.5, with this approach, we can guarantee proper failure handling for the following scenarios: i) controlled VM evictions that grant a termination grace period; ii) long-running functions approaching the platform's timeout. For the particular case of function timeouts, we advocate that the Serverless platform emits a signal some time before the execution times out (possibly the same as the eviction grace period) to allow our checkpoint mechanism to take action.

By dealing with these faults, we are approximating the FaaS fault model to the IaaS one since, in IaaS, migration of workloads is already common, especially for scheduled maintenance events. Particularly, all the workloads that are already supported in IaaS will also be supported with R- Check. This ultimately means that, by employing our solution, we are contributing to the generalization of Serverless, lifting current restrictions from those platforms concerning the type of workloads they can run. However, like IaaS, this model does not handle critical applications that must be able to withstand the unexpected crash of a machine (e.g., due to a power outage or a hardware fault). In our view, in today's cloud deployments, such critical applications must already employ replication protocols (e.g., Paxos [54]) to protect their critical state in the presence of such events and, therefore, can be seen as orthogonal to our proposal.

4.4 Architecture

In this section, we present the overall architecture of our solution, which we name R-Check[1].

Depicted in Figure 4.1 is the high-level architecture we propose. R-Check 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.

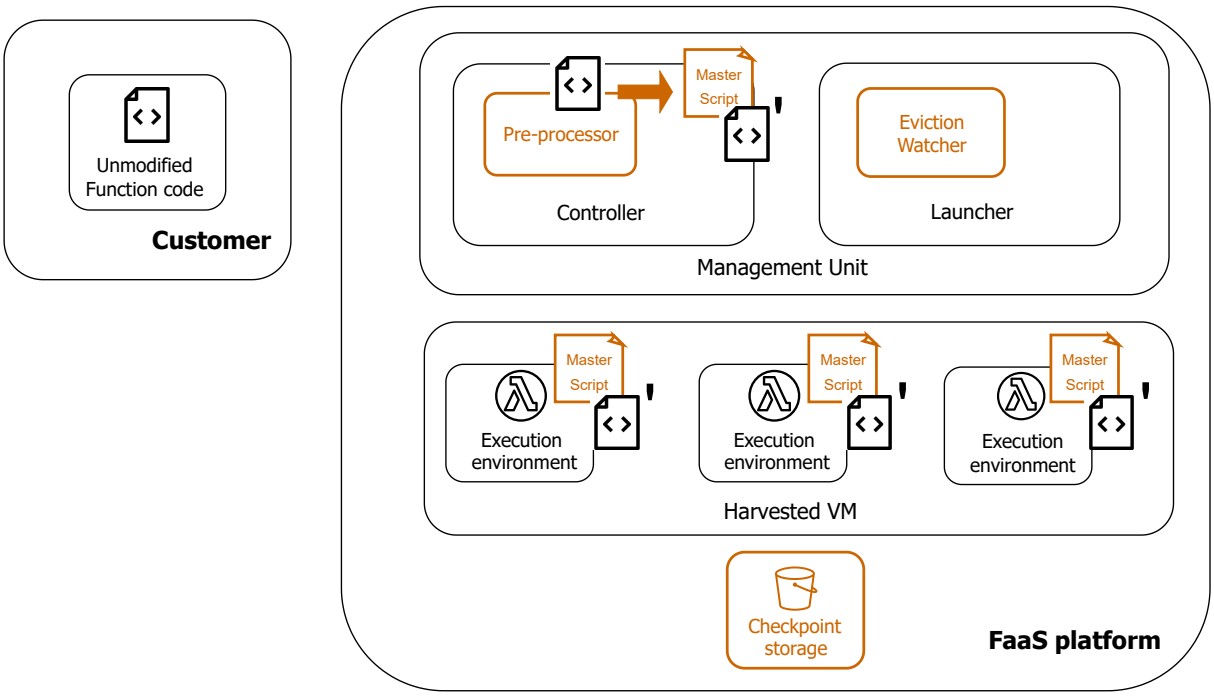


Figure 4.1: Overview of a generic FaaS platform with R-Check's specific components highlighted in orange.

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, there is no added programming effort than what would be expected from any well-known FaaS offering. 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. To realize R-Check, 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. Once the function is successfully created, users can request invocations for

[1]Short for Reactive Checkpointing. To the best of our knowledge, it is the first mechanism that combines a checkpointing approach with a reactive timing of action to achieve fault tolerance in Serverless.

it.

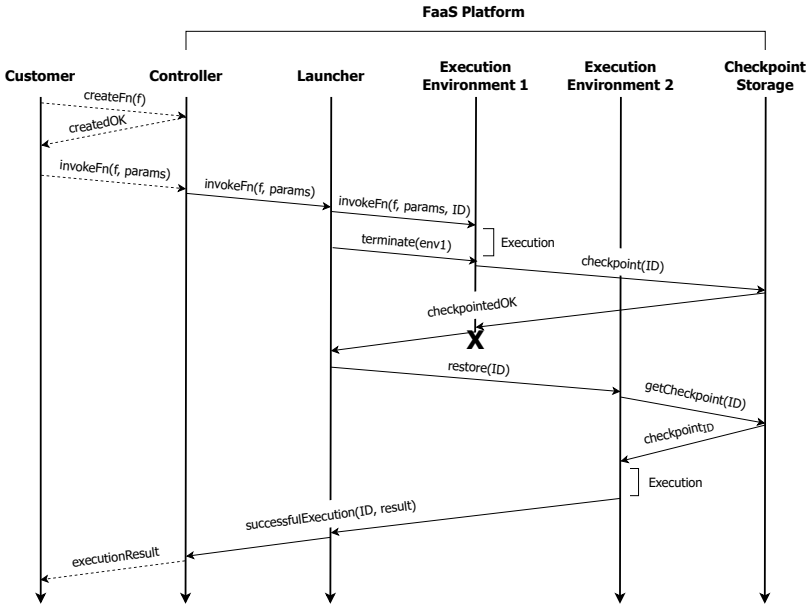
Algorithm 4.1: R-Check master script

```
// Checkpointing segment of the master script
Input  : function, params
Output: r_check_result
begin
    execution_id ⟵ run_function(function, params)
    fault_occurred ⟵ FALSE
    function_result ⟵ get_execution_result(execution_id)
    while not fault_occurred and function_result == null do
        fault_occurred = fetch_termination_notice()
        function_result ⟵ get_execution_result(execution_id)
    if fault_occurred then
        checkpoint_result ⟵ checkpoint_function(execution_id)
                                  ▷ includes checkpoint process + checkpoint object upload
        r_check_result ⟵ checkpoint_result, CHECKPOINTED_OK
    else
        if success(function_result) then
            r_check_result ⟵ function_result, SUCCESSFUL_EXECUTION
        else
            r_check_result ⟵ function_result, ERROR
```

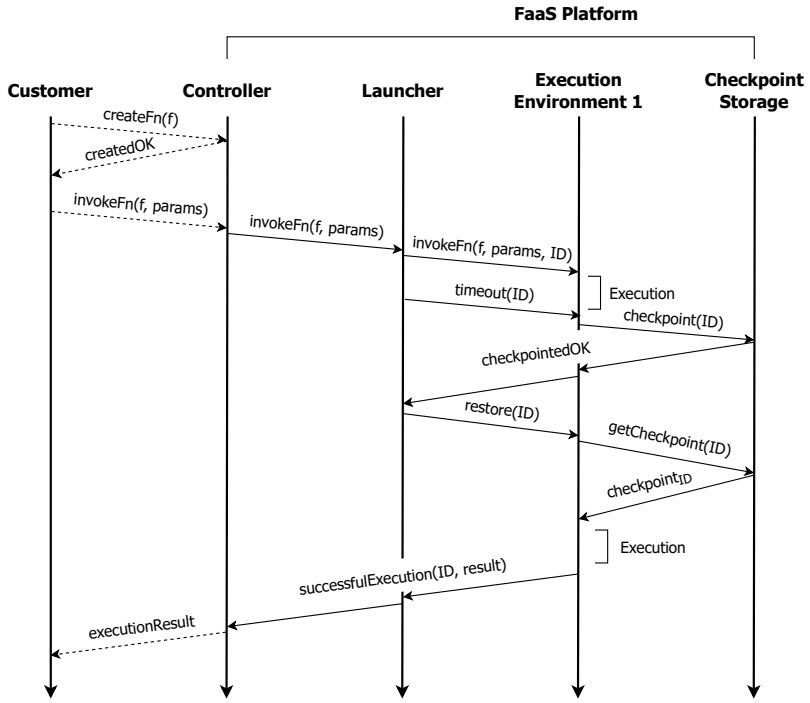
```
// Restoring segment of the master script
Input  : execution_id
Output: r_check_result
begin
    restore_function(execution_id)      ▷ includes checkpoint object download + restore process
    ...                                 ▷ checkpointing may be triggered again until the function returns
    function_result ⟵ get_execution_result(execution_id)
    if success(function_result) then
        r_check_result ⟵ function_result, SUCCESS
    else
        r_check_result ⟵ function_result, ERROR
```

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, 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.

