Evicting for the greater good: The Case for Reactive Checkpointing in serverless computing

Rafael Alexandre, Rodrigo Bruno, João Barreto, Rodrigo Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa

Abstract

Evictable cloud resources give providers extra flexibility in managing their infrastructure and help reduce resource idleness. Due to its user-transparent and ephemeral nature, serverless computing appears to be a logical fit to make use of this type of resources. However, in its current form, this flexibility comes at the cost of semantics, namely by imposing execution time limits and only guaranteeing *at least once* semantics. As such, developers must either implement idempotent code or resort to proactive checkpointing or logging approaches, which pessimistically introduce runtime overheads in every invocation. In this paper, we make the case for an alternative approach to handling function interruptions, based on a reactive checkpoint-based mechanism. Following this approach, we present R-Check, a system designed to efficiently migrate function instances running on top of evictable cloud resources when interruptions are imminent. Our evaluation shows that significant resource savings can be achieved using such resources, and that function migration is possible with minimal runtime overhead.

CCS Concepts: • **Software and its engineering** → **Checkpoint / restart**; • **Computer systems organization** → **Cloud computing**.

Keywords: Serverless Computing, Checkpoint/Restore, Fault tolerance

ACM Reference Format:
Rafael Alexandre, Rodrigo Bruno, João Barreto, Rodrigo Rodrigues. 2023. Evicting for the greater good: The Case for Reactive Checkpointing in serverless computing. In *4th Workshop on Resource Disaggregation and Serverless (WORDS '23), October 23, 2023, Koblenz, Germany.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3605181.3626289

1 Introduction

To achieve higher resource density, cloud providers now offer the opportunity to take advantage of cheaper ephemeral resources that could be evicted at any moment if a high-priority job needs to use the same resources. This momentarily surplus of resources is made available through services such as Spot [6], and Burstable [4, 22] instances. Harvest VMs [35] further build up on this idea of using harvestable resources by expanding or contracting individual Virtual Machine (VM) instances at runtime (e.g., to add/remove cores).

While the idea of opportunistically taking advantage of idle resources increases hardware efficiency, it significantly hampers the programming model as applications now need to be able to adapt to use more (or less) resources, or completely terminate in the case of an eviction. In the face of this adversity, serverless has been proposed as a natural model to take advantage of such resources [8].

Serverless computing is becoming an increasingly popular cloud programming paradigm, especially in the form of Function-as-a-Service (FaaS). It is now supported by major cloud providers (e.g., Amazon's AWS Lambda [5], Microsoft's Azure Functions [23], Google's [18] and IBM's Cloud Functions [19]) as well as through open-source projects (such as Apache's OpenWhisk [9] and OpenFaas [24]).

FaaS offers an intuitive, event-based interface for developing cloud applications. The vision behind this paradigm is to completely hide the management of VM, runtimes, and resources from the programmer so that cloud users can focus on the application logic. Each of these functions can then be invoked on demand and independently from one another. Function instances have been shown to start extremely fast compared to traditional VMs [1], allowing serverless applications to quickly scale to many compute units without provisioning a long-running cluster. Cloud providers are responsible for handling VM/container allocation, deploying the user code, and scaling the resources up and down. All of this infrastructure management is transparent to users.

Serverless offerings encourage users to write stateless and short-running functions [11]. In particular, services like AWS Lambda set a maximum execution time on the order of minutes [2] and, in case it is necessary to halt the function execution (namely for resource scheduling purposes), the default action is to redeploy the function in a different node and simply retry it. For this reason, the functional programming model of serverless computing is a good fit to exploit harvested resources and it has even been shown to result in a low overall function eviction rate [8].

However, as a consequence of its growing popularity, more and more applications are being ported to serverless and, as a result, the idea of using serverless to harness harvested resources becomes challenged. First, stateful functions [12, 28, 29], which store and access external state, may lead to inconsistencies and/or undesired behaviors when evicted at an arbitrary point (for example, duplicate items appearing in a shopping cart). To prevent these pathologies, providers of FaaS services guide developers to write idempotent code [7] when devising stateful functions. Second, as the number of use cases continues to grow, we expect an increase in the average function invocation lifetime, which increases the prevalence of the first problem.

In light of these challenges, recent work proposed methods for saving computation steps, namely by recording the computation through logging or periodic checkpointing [20, 33, 34]. However, these approaches introduce considerable performance overheads in every function invocation, even if only a fraction needs to handle an eviction. Furthermore, these overheads tend to increase as the range of functions deployed becomes wider, namely when FaaS services start supporting longer-running functions [2].

