Serverless: Beyond the Cloud

Position Paper

Ali Kanso
IBM Research
Yorktown heights, New York
akanso@us.ibm.com

Alaa Youssef
IBM Research
Yorktown heights, New York
asyousse@us.ibm.com

ABSTRACT

Serverless computing is widely known as an event-driven cloud execution model. In this model, the client provides the code and the cloud provider manages the life-cycle of the execution environment of that code. The idea is based on reducing the life span of the program to execute functionality in response to an event. Hence, the program's processes are born when an event is triggered and are killed after the event is processed. This model has proved its usefulness in the cloud as it reduced the operational cost and complexity of executing event-driven workloads. In this paper we argue that the serverless model does not have to be limited the to the cloud. We show how the same model can be applied at the micro-level of a single machine. In such model, certain operating system commands are treated as events that trigger a serverless reaction. This reaction consists of deploying and running code only in response to those events. Thus, reducing the attack surface and complexity of managing single machines.

CCS CONCEPTS

• **Software and its engineering → Software as a service orchestration system**; *Functionality*; *System administration*;

KEYWORDS

Serverless Computing; Linux containers; Function as a Service; cloud computing.

ACM Reference Format:
Ali Kanso and Alaa Youssef. 2017. Serverless: Beyond the Cloud: Position Paper. In *WoSC'17: WoSC'17: Workshop on Serverless Computing, December 11–15, 2017, Las Vegas, NV, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3154847.3154854

1 INTRODUCTION

Recent studies show that the infrastructure as a service (IaaS) and platform as a service (PaaS) had the highest yearly growth rate in revenues [23]. Which is a clear indication that cloud tenants are realizing the value of offloading the management of their infrastructure. In fact, most cloud tenants wish to reap the functionality of their code while minimizing their investments in managing the hardware and software stack underlying their code. Serverless computing is a natural evolution of PaaS [10, 11]. Unlike PaaS, in serverless the code pushed to cloud sits dormant until it is triggered by an event that requires the code's functionality to be executed. While the added value of serverless is evident at the macro-scale of the cloud, it has not been fully explored at the micro-scale of individual machines. In this paper we argue that the functionalities needed on a single machine can be made serverless. This would reduce the operational complexity, as well as the attack surface and in some cases the resource utilization on the machine.

Minimal operating systems are on the rise [7, 8]. With the surge of SaaS (Software as a Service) products, the need to clutter one's machine with utility software is diminishing. For instance, a web browser can now substitute a wide variety of software that can create and/or edit text, spread sheets photos, videos, etc. However, some needed functionalities cannot be exported to the cloud. For example, when using a packet analyzer to debug network issues, the process executing this functionality has to be running on the local machine with certain privileges. Nonetheless, the functionality of analyzing network packets is rarely used by most users which raises the question of whether it needs to sit dormant on the machine after it successfully performs its operations. Therefore, we believe that such functionality can be made serverless, whereby, the software performing this functionality is deployed on demand and removed thereafter. Although this may add some overhead and latency on the short run, we believe the added benefits in terms of simplicity and added security can outweigh those shortcomings. In the context of this paper, we characterize serverless as the ability to achieve functionality-on-demand, in a transparent way where the user of said functionality is not aware of the underlying complexity involved in attaining it. We introduce the term functionality-on-demand to refer to our proposed serverless approach on the local machine. This paper is organized as follows: in Section II we present the background of our work explaining the technology behind servereless. In Section III we discuss how serverless can be applied at the machine level. In Section IV we survey the literature for related work. Finally, we conclude and discuss our future work in Section V.

2 BACKGROUND

2.1 Serverless Computing

Serverless computing is mainly defined in the context of Function as a Service. The users register their functions (code) with the cloud serverless provider. Those functions are triggered based on events. A trigger can be an HTTP route that is invoked, a timer, an object added/removed from storage among others. In most serverless implementations, when a function is triggered, the code is loaded

WoSC'17, December 11–15, 2017, Las Vegas, NV, USA

ACM ISBN 978-1-4503-5434-9/17/12…$15.00
https://doi.org/10.1145/3154847.3154854

in a container [22] where is executes in an isolated sandbox. The container is deleted or cached for future reuse after the function is executed. Hence, containers are considered key enablers of the serverless paradigm. While serverless computing is not ideal for long lived state-full workloads such as databases. It can support a wide variety of stateless event-driven workloads.