A typical function execution in R-Check works as follows. The master script, whose logic is described in Algorithm 4.1, runs the user functions with the user's desired parameters and waits until its completion. Putting aside the case where the execution finishes without any interference, there are two relevant use cases that trigger custom R-Check actions.



(a) Use case 1 - VM eviction



(b) Use case 2 - Function timeout

Figure 4.2: Communications in a R-Check-enabled FaaS platform. Solid arrows represent communication within the FaaS platform and dashed arrows represent communication customer/platform over the network.

Figure 4.2 introduces those use cases. Figure 4.2(a) represents a case where a VM is evicted while running a function. In this case, we assume that execution environment 1 is placed on VM1, and execution environment 2 is placed on a different VM2. When a termination is detected for VM1 by the eviction watcher, the launcher relays a termination request to each execution environment from VM1. This request is captured by the function's corresponding master script, which 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 from that VM1 have been taken. The master script, now running on a new execution environment on VM2, is requested to retrieve the checkpoint object with the previous ID from storage. After that, it re-establishes the context and state of the function execution, avoiding restarting it from the beginning. 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.

Figure 4.2(b) represents the case where a function runs for too long, approaching the execution time limit. When a given invocation approaches that time limit, the checkpoint/restore process is triggered. From here, the flow of operations is very similar to the previous case. The only noticeable difference is that, since this scenario does not contemplate an eviction, the function might be restored in the same execution environment, depending on the decisions made by the launcher's load balancer.

4.5 Checkpointing in R-Check

In this section, we describe how R-Check creates checkpoints and restores them. Particularly, we present the tool we used for this purpose and optimizations made so that it would more closely fit the Serverless context.

4.5.1 Checkpoint & Restore tool: CRIU

The concept of stopping, checkpointing and restoring processes and computations at system-level and computations has been around in systems literature for a long time [55]. As stated in Section 3.3.2, many of these approaches, however, are not entirely suitable for Serverless computations because they either require modifications to the kernel (BLCR [46]), which the cloud provider would have to implement or introduce additional overheads on system calls (DMTCP [47]). On the other hand, CRIU [48] 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.

CRIU can dump the complete state of a running Linux process to a file and start an identical process from just the information in that file, with memory, file descriptors and even its Process Identifier (PID)

all in order. To better understand the key techniques used by it, we now describe a typical process of checkpoint and restore.

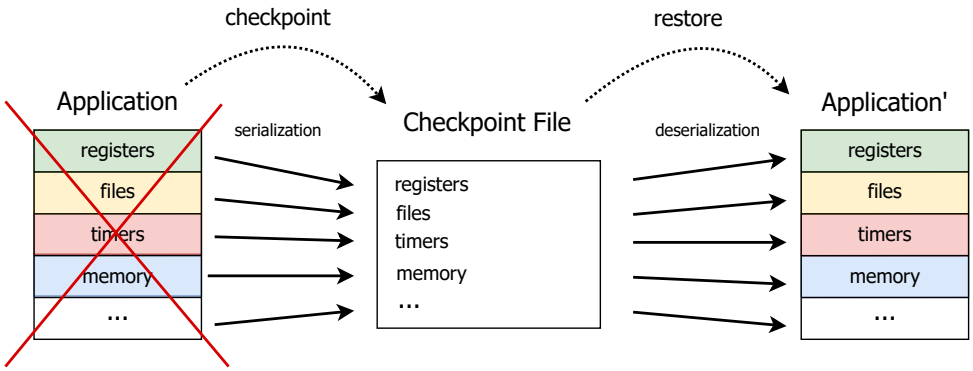


Figure 4.3: CRIU checkpoint/restore architecture.

When **checkpointing** a process with a PID `p` (command: `criu dump -t PID -vvv -o dump.log`), the entire tree of processes associated with PID `p` as the root process must also be checkpointed. CRIU starts by recursively freezing (i.e. stopping via cgroups[2] all of these processes and their respective threads). While doing so, CRIU *attaches* to these processes through the `Ptrace` system call[3]. 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 (as shown in Figure 4.3), the parasite code is removed, and the processes can resume or be killed, depending on the user's intentions.

Upon **restoring** (command: `criu restore -vvv -o restore.log`), CRIU first looks for the path `/proc/sys/kernel/ns_last_pid`. This file contains the last PID that was assigned by the kernel. When the kernel needs to assign a new PID, it looks into `ns_last_pid`, gets the `last_pid` and assigns `last_pid + 1`. To restore the desired PID `P` from the root process of the application, CRIU locks `ns_last_pid`, writes `P-1` and calls $clone()$[4]. 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.

[2]Cgroups is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes. The cgroup interface also possesses a *freezer* that can arbitrarily start and stop processes without using signals.

[3]`Ptrace` is capable of monitoring and controlling other processes.

[4]$clone()$ create a new "child" process, in a manner similar to $fork()$.

CRIU Limitations. While CRIU is a broadly applicable tool, we acknowledge that it does not work for every single use case. For example, it is not possible to checkpoint processes that are already using `Ptrace` (e.g., `gdb`, `strace`). Also, CRIU doesn't dump tasks with held file locks nor sockets other than TCP, UDP, UNIX, packet and netlink. For a more exhaustive list of limitations, we refer to CRIU's website. These limitations, however, seem to be a negligible number of use cases and, therefore, do not make this tool unfeasible in our context.

4.6 Employing CRIU in R-Check

As mentioned in Section 4.5, we use CRIU to enable a seamless checkpoint and restore procedure, which successfully supports various kinds of applications and processes. Nonetheless, CRIU was originally built to snapshot processes in serverful computing. Thus, it is not equipped with out-of-the-box support for our use case in terms of features and performance. We now expand on some of those unsuitable features and explore improvements made to fix them.

4.6.1 Checkpoints directly in remote storage

As described before, CRIU retrieves the information needed to perform a checkpoint operation from the application and then saves the data on disk, creating the checkpoint image files. Given that we intend to migrate that data into a new environment, those files need to be exported out of local disk to remote storage units. With vanilla CRIU, this would be possible by bundling the checkpoint files into an archive, which would then be compressed and uploaded to remote storage. This process is depicted in Figure 4.4.

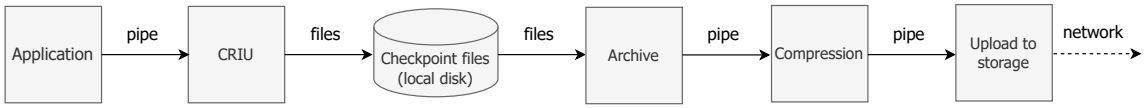


Figure 4.4: CRIU conventional data flow.

Although this approach works, it raises some concerns regarding performance and memory usage. The primary issue in this approach is that we are hitting local disk, both for writing checkpoint files and, later, for reading them, 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 4.5) 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.

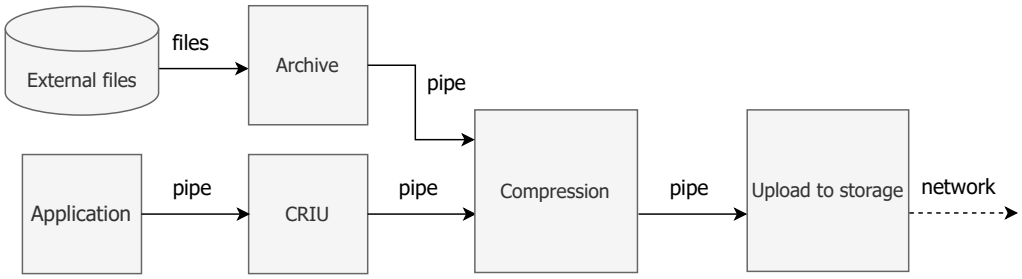


Figure 4.5: Modified CRIU data flow.

Since this approach dramatically reduces data copies to the local filesystem, the checkpointing process achieves much higher throughput than vanilla CRIU.

From the restore side, the process is relatively similar to its checkpointing counterpart, except we reverse the data flow that we just presented: the data is decompressed and then unarchived before going through CRIU and, finally, restoring the execution. This restore process, however, presents a problem: CRIU reads the checkpoint files out-of-order. For example, CRIU's file `inventory.img`, a top-level description of all the other image files, is written at the end of the stream during the checkpoint. On the other hand, during restoration, this file is read at the beginning of the stream. 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$\times$ 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. In the worst case scenario, if the application consumes all the memory allocated to that container, we might need twice as much memory as that container has. To deal with this potential issue, during restore, as CRIU reads data on the pipe, that memory is deallocated immediately after, thus avoiding this blowup in memory usage.

