Memoryless: A Two-phase Methodology for Setting Memory Requirements on Serverless Applications

Rodrigo da Rosa Righi[a], Gabriel Borges, Cristiano André da Costa[b]
and Vinicius Facco Rodrigues[c]
Applied Computing Graduate Program, Universidade do Vale do Rio dos Sinos, São Leopoldo, Brazil

Keywords: Serverless, Abstraction, Performance, Memory, FaaS, Microbenchmark.

Abstract: Serverless computing, also known as Function as a Service, is a new paradigm that aims to separate the user of the platform from details about any infrastructure deployment. The problem lies in the fact that all the current Serverless platforms require the user to specify at least the needed memory usage for their Serverless offerings. Here we have a paradox since the users must be involved in technical issues to run their applications efficiently, both in terms of execution time and financial costs. To the best of our knowledge, the state-of-the-art lacks on providing studies regarding the best memory size for a particular application setting. In this context, this work presents Memoryless, a computational methodology that is in charge of removing the completion of memory limits by the user when launching Serverless demands. To accomplish this, we introduce in the literature a two-pass algorithm composed of a microbenchmark where users inform simple application parameters (first pass) and receive from the hypervisor the memory required to run their demands. In addition to user abstraction, financial cost also drives our research, since commonly this metric is directly proportional to the selected memory size. We implemented Memoryless using NodeJS, Kubeless, and Kubernetes. The result confirms that the proposed methodology is capable of lowering the memory needs to run an application while maintaining expected execution times. This benefits both cloud administrators (who can run more Serverless demands for different users in parallel) and cloud users, who will pay less on using the cloud, so exploring better the pay-as-you-go policy.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Every major computation revolution surpasses the last one by adding abstraction layers and novel functionalities for client applications. Client-server topped the mainframe usage, while cloud computing topped the client-server paradigm (Fox et al., 2009; Kanso and Youssef, 2017; Nguyen et al., 2019). In particular, cloud differs from client-server applications because it provides the notion of resource elasticity. The ideas consist of on-the-fly adding or removing, or yet reconfiguring, resources in such a way their number best fits a particular moment of the application execution. In this way, threshold-based cloud elasticity with rules and action statements is mainstream when thinking about the malleability of resources (Nasr, 2019; Villano, 2020). Using either a graphical interface, a command-line approach, or an API (Application Program Interface), users must handle lower and upper load thresholds (frequently linked with the CPU usage metric). Besides, rules and actions must also be analyzed, which is not trivial for non-cloud experts (Kim and Lee, 2019).

Today, we are living in the next revolution of resource management named Serverless computing (Jonas et al., 2019; Kanso and Youssef, 2017; Kim and Lee, 2019; Nguyen et al., 2019). Serverless computing is a new architecture paradigm, in which the user only needs to directly upload the code into a hypervisor and specify how much memory is required to run that piece of code. The system will allocate computational resources (for example, CPU) in proportion to the main memory size (Kanso and Youssef, 2017; Nguyen et al., 2019; Winzinger and Wirtz, 2019). The larger the size, the higher the CPU allocation. Resource usage is measured and billed in small increments (for example, 100ms) and users pay only for the time and resources used when their functions are running. The submitted code is known in

[a] https://orcid.org/0000-0001-5080-7660
[b] https://orcid.org/0000-0003-3859-6199
[c] https://orcid.org/0000-0001-6129-0548

Righi, R., Borges, G., André da Costa, C. and Rodrigues, V.
Memoryless: A Two-phase Methodology for Setting Memory Requirements on Serverless Applications.
DOI: 10.5220/0010707500003058
In *Proceedings of the 17th International Conference on Web Information Systems and Technologies (WEBIST 2021)*, pages 84-94
ISBN: 978-989-758-536-4; ISSN: 2184-3252
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

the industry as Function as a Service (FaaS), to allude to the fact that only a procedure is being uploaded to the platform, instead of any compiled or packaged objects (Amazon, 2019). Serverless functions can be used to run code, build mobile and web applications, manage containers and handle other cloud computing tasks (Nguyen et al., 2019). All the infrastructure needed to run the code is hidden from the user. Also, the user pays only for code usage (procedure calls), *i.e.*, for the period comprised between starting the code and the final execution of the last piece of code (Kuhlenkamp et al., 2020). Thus, a large and variable number of resources (usually, containers) take place to execute a collection of stateless functions inside a time limit imposed by the cloud provider. These conditions allow Serverless to implement new microservices architectures and to drive down the costs related to threshold-based elasticity. In this last technique, the charging method typically considers an entire time window, independently of the user spent only the first few seconds on that window (Jangda et al., 2019; Jonas et al., 2019; Kuhlenkamp et al., 2020; Winzinger and Wirtz, 2019).

