A Migration-Based Approach to Execute Long-Duration Multi-Cloud Serverless Functions

Boubaker Soltani Afifa Ghenai Nadia Zeghib

LIRE Laboratory

Constantine2 – Abdelhamid Mehri University, Constantine, Algeria

{boubaker.soltani, afifa.ghenai, nadia.zeghib} @univ-constantine2.dz

Abstract

Serverless Computing is emerging as an undeniable paradigm for the deployment of (multi)cloud applications. It is mainly characterized by the use of stateless loosely-coupled functions that are composed together to perform useful actions. This approach, contrarily to monolithic one, makes easier the maintenance and the evolution of the applications, since the functions can be independently revised and reprogrammed. However, one principle in Serverless computing is that function execution should be within a short duration (five minutes max in most Cloud provider platforms), after which the function is abruptly terminated even if it has not completed its task. Moreover, the max duration cannot be extended without a negative effect on the platform performance. This leads to prevent functions requiring longer time from being adopted as Serverless functions. This paper deals with this drawback. It proposes a distributed migration-based approach which promotes the execution of long-duration Serverless functions: each running function that reaches the maximum duration limit is repeatedly transferred to another cloud platform where it is carried on. In this aim, the migration-based system architecture, the migration technique and the migration algorithm are described. The proposed approach use is illustrated by a case study: a generic machine learning application built over the scientific platform ANTDROID.

1. Introduction

Cloud computing is now commonly used to describe the delivery of software, infrastructure and storage services over the internet. In this field, there are generally two parts that may be made available under the client control: the application code and the underlying infrastructure hosting that application. In this context, Serverless computing provides a great opportunity for developers seeking relief from the burden of infrastructure. This computing model allows building and running applications and services without having to manage infrastructure. In fact, Serverless Computing is an event-driven approach that abstracts the infrastructure management away from the client. Aspects like scalability, provisioning and fault tolerance are automatically handled by a Serverless platform, while the Cloud user focuses only on his functional code [Bald17]. This code comes in the form of a set of stateless functions that are agnostic of where are they going to be executed [Bald17]. Serverless platforms adopt a pay-per-execution billing strategy, i.e., contrarily to traditional Cloud platforms that host the client program in a listening running server (hence, the client pays as long as the server is running even if there are no requests), a Serverless platform does not start a server until a request is made. In this way, the client is billed for each invocation, meaning that no payment if no requests [Kuh17]. The Serverless architecture is relatively a new paradigm and only few pioneers have

In: Proceedings of the 3rd Edition of the International Conference on Advanced Aspects of Software Engineering (ICAASE18), Constantine, Algeria, 1,2-December-2018, published at http://ceur-ws.org

investigated this domain. Amazon AWS Lambda is a compute service that supports many languages like node.js, C#, Python and Java on AWS infrastructure. Source code is provided as ZIP file and deployed in a container that is allocated the necessary hardware resources. The combination of code, configuration and dependencies is what is called a *Lambda function* [Sbar17]. The author admits that a Lambda function can only run for a maximum of five minutes. Azure Functions [Kum17] is Microsoft's version of a Serverless platform. They provide a technique named the consumption plan to enhance function scalability during execution [Azr18]. A function in such plan is limited (by default) to five minutes, but this value can be increased up to ten minutes [Azr18]. Google Cloud Functions [Stig18] is another promising Serverless platform which limits the function to nine minutes max [Goo18]. Moreover, according to [Fox17], Serverless platforms are still hosting short- running functions due to the faced difficulty of scheduling long-running tasks and their SLA (service level agreements) management, but the authors advocate that it is possible in the future that Serverless platforms will host such tasks. Current procedure to execute long-duration workloads is not to provision them as Serverless functions (e.g. as a listening server instead) [Fox17] [Sbar17].

Consequently, in current Serverless platforms, the function execution should be within a short duration (five minutes max in most Cloud provider platforms), after which the function is abruptly terminated even if it has not completed its task. This leads to prevent functions requiring longer time from being adopted as Serverless functions.

In the aim to overcome this drawback, we take advantage from Multi-cloud paradigm features and propose a distributed migration-based approach which promotes the execution of long-duration Serverless functions.

The Multi-Cloud paradigm allows considering the distribution and provider-independence aspects. In fact, Multi Cloud [Groz14] is the usage of multiple, independent clouds by a client or a service. This paradigm is dedicated to accomplish the task of scheduling workloads to resources deployed across multiple clouds. A Multi-cloud is, however, not the same as Cloud federation which refers to a set of cloud providers that intentionally interconnect their infrastructures to make a unified resource pool [Groz14]. This distribution and independence allows us to apply our repeated migration-based approach to cope with the aforementioned problem.