This paper takes the position that serverless computing can and should evolve to safely deploy non-idempotent stateful functions on evictable resources while also inflicting minimal runtime overheads. Our vision is based on the insight that evictions are controlled events and not crash faults. Thus, by co-designing the function runtime together with provider scheduling and scaling decisions, we advocate for a reactive and application-transparent checkpointing model that automatically migrates active invocations upon an eviction.

To realize this idea, we present a proof of concept system called R-Check, a reactive and fully transparent checkpoint-based framework for serverless functions. R-Check leverages the fact that cloud providers grant a termination grace period before they relocate a resource, e.g., two minutes in the case of EC2 spot instances [26]. This way, R-Check attempts to make use of that period to efficiently snapshot the application state and resume from it afterwards. This allows R-Check's functions to be stateful and still be interrupted at any point during their execution.

We start this paper by studying the resource usage of serverless clusters with and without R-Check, estimating resource savings up to 7.24% when using evictable resources compared to a function compaction policy similar to the one used by commercial FaaS platforms. Then, we present the design, implementation, and preliminary evaluation of our work-in-progress R-Check prototype, showing that the system can successfully migrate function instances, achieving minimal overheads for the vast majority of function invocations (less than 1 second and 0.5 seconds for checkpoint and restore, respectively).

2 Benefits of assertive function compaction

Current FaaS offerings, e.g., AWS Lambda and Azure Functions, schedule function invocations by packing them together with the goal of maximizing memory utilization per physical node [31]. Despite this bin-packing effort, two main factors contribute to per-node sub-optimal memory utilization: i) highly elastic function invocation bursts, coupled with ii) unpredictable function invocation times. As a result, resource utilization across large fleets of nodes becomes fragmented resulting in low resource utilization.

We argue that it is possible to further consolidate function invocations by migrating running function instances to other physical nodes to reduce the amount of idle resources. In particular, we propose that nodes whose load falls below a certain threshold can be proactively evicted, migrating the functions they accommodate to other available nodes so that the provider can reduce the overall resource and energy footprint.

To test our vision, we quantify the influence of evicting nodes that fall below a certain minimum load. We define load of an active node n at an instant t as the sum of the memory of all f function invocations running on node n at instant t:

$$load(n, t) = \sum_{f \in F} memory_f \tag{1}$$

To do so, we simulate a large FaaS cluster that we use to replay invocations from the Azure Functions public trace [27]. The selected trace spans a 24-hour period (day 5) containing 178 million function invocations. The simulation takes into account the duration and average memory consumption and schedules functions on a cluster of 32GB nodes following the compaction-oriented scheduling strategy described in previous work [31][1]. In the simulation, if a node's load falls below the minimum load to evict, active function invocations on that node are migrated and the node is removed from the available pool of nodes. To measure resource usage, we define the following metric:

$$instance\text{-}seconds = \sum_{n \in N} lifetime_n \tag{2}$$

as the sum of the *lifetime*, in seconds, of all the n active nodes from the cluster for the duration of the experiment. A lower value of this metric means that there were more opportunities to either power down nodes or allocate them to other purposes, thus representing a lower resource utilization.

Figure 1 shows both the instance-seconds metric and the eviction rate, both as a function of the minimum load to evict threshold. We can see that there is a steep decline in resource usage up to around 20% of minimum load, achieving ≈7.24%

[1]We simulate scheduling according to memory consumption because the trace does not contain other load metrics, namely CPU utilization.

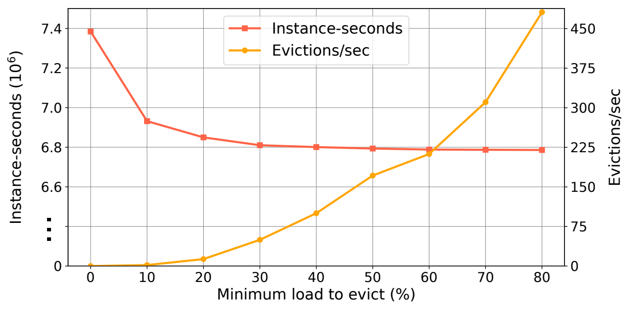

Figure 1. Instance-seconds needed to run a 24-hour FaaS cluster (left y-axis), for different minimum thresholds to evict an instance, and number of function evictions per second for those settings (right y-axis). The function invocation throughput in this cluster is, on average, 2060 invocations per second.