Analyzing the state-of-the-art, we perceive that works are trying to address the concern related to easying the use of Serverless applications to non-expert users. Most of the works tend to attack the issue called cold start that causes the first computation of a Serverless program to be slow, as shown by (Abad et al., 2018; Horovitz et al., 2019; Kim and Cha, 2018; Oakes et al., 2017; Winzinger and Wirtz, 2019). Albeit there is much progress in this area, the authors in (Jonas et al., 2019) point us to broader needs for the Serverless computing platforms. In particular, the article cites that the users of Serverless computing still need to specify the memory usage of their programs that do not match the concept behind Serverless computing, where the users should only worry about their code. Yet, in (Kuhlenkamp et al., 2020) the authors still cite that the problem of the user abstraction level is a concern for both the hypervisor and the own user. When a Serverless computing platform bounds the user into specifying some infrastructure requirements, this, in turn, bounds the provider to always providing at least that infrastructure for running the code. Thus, we affirm that today remains an open gap in the literature regarding the need for removing barriers to specifying infrastructure requirements on using the Serverless concept.

In this context, this work proposed a two-pass algorithm named Memoryless, who is in charge of facilitating the use of Serverless applications in such a way users must not worry about filling up details about memory size. To accomplish this, this algorithm executes microbenchmark where: (i) in the first pass we have an estimation of the needed memory size to rub a single container in the face of a set of application description; (ii) in the second phase, we have a launching of the Serverless application with the parameter previously discovered as memory limit. In addition to diffusing the employment of FaaS, Memoryless is also pertinent to reduce financial cost to end-users, since providers typically charge taking into account the selected memory requirements. In other words, Memoryless avoids overestimation in the memory parameter, so impacting directly in the cost for end-users. We developed a prototype that uses NodeJS, Kubeless, and Kubernetes, which are representative tools when assembling a container-based Serverless environment. Memoryless contributions are twofold.

- Proposal of two-pass methodology to run Serverless applications with the most suitable memory limit, so allowing users for not caring about such parameter beforehand;
- A proof-of-concept with state-of-the-art technologies, enabling the proposed methodology as viable to support Serverless infrastructure effortlessly.

The remainder of this article will first introduce the related work in Section 2. In this point, our idea is to present a comparison table, as well as the open gaps in the literature. Section 3 describes Memoryless, including its architecture, functioning and microbenchmark. Section 4 reveals the evaluation methodology. A discussion of the results appear in Section 5. Finally, Section 6 emphasizes the scientific contribution of the work and notes several challenges that we can address in the future.

2 RELATED WORK

In this section, we analyze initiatives in the state-of-the-art on Serverless usage and applications. Jonas et al. (Jonas et al., 2019) present an article that covers the current state of research and industry on the topic, detailing the main gaps on the Serverless today. They reveal an analysis of the industry viewpoint, detailing what the industry has to offer for the users of Serverless. The authors go a step further to analyze the current state of the Serverless offerings. The authors created several implementations of typical software applications in the context of Serverless. They then compared them, showing a considerable gain in resource and cost when it comes to high-level applications. However, they point out that Serverless is not

the right fit for running low-level applications, such as databases.

Horovitz et al. (Horovitz et al., 2019) present a technical work that tries to solve the "cold start" problem on Serverless today. By provisioning the same architecture in a Serverless and a usual cloud VM, it uses machine learning to predict the cost for the service to operate. It routes the call for the cheaper instance, be it Serverless or cloud VM. It is important to note that this work was made from the perspective of the user of Serverless and cloud infrastructures, different from other found technical jobs. The proposal is implemented on top of the Fission open-source Serverless engine. It uses Python Scikit Learn with Decision Trees to both: (i)estimate when the procedure calls that would arrive in the system and; (ii) to route the calls to the VMs, in such a way Serverless platform would not handle it. The result of the classification algorithm then makes fake calls to the Serverless application to remove the "cold start" and spin up a single VM to handle the requests in the meantime.

Oakes et al. (Oakes et al., 2017) aim to raise the abstraction and the execution speed of Serverless applications. The idea is to require the user of a Python Serverless application to deploy, along with his/her code, a list of the needed packages that the service depends on. By bringing the need to bundle the application onto the hypervisor, this strategy enables the hypervisor to load the application faster by caching the packages needed for all users of the Serverless platform. Thus, we have less FaaS to load into memory. The model proposed in (Oakes et al., 2017) is supposed to be security-aware. By forking new processes from the running dependencies, it isolates all the callers from seeing each other's data should they share any package. The authors use a copy-on-write-like directive to avoid package sharing while still allowing for heavy package reuse when the user provides the code for them to run.

Abad, Boza, and Eyk (Abad et al., 2018) build on top of the work of Oakes et al. (Oakes et al., 2017) by inserting a cache hit/miss algorithm, in addition to an algorithm that is aware of the packages needed to run a Serverless provision. These changes increase the performance of dependency management by 66%. Kim and Cha (Kim and Cha, 2018) bring a two-step algorithm to lower the latency of the calls to Serverless. The first is a model to allocate a three-state worker. Here, we have a swarm of "template workers" (the base images needed for any Serverless function), which would be converted to a "Ready worker" that would run all the pre-running steps to leave the worker ready to be deployed, which could then be promoted an "Active worker" (the actual code executing). The second step of the algorithm is a sliding window dynamic prediction to determine when a "Template worker" would be allocated to a "Ready worker" and then to an "Active worker."