The remainder of this paper is organized as follows: Section 2 discuses some works based on monolithic application decomposition that are relevant to our work. Section 3 is dedicated to the presentation of our migration-based system architecture. Section 4 illustrates our proposal via a case study. Finally, Section 5 concludes the paper with ongoing work.

2. Related Work

The investigated literature is classified switch to two categories: works that focused on well-designing functions at design time, i.e., from the beginning, be careful to program functions that do not surpass the maximum limit; and works that focused on decomposing pre-existing monolithic applications to smaller independent functions.

2.1. Design-Time Function Creation

In [Ader17], set of requirements assisting the development of microservices benchmark application for repeated empirical research in software architectures was provided [Ader17]. The e-commerce European software Otto is planned to be rebuilt from scratch switch to a microservice concept of Verticals (isolated parts from each other, no shared states or infrastructures) [Hass17]. Sascha et al. [Alpr15] propose a microservice-based architecture for business process modeling. An application in their model is decomposed to resources and entities, where a service is dedicated for each resource. Authors of [Tarm17] are tackling the limit of the traditional method for analyzing and designing microservice architectures by proposing a new one based on holistic analysis and design (from identifying the organizational identity and objectives to modeling the microservice dependency graph) [Tarm17]. Containerized microservice [Venu17] is a typical combination nowadays, due to the promising container technology solutions like Docker. Nevertheless, authors notice lack of performance consideration in microservice building and identify open issues and research guidelines. The Table 1 summaries the main advantages and limits of the design–time function creation approaches.

Table 1: Design-Time Function Creation Approaches Summary

Papers	Application domain	Advantages	Limits
[Ader17]	Software empirical research	Evaluative Requirements	Function execution time is once fixed in design-time. So, (in/de)creasing the max limit requires redesigning the function to meet the new time limit.
[Hass17]	E-commerce apps	Scalability and agility	
[Alpr15]	Business process modeling	Flexibility and agility	
[Tarm17]	Holistic analysis/design	More details are dragged	
[Venu17]	Containerized microservices	Lightweight virtualization	

2.2. Monolithic Application Decomposition

In [Kecs16], authors offer an approach based on ENTICE project's Image synthesis technique. The principle is to generate dynamic VM images and their required software dependencies (without irrelevant parts) [Kecs16]. Service Cutter [Gys16] uses 16 software's coupling criteria extracted from academia and industry practices. It uses domain models and other artifacts as coupling information sources; represents them as a graph; before estimating the parts that may be split to independent loosely-coupled services. Authors in [New15] give the outlines of decomposing a monolithic application by determining the different contexts that the monolith encompasses, moving the source code selectively to each context package, then choosing one of them to begin the separation process on it (priority criteria like: pace of change, security). In [Maz17], authors present a formal microservice extraction model to allow algorithmic recommendation of microservice candidates in a refactoring scenario. This model is similar to [Gys16] approach. It has two transformations: construction, which converts the monolith to graph and clustering, which extracts microservices from graph. The major advantages and limits of these approaches is summarized in Table 2.

Table 2: Monolithic Application Decomposition Approaches Summary

Paper	Decomposition based on	Advantages	Limits
[Kecs16]	VM images	Facilitated by Cloud platforms	• Requires manual intervention. • Decomposition may be impracticable. • Application altering may be undesirable.
[Gys16]	Design artifacts	Closer to user goals isolation	
[New15]	Context packages	Explicit domain isolation	
[Maz17]	Source code	Directly executable result	

2.3. Discussion

According to the above works summarized in Table 1 and Table 2, the decomposition may be impracticable or leads to fixed-duration functions (do not fit a frequently-changing limit environment). Moreover, it may require manual intervention, or may have a specific application domain. Hence, it seems that the decomposition has significant limits and it is not suitable to deal with long-duration Serveless functions. In the following, we propose an approach which does not imply any structure reforming (no redesign consideration concerns). It is mainly based on the migration principle in Multi-cloud computing. The key idea is that each running serverless function that reaches the maximum duration limit is repeatedly transferred to another cloud platform where it is carried on.