4.6.2 Local files of a function

In various workloads, functions create, read, update and delete local files throughout their execution. Thus, 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. CRIU itself expects that the filesystem remains the same on both nodes but does not provide any support for cloning relevant files from one side to the other. While this can be achieved by manually copying the files from the original container to remote storage and then downloading them back into the destination container, this approach would require opening another connection to the storage service and transferring those files separately.

We mitigate the above limitation by taking advantage of the fact that Serverless environments gener-

ally provide a small writable folder which developers can use (usually `/tmp`). With R-Check, we bundle all the files and subfolders 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.

4.7 Implementation

In this section, we discuss the implementation details of our R-Check prototype. We use Apache OpenWhisk as the underlying Serverless infrastructure. Our prototype communicates with the OpenWhisk cluster to create, delete and invoke functions. It is written in Golang mainly because the official client library to access the OpenWhisk API is written in Golang and functions were mainly written in Python. Our implementation also depends on a modified version of CRIU, CRIU-Image-Streamer [56], written in Rust, that provides support for streaming of checkpoint images to and from CRIU during checkpoint/restore, as explained in Section 4.6, and S3 for stable storage.

4.7.1 Serverless platform of choice

In order to choose the Serverless platform on top of which we build our system, it was necessary to look at the requirements of its building blocks. In this case, we must consider the restrictions of CRIU. CRIU needs access to the kernel capabilities `CAP_SYS_ADMIN`, which performs a wide range of system administration and privileged operations, and `CAP_SYS_PTRACE`, which enables monitoring of arbitrary processes. Given that `CAP_SYS_ADMIN` is an overloaded capability that, if enabled, can raise several security issues, more recently, CRIU maintainers have developed a patch to run CRIU without it by enabling a brand new kernel capability `CAP_CHECKPOINT_RESTORE`, that has tailored permissions for CRIU. However, for security reasons, popular commercial Serverless environments, such as AWS Lambda, Azure Functions or Google Cloud Functions, cannot be granted elevated kernel capabilities. Besides, their platforms' code and detailed infrastructure are kept concealed so they cannot be modified nor fully comprehended.

Considering these constraints, we opted for an open-source Serverless platform that would let us modify the infrastructure as needed. With this in regard, we chose Apache OpenWhisk as the platform to implement our prototype. Openwhisk is a widely adopted open-source FaaS platform started by and still used extensively by IBM in their cloud functions' offerings. We now provide a top-level overview of the OpenWhisk architecture.

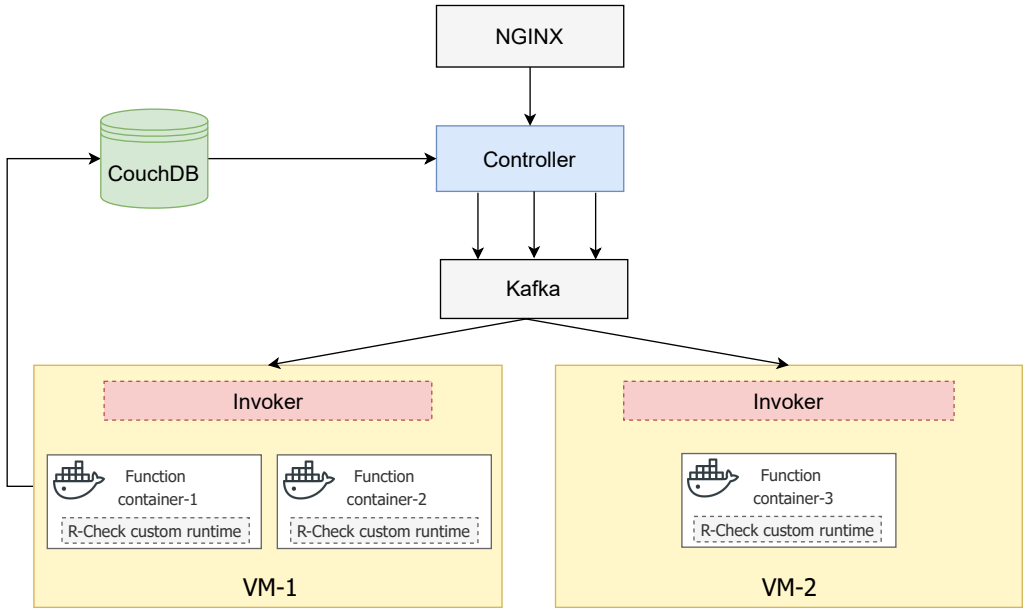


Figure 4.6: Architecture of Apache OpenWhisk with R-Check custom runtime.

4.7.2 OpenWhisk Architecture

The OpenWhisk platform internally works as follows. Firstly, an Nginx web server exposes a public-facing HTTP endpoint primarily used as a reverse proxy to where clients send all their requests. After the request passes through the reverse proxy, it reaches the controller that acts as a gatekeeper to the system: it performs authentication and authorization of every request and hands it over to the next component. The controller then assigns requests to invokers through a Kafka [57] messaging queue. If the function code is not already available, the invoker fetches it from CouchDB [58] and injects it into the function container. The invoker is responsible for either reusing a "hot" container or launching a new "cold" container based on the current system load. The function runs, and once the execution is finished, the result and other execution metadata are pushed back into the CouchDB system for later retrieval. Running idle containers are killed after a fixed keep-alive period (10 minutes by default).

4.7.3 Integration of R-Check in OpenWhisk

In order to implement a checkpoint/restore mechanism in OpenWhisk based on CRIU, we modified the invoker code so it could launch containers with the required kernel capabilities needed by CRIU and an attached Teletypewriter (TTY), a virtual terminal session that allows us to retrieve information printed to `stdout` by functions.

Checkpoint from inside or outside the container? Customers often over-provision their function resources and waste many resources. This would mean that checkpoint sizes would be much bigger than

required for the amount of memory being used in reality. For this reason, we opted for an approach that checkpoints function from within the containers themselves, reducing the memory footprint introduced by the checkpointing process.

To make this work, we have to make CRIU binaries and libraries available in the execution container. For this, we develop a modified runtime image that includes our modified CRIU binaries, making sure every container is capable of checkpointing and restoring applications if needed. Each function is run on top of our custom runtime image as shown in Figure 4.6.

In this current scenario, a problem arises: applications cannot checkpoint themselves with CRIU. CRIU only allows to checkpoint other processes, so effectively, we cannot launch functions directly with R-Check because there would not be a way to checkpoint them from inside the container. This is the reason behind the addition of a master script, introduced in Section 4.4.

```
import subprocess as sp

ROOT_APP_PID = 10000

def master_handler(metadata, user_args):

    waiting_period_before_timeout = metadata["timeout"]

    # Set a custom previous PID
    set_last_pid(ROOT_APP_PID - 1)


    # Invoke user function in a new process
    sp.Popen(["/bin/bash", "-c", "python user_function.py " + user_args])


    # Sleep X seconds before taking a checkpoint
    time.sleep(waiting_period_before_timeout)

    # All files written will be stored in the writable '/tmp-action' folder
    checkpoint_command = "criu-image-streamer --external-files /tmp-action/" +
                "--app-pid " + str(ROOT_APP_PID) + " dump"

    # Checkpoint application
    checkpoint_process = sp.Popen(["/bin/bash", "-c", checkpoint_command])

    # Retrieve the checkpoint process output
    checkpoint_process_output = checkpoint_process.stdout.read()

    # Check that checkpoint was good
    try:
        checkpoint_stats = check_correct_processing(checkpoint_process_output)
    except Exception as e:
        return {"status":"ERROR", "message":"(Internal) {}".format(str(e))}

    return {"status":"CHECKPOINTED", "metadata":checkpoint_stats}
```

Listing 4.2: Master script flow of operations - Checkpoint.

This script is packaged with the user code on every function creation. Listing 4.2 shows the actual implementation of the master script checkpoint function. Once an invocation for that user function is requested, OpenWhisk will invoke our master script as if it were the user function. In order to easily evaluate checkpoint and restore metrics, in our evaluation (Chapter 5), we simulate eviction events in the master script. Thus, it retrieves a function timeout period passed as a metadata argument to simulate

an eviction – this is the time the script will wait before triggering the checkpoint process. It then sets up a custom previously assigned PID, preferably a high one, so we can control which PID will be used and avoid PID collisions on restore. After that, it runs the user function with the given arguments, and when the timeout period elapses, it triggers the checkpointing process. Note that we include all the contents of `/tmp-action` in the checkpoint image since it simulates the only writable folder available to the user and needed for a successful restoration. Once the process returns, we check if it was successful and, if so, return a `CHECKPOINTED` flag and some checkpointing metrics to the platform.