Hall and Ramachandran(Hall and Ramachandran, 2019) present an interesting take on the Serverless world. The authors first assess the current problems in Serverless computing, namely the cold start problem, and pinpoint the main issue with it in the use of containers to run native code. The authors then propose a new container-like work by using the V8 engine to run WebAssembly code, which has a faster initialization time than native code running inside typical containers. By using existing compilers targeting WebAssembly from native-targeting languages, there is a quicker initialization time. However, the main issue lies in the current speed of WebAssembly, which still is slower than native code.

To analyze and compare the aforementioned works, we have devised the following guiding questions:

1. Does the work deal with Serverless?
2. Does the result of the work need to be implemented by the hypervisor?
3. Does the work aim to improve the performance of the Serverless application execution?
4. Does the work consider memory usage?
5. Does the work use cache to improve the performance of the execution?
6. Does the work use machine learning?
7. Does the work aim to fix the "cold start" problem?
8. Does the practice aim to raise the usage abstraction for the user of a Serverless platform?

While most of the works were able to answer questions 1, 2, 3, 5, 6, and 7, no work was able to answer questions 4 and 8. Then, we envisage a gap in the state-of-the-art, as presented in Table 1, that offers an analysis of the works mentioned above. A better alternative to having the user specify resources would be to raise the level of abstraction, having the cloud provider infer resource requirements instead of having the developer define them. Provisioning just the right amount of memory automatically is particularly appealing but especially challenging when the solution must interact with the automated garbage collection used by high-level language runtimes.

3 MEMORYLESS MODEL

We plan to complete the gap existing in the literature that consists of the necessity to inform the mem-

Table 1: Comparison of the selected works.

Feature	(Jonas et al., 2019)	(Horovitz et al., 2019)	(Oakes et al., 2017)	(Abad et al., 2018)	(Kim and Cha, 2018)	(Hall and Ramachandran, 2019)
Does the work deal with Serverless?	✓	✓	✓	✓	✓	✓
Does the result of the work need to be implemented by the hypervisor?		✗	✓	✓	✓	✓
Does the work aim to improve the performance of the Serverless application execution?		✓	✓	✓	✓	✓
Does the work consider memory usage?		✗	✗	✗	✗	✗
Does the work use cache to improve the performance of the execution?		✗	✓	✗	✗	✗
Does the work use machine learning?		✓	✗	✗	✗	✗
Does the work aim to fix the ”cold start” problem?		✓	✓	✓	✓	✓
Does the practice aim to raise the usage abstraction for the user of a Serverless platform?	✗	✗	✗	✗	✗	✗

ory limit when launching a Serverless demand to the cloud. We agree that the concept of Serverless is up-and-coming, and to accomplish this, it is important to provide better abstraction on its usability. In this way, Memoryless follows this idea by hiding from the user any details related to memory resource allocation and management of machine instances in the cloud. Instead, they can run code on cloud servers without having to configure or maintain the servers at all. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity.

This section first presents the design decisions of Memoryless modeling. Second, we reveal the system architecture, highlighting the user interaction. Finally, the proposed algorithm is described, where we also highlight the particular points that reside our contribution.

3.1 Design Decisions

We developed Memoryless to able users to launch Serverless applications without needing to describing any detail related to memory. Memoryless then changes the way users from today's Serverless use the platform in the following manner: (i) users would have to submit the expected parameters of their program along with their source code; (ii) the hypervisor can run a proposed microbenchmark to determine the most suitable memory needs for the Serverless application. Thus, Memoryless can be seen as a two-pass algorithm. The idea in (i) is to receive both the FaaS (UC) to be run on the Serverless platform and a set of parameters (PR) in the form of events that trigger the Serverless program. The Serverless platform should then start a modified 'function container' in (ii) that should have the ability to query for the used memory in any given moment in time. We understand that using a microbenchmark to determine the best value of the memory brings the following benefits:

- For the user, who will pay exactly what his/her application really needs, so avoiding overestimation of this parameters and an subsequent overcharging on running a Serverless application;
- For the cloud provider, since more users can be scheduled concomitantly to use the available cloud resources.

To enable these benefits, we bring the idea that the cloud provider must do not charge the user when running a sample application with determined parameters. The output is the most suitable memory size and the own provider profits on doing this procedure as clarified earlier. Technically, the platform should then test for the needed memory by executing the function UC with the parameters PR and querying the memory used after each test set be completed. After all the querying done, the platform gets the maximum used memory of the execution. This value is used to launch the Serverless application subquently. Our idea then is to exchange the low-level representation of memory to specifying a high-level parameter list. This list

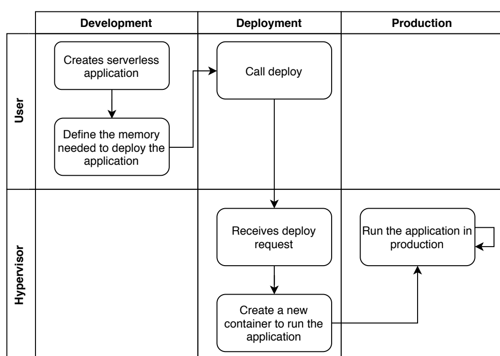


Figure 1: Current Serverless model.

would be application-specific and, more importantly, more relevant to the business reason for the application rather than the technical basis. Also, considering that the final cost is proportional to the allocated memory, the proposed strategy also goes towards providing a solution for saving money in such a way we nave the most suitable memory limit to run a program in the cloud.