3. The Proposed Migration Approach

In this section, we propose a distributed migration approach based on Multi-Cloud Serverless architecture. First, we describe briefly this architecture that uses Docker containers at runtime for executing user functions. This platform is the distributed link that makes the different clouds find each other. Second, we describe the used migration-based technique which is accomplished in two phases: monitoring and migration.

3.1. Distributed Serverless Architecture

Docker [Vohr16] is a container technique that bundles the dependencies of a specific application in a single deployable unit called Docker image that can be executed at any node running Docker engine. Docker Containers can be connected to combine their resources for the software running within. Dockerfile is a collection that embraces all commands of configuration and downloads required API for the application to be ready for execution. Then it builds the container either from scratch or upon other base image (usually, a base container consists of an operating system and some very basic tools) [Vohr16]. Each function in the system needs a finite set of container images in order to be able to properly work (its software dependencies).

We adopt our distributed Serverless architecture shown in Figure 1 [Solt18]. This architecture is a Peer to Peer (P2P) system that spans several Cloud provider platforms (a Middleware for Multi Cloud). In this paper, we show the role of two needed components.

(a) Container Image Registry (CIR): a registry of container images for different programming language runtimes. This registry is extensible, so runtimes can be added/ removed from it and each node in the Multi Cloud system has its own CIR (for its functions) [Solt18].

(b) *The Function Manager (FM):* among its roles, it ensures the communication between different nodes in the Multi Cloud Serverless system. In this paper, we use it to migrate functions from node to another along with its all dependencies, so, no worry about their existence in destination [Solt18].

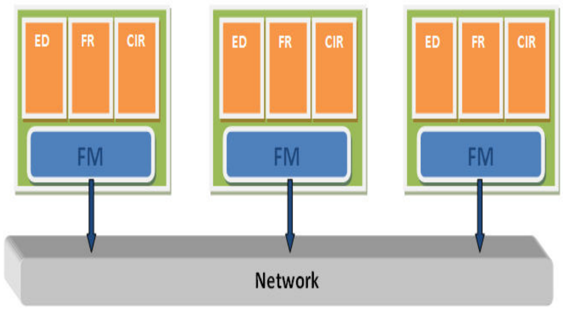

Figure1: Distributed Serverless Architecture[Solt18]

3.2. Migration-Based Thechnique

First, we define a variable in our system: *MaxLimit.* It represents the maximum allowed time (in seconds) for a function to be executed before it is scheduled for migration (abruptly terminated in traditional Serverless) and it is configurable by system administrators switch to their needs (five minutes by default).

The novel contribution is adding new specific images and functions to the *Container Image Repository (CIR)* and *the Function Registry (FM)* respectively. The new added elements details are as follows:

3.2.1. **Function Monitor Image (FMI)**: It is the runtime image for the monitoring task. We are choosing Prometheus as implementation for it. Prometheus [Prm1t18] is an open-source system monitoring and alerting toolkit and it can be configured with Docker [Prm2t18]. This image has the interfacing API for *monitor* (see next point 3.2.2) to use Prometheus. The Monitor API Docker image is represented in Figure 2.

3.2.2. **Monitor**: It is a specific function that has as role to count the consumed time during execution, for each function in the local node using FMI API and all functions that have ran more than or equals *MaxLimit* are passed to FUS for migration (see next point). The *monitor* is periodically invoked to ensure continuous function refresh (thirty second by default, but it is configurable).

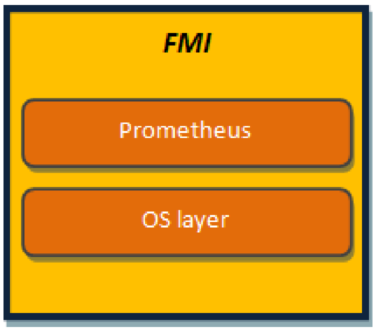

Figure 2: Monitor API Docker Image

3.2.3. **Function Urgent Scheduler (FUS):** this function is the scheduler that receives the list of functions that have reached *MaxLimit* (from *monitor*). It selects, for each one, a random node and informs the selected node for the function's dependencies (names only). The node responds back by asking for its missing images; the set passed by FUS to its local FM. Finally, the local FM migrates the function towards the remote FM all along with its missing images (delta images) and its current memory state. Algorithm 1 describes how FUS works.

```
Algorithm 1

Input: List<function> long
Output: List<function, nodeID, deltaImages> migr
For each f in long:
    (1) fmid = Selects random node
    (2) sends(fname, dependenciesNames) to fmid
    (3) receives (deltaImages) from fmid
    (4) adds (f, fmid, deltaImages) to migr
End for;
(5) returns migr to local FM
(6) FM adds the complete deltaImages to migr
(7) FM sends each f to its target
```

Figure 3: Urgent Scheduler Algorithm

4. Case Study

In the aim of showing the actual contribution of our approach and its usefulness, we consider an application with at least one long-running function. Our illustrating case study is a generic machine learning application of three Java-based Serverless functions built over the scientific platform called ANTDROID [Hadr12].