resource savings compared to the baseline scenario (that represents current FaaS scheduling policies with no compaction through migration). From 30% onwards, the marginal savings in terms of instance-seconds become less noticeable as the load to trigger an eviction increases, and therefore increasing this threshold beyond 30% is likely not going to be worth the benefits. Such remark is corroborated by the number of evictions per second (right-hand axis of Figure 1) where evictions are fairly limited for thresholds up to 30% (less than ≈50 evictions per second). Thus, by setting a minimum load to evict of around 20%, there is the potential to achieve a considerable reduction in resource usage while affecting a small number of function executions.

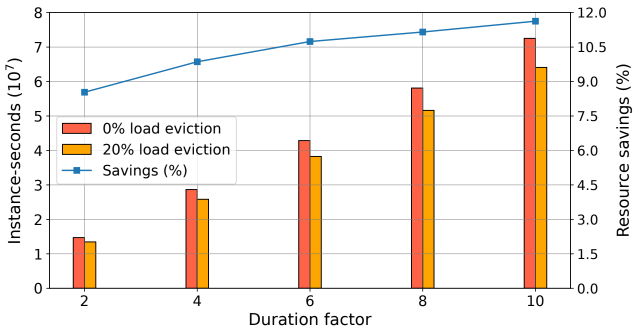

Figure 2. Instance-seconds needed to run a 24-hour Azure Functions trace on 0% and 20% load eviction policies when increasing the average function invocation time by a constant duration factor (left y-axis) and savings obtained for each factor (right y-axis).

The increasing attention that serverless has been receiving also makes it reasonable to envision a future where more diverse and general-purpose workloads are deployed on such platforms. This implies that functions will tend to be longer-running. In fact, the observation that FaaS platforms have increased their execution time limits over recent years [2] is a clear hint in this direction. Hence, we also measure resource savings in this hypothetical scenario where functions have increased function invocation times. We simulate longer function execution times by multiplying all original function execution times from the trace by a constant factor. We compare the baseline of a 0% to a 20% minimum load to evict policy and display the results in Figure 2. With the increased duration, we observe that the number of instance-seconds for the 20% load scenario grows slower than in the baseline 0% load scenario, achieving savings of up to ≈11.6% with a duration factor of 10. This means that, in the future, as the average function duration increases, resource savings for systems that follow an aggressive function compaction policy (as proposed in this work) will likely increase. We also tested increasing the memory footprint of each function by multiplying it by a constant factor but we observed no relevant changes in the instance-seconds metric when compared to the original setting.

Motivated by the aforementioned results, we argue that it is possible to achieve significant cloud resource savings by terminating nodes when they are underutilized. However, at the same time, this aggressive scheduling policy brings up new challenges as it forces developers to either restrict application semantics or introduce large runtime overheads to deal with evictions. In the rest of this paper, we present an approach that allows serverless applications to be deployed on evictable resources and keep the same semantics, while adding no runtime overheads to the normal-case executions where there are no evictions. But before presenting this proposal, we first put in perspective how the notion of controlled evictions affects the semantics of cloud services.

3 Serverless fault tolerance: the circle is now complete

To understand the advantages and drawbacks of various cloud solutions in the different metrics relevant to this work, namely fault tolerance, expressiveness, and ability to migrate instances for better resource utilization, we analyze systematically how the offer of cloud services evolved throughout the years along these axes.

The first cloud service to be launched by Amazon was EC2, which was the first example of Infrastructure as a Service (IaaS). In this class of services, the customer can launch VM instances, which run the entire software stack that the customer desires. While this is optimal in terms of expressiveness, VMs expose direct access to the infrastructure, making it difficult for providers to automatically migrate running VMs transparently to users. For example, VMs have public IPs that might become temporarily unreachable.

FaaS enables a more proactive approach to resource management, as we previously explained, but does so at the expense of semantics: to allow the provider to perform fine-grained management of its resources, current FaaS offerings bound the function execution time by a limit that is fixed for each provider, after which the function is forcibly terminated. Furthermore, there is an expectation that the provider may terminate the function execution at any point before that limit. This way, invocations can be evicted if resources become necessary to host a higher-priority tenant/service, or if more convenient resources become available elsewhere.