3.2 Architecture

This section presents the Memoryless architecture, where we present interactions of the user and hypervisor in three moments: (i) development; (ii) deployment; (iii) production. Figure 1 illustrates these actors and moments in a traditional Serverless model. At the development stage, users must be involved with memory details of their applications. Figure 2 depicts the proposed methodology, where we divide the hypervisor offering in two steps involving Microbenchmark and Production. The microbenchmark is responsible for determining how much memory the application will need to run effectively. The Production Step, in its turn, is the same as the current Serverless offerings; but here, the memory requirement will come from the microbenchmark step rather than from the user. Thus, the cloud service provider manages the infrastructure and the software and maps the function to an API endpoint, transparently scaling function instances on demand.

The proposed architecture takes into account the moment from which the user submits his/her code to the hypervisor. Then, we have the replacement of the current requirements of FaaS and memory to FaaS and parameters. These parameters could be informed through command-line or by using auxiliary files that are uploaded when launching a Serverless application. Considering the settings, we agree that the user must be aware that the fewer parameters the user adds, the worst will be the return of the microbenchmark.

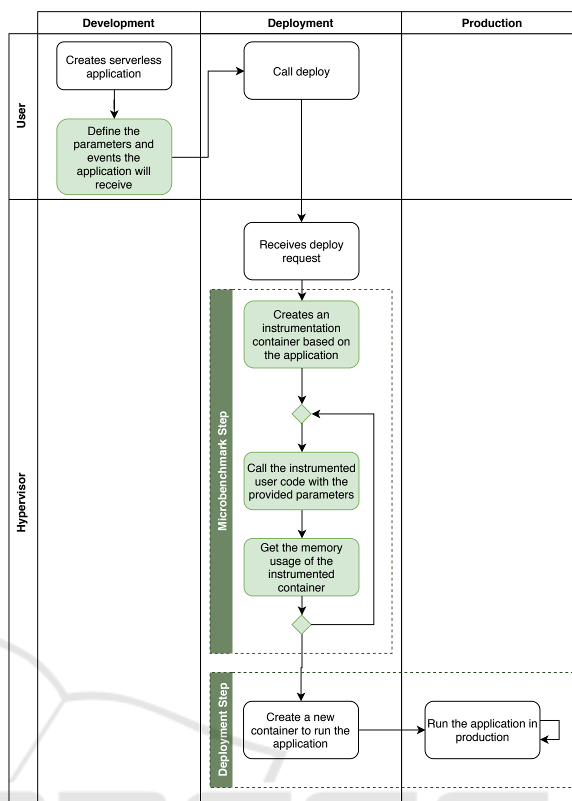


Figure 2: Memoryless architecture.

Therefore, the more diverse and rich the parameters are, the more accurate the memory number yielded by the microbenchmark step will be.

The parameters needed to determine the memory limit on running a Serverless application vary depending on the implementation of the Serverless infrastructure. Memoryless model must support all the parameters that the Serverless infrastructure allows. Assuming that a Serverless provider supports both HTTP events and an internal event mechanism, the user must be able to provide examples of such events so that the microbenchmark step can effectively mimic a real-world usage of the Serverless infrastructure. So, the format of such parameters is implementation-specific. The parameters must be stateless and must not depend on one another; in such a way, the microbenchmark can run calls in parallel with a different load of events for the Serverless application.

The proposed model does not make a distinction between a call correctly placed or if an error occurred inside the FaaS. The model will query for the used memory should the call produce an error or not. The user should decide whether the error handling mechanism should reflect any changes in the memory available for his/her program or not. Finally, this step of

training the Serverless execution to obtain the memory limit to execute the application is not charged to the user, *i.e*, it corresponds to a service offered by the cloud providers to users to enable them to tune their applications to consume the most suitable number of resources. Here, it is pertinent to remember that the amount of allocated meory is commonly proportional to the number of containers used to run a Serverless demand.

3.3 Algorithm Proposal

Algorithm 1 presents the microbenchmark step on the Memoryless execution. This algorithm needs to receive three parameters: (i) `faas`: The function to be run; (ii) `pr`: the parameters needed to run the argument FaaS; (iii) `runs`, which refers to the number of times the parameters will be called. While the arguments `faas` and `pr` will be provided by the user of the Serverless platform, the platform itself will have to decide the maximum load allowed for the item runs. To test for accurate loads, the platform should set it to the maximum number of connections and events available to a single container (before the horizontal scaling starts).

Algorithm 1: Microbenchmark Step.