4.1. System Description

ANTDROID [Hadr12] is a federated SaaS available to scientists for sensing the activities of mobile users for their own experiments. The *scientist component* of the platform provides an interface allowing them to connect external services to the platform to extract and reuse dataset collected from their experiments' participants (*e.g.*, *visualization*, *analysis)* as shown in Figure 4.

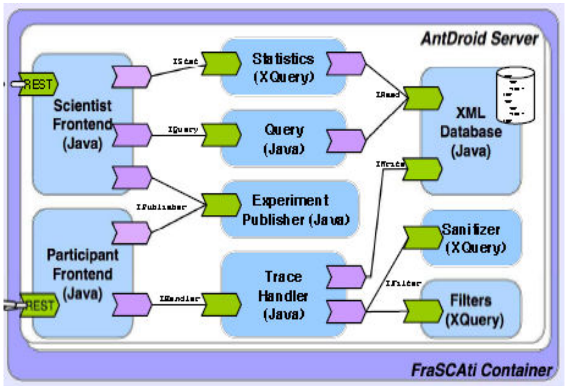

Figure 4: AntDroid Architecture [Hadr12]

This interface will be exploited by a special *docker* image of our Serverless system (stored

in CIR) to build the *AntDroid Connect Service or ACS* as shown in Figure 5.

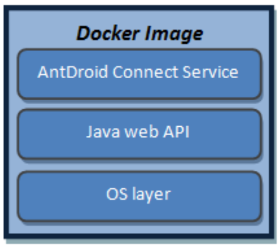

Figure 5: ACS Docker Image

ACS image is now the dependency runtime for all the application's three functions (as they do not need more than usual Java runtime environment functionality). The details of the functions as well as their minimum estimated time before termination are given in Table3. The three functions are executed sequentially one after another switch to their order in Table 3.

Table 3: Example of Functions Set

Function name	Role	Duration
Filter	Accesses the scientist collected dataset to extract the subset of valuable data.	20 min
Learner	Iterates the filtered data to acquire new knowledge (statistical indicators).	8 min
Viewer	Shows the calculated results in graphical plots (like Pie plots).	few seconds

4.2. Application of The Proposed Approach to AntDroid

Since the dataset collection may be very large and continuously fed by interactions (tens of gigabytes), the two functions *filter* and *learner* are estimated to be long, and hence, subjects for this paper procedure to be applied upon. The *filter* function, hence, will be migrated 3 times in the respective moments 5min, 10min and 15min after it starts. We show in the following figures how does that scenario happen (only once for filter for the sake of illustration). Table 4 discusses the three figures steps and meanings.

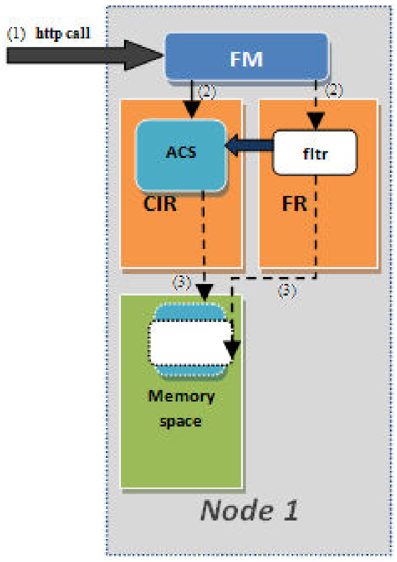

Figure 6: Migration Scenario (a)

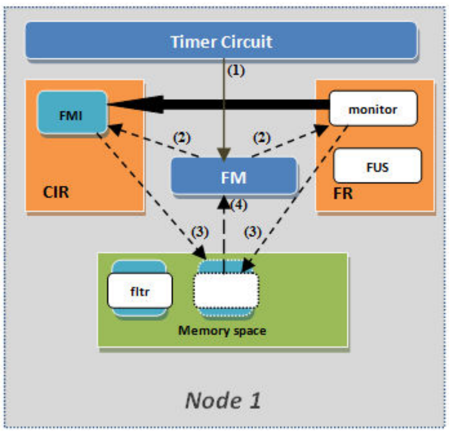

Figure 7: Migration Scenario (b)