2.2 The Serverless Framework

Serverless frameworks are typically implemented as distributed systems with multiple components working together to abstract the system and its configuration from the end-user, and thus giving the end user the serverless experience. For instance, OpenWhisk [17] has a front end web server accepting user requests and forwarding them to a system controller. The controller in turn analyses the requests and invokes an action to be executed. The invoker is an agent that sits on worker node waiting to receive an action from the controller, at which time, it will instantiate a container, inject the function's code into the container and executes it. The users'functions are stored in a database, as well as the system state as shown in Figure 1.

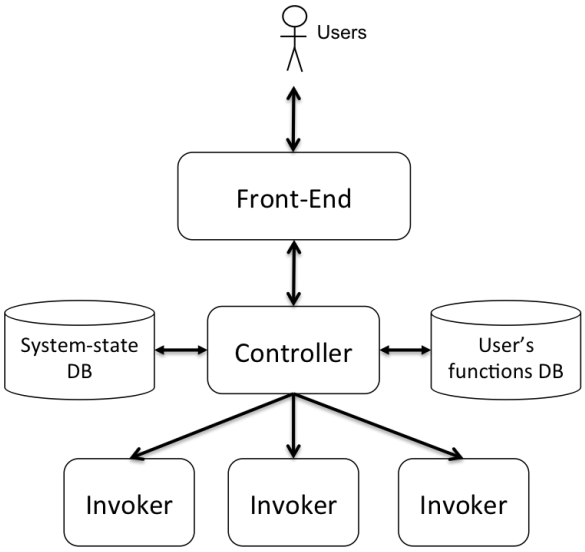

Figure 1: The OpenWhisk overall architecture.

2.3 Linux Containers

Linux containers are perceived as a lightweight virtualization technology. They act as process-level wrappers that isolate the application and account for its resource consumption. Containers leverage the Linux namespaces [13] for isolation (network, process ID, mount ...) and control groups (cgroups) [2] to limit and account for the resources (CPU, Memory, disk I/O ...) consumed by processes. A container is spawned based on an image. An image is an immutable file-system and a set of parameters to use when instantiating the container at runtime. Such image normally includes the executable files of the process(es) (in the containers root file system once started). The image typically includes a specific program for which it is created, e.g. a database management system, and all the libraries and dependencies needed to execute the program. As a result, the container image can run on all Linux distribution that share the standard kernel. There is even support for containers across none Linux platforms [14, 15]. The container can also point to volumes that are mounted to the container when it is instantiated. There are several technologies today for managing the containers life-cycle (creation, update and deletion). They are often referred to as the container runtime or container engine [5, 6].

3 SERVERLESS AT THE MACHINE LEVEL

With the emergence of SaaS, a lot of the functionalities that were once accessible by installing and configuring software of the local machine are now offered as a service. Hence, the average user can get a lot more functionality from the personal computer without the complexity of software management and configuration. On the other hand, we see that for the software that needs to run on the local machine, the user does not get much assistance beyond the package manager ability [20, 21]. We believe that a lot of the functionality needed on the user's machine can be made serverless, in the sense that the user gets the functionality without the hassle of installing and managing the software.

3.1 The functionality-on-demand approach

When a user invokes a program on the local machine, the successful invocation depends on having the program (and its dependencies) installed on the local machine. We propose an alternative approach, where the system is smart enough to realize that even if the software is not installed, a container image containing this software can be found and instantiated, and then, after the successful completion of the invocation, the container can be cleaned up. In other words, the container does not need to outlive its usefulness. From this perspective, the user will get the functionality needed in a transparent manner, without having the burden to install and configure the needed programs. With less programs installed on the machine, the attack surface is reduced. Moreover, since the running programs are wrapped in containers, they are more secure as they have limited impact on the system and other programs. The rational behind using container to implement our approach is as follows: (1) containers are simple to package, deploy and remove. (2) each container can be assigned limited resources, hence a memory leak in one containerized process will not affect other containers. (3) Each containerized process will run in a sandbox with limited to no visibility to other containers. (4) Containers are now supported on multiple platforms and hence the functionality-on-demand solution becomes portable across platforms. (5) Using containers makes it easier to keep track to the latest (and usually more secure) patched versions of the container images.