On restore, as seen in Listing 4.3, the master script retrieves the checkpoint key and the app PID in which the original process will be restored. Then, it immediately triggers a restoration process, re-launching the interrupted function. When it finishes, the master script checks if the restore was successful and waits for the user function result. Finally, we return with a a `SUCCESS` flag, the function result and restore metrics.

While our prototype focuses on python functions, it is trivial to reproduce this master script logic for other OpenWhisk supported languages.

```
import subprocess as sp

def master_handler(metadata):

    # Retrieve checkpoint key to identify the corresponding checkpoint object
    checkpoint_key = "checkpoint-" + metadata["invocation_ID"]

    # Retrieve the desired PID for the application
    root_app_pid = metadata["root_app_pid"]

    # Restore the application with the same PID
    restore_command = "criu-image-streamer --checkpoint-key " + checkpoint_key
        + " --app-pid " + str(ROOT_APP_PID)  + " restore"

    restore_process = sp.Popen(["/bin/bash", "-c", restore_command])

    # Retrieve the restore process output
    restore_process_output = restore_process.stdout.read()

    # Check that restore was good
    try:
        restore_stats = check_correct_processing(restore_process_output)
    except Exception as e:
        return {"status":"ERROR", "message":"(Internal) {}".format(str(e))}

    # Wait for user function to finish and retrieves result
    function_result = wait_function_termination()

    return {"status":"SUCCESS", "result":function_result, "metadata":restore_stats}
```

Listing 4.3: Master script flow of operations - Restore.

4.7.4 Remote Storage

R-Check uses external storage units to store checkpoint objects. Currently, we use S3 as the primary storage service because it is very cost-effective, reliable and does not require additional setup burden.

Besides, because we used AWS as our cloud provider of choice during our evaluation (see Chapter 5), we could achieve the best networking performance possible by communicating to S3 through a Virtual Private Cloud (VPC) inside AWS internal infrastructure.

We also tested Redis [59], an in-memory database that persists on disk. Although more performant, Redis imposes a 512 MB value entry size limit for any data type. R-Check's checkpoint objects can be very large, possibly exceeding this threshold which deems this alternative impractical.

5

Evaluation

Contents

5.1 Overview

In this chapter, we evaluate R-Check as a serverless framework to deploy cloud functions and compare it with other state-of-the-art systems for fault tolerance in the Serverless domain. The main objective of this system is to provide an efficient checkpointing mechanism to handle controlled faults. Towards that end, during the evaluation, we try to answer different questions regarding the solution's applicability and performance, stated as follows:

- What is the overhead introduced by R-Check? Moreover, how does this overhead vary with how much memory each function consumes? (Section 5.3)
- Is this approach effective and efficient for various real applications? (Section 5.4)
- How does R-Check compare with an application-specific checkpointing mechanism in terms of performance and memory consumption? (Section 5.5)
- How does R-Check's reactive checkpointing scheme compare with state-of-the-art preventive approaches, both logging-based and checkpoint-based? (Section 5.6)
- What are the limits of realistic checkpointing: how many functions can be concurrently checkpointed within a reasonable grace period? (Section 5.7)

We believe these experiments represent the most relevant metrics to verify the validity of R-Check in this problem space.

5.2 Experimental Setup

Having mentioned the question we want to answer, we now describe the experimental setup for evaluating our prototype.

For these experiments, we ran our implementation on an Apache OpenWhisk deployment, as discussed in Section 4.7. 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 instance and one invoker instance because, for these experiments, we do not expect to accommodate heavy compute loads. 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 is equipped with 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

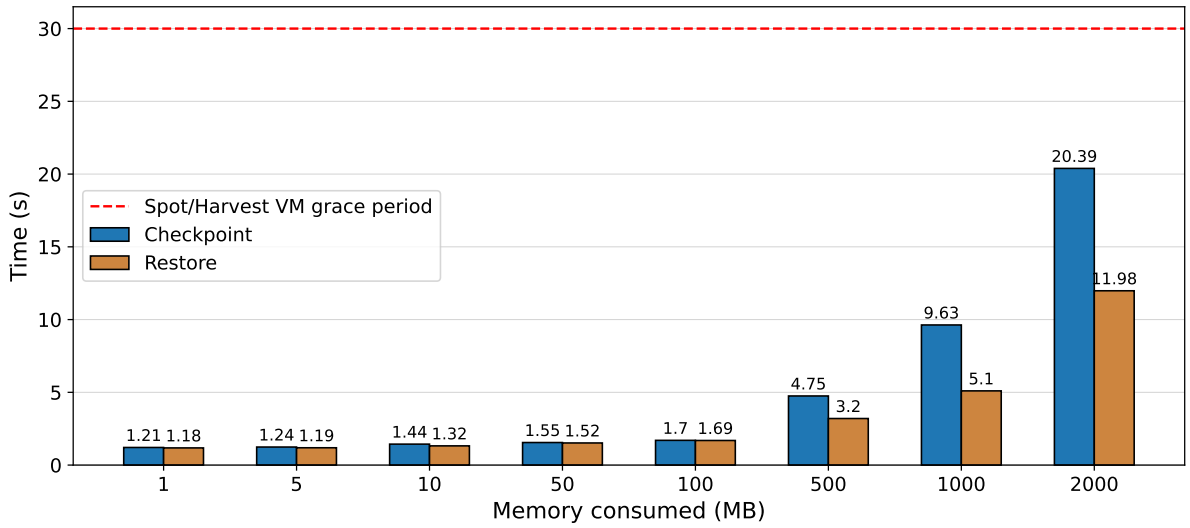


Figure 5.1: Checkpoint/restore times for various sizes of allocated memory. Each bar represents the median (P50) over 50 executions.

an internal VPC gateway endpoint, allowing for the maximum possible network throughput between EC2 and S3.

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. The relevant performance metrics were stored in an S3 bucket specific to that workload, available for later retrieval. Similarly, every function's input, output, and checkpoint objects were stored in their own S3 bucket. For compression of the checkpoint objects, we use the LZ4 [60], a popular and fast lossless compression algorithm that achieves a $\approx 3\times$ compression factor for a typical Python application.

Unless otherwise stated, each one of the individual experiments we describe was executed 50 times.

5.3 R-Check Overheads

In order to assess how R-Check performs, we need to look at the limiting factors for our checkpointing tool: CRIU. 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. Halfway through the execution, at 15 seconds, the function is interrupted, and the checkpointing/restore process is triggered.

Figure 5.1 presents the median checkpoint and restore times for allocated byte array sizes ranging

Operation	Average Throughput (MB/s)
CRIU Checkpoint / Restore	1650 / 640
LZ4 Compress / Decompress	1022 / 2345
S3 Upload / Download	50 / 82

Table 5.1: Average throughputs observed for each step of our checkpoint/restore approach.

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 ($\approx$ 20.4 seconds), namely within the time allowed by the grace periods granted in the current cloud offer (typically 30 seconds).

From the plot, we can also depict that restore operations take less time to complete than checkpoint operations for all values measured. To understand the reason behind this, we must analyze the throughput of each operation of R-Check: CRIU checkpoint, compression and upload to S3. Table 5.1 presents the average throughput for each stage in the checkpoint/restore procedure observed during the aforementioned experiment. From the table, we can understand that the primary bottleneck of checkpoint/restore is the network since it is, by far, the stage with the lowest throughput, dominating all executions. Moreover, we see that the download rates observed are higher than the upload rates, contributing to the higher overhead of checkpointing compared to the restore procedure.

To understand the variation of memory in the checkpoint object, Figure 5.2 presents checkpoint sizes as a function of the allocated memory in the function. It shows not only a near-linear growth of checkpoint sizes with the increase in memory allocated, but also very 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.

5.4 Real Applications

In this section, to demonstrate the generality of R-Check for Serverless applications, we present an evaluation of using our system for multiple real applications and functions. 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 benchmark applications from SeBS [61] and FunctionBench [62], two popular benchmark suites in the Serverless domain. We now describe each of the chosen benchmarks, written in Python.

Compression (FunctionBench). This benchmark takes a file as input and compresses it through GZIP, a single-file/stream lossless data compression utility. The primary objective of this application is to represent realistic disk I/O-heavy operations and assess if R-Check can handle interruptions during read/write

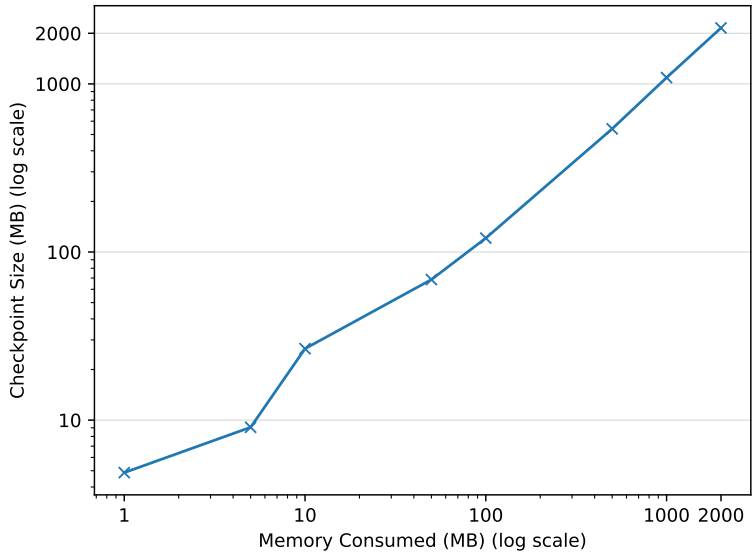


Figure 5.2: Checkpoint sizes for various sizes of allocated memory. Each point represents the median (P50) over 50 executions.