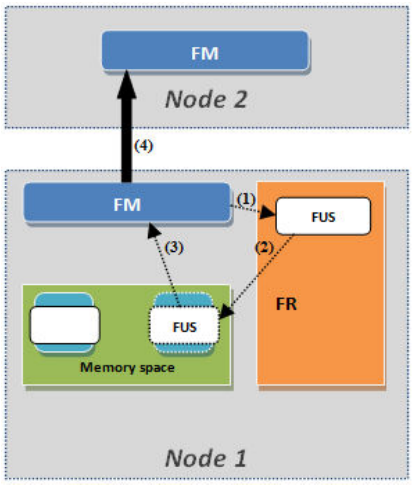

Figure 8: Migration Scenario (c)

Table 4: Migration Scenario Details

Figure part	Details
(a) Application starting point. As shown in Figure 6.	(1) The client calls the http link: fmid-url/filter. (2) The FM selects the *filter* function and its dependency *ACS* image. (3) The *ACS* is started and *filter* function now executes.
(b) After 5 min. As shown in Figure 7.	(1) The local timer notifies the FM (each 30 sec). (2) The FM prepares to start the new monitoring cycle. (3) The *FMI* image and *monitor* function are loaded to memory. (4) *monitor* detects that *filter* reached its max limit. It notifies FM.
(c) Migration phase. As shown in Figure 8.	(1) FM prepares the FUS scheduler function. (2) FUS is started within its container. (3) FUS executes its algorithm, determines the migration destination and informs FM. (4) FM migrates *filter* from Node1 to Node2.

From this presented illustration of our technique where the *filter* function was (same for *learner*) migrated as much as needed (3 times here) until its termination, the developer concerns himself less about time constraints and any factors affecting them during design-time (like unpredictable overloading of Serverless infrastructures, data volumes), focusing more on functional requirements of his application.

The above case study shows clearly that our proposed approach allows the execution of long duration Serverless functions. We intend to consider some key aspects to improve its efficiency. First, during the function migration, the destination cloud may not provide the same level of QoS (Quality of Service) desired by the migrated function (although all the dependencies are ensured) for many reasons, e.g. the new host may be in a period of high charge workloads. Another issue is the time necessary to execute the algorithm of this paper added to the required time to transfer the function state and its dependencies. This time can differ from Cloud-pair network links affected by parameters like geo-locations, links' bandwidth and size of transferred functions and libraries. The random aspect of the Algorithm1 will be studied as a more fair function distribution between nodes. An evaluation of the incurred latency of Algorithm1 will be conducted by measuring the diverse parameters mentioned in this paragraph, which gives more realistic dimension to this architecture. The main tool for this evaluation will be Semi-physical simulation which is essentially the adoption of virtualization techniques to simulate a big number of physical machines, while in reality there are only a few of them.

5. Conclusion

In this paper, we have tackled the limit of the Serverless principle: all functions should terminate their execution in a short period (typically five minutes). We have proposed a distributed migration approach based on Multi-Cloud Serverless architecture. This migration-based technique deals with Serverless functions that run for a long duration. Our technique consists of transferring a target function that reaches its limit in term of execution time, to another Cloud platform to be resumed there. By repeatedly applying this step, a function – no matter how long – finishes its task normally. This algorithm is designed to be fast and simple in term of scheduling decision, because its main goal is not optimizing resource utilization, but to save long functions from being cut off. We have illustrated the use of the proposed approach through the case study of a generic machine learning application built over the scientific platform ANTDROID. In future, we plan to investigate in the optimization of the scheduler algorithm. Another interesting work may concern considering more fair function distribution between nodes instead of mere randomness-based selection, all without losing the urgent aspect of the algorithm.

6. References

[Bald17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell,Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, Philippe Suter. Serverless Computing: Current Trends and Open Problems. Springer, December,2017. https://arxiv.org/pdf/1706.03178.pdf

[Kuh17] Jorn Kuhlenkamp and Markus Klems. Costradamus: A Cost-Tracing System for Cloud-based Software Services. Service-Oriented Computing: 15th International Conference, ICSOC 2017, Malaga, Spain, November 13–16, 2017, pp.657-672.