3.2 Serverless Shell

To illustrate our proposed approach of functionality-on-demand, we introduce the serverless shell. In Linux operating system, the shell is a user program that acts as a mediator between the user and the kernel. The shell is a command interpreter that forwards user commands into the kernel in order to execute programs, create files, manage users, etc. We created a simple prototype serverless shell that instead of directly forwarding the commands to the kernel, it

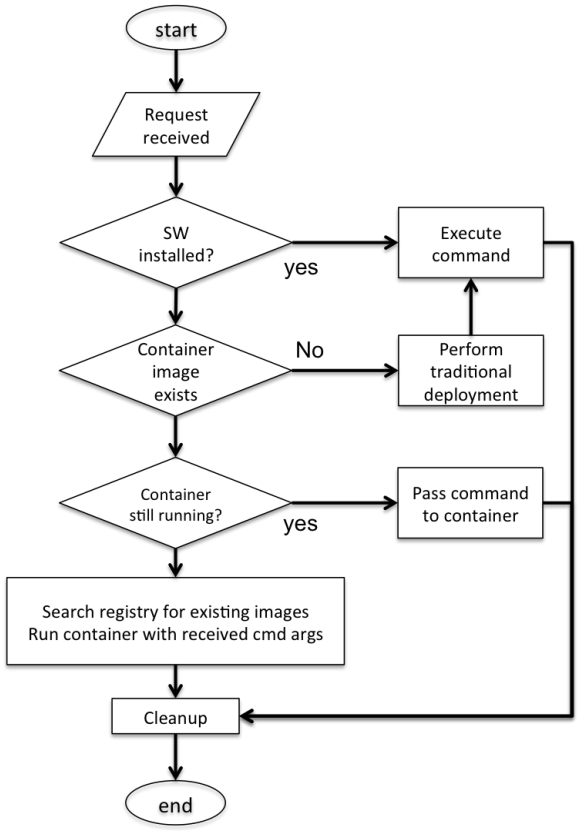

Figure 2: The Serverless shell work-flow.

follows a different behavior illustrated in Figure 2. Upon receiving a user command, the serverless shell would first check if the program specified in the command is installed, in such case, it will execute the command. However, if the program is not installed, the shell will probe the image-registry to check if there is a container image that includes the needed program. If not, it will perform a regular deployment and invoke the command thereafter. On the other hand, if such an image exists, the shell will check if there is an instantiated container of this image, and passes the command to the container to be executed. Finally if the image exists and the container is not instantiated, it will pull the image locally, and instantiate the container.

3.3 Implementation

Our serverless shell prototype implements the logic shown in Figure 2. It implements multiple interfaces as shown in Figure 3.

In our prototype, we use a repository of container images of system utility tools such as TCP-dump [24], wireshark [26], etc. We also implemented a mapping function that maps each command to an image. The shell expects the command to be structured according to (1), the command (representing the program name) followed by the options and finally the arguments.

$$command[-option(s)][argument(s)] \quad (1)$$

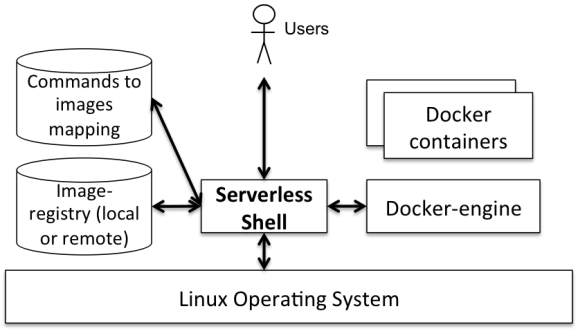

Figure 3: Serverless shell overall architecture.

Our serverless shell implementations shares some similarities with the serverless framework architecture shown in Figure 1. The shell acts as a front-end, controller and invoker. It depends on the databases (and image-registry) to store the container images and commands mappings.

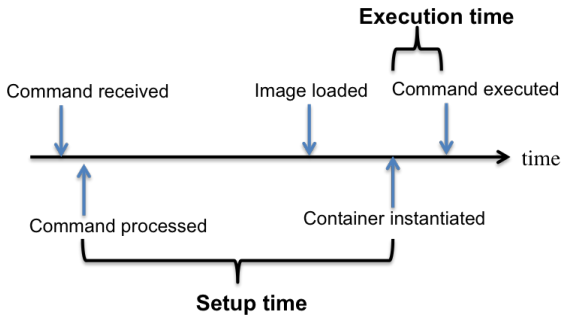

Figure 4: Serverless execution time composition.