However, this constrains not only the duration but also the semantics of cloud functions, namely by requiring these to be idempotent so that they can be seamlessly restarted upon eviction. This constraint can be an entry barrier for developers of serverless code, as it precludes, for example, any interaction with external resources.

To address this problem, a few recent research papers proposed the use of either checkpointing (Kappa [34]) or logging and replay (Beldi [33] and Boki [20]) on top of serverless systems. The idea is that, during every function execution, the execution environment either takes periodic checkpoints of the state of the runtime or logs every single output that was already issued by the function. Then, upon halting the execution of a function and restarting it elsewhere, either the checkpoint allows for resuming the function from a point near where it was stopped, or the log allows for replaying the execution and suppressing duplicate outputs. However, the problem with both techniques is that they impose a runtime overhead that was not present before, which is paid for by every function execution, even though only a fraction of these are expected to be restarted.

In R-Check, our approach is that functions should not be restarted, but instead migrated. More specifically, our proposal is grounded on the following insight: *evictions are controlled faults*, in contrast to unexpected crashes (e.g., faults caused by a sudden power outage), which are unplanned, and thus more difficult to recover from. As a result, the requirement of idempotency can be lifted by leveraging the eviction notice that is present in existing evictable resource-based cloud services (namely a 30-second up to 5-minute grace period in spot instances [10, 15, 26]) to conduct a reactive checkpoint of the cloud function, thus avoiding any incorrect behavior or loss of application state upon migration.

Our insight is further backed by the observation that proposals that resort to logging or checkpointing offer protection against a class of faults that today's cloud services do not address, which are unexpected crash faults. In particular, both IaaS and FaaS do not tolerate these faults, and therefore the developers that use these cloud services have to resort to some application-level mechanism in case there is the need to handle sudden crashes. (In practice, though, we argue that most applications do not use such extra mechanisms, probably due to a combination of faults being rare and the tolerance to exposing some errors to users, who have the choice of retrying their interactions with the service.) Logging and checkpointing, in turn, go to the extent of guaranteeing that crashes are tolerated automatically, using the same mechanisms that are used for function restarts. In our view, removing the semantic limitation of functions having to be stateless and idempotent does not require us to tolerate more faults than what FaaS and IaaS handle today, and, by equating the level of fault tolerance to Iaas/FaaS, we are also able to avoid paying expensive runtime overheads for the fault handling mechanisms.

	Full expressiveness[1]	Crash faults	No runtime overhead	Fine-grained resource mgt.
IaaS	✓	×	✓	×
FaaS	×	×	✓	✓
Kappa [34]	✓	✓	×	✓
Beldi [33]	✓	✓	×	✓
R-Check	✓	×	✓	✓

[1] Namely support for non-idempotent code.

Table 1. Relevant characteristics of different cloud services and research proposals.

To demonstrate the feasibility of this approach, we assembled a proof of concept system called R-Check, which we briefly describe in the rest of this paper. To put the relevance of R-Check in perspective, Table 1 presents how different cloud platforms and systems handle different types of faults and the additional requirements they impose on the application code, following our previous discussion.

In summary, R-Check shows that there is a middle ground between today's FaaS (which does not support general code) and systems that support both general code and handle crash faults [33, 34], by proposing a fault model that, like IaaS, supports planned evictions but not crash faults. The resulting system enables users to choose a sensible balance between performance overhead, expressiveness, and handling faults.

4 Reactive checkpoint & restore

Checkpoint/restore has been explored extensively in serverless computing to reduce cold start times [14, 17, 30]. In this work, we look at it not to reduce initialization time, but to migrate running function invocations upon an eviction.

One way to implement checkpoing/restore is to request users to provide their own checkpoint/restore logic. Although this empowers the user to make the eviction handling as optimized as possible, this poses a significant extra programming effort. Instead, we propose a completely **transparent** and **function-agnostic** checkpoint/restore approach where the provider triggers its own snapshot mechanism at system-level with no extra burden for the developer. In the case of R-Check, this is achieved by the use of Checkpoint Restore in Userspace, CRIU [16].

In R-Check, when an instance is about to be terminated due to the cloud provider's aggressive resource management, a request is sent to the function instance to save the current context of execution and store it in a remote storage service or platform. Once checkpointing finishes, a new function instance on another available node triggers the restoring logic code that resumes the previously interrupted function.