Input:
Faas - The Function as a Service
pr - Set of parameters for the FaaS
runs - Times to run the tests
Output:
MM - Maximum required memory for the provided faas

1: $i \leftarrow j \leftarrow 0$
2: $MU \leftarrow \{\}$
3: $ic \leftarrow createMicrobenchmarkContainer(faas)$
4: **for** i=0; i ¡ runs; i++ **do**
5:     **for** j=0; j ¡ num(pr); i++ **do**
6:         $inside\ ic\ call\ faas(pr[j])$
7:         $memory \leftarrow getCurrentMemoryInContainer(ic)$
8:         $MU \leftarrow MU \cup memory$
9:     **end for**
10: **end for**
11: $MM \leftarrow Max(MU)$
12: **return** MM

Also, we are assuming that the provider will delivery two other functions:

1. `createMicrobenchmarkContainer`: a function that upon receiving the FaaS will create a new container with that FaaS and with instrumentation abilities, namely the possibility to query for the total used memory at any given time.
2. `getCurrentMemoryInContainer`: a function that returns the current memory usage of the container specified.

The last assumption needed for executing the ideas of Memoryless is that we are working with a runtime that performs memory management with a Garbage Collector. In this way, line number 7 of Algorithm 1 will yield all the memory used when executing the Serverless demand. If not working with a Garbage Collector, then the microbenchmark container must record the used memory during line 6, since at this time we need to know the current used memory because it is not freed during previous executions. Thus, logically, the second execution will retiorn more memory if compared to the first because of memory never goes to zero. The high-level production step should replace the current infrastructure by having an action to call the Microbenchmark step before starting the container. Finally, we must get the output of Algorithm 1 and run the final Serverless application.

4 EVALUATION METHODOLOGY

Today, the literature agrees that there is not a standard methodology to evaluate Serverless applications (Kim and Lee, 2019; Kuhlenkamp et al., 2020; Winzinger and Wirtz, 2019). For example, in (Kim and Lee, 2019) the authors used arbitrary applications in the fields of Big Data, Web and security. On the other hand, Kuhlenkamp et al. (Kuhlenkamp et al., 2020) explored a synthetic application with three distinct phases: warm-up, scaling and cool-down. Yet, dependency testing between different application modules was addressed in a Serverless infrastructure in (Winzinger and Wirtz, 2019). Therefore, in this moment we opted by developing our own evaluation methodology, which was focused on analyzing the memory impact on Serverless deployment. Our methodology comprises the answer for a collection of questions and the implementation of a prototype and experimental memory-targeted applications. The following guiding questions are planned to be used to evaluate the Momeryless prototype:

EV1 Can the implementation run in a two-pass configuration?

EV2 Can the Microbenchmark step yield the maximum memory used with the given parameters?

EV3 Can the system assign different memory requirements for two applications with distinct memory usages?

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

In brief, for any given program, Memoryless should be able to provide the following software requirements: (SR1) determine the maximum memory needed for running the program on a single container; (SR2) use the information obtained in S1 to instantiate a container; (SR3) remove the need from the user to specify memory limit on FaaS implementations.

The developed prototype has a set of requirements for the implementation. Regarding the Function as a Service runtime: (i) it must use garbage collection as memory management; (ii) there must be an open API to query the memory usage at any given moment in time. Regarding the Serverless hypervisor implementation: (iii) It must be extensible to allow for the creation of the Microbenchmark step; (iv) It must be based on containers so it will be possible to create the microbenchmark Container. Regarding the parameters: (v) the only event parameters supported will be HTTP events; (vi) the HTTP events shall not carry any payload. Regarding the FaaS: (viii) there should be a simple Function as a Service implementation that queries a database and return data to the caller; (viii) there shall be a memory-heavy Function as a Service implementation that inflates its memory until a given threshold and stays in that threshold while being called.

The chosen runtime was NodeJS (NODE.JS FOUNDATION, 2019a), as it provides memory management via Garbage Collection and it has an API for querying the memory usage during the execution of a process (NODE.JS FOUNDATION, 2019b). This satisfies the requirements (i) and (ii). For the implementation, we are using Kubeless (KUBELESS, 2019), since it is an open-source implementation of a Serverless provider. Being open-source enables us to change its behavior to implement our model, thus satisfying (iii). It also is built on top of Kubernetes (THE KUBERNETES AUTHORS, 2019), which is an orchestration mechanism for containers, thus meeting (iv) as well. Following the same ideas from (Horovitz et al., 2019), here we also use the nodecellar application to mimic a simple FaaS that queries a database, after converting it to be able to run on the platform Kubeless. This satisfies (vii). For (viii), the authors developed a simple application that randomly generates strings and stores them into memory, using the same memory usage API (NODE.JS FOUNDATION, 2019b) to set the memory size used to a specific limit. Finally, both sample applications satisfy (v) and (vi).

5 RESULTS

In this section, we present the results obtained when running the prototype with the evaluation methodology details described earlier.