In terms of performance overhead, the processing delay per command is on average less than 5ms. However, the latency of loading the image and starting the container can vary depending on the network bandwidth the machine resources, and the image size. This can take 10s of seconds. When the image is cached locally, the setup time drops significantly to an average of 1 second. The execution time varies according to the functionality invoked within the container. The execution and setup times are illustrated in Figure 4.

3.4 Overhead Observation

While the functionality-on-demand approach reduces the management complexity, it adds latency and execution overhead. Hence, in order to examine the efficiency of this approach, we define the (ε) measure. Where ε represents the mean time between calls of a given functionality (MTBC) minus the mean time to setup (MTTS) and mean time to execute (MTTE). In short, when $\varepsilon < 0$, then the functionality is called again before the container containing the program producing the functionality is removed, and hence the only added overhead is the latency and resources (CPU, Disk, Memory, Network, ...) consumed by the container which is typically negligible [19]. However, when ε is slightly larger than zero, then

the container is recreated shortly after is has been cleaned up, and therefore the computational overhead increases and we see a drop in resource saving. However when ε grows larger, i.e. when the functionality is rarely called, then we see that the resource saving increases.

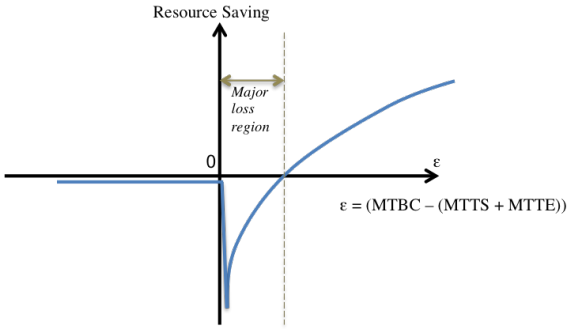

Figure 5: Efficiency measure of functionality-on-demand.

3.5 Optimization

Based on the above observation, we can clearly optimize our functionality -on-demand implementation by calculating ε and defining a caching mechanism accordingly. The idea here is to remove the major loss region shown in Figure 5 by caching the container long enough for the next proximate call to occur. Cache optimization techniques are outside the scope of this paper, however the literature is rich of contributions in this domain [3, 12, 25]. We can further optimize the efficiency by applying pattern recognition techniques, and detect patterns of execution of certain programs. Here we can distinguish between: (a) the pattern of running periodically (e.g. security scanning, update, or data compression) and (b) the pattern of having a sequence of consecutive calls (e.g. realizing that "tcpdump" command are typically followed by a "traceroute" command). When a pattern is detected whether based on frequency or sequence, we can preload the container image (or next container in case of sequence patterns) beforehand in order to minimize the setup time.

4 RELATED WORK

The related work falls under two categories, (1) the general work done on cloud functions, i.e. the typical serverless approach, and (2) the work done in the area of minimal operating systems.

4.1 Cloud Funtions:

Amazon Lambda [1] the first widely used commercial serverless solutions. It offers the users the ability to push their code to the cloud, and have their code (functions) instantiated on demand based on several available triggers. Such triggers include adding objects to storage with Amazon S3, and stream-based triggers such as Amazon Kinesis. Moreover, custom applications can trigger a function by directly interacting with the Amazon Lambda web API. Many cloud providers offer their own variation of serverless that share the main characteristic of Lambda. IBM cloud functions [4], Google cloud functions, Microsoft Azure functions [9] are all commercial offering of serverless services. We also see more opensource implementations of serverless such such OpenWhisk released by IBM as an apache project, and OpenLambda [16]. All of the above solution do not target the problem of enabling functionality-on-demand on the local machine.

4.2 Minimal Operating Systems:

Perhaps the closest vision to our proposal is the CoreOS Container Linux vision [8]. This lightweight operating system is optimized for running containers and therefore provides no package manager as a way for deploying applications. Applications with Container Linux run in containers, allowing Container Linux to include only the minimal functionality required for deploying applications inside containers. Nonetheless Container Linux does not support the users with deploying the containers transparently, instead, it is the user's responsibility to figure out what containers to deploy and when to remove them. And hence the serverless experience (functionality-on-demand) is not present. OSV [18] is another cloud focused and special purpose operating system that makes the deployment and management of the operating system itself trivial for the developers, however it offers no support to seamlessly deploy software application, and that makes it less relevant to our work.