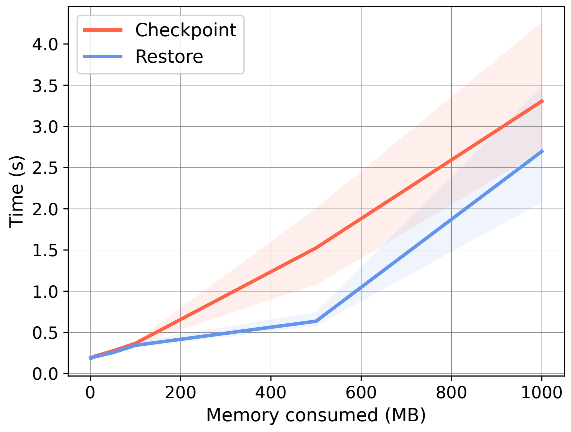

Figure 3. Checkpoint and restore times as a function of the memory consumed in the execution. Shaded regions represent the 90% confidence interval of measured times.

We now analyze the potential overhead that is introduced by using the transparent checkpoint/restore technique proposed above. In this experiment, we run a Python microbenchmark function that (1) allocates a specific amount of memory during its execution and (2) sleeps for a given period of time. We use CRIU as our snapshotting tool and measure the cost of checkpointing and restoring function instances locally (leaving room for different remote storage solutions to be explored in future work). All experiments were run 100 times on an AWS EC2 t3a.large machine (2 virtual CPUs, 4 GB RAM).

The results are presented in Figure 3. We can see that function checkpoint and restore times are kept below 1 second and 0.5 seconds, respectively, when up to 300 MB of memory are consumed. From the Azure trace used in Section 2, we learn that 93% of functions allocate between 100 MB and 300 MB of memory during execution. Taking this estimation as a reference for how much memory functions may consume, we anticipate that the overhead introduced by R-Check, as reported above, is manageable for the majority of function instances.

5 Discussion

While R-Check facilitates the deployment of serverless on evictable resources, it also opens a number of challenges, opportunities, and open questions.

Doesn't FaaS already have a fault handling mechanism? While current FaaS platforms tolerate faults by design, the mechanism used to recover from such faults – retrying the function invocation – may cause semantic problems if a function is not idempotent. For example, imagine a function that logs user actions in a database. If there is a fault after updating the database but before the invocation finishes, the platform will retry the invocation leading to a duplicate database record. Avoiding duplicate records even in this trivial example is complex, as it requires developers to carefully check if the record has been added before. Note that simply disabling the auto-retry functionality would prevent inconsistencies (and duplicate records in our example), but this would also create additional tension by failing requests when deploying functions atop environments with ephemeral resources such as spot instances. Therefore, instead of relying on developers to guarantee correctness, R-Check proposes to migrate function executions instead of retrying them.

Is IaaS obsolete? We envision that serverless will continue growing, and, as it does, so will need to accommodate a significantly larger set of use cases. In particular, this may include cases where applications have an unpredictable load that results from bursty requests and unpredictable computation times. R-Check further widens the scope of FaaS to be able to handle long-running functions, but, to this day, IaaS is still the most cost-effective deployment type to host long-running computations with predictable loads.

What if the migration fails? Checkpoint/restore has received significant attention in recent years and is now even used in production serverless platforms (e.g., AWS Lambda uses checkpoint/restore to speed up instance startup [3]). However, a migration could still fail due to overrunning the termination period, or because of an unsupported resource that cannot be checkpointed. In such an event, the platform should not retry the invocation as it may break the intended semantics (as described above), but instead, it should provide the developer with a failure log that allows him/her to resolve any consistency issues. Note that this is similar to IaaS, where service crashes may require developer attention to analyze the logs and take action. In the case of overrunning the function timeout, we envision that providers could extend this period even if charging at a higher price.

Can R-Check scale to large function instances? In this paper, we present a preliminary evaluation of what checkpoint/restore looks like in R-Check. We plan to extend the evaluation of R-Check's snapshotting capabilities in future work. We also propose a number of avenues to further optimize snapshotting:

I) Different snapshot communication mechanisms: One could possibly have an in-memory cache holding snapshots waiting to be pulled for a restore. A combination of different storage types could also be used depending on the size and available bandwidth as previously proposed [21, 25]. Another alternative would be to take advantage of direct communication [32] between the evicted node and the node that will restore the snapshot;

II) Snapshot-aware language runtimes: FaaS platforms offer a number of language runtimes capable of running the most popular languages (e.g., Python, Java, Javascript). These runtimes pre-allocate large chunks of memory, commonly referred to as the heap, which is then used for object allocations at runtime. However, since the heap memory is allocated (and committed) by the runtime, snapshotting tools cannot distinguish between used and unused heap memory. One possible research direction is to adapt runtimes to become snapshot-aware, for example, to export the memory pages that contain unused heap memory which could, therefore, be discarded during snapshot generation [13].