[Sbar17] Peter Sbarski with Forewords by Patrick Debois, Donald F. Ferguson. Serverless Architectures on AWS. Manning Shelter Island; 2017.

[Kum17] Praveen Kumar Sreeram. Azure Serverless Computing Cookbook. Packt Publishing Ltd;August,2017. https://azure.microsoft.com/mediahandler/files/resourcefiles/ff388333-6cd0-4b47-a33e3c3296a5141b/Azure_Serverless_Computing_Cookbook. pdf

[Azr18]https://docs.microsoft.com/en-us/azure/azure-functions/functions-scale, accessed in 5/23/2018.

[Stig18] Maddie Stigler. Beginning Serverless Computing, Developing with Amazon Web Services, Microsoft Azure and Google Cloud, Apress; 2018. https://www.apress.com/us/book/9781484230831

[Goo18] Google Cloud Functions documentation: https://cloud.google.com/functions/quotas, accessed in 5/23/2018.

[Fox17] Geoffrey C. Fox, Vatche Ishakian, Vinod Muthusamy and Aleksander Slominski (IBM). Report from workshop and panel on the Status of Serverless Computing and Function-as-a-Service (FaaS) in Industry and Research. First International Workshop on Serverless Computing (WoSC), June 5-8, 2017, Atlanta,GA,USA.

[Groz14] N. Grozev and R. Buyya, “Inter-Cloud architectures and application brokering: taxonomy and survey,” in Journal of Software: Practice and Experience, vol. 44, no. 3, pp. 369–390, Mars. 2014.

[Ader17] Aderaldo C., Mendonça N., Pahl C., Jamshidi P., Benchmark requirements for microservices architecture research. In Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering, ECASE, May 2017, Buenos Aires, Argentina, pp. 8-13.

[Hass17] Wilhelm Hasselbring and Guido Steinacker. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), 5-7 April 2017, Gothenburg, Sweden.

[Alpr15] Sascha Alpers and Christoph Becker. Microservice Based Tool Support for Business Process Modelling. Enterprise Distributed Object Computing Workshop (EDOCW), IEEE 19th International, Sept. 2015.

[Tarm17]Ahmad Tarmizi Abdul Ghani, Mohd. Shanudin Zakaria. A Method for Analyzing and Designing Microservice Holistically. (IJACSA) International Journal of Advanced Computer Science and Applications, Vol. 8, No. 12, 2017.

[Venu17]M.V.L.N.Venugopal. Containerized Microservices architecture. International Journal Of Engineering And Computer Science ISSN:2319-7242 Vol.6(11), pp. 23199-23208, November 2017.

[Kecs16] Kecskemeti G., Marosi A., Kertesz A., The ENTICE approach to decompose monolithic services into microservices. In International Conference on High Performance Computing & Simulation (HPCS 2016), July 18-22, 2016, Innsbruck, Austria, pp. 591-596.

[Gys16]Michael Gysel,Lukas Kölbener, Wolfgang Giersche and Olaf Zimmermann. Service Cutter: A Systematic Approach to Service Decomposition. 5th European Conference on Service-Oriented and Cloud

Computing (ESOCC), Sep 2016, Vienna, Austria. Springer International Publishing, LNCS-9846, pp.185-200, 2016.

[New15] Sam Newman. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, Feb. 2015 (chapter 5).

[Maz17] Genc Mazlami, Jurgen Cito, Philipp Leitner. Extraction of Microservices from Monolithic Software Architectures. The 24th IEEE International Conference on Web Services (ICWS 2017) June 25-30, 2017, Honolulu, HI, USA

[Vohr16] Deepak Vohra. Pro Docker. Apress, ISBN-13 (pbk): 978-1-4842-1829-7, ISBN-13 (electronic): 978-1-4842-1830-3; 2016.

[Solt18] Boubaker Soltani, Afifa Ghenaï and Nadia Zeghib. Towards Distributed Containerized Serverless Architecture in Multi Cloud Environments. The 15th International Conference on Mobile Systems and Pervasive Computing (MobiSPC) Gran Canaria, Spain, August 13-15, 2018.

[Prm1t18] Prometheus - Monitoring system & time series database. https://prometheus.io/, accessed 27/05/2018.

[Prm2t18]https://docs.docker.com/config/thirdparty/prometheus/, accessed 27/05/2018.

[Hadr12] N. Haderer, R. Rouvoy and L. Seinturier, "AntDroid: A Distributed Platform for Mobile Sensing," Research Report RR-7885, Inria, Feb. 2012.