5.1 Winecellar Application

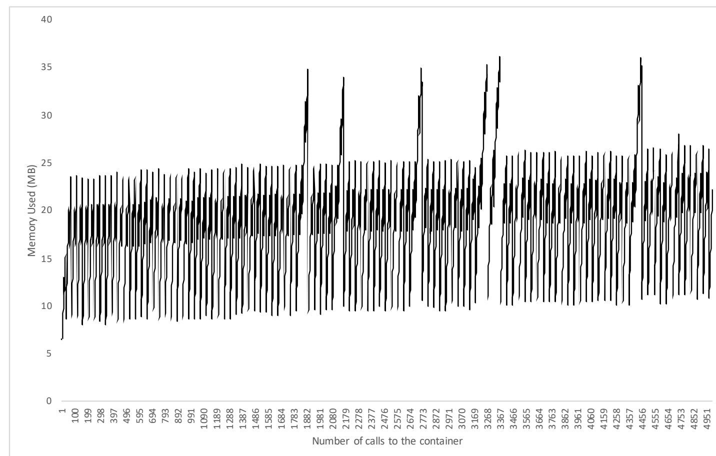
Winecellar (https://www.vinfolio.com/cellar-management) is a Web application where we have a collection of wines and operations such as query, addition and removal are allowed over such collection. Thus, we deployed the functions (add, remove and query) of this application in the Kubeless Serverless implementation. When running the Winecellar, the microbenchmark step was able to run 5 thousand calls to the function that read all the wines stored in the database. The results can be found in Figures 3a and 4a. The graphs were made by storing all the values returned at the line 13 of the Algorithm 1.

The application has shown to have peak memory at 36.11 MB, which can be found around call number 4456 on the graph of Figure 3a. Here, we achieved a median memory usage of 8.96 MB, where more than half of the calls were situated between 14.83 MB and 21.73 MB, as seen in graph 4a. The return of the Algorithm 1 in this instance was 36.11 MB. The production step was then able to deploy a production-ready container with the *FaaS* code and a memory limit of 36.11 MB. Without this number, the user possibly will complete with a higher parameter; for example, 64 MB or more.

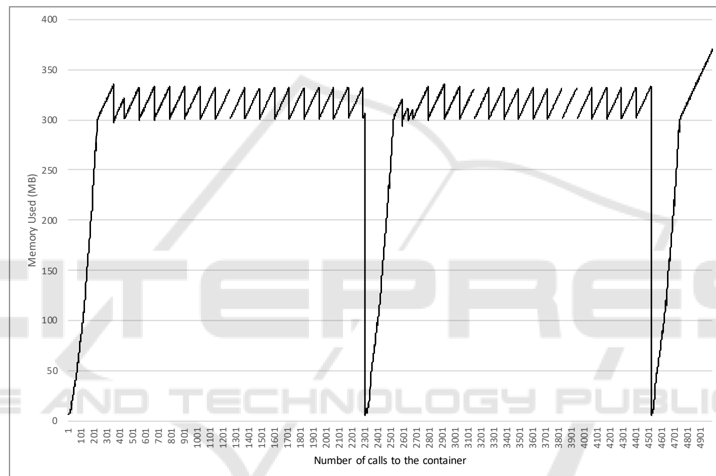
We observe that in Figure 3a, there is a trend to use more and more memory, as the minimum memory needed to run keeps increasing as the calls keep being made. This could be an indication of a memory leak either in the Winecellar application or in the Serverless provider implementation. Further investigation would be required to determine the exact nature of this behavior. Another factor to investigate further would be the memory peaks shown in the calls with numbers equal to 1822, 2179, 2674, 3169, 3268, and 4456. These peaks clearly show that in these moments, something is stressing the memory usage abruptly. Further investigation would be necessary to pinpoint what the cause such peaks.

5.2 Memory-stress Test Application

The application designed to stress the memory running inside the containers presented the results shown in Figure 3b. This application was designed to add a



(a) Memory usage for each call in the Winecellar application.



(b) Memory usage in memory-stress test application.

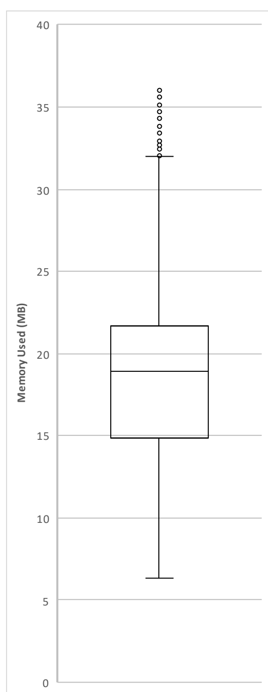
Figure 3: Timeline of memory usage per call for both evaluated applications.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

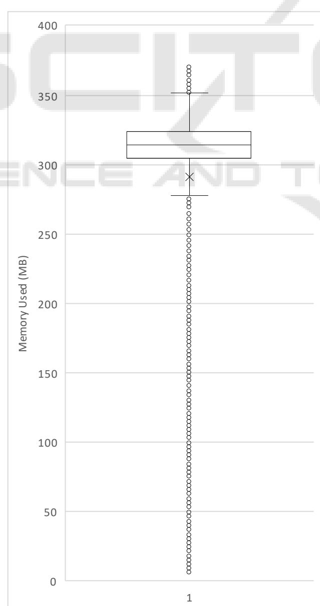
random string into memory in each call until the process had 300 MB. If the process had more than 300 MB, the call would not execute anything. This application presents us a more interesting scenario to analyze. The peak of memory usage was 370.22 MB, as seen in the graph of Figure 3b. We observe that the median was 313.99 MB, and half of all the calls used between 304.89 MB and 323.61 MB of memory (see Figure 4b). The production step was then able to deploy a production-ready container with the *FaaS* code and a memory set of 370.22 MB.