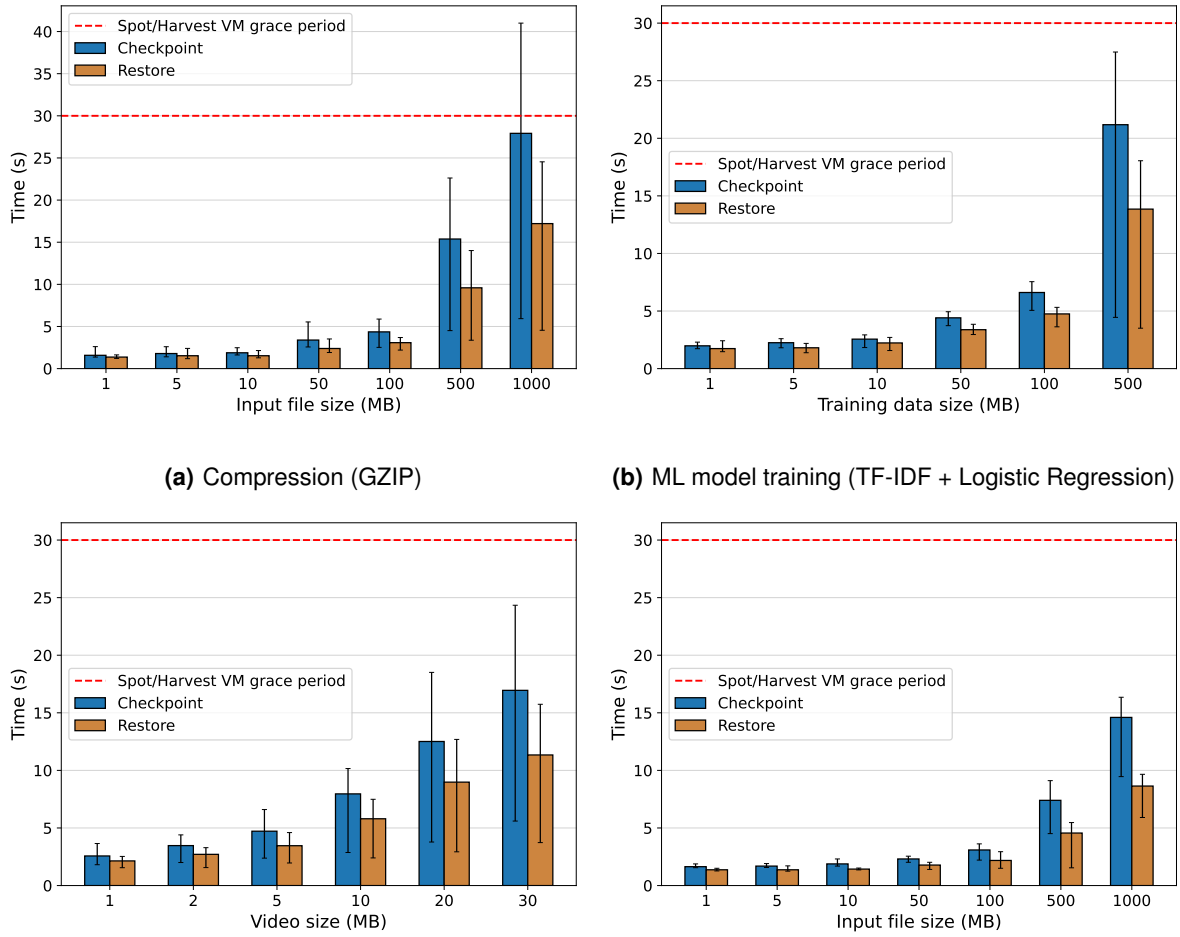
operations.

Machine learning model training (FunctionBench). This benchmark uses Python's `scikit-learn` package and reads `.csv` training sets from Amazon Fine Food Reviews [63], transforming each review into a TF-IDF vector. The outcome of the featurization process is then applied to a Logistic Regression classifier to build a model that predicts reviews' sentiment scores. Since Training ML models in Serverless environments are increasingly popular [34–36], it makes sense to include an application of that sort in our evaluation.

Video Processing (SeBS) This application receives a video file and converts it into 1-second GIF file with 10 fps using a static build of `FFmpeg`, a software that contains a suite of libraries useful for handling video, audio, and other multimedia files.

Network I/O (SeBS) This is 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, a feature that most other checkpointing mechanisms cannot guarantee, especially those implemented at an application-level scope.

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 5.3 shows the 5th, 50th and 95th percentile checkpoint/restore latency for each one of those applications and multiple input



(a) Compression (GZIP)

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

(c) Video Processing (GIF extraction)

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

Figure 5.3: Evaluation of R-Check on real applications from SeBS and FunctionBench benchmark suites. Each bar represents the median (P50), 5th percentile (P5) and 95th percentile (P95) of 50 executions.

object sizes. Median checkpoint times range from 1.2 seconds for small input sizes up to around 27.9 seconds for sizes that approach the memory limits of the containers, all of them still under a typical 30-second eviction grace period as required. On the other hand, median restore times go from 1 second to 17.2 seconds. On the upper end, we notice that some results vary slightly from the ones inferred from the previous experiment in Section 5.3: for example, Figure 5.3(a) shows that, in the 95th percentile, the compression benchmark takes around 42 seconds to checkpoint. This is most 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. While this is a limitation in our current approach, we refer to optimizations in the networking process in future work perspectives (see Section 6.1).

We also observe that the checkpoints produced by R-Check are, in practice, lower bounded in terms

	Compression			Machine Learning			Video Processing			Network I/O		
Input size (MB)	1	100	1000	1	50	500	1	10	30	1	100	1000
Checkpoint size (MB)												
Median	5.5	136.9	1285.0	63.0	285.0	1716.5	104.5	620.3	1079.2	35.3	126.7	965.7
P5	5.5	58.0	200.7	39.8	224	289.0	41.8	132.9	304.1	35.3	57.9	506.2
P95	5.5	193	1886.5	93.8	303.4	2022.7	141.1	817.6	1675.3	35.3	152.9	1066.4

Table 5.2: Checkpoint object sizes for different real applications and input sizes.

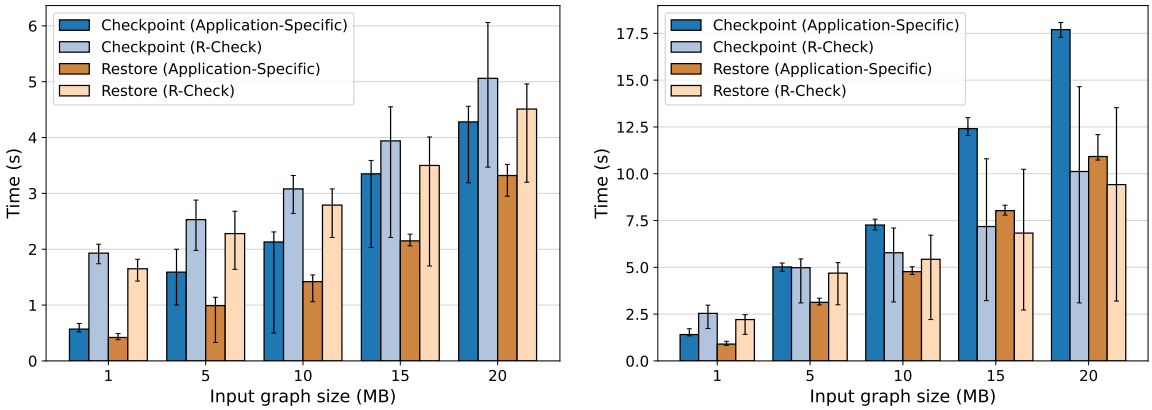
of latency and size. There is a checkpoint overhead of at least 1.2 seconds in the smallest checkpoints taken, with a corresponding checkpoint object size of 5.5 MB, as depicted in Section 5.4. This happens because the checkpointing algorithm needs to capture the bare minimum context to restore a function. In Python, that includes standard libraries imported for every function in every execution, the stack and the application code adding up to the minimum overhead obtained. It is relevant to note that this applies not only to our tested applications in Python but also to applications written in other languages that use standard libraries and packages.