5 CONCLUSION AND FUTURE WORK

In this paper we propose enabling the serverless paradigm on the local user machine. We believe that the concept of functionality-on-demand can significantly reduce the complexity of managing and configuring software, allowing users to focus on reaping the value of the functionality. The approach also enhances security by isolating processes in containers and removing unused programs, thus reducing the potential attack surface. While this approach may not be suitable for every setup and every type of workload, we see a lot of added benefits that can outweigh the shortcoming of added latency. As a future work we will look into extending this approach to other IoT devices where storage is limited.

ACKNOWLEDGMENTS

The authors would like to thank Dr. Ioana Baldini and Dr. Asser Tantawi from IBM research for their valuable insights.

The authors would also like to thank the anonymous referees for their valuable comments and helpful suggestions.

REFERENCES

[1] Lambda AWS. 2017. (2017). Retrieved October 2, 2017 from https://aws.amazon.com/lambda/
[2] Linux cgroups. 2017. (2017). Retrieved October 2, 2017 from http://man7.org/linux/man-pages/man7/cgroups.7.html
[3] et al Choi, Jaeyoung. 2011. A survey on content-oriented networking for efficient content delivery. *IEEE Communications Magazine* 49, 3 (2011).
[4] IBM cloud functions. 2017. (2017). Retrieved October 2, 2017 from https://console.bluemix.net/openwhisk/
[5] Linux containers. 2017. (2017). Retrieved October 2, 2017 from https://linuxcontainers.org
[6] Rocket containers. 2017. (2017). Retrieved October 2, 2017 from https://coreos.com/rkt/
[7] Tiny core Linux. 2017. (2017). Retrieved October 2, 2017 from http://distro.ibiblio.org/tinycorelinux/
[8] CoreOS. 2017. CoreOS Container Linux. (2017). Retrieved October 2, 2017 from https://coreos.com/why

[9] Azure functions. 2017. (2017). Retrieved October 2, 2017 from https://azure.microsoft.com/en-us/services/functions/
[10] Google. 2017. Google App Engine. (2017). Retrieved October 2, 2017 from https://cloud.google.com/appengine/
[11] IBM. 2017. IBM platform as a service. (2017). Retrieved October 2, 2017 from https://www.ibm.com/us-en/marketplace/imi-managed-platform-as-a-service
[12] H. Luo M. Zhang and H. Zhang. 2015. A Survey of Caching Mechanisms in Information-Centric Networking. *IEEE Communications Surveys and Tutorials* 17 (2015), 1473–1499.
[13] Linux namespaces. 2017. (2017). Retrieved October 2, 2017 from http://man7.org/linux/man-pages/man7/namespaces.7.html
[14] Containers on mac. 2017. (2017). Retrieved October 2, 2017 from https://docs.docker.com/docker-for-mac/
[15] Containers on Windows. 2017. (2017). Retrieved October 2, 2017 from https://docs.docker.com/docker-for-windows/
[16] OpenLambda. 2017. (2017). Retrieved October 2, 2017 from https://open-lambda.org
[17] OpenWhisk. 2017. (2017). Retrieved October 2, 2017 from https://openwhisk.incubator.apache.org
[18] Osv. 2017. (2017). Retrieved October 2, 2017 from http://osv.io
[19] P. Shenoy P. Sharma, L. Chaufournier and Y. C. Tay. 2016. Containers and Virtual Machines at Scale: A Comparative Study. In *Proc. 17th International Middleware Conference (Middleware '16)*. ACM, 1–13.
[20] Apt package manager. 2017. (2017). Retrieved October 2, 2017 from https://wiki.debian.org/Apt
[21] Yum package manager. 2017. (2017). Retrieved October 2, 2017 from https://fedoraproject.org/wiki/Yum
[22] A. R. Raja R. Dua and D. Kakadia. 2014. Virtualization vs Containerization to Support PaaS. In *Proc. International Conference on Cloud Engineering*. IEEE, 610–614.
[23] Synergy research group. 2016. Synergy cloud survey. (2016). Retrieved October 2, 2017 from https://www.srgresearch.com/articles/2016-review-shows-148-billion-cloud-market-growing-25-annually
[24] Tcpdump. 2017. (2017). Retrieved October 2, 2017 from http://www.tcpdump.org/tcpdump_man.html
[25] Jia Wang. 1999. A survey of web caching schemes for the Internet. In *Proc. International Conference on Cloud Engineering*. SIGCOMM Computer Communications Revision 29, 36–46.
[26] Wireshark. 2017. (2017). Retrieved October 2, 2017 from https://www.wireshark.org