Observing the results, we highlight information regarding the period that declines the memory usage, which appears close to the calls 2501 and 4501. This can be explained by the lifespan on the Serverless container timing out, as the call to create the strings takes more time than the call to query a database (albeit the query has network traffic to fetch from the database). Thus, after around 2 thousand calls, the Serverless provider ends up destroying the container running the function and has to start it again when the next call is executed.

Another factor to observe refers to the application, which usually shows a peak memory of 323.61 MB. This can be explained as 300 MB being the limit in memory usage: summing the random string in memory and the heap memory needed by the NodeJS engine to run the function when called. The magnitude of the offset, from 20 MB to 30 MB, is comparable to the memory usage shown in Subsection 5.1, where the memory cost for the NodeJS application is minimal. The sawtooth shape of the graph also indicates that the memory needed don't go under 300 MB (as is specified by the function), but the runtime keeps garbage collecting the memory required to run the service. Thus, we can then say that around 300 MB is



(a) Boxplot for memory usage for each call in Winecellar.



(b) Boxplot for memory usage in memory stress test.

Figure 4: Box plot graph of the tested applications.

being used for the runtime stack, and the remaining memory is being used for the heap of the runtime.

In all instances of the lifespan, the first calls always show some variation in memory usage, such as in the calls 201 to 601, and calls 2501 to 2801. Therefore, this could be the result of the heap misbehaving, and, if run for more calls, the runtime could stabilize. It is essential to note that this example only has one parameter P_r and one type of call to be run in the Microbenchmark Step. If there would be more types of calls and more behaviors testing for this same application, the graph could have a better representation of reality, and the uniformity of the saw-tooth peaks would be more fluent.

5.3 Tools Shortfalls

We used computing tools that presented some shortfalls. Most notably were the limitations of the Kubeless implementation. Kubeless lacks a "scale to zero" capability, one of the hallmarks of Serverless computing (Jonas et al., 2019). On the upside, it does not possess a "cold start" (Jonas et al., 2019) issue since it never stops executing (thus being pricier). Also, the lack of debugging capabilities makes it harder for the user to develop targeting this platform. We also observed the lack of active development in (KUBELESS, 2019). The open-source community seems to have migrated from Kubeless to the Google-sponsored Knative (THE KNATIVE AUTHORS, 2019), as it is described as a Kubernetes-based platform to deploy and manage modern Serverless workloads. Future works include implementation and testing of Memoryless on top of Knative instead of Kubeless and verification whether the caveats presented here are still present.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

5.4 Discussion

This section goes back to the questions raised in Section 4 to check if the evaluation yields a positive result and if the specific objectives were fulfilled. For all the questions, we raise the same three points: (i) Was it answered? Was it a positive outcome? (ii) Does it have some drawbacks? Which?; (iii) Is it possible to fix this drawback? Would it need future work? In Tables 2, 3, 4 and 5, we have a complete breakdown of the questions and their answers. Some questions and evaluation methods are very similar that they were joined in single table.

6 CONCLUSION

The Serverless computing paradigm is a game-changer in cloud computing. The current caveat of having to specify how much memory will a Serverless application has to use beforehand is indeed against the current trend of raising the abstraction layer. For

Table 2: Answers and drawbacks of EV1 and SR2.

(EV1) Can the implementation run in a two-pass configuration? (SR2) Use the information obtained in SR1 to instantiate a container that is limited to the needed memory on the application	
Answer	Yes, the implementation has proven that it can indeed run in a two-pass configuration by extending the current deployment step to run the implementation first and only start the production container after the memory has been determined.
Drawback	A major drawback is the time to start executing the deployment function in production, as the *Microbenchmark Step* can take T seconds to run all the *Microbenchmark Step*, where $T = \lvert P_a \rvert * P_r * t$ where $\lvert P_a \rvert$ is the number of parameters received, P_r is the precision required by the hypervisor and t is the time in seconds to run the *FaaS* application.
Possible solution to drawback	A solution for future work could be to parallelize the execution of the Microbenchmark Step and the Production Step so that the Production Step can run with the maximum memory until the Microbenchmark Step is ready to re-create the production container with the minimal viable memory allocation.

Table 3: Answers and drawbacks of EV2 and SR1.

(EV2) Can the Microbenchmark Stepyield the maximum memory used with the givenparameters? (SR1) Determine the maximum memory needed for running the program on a single container	
Answer	Yes, the Microbenchmark Step has proven to be able to yield the correct answer, as can be checked on Figures 3 and 4.
Drawback	It only works with garbage collected languages, as it can only synchronously query memory usage, and only a garbage collected language would still hold irrelevant items in memory.
Possible solution to drawback	The model could be improved in future works to query the memory usage at a fixed time interval asynchronously, or it could continuously register the memory usage of the container running the user code during the maximum memory test and return that value when queried.

Table 4: Answers and drawbacks of EV3.

(EV3) Can the system assign different memory requirements for two applications with distinct memory usages?	
Answer	Yes, the system can assign different memory limits to different programs, as seen in Section 3.
Drawback	None.
Possible solution to drawback	None.

Table 5: Answers and drawbacks of SR3.