III) Incremental checkpointing: The snapshot size can be reduced by creating an initial snapshot and subsequently only tracking the memory that is modified. Such incremental snapshots could be created after launching the runtime, or after loading the function code. Then, upon an eviction, an incremental snapshot would only include modified memory pages.

6 Conclusion

In this work, we advocate for the placement of serverless functions on top of evictable resources for better cloud resource utilization. To overcome function failures and relax idempotency requirements of serverless platforms, we propose R-Check, a system that reactively checkpoints and restores functions when evictions occur. Our approach adds marginal overheads to the overall execution time and requires no effort from application developers.

Acknowledgements

We thank the anonymous reviewers for the feedback. This work was supported by the Fundação para a Ciência e Tecnologia through projects UIDB/50021/2020, PTDC/CCI-INF/6762/2020 (MS3) and UID/CEC/50021/2019 (INESC-ID).

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 419–434.

[2] Amazon Web Services 2018. AWS Lambda enables functions that can run up to 15 minutes. https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/

[3] Amazon Web Services 2022. Accelerate Your Lambda Functions with Lambda SnapStart. https://aws.amazon.com/blogs/aws/new-accelerate-your-lambda-functions-with-lambda-snapstart/

[4] Amazon Web Services 2023. AWS Burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html

[5] Amazon Web Services 2023. AWS Lambda. https://aws.amazon.com/lambda/

[6] Amazon Web Services 2023. AWS Spot Instancess. https://aws.amazon.com/ec2/spot/

[7] Amazon Web Services 2023. How do I make my Lambda function idempotent? https://aws.amazon.com/premiumsupport/knowledge-center/lambda-function-idempotent/

[8] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. 2020. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 735–751.

[9] Apache OpenWhisk 2023. Open Source Serverless Cloud Platform. https://openwhisk.apache.org/

[10] Microsoft Azure. 2023. Use Azure Spot Virtual Machines. https://learn.microsoft.com/en-us/azure/architecture/guide/spot/spot-eviction.

[11] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. Serverless Computing: Current Trends and Open Problems.

[12] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. 2022. Stateful Serverless Computing with Crucial. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 39 (mar 2022), 38 pages.

[13] Rodrigo Bruno and Paulo Ferreira. 2016. ALMA: GC-Assisted JVM Live Migration for Java Server Applications. In *Proceedings of the 17th International Middleware Conference* (Trento, Italy) *(Middleware '16)*. Association for Computing Machinery, New York, NY, USA, Article 5, 14 pages.

[14] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. 2020. SEUSS: Skip Redundant Paths to Make Serverless Fast. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 32, 15 pages.

[15] Alibaba Cloud. 2023. Receive a preemptible instance interruption event. https://www.alibabacloud.com/help/en/ecs/use-cases/receive-a-preemptible-instance-interruption-event.

[16] CRIU 2023. Checkpoint/Restore in Userspace. https://criu.org/Main_Page

[17] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. *Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting*. Association for Computing Machinery, New York, NY, USA, 467–481.

[18] Google Cloud 2023. Google Cloud Functions. https://google.com/functions/

[19] IBM Cloud 2023. IBM Cloud Functions. https://cloud.ibm.com/functions/

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

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

[22] Microsoft Azure 2023. Azure Burstable VMs. https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable

[23] Microsoft Azure 2023. Azure Functions. https://microsoft.com/en-us/services/functions/

[24] OpenFaas 2023. OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com

[25] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 193–206.

[26] Amazon Web Services. 2023. Spot Instance interruption notices. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html.

[27] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. *CoRR* abs/2003.03423 (2020).

[28] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 419–433.

[29] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst. *Proceedings of the VLDB Endowment* 13, 12 (Aug 2020), 2438–2452.

[30] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. 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).

[31] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. 2018. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 133–146.

[32] Michal Wawrzoniak, Ingo Müller, Rodrigo Bruno, and Gustavo Alonso. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *CIDR*.

[33] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204.

[34] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343.

[35] Yanqi Zhang, Íñigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. 2021. Faster and Cheaper Serverless Computing on Harvested Resources. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 724–739.