Another observation we can extract from the Figures is that, as the input sizes grow, the disparity between the 5th and 95th percentiles also grows accordingly. It is an expected behaviour since we use random timeouts to decide when to checkpoint, and as functions grow in the input size, they become more prolonged and memory intensive. This 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.

5.5 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 based on CRIU. The intuition, in this case, is that a programmer- or compiler-written checkpoint and restore procedure is more efficient as it does not unnecessarily save and restore the entire address space of the process but only the relevant data structures and variables.

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 cannot take much advantage of such mechanisms since that would entail waiting for the end of the call, possibly missing the checkpointing window. This factor largely restricts the kind of workloads we may use, so choosing suitable functions was the biggest challenge we faced in this part of the evaluation. In light of these circumstances,



(a) Checkpoint/Restore times by input graph size (BFS)

(b) Checkpoint/Restore times by input graph size (PageRank)

Figure 5.4: Comparison of R-Check's checkpoint/restore times against an application-specific approach that saves only relevant variables. Each bar represents the median (P50), 5th percentile (P5) and 95th percentile (P95) of 50 executions.

we chose two Python benchmarks that present the most suitable characteristics for application-level benchmarking:

- A Breadth-First Search (BFS) algorithm which takes a graph as input and traverses it.
- An instance of PageRank, which is an algorithm that 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`[1], 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 for retrieval upon restoration. When restoring, the checkpoint object is retrieved and loaded into the function where the context is unwinded, 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. It is also important to emphasize that 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. This contrasts with R-Check where, apart from some limitations stated in Section 6.1, can checkpoint anytime during the execution.

The results obtained are depicted in Figure 5.4, specifically for the BFS function in Figure 5.4(a) and for the PageRank function in Figure 5.4(b). We see that, for small input sizes (1 MB and 5 MB

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

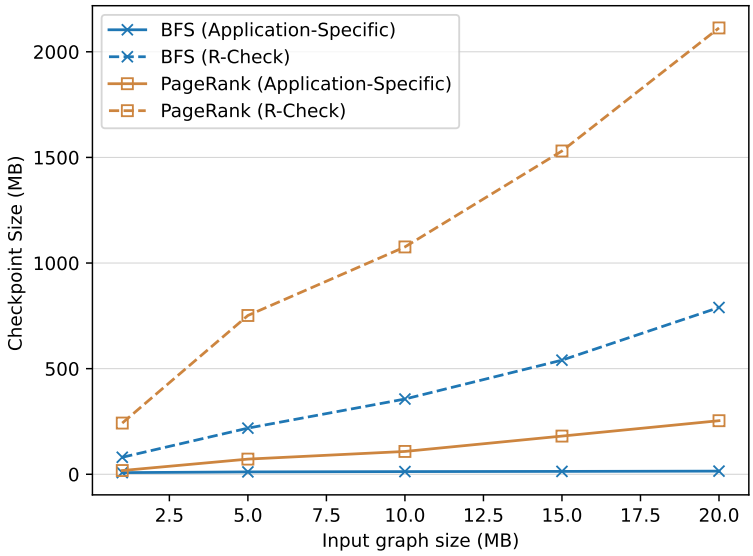


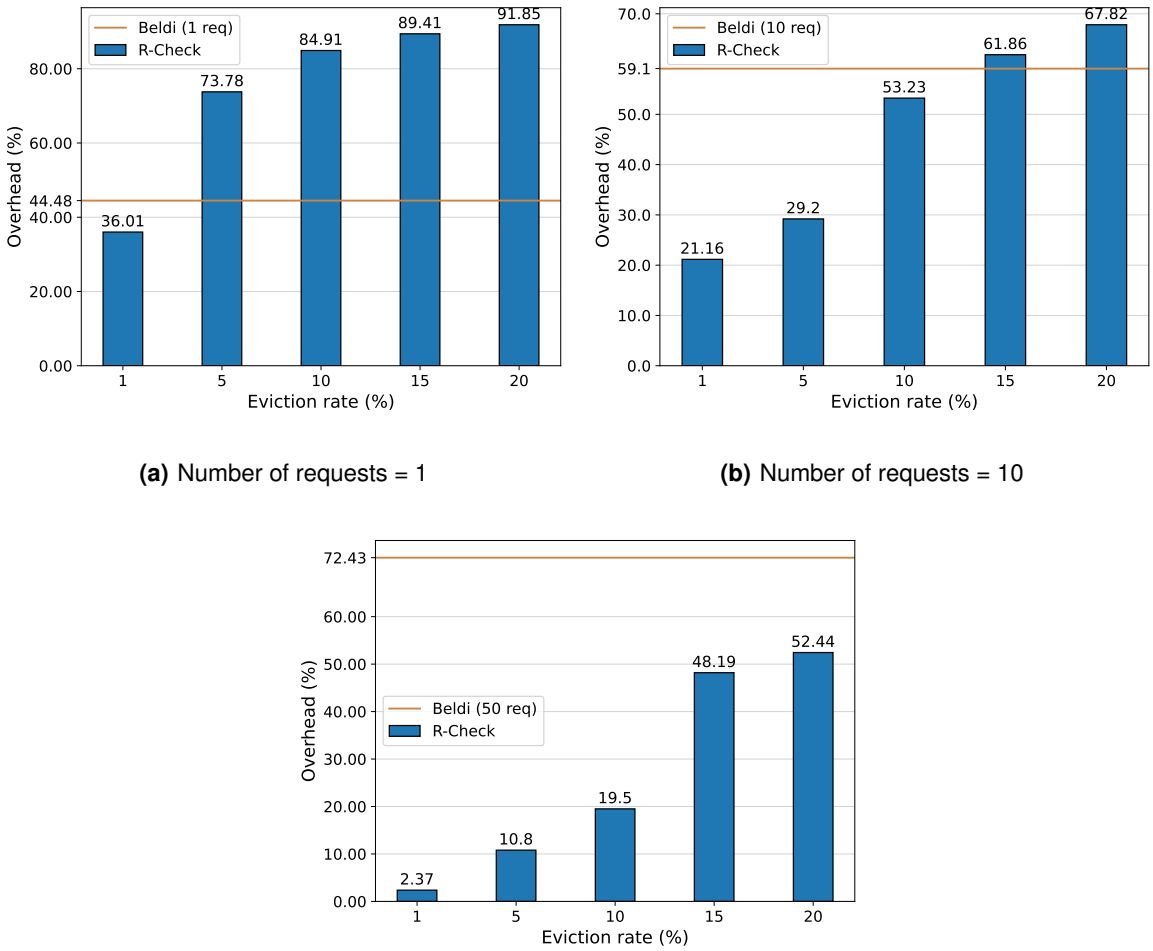
Figure 5.5: Comparison of R-Check's checkpoint object sizes against an application-specific approach that saves only relevant variables. Each line represents the median (P50) of 50 executions.

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. However, breaking down the checkpoint/restore numbers for the application-specific method sheds light on the reason behind this behaviour. Consider, for example, the 15 MB BFS experiment. According to Figure 5.5, 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$\times$ compression ratio). At a $\approx$ 50 MB/s upload rate to S3, 16.48 MB of checkpoint size translates into 0.33 seconds to upload the checkpoint object to S3. Considering a negligible cost for compression, it took $\approx$ 3 seconds ($\approx$ 90% of the total time) to extract the variables and serialize them with `pickle`, which 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 5.5, checkpoint sizes in R-Check grow more rapidly than for the application-specific method, as expected, given that the whole application address space is

checkpointed. However, we believe this does not represent a high cost for the cloud provider in terms of storage requirements because checkpoint objects are kept for a short time until the execution is restored and can be garbage collected immediately. Since storage services like S3 charge in a Memory-Duration granularity, the financial impact of each checkpoint is minimal.

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.



(a) Number of requests = 1

(b) Number of requests = 10

(c) Number of requests = 50

Figure 5.6: 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.

5.6 Comparison with the State-of-the-Art

We compared R-Check with the state-of-the-art for lifting the idempotence requirements in current serverless models, namely those solutions that employ preventive approaches for fault tolerance in Serverless: Beldi [14], a logging-based approach and Kappa [8], a checkpointing-based approach. Our key premise is that, by checkpointing 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 their 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.

5.6.1 Comparison with Beldi

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 relative overhead introduced by such calls and compared it with the overhead observed using R-Check with multiple eviction rates. We tested 3 different numbers of requests per function invocation: 1, 10, 50.

Figure 5.6 shows the overhead introduced by both approaches relative to the whole execution time. Beldi performs better than R-Check for functions that contain a small number of requests, in this case, 1 request, where R-Check's overhead is lower only at a 1% eviction rate. However, Beldi's performance degrades as the number of requests increases. At 10 requests per invocation, 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.