(SR3) Remove the need for the user-specified memory limit on FaaS implementations	
Answer	Yes. One of the steps for the model is to remove the need for user-specified memory limits
Drawback	By removing the option for the user to specify the memory the user is losing low-level control over their application, but, as seen before, this is the whole intent of the Serverless computing movement.
Possible solution to drawback	Future works could create a questionnaire and ask users of Serverless computing solutions if they prefer to have more low-level control over their programs, or they would rather have their infrastructure calculate everything for them. They only can worry about their application logic.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

example, all the recent Serverless offerings, including Google Cloud Functions (released by Google in 2017), AWS Lambda (introduced in 2014), and Microsoft Azure Functions (presented in 2016), present this drawback mentioned above. In this context, this article introduced a new proposal named Memoryless in such a way the memory limit is obtained through a microbenchmark, which should be offered by the cloud provider. In particular, we filled the gap pointed out by Jonas et al. (Jonas et al., 2019) who present the need to solve this memory issue to in fact elevate the use of this computing principle.

By building a prototype of the model and running two different types of workloads, we can denote that the proposed model obtained encouraging results being feasible to launch Serverless effortlessly. We must also highlight the financial cost factor. In a Serverless computing deployment, the cloud customer only pays for service usage; there is never any cost associated with idle, down-time. However, this payment is proportional to the registered memory; then, Memoryless goes towards saving money since here we will not have memory overestimation when running Serverless demands. Finally, our contributions can be employed to bring to an elastic-cloud platform yet more applications from different areas, including the Internet of Things, event-triggered computing, mobile apps, backend procedures, and high-volume of data treatment.

ACKNOWLEDGEMENTS

The authors would like to thank the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - CAPES (Finance Code 001) and Conselho Nacional

de Desenvolvimento Científico e Tecnológico - CNPq (Grant Number 303640 / 2017-0).

REFERENCES

Abad, C. L., Boza, E. F., and van Eyk, E. (2018). Package-aware scheduling of faas functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 101–106, New York, NY, USA. ACM.

Amazon (2019). Serverless computing – amazon web services. https://aws.amazon.com/serverless/. Accessed on 12/06/2019.

Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica, I. (2009). Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28(13):2009.

Hall, A. and Ramachandran, U. (2019). An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, pages 225–236, New York, NY, USA. ACM.

Horovitz, S., Amos, R., Baruch, O., Cohen, T., Oyar, T., and Deri, A. (2019). Faastest - machine learning based cost and performance faas optimization. In Coppola, M., Carlini, E., D'Agostino, D., Altmann, J., and Bañares, J. Á., editors, *Economics of Grids, Clouds, Systems, and Services*, pages 171–186, Cham. Springer International Publishing.

Jangda, A., Pinckney, D., Brun, Y., and Guha, A. (2019). Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA).

Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C.-C., Khandelwal, A., Pu, Q., Shankar, V., Menezes Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J., Popa, R. A., Stoica, I., and Patterson, D. A. (2019). Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley.

Kanso, A. and Youssef, A. (2017). Serverless: Beyond the cloud. In *Proceedings of the 2nd International Workshop on Serverless Computing*, WoSC '17, page 6–10, New York, NY, USA. Association for Computing Machinery.

Kim, J. and Lee, K. (2019). Practical cloud workloads for serverless faas. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 477, New York, NY, USA. Association for Computing Machinery.

Kim, Y. and Cha, G. (2018). Design of the cost effective execution worker scheduling algorithm for faas platform using two-step allocation and dynamic scaling. In *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, pages 131–134.

KUBELESS (2019). Kubeless. https://kubeless.io/. Accessed on 21/11/2019.

Kuhlenkamp, J., Werner, S., Borges, M. C., Ernst, D., and Wenzel, D. (2020). Benchmarking elasticity of faas platforms as a foundation for objective-driven design of serverless applications. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, page 1576–1585, New York, NY, USA. Association for Computing Machinery.

Nasr, . H. M. E. A. E. K. M. M. (2019). An adaptive technique for cost reduction in cloud data centre environment. *International Journal of Grid and Utility Computing*, 10:448–464.

Nguyen, H. D., Zhang, C., Xiao, Z., and Chien, A. A. (2019). Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 1–6, New York, NY, USA. Association for Computing Machinery.

NODE.JS FOUNDATION (2019a). Node.js. https://nodejs.org/en/. Accessed on 21/11/2019.

NODE.JS FOUNDATION (2019b). Process — node.js v13.1.0 documentation. https://nodejs.org/api/process.html. Accessed on 21/11/2019.

Oakes, E., Yang, L., Houck, K., Harter, T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2017). Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400.

THE KNATIVE AUTHORS (2019). Knative. https://knative.dev/. Accessed on 22/11/2019.

THE KUBERNETES AUTHORS (2019). Production-grade container orchestraion - kubernetes. https://kubernetes.io/. Accessed on 21/11/2019.

Villano, V. C. A. D. B. M. R. U. (2020). A methodology for automated penetration testing of cloud applications. *International Journal of Grid and Utility Computing,*, 11(2):267–277.

Winzinger, S. and Wirtz, G. (2019). Model-based analysis of serverless applications. In *Proceedings of the 11th International Workshop on Modelling in Software Engineerings*, MiSE '19, page 82–88. IEEE Press.