5.6.2 Comparison with Kappa

Kappa checkpoints functions by calling Kappa's library. In this evaluation, we faced some of the same challenges as we did in Section 5.5, 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,

Function	Duration (s)	Distribution (%)
BFS (1 MB)	2	95.6
BFS (5 MB)	12	3.5
BFS (10 MB)	21	0.2
BFS (15 MB)	54	0.32
BFS (20 MB)	89	0.33
BFS (25 MB)	112	0.02
BFS (30 MB)	143	0.03

Table 5.3: Distribution of function executions for different input sizes of the BFS function. This distribution is based on a public Azure Functions trace [1] distribution of function invocations given their execution duration.

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 (the same as in Section 5.5).

To evaluate both approaches in a more realistic scenario, we used a public Azure Functions dataset [1], that includes real traces of Serverless functions executions on a 24-hour period. These traces contain information on the memory allocated to each function, number of executions and duration of each execution. We grouped functions based on their execution duration to come up with an approximate distribution that we could use to model our experiment. Then, we ran a set of 1000 of our functions that would match that distribution. The distribution used is presented in Table 5.3

For 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 context of harvested resources (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 5.7 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. With these results, we can be confident that, for most evictable resources, R-Check performs better than Kappa while not adding any extra user programming burden.

5.7 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

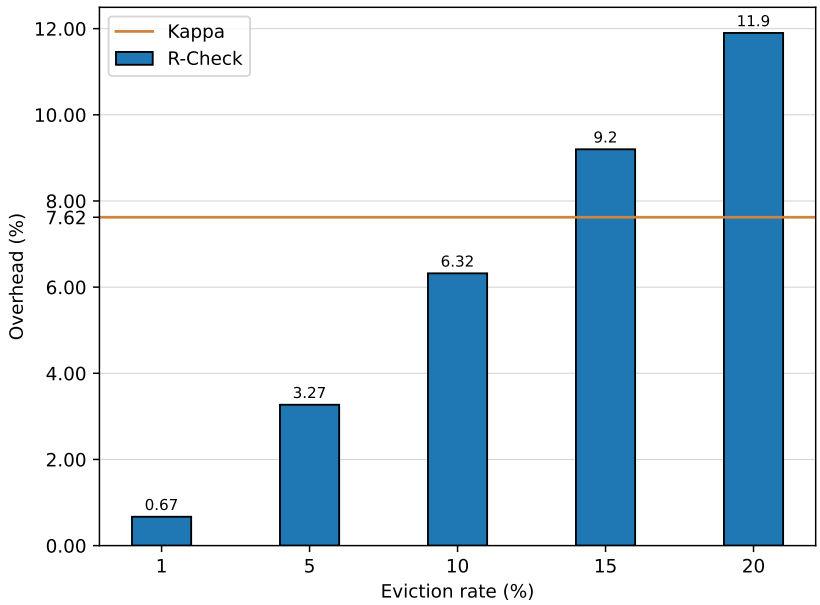


Figure 5.7: Comparison of relative R-Check's overhead against Kappa for different evictions rates on a set of 1000 BFS function executions.

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 5.4 (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 5.8 shows that the median checkpoint latency remains slightly over 2 seconds (with an average individual checkpoint time of 1.5 seconds) and median restore latency under 1.3 seconds (with an average individual restore time of 1.2 seconds) even for 16 concurrent functions. This is because, 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.

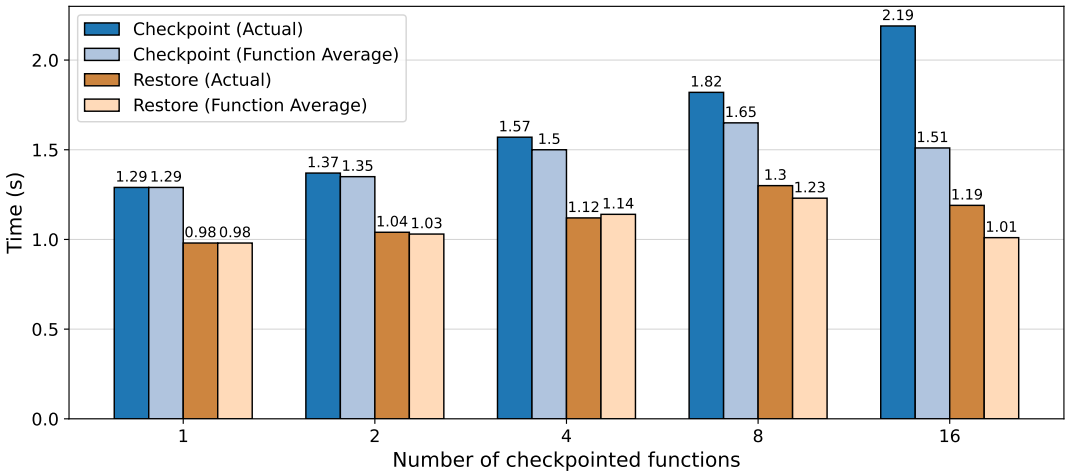


Figure 5.8: Evaluation of checkpoint/restore overhead when scaling concurrent checkpoints on a Compression (GZIP) benchmark with an input size = 50 MB. ”Actual” values represent the median latency (P50) over 50 measurements and ”Function Average” represent the average overhead of a function for each set of checkpointed functions.

6

Conclusion

Contents

Serverless computing is becoming a viable cloud paradigm for an increasing number of use cases. Thus, most prominent cloud computing vendors have released their Serverless platforms, and there is a tremendous amount of investment and attention around this space, both from industry and academia. Serverless initially targeted functions that thrive in the stateless and volatile nature of the paradigm. Contrariwise, the deployment of non-idempotent and long-running applications remains a challenging task. Since functions can fail through the execution, a re-execution might unnecessarily duplicate resources or cause 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 any given moment.

Towards the generalization of workloads, we envision that the future of serverless computing should include a fault-tolerant mechanism to deal with planned 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 ran experiments within an OpenWhisk environment, evaluated R-Check for various real applications and compared it to state-of-the-art systems in this space, namely Kappa [8] and Beldi [14]. These fault-tolerant approaches affect 100% of the executions, while R-Check only checkpoints when strictly necessary. In these experiments, we aimed attention at how checkpoint/restore time varies with function memory consumption and demonstrated the feasibility of our approach with checkpoints taking only up to 3 seconds for 100 MB of function memory used. Additionally, we achieved safe checkpoints within a 30-second termination period, even when exhausting the container's total allocated memory. This study also sets the ground for a new paradigm of solutions in the fault tolerance space in Serverless: while employing a reactive approach, a diminished set of invocations will be affected by checkpointing, introducing no runtime overhead to most executions.

6.1 System Limitations and Future Work

While the results are promising, and R-Check can support a wide range of applications and workloads, it has a few limitations and opportunities for improvement. Next, we discuss those limitations and possible directions to build on top of our model.

Unexpected crashes. As previously discussed in Section 4.3, R-Check can only checkpoint functions when failures are signaled in advance, 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 (with or without R-Check) 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 machine crashes.

Usage of real evictable VMs. Our experiments were conducted entirely on regular VMs and evictions were simulated so that we could gather tangible results in a reasonable time-frame. A logical next step is to deploy our model on top of actual evictable VMs and understand if the results obtained in our evaluation hold in an environment that resembles more closely that of our end objective.

Identification of critical checkpoints. 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 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 [11], it would be possible to characterize function invocations and predict which workloads would likely finish within the grace period and, consequently, do not need to be checkpointed. Naturally, this would only work for workloads seen before, while new workloads would assume a conservative approach and be checkpointed regardless.

Reduction in network latency. As seen in Chapter 5, uploading and downloading checkpoint objects is currently the major bottleneck of R-Check. However, some observed throughputs are much lower than the actual per-instance advertised bandwidth. Some alternative storage options and approaches should be more thoroughly tested to increase network throughput. One possible improvement is to leverage checkpoint image sharding to maximize horizontal upload/download throughput.

Bibliography

[1] “Microsoft azure traces.” [Online]. Available: https://github.com/Azure/AzurePublicDataset

[2] “AWS Lambda.” [Online]. Available: https://aws.amazon.com/lambda/

[3] “Azure Functions.” [Online]. Available: https://microsoft.com/en-us/services/functions/

[4] “Google Cloud Functions.” [Online]. Available: https://google.com/functions/

[5] “IBM Cloud Functions.” [Online]. Available: https://cloud.ibm.com/functions/

[6] “Apache OpenWhisk Open Source Serverless Cloud Platform.” [Online]. Available: https://openwhisk.apache.org/

[7] “OpenFaaS - Serverless Functions Made Simple.” [Online]. Available: https://www.openfaas.com

[8] “Kappa: A programming framework for serverless computing,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 328–343. [Online]. Available: https://doi.org/10.1145/3419111.3421277

[9] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, “Serverless Computing: Current Trends and Open Problems,” 2017.

[10] “AWS Lambda - How do I make my Lambda function idempotent?” 2022. [Online]. Available: https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/

[11] Y. Zhang, I. n. Goiri, G. I. Chaudhry, R. Fonseca, S. Elnikety, C. Delimitrou, and R. Bianchini, “Faster and Cheaper Serverless Computing on Harvested Resources,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 724–739. [Online]. Available: https://doi.org/10.1145/3477132.3483580

[12] B. Przybylski, M. Pawlik, P. Żuk, B. Łagosz, M. Malawski, and K. Rzadca, “Using Unused: Non-Invasive Dynamic FaaS Infrastructure with HPC-Whisk,” 2022.

[13] "AWS Lambda enables functions that can run up to 15 minutes." 2018. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/

[14] H. Zhang, A. Cardoza, P. B. Chen, S. Angel, and V. Liu, "Fault-tolerant and transactional stateful serverless workflows," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 1187–1204. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

[15] Z. Jia and E. Witchel, "Boki: Stateful Serverless Computing with Shared Logs," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 691–707. [Online]. Available: https://doi.org/10.1145/3477132.3483541

[16] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, and D. A. Patterson, "What Serverless Computing is and Should Become: The next Phase of Cloud Computing," *Commun. ACM*, vol. 64, no. 5, p. 76–84, apr 2021. [Online]. Available: https://doi.org/10.1145/3406011

[17] T. Wagner, "Debunking serverless myths." 2018. [Online]. Available: https://www.slideshare.net/TimWagner/Serverlessconf-2018-keynote-debunking-Serverless-myths

[18] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter, "A Survey on Reactive Programming," *ACM Comput. Surv.*, vol. 45, no. 4, aug 2013. [Online]. Available: https://doi.org/10.1145/2501654.2501666

[19] "Amazon S3 - Object storage built to retrieve any amount of data from anywhere." [Online]. Available: https://aws.amazon.com/s3/

[20] "Simplifying serverless best practices with Lambda Powertools." [Online]. Available: https://aws.amazon.com/blogs/opensource/simplifying-serverless-best-practices-with-lambda-powertools/

[21] I. Blue and G.-X. Team, "Overview of the IBM Blue Gene/P Project," *IBM J. Res. Dev.*, vol. 52, pp. 199–220, 2008.

[22] "AWS Spot Instances." [Online]. Available: https://aws.amazon.com/ec2/spot/

[23] P. Ambati, I. Goiri, F. Frujeri, A. Gun, K. Wang, B. Dolan, B. Corell, S. Pasupuleti, T. Moscibroda, S. Elnikety, M. Fontoura, and R. Bianchini, "Providing SLOs for Resource-Harvesting VMs in cloud platforms," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 735–751. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/ambati

[24] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, “Sprocket: A Serverless Video Processing Framework,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 263–274. [Online]. Available: https://doi.org/10.1145/3267809.3267815

[25] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, “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)*. Boston, MA: USENIX Association, Mar. 2017, pp. 363–376. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[26] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, “Serverless Linear Algebra,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 281–295. [Online]. Available: https://doi.org/10.1145/3419111.3421287

[27] S. Werner, J. Kuhlenkamp, M. Klems, J. Muller, and S. Tai, “Serverless Big Data Processing using Matrix Multiplication as Example,” 12 2018, pp. 358–365.

[28] A. Aytekin and M. Johansson, “Harnessing the Power of Serverless Runtimes for Large-Scale Optimization,” *ArXiv*, vol. abs/1901.03161, 2019.

[29] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, “Pocket: Elastic Ephemeral Storage for Serverless Analytics,” in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/klimovic

[30] Q. Pu, S. Venkataraman, and I. Stoica, “Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure,” in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/pu

[31] J. Sampé, G. Vernik, M. Sánchez-Artigas, and P. García-López, “Serverless Data Analytics in the IBM Cloud,” in *Proceedings of the 19th International Middleware Conference Industry*, ser. Middleware ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1–8. [Online]. Available: https://doi.org/10.1145/3284028.3284029

[32] Y. Kim and J. Lin, “Serverless Data Analytics with Flint,” 07 2018, pp. 451–455.

[33] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi, “Understanding Ephemeral Storage for Serverless Analytics,” in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC ’18. USA: USENIX Association, 2018, p. 789–794.

[34] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, “Cirrus: A Serverless Framework for End-to-End ML Workflows,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 13–24. [Online]. Available: https://doi.org/10.1145/3357223.3362711

[35] V. Isahagian, V. Muthusamy, and A. Slominski, “Serving Deep Learning Models in a Serverless Platform,” 04 2018, pp. 257–262.

[36] M. S. Kurz, *Distributed Double Machine Learning with a Serverless Architecture*. New York, NY, USA: Association for Computing Machinery, 2021, p. 27–33. [Online]. Available: https://doi.org/10.1145/3447545.3451181

[37] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, “From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers,” in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, Jul. 2019, pp. 475–488. [Online]. Available: http://www.usenix.org/conference/atc19/presentation/fouladi

[38] “Azure Durable Functions.” [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/

[39] S. Burckhardt, C. Gillum, D. Justo, K. Kallas, C. McMahon, and C. S. Meiklejohn, “Durable Functions: Semantics for Stateful Serverless,” *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485510

[40] S. Setty, C. Su, J. R. Lorch, L. Zhou, H. Chen, P. Patel, and J. Ren, “Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 501–516. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty

[41] V. Sreekanti, C. Wu, S. Chhatrapati, J. E. Gonzalez, J. M. Hellerstein, and J. M. Faleiro, “A Fault-Tolerance Shim for Serverless Computing,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387535

[42] P. Karhula, J. Janak, and H. Schulzrinne, “Checkpointing and Migration of IoT Edge Functions,” in *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, ser. EdgeSys ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 60–65. [Online]. Available: https://doi.org/10.1145/3301418.3313947

[43] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen, *Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting*. New York, NY, USA: Association for Computing Machinery, 2020, p. 467–481. [Online]. Available: https://doi.org/10.1145/3373376.3378512

[44] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, “Benchmarking, analysis, and optimization of serverless function snapshots,” *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr 2021. [Online]. Available: http://dx.doi.org/10.1145/3445814.3446714

[45] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo, “SEUSS: Skip Redundant Paths to Make Serverless Fast,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3392698

[46] J. Duell, “The design and implementation of berkeley lab’s linux checkpoint/restart,” 2005.

[47] “DMTCP - Distributed MultiThreaded CheckPointing.” [Online]. Available: https://dmtcp.sourceforge.io/index.html

[48] “Checkpoint/Restore in Userspace.” 2022. [Online]. Available: https://criu.org/Main_Page

[49] Y. Tan, D. Liu, N. Li, and A. Levy, “How Low Can You Go? Practical cold-start performance limits in FaaS,” 2021.

[50] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, “Cloudburst: Stateful Functions-as-a-Service,” *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, sep 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407836

[51] F. Romero, G. I. Chaudhry, I. n. Goiri, P. Gopa, P. Batum, N. J. Yadwadkar, R. Fonseca, C. Kozyrakis, and R. Bianchini, “Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications,” in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 122–137. [Online]. Available: https://doi.org/10.1145/3472883.3486974

[52] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, and A. Tchana, “OFC: An Opportunistic

Caching System for FaaS Platforms," in *Proceedings of the Sixteenth European Conference on Computer Systems*, ser. EuroSys '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 228–244. [Online]. Available: https://doi.org/10.1145/3447786.3456239

[53] "AWS Spot - Spot Instance advisor." [Online]. Available: https://aws.amazon.com/ec2/spot/instance-advisor/

[54] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, December 2001. [Online]. Available: https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[55] G. M. Lohman and J. A. Muckstadt, "Optimal Policy for Batch Operations: Backup, Checkpointing, Reorganization, and Updating," in *Proceedings of the 1977 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '77, 1977, p. 157.

[56] "CRIU Image streamer." [Online]. Available: https://github.com/checkpoint-restore/criu-image-streamer

[57] "Apache Kafka: A distributed streaming platform." [Online]. Available: https://kafka.apache.org

[58] "Apache CouchDB: A distributed streaming platform." [Online]. Available: https://couchdb.apache.org

[59] "Redis - The open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker." [Online]. Available: https://redis.io

[60] "LZ4 - Extremely fast compression." [Online]. Available: https://github.com/lz4/lz4

[61] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "SeBS: A Serverless Benchmark Suite for Function-as-a-Service Computing," 2021.

[62] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 502–504.

[63] "Amazon Fine Food Reviews." [Online]. Available: https://snap.stanford.edu/data/web-FineFoods.html