BND: A Free-to-Decompose Function-as-a-Service Application Framework

ABSTRACT

Decomposition is a double-edged sword in micro-service architecture, which helps to make full use of cloud computing, and is also the main challenge in the development and evolution of distributed systems. Typically, cloud computing service providers use "bottom-up" Serverless solutions, which focus on effectively leveraging the infrastructure. Nevertheless, the problem of decomposing is still far from solved, even worse due to better granular computing resources. On the contrary, this work from the "top-down" viewpoint of software development, introduces BND, a Function-as-a-Service application framework, which is a "free-to-decompose" solution by decoupling compile-time and runtime properties of function and service, making the instance of function can be invoked in a unified way no matter it is a inner function or a remote service, so that it can be transparent to developer while decomposing monolithic application or existing distributed system.

KEYWORDS

Cloud Computing, Micro-Service, Function-as-a-Service, FaaS, Free-to-Decompose

1 Introduction

Recently, micro-service architecture has been increasingly noticed in academia and applied in more and more companies for different kinds of businesses, including e-commerce[1], IoT[2], etc. According to the classic software architecture model[3], its emergence could be an inevitable consequence of the development of cloud computing, which is characterized as virtualization, distribution and dynamically extendibility[4]. Also, due to its discrete and heterogeneous nature, micro-service architecture has been considered as an ideal match with cloud computing and appears poised to replace SOA[5, 6, 7, 8, 9, 10, 11].

The idea of micro-service architecture is to decompose monolithic application into a set of micro-services[6], which are comparatively isolated, light-weigh modules or functions, communicating through message systems, RPC protocols or HTTP/HTTPS protocol in RESTful style. Further, it can be separated into two major categories according to the communication mechanism, which are event-driven and service-oriented respectively, and sometime compounded. While decomposing gives it more than ever effective leverage from cloud computing techniques, digging out more flexibility, modularity, and evolvability for large scale distributed systems, it is a double-edged sword which also bring in problems of service design, continuous delivery, monitoring and scheduling etc.[5, 14, 13], and making contemporary practices far from mature.

Some studies have shown the major impediment of migrating to micro-service architecture is none other than decomposing, which is also connoted as “Decoupling”[10, 12]. Certain popular theories may help in the designing stage, such as “Domain Driven Design”. But in practice, it is still not easy to tell whether a design is good or not. And micro-service systems are typically constructed and managed with certain indispensable architectural elements, which in charge with service registering, naming, searching, balancing, authorizing etc., making it too costly to have everything prepared at the beginning of any agile project, and requiring too much expertises for the startup team[8]. Moreover, these architecture-related-only elements, are usually hard coded in the source code, which causing the cohesion of compile-time and runtime properties, thereby the separated functions or modules have to be converted into services during decomposing. Thus, no matter what kind of decoupling scheme has been made, the changes committed to system always involve mammoth tasks, affecting almost every aspect of system, and eventually, very difficult to test[8]. It seems we have fallen into a vicious circle, in which neither do we able to transform an existing system, nor to build a new one, in a convenient way.

The key of solving this problem is to decouple runtime properties from compile-time, to ensure functions or modules unaware of its role transition during decomposing. Which means they could have two different runtime states: 1) Inner function or module resides in the same process, or 2) Remote services running on different processes or different hosts; But they will always be invoked in the same way in source code throughout the system and lifecycle.

This work proposes a “free-to-decompose” framework (BND) implementing FaaS, to keep source code untouched when separating monolithic application or part of system into a set of micro-services, so that the decomposing could be performed in any arbitrary way, and keeping the system as congruent as before.

The rest of this paper is structured as follows. Section 2 introduce most recent research works about FaaS. Section 3 introduces the motivation behind this work. Section 4 presents the design of the abstraction of a middle level layer between compile-time and runtime. Section 5 presents the design of adaptative architecture based on the abstraction. Section 6 presents an implementation based on the design. Finally, Section 7 summarizes this work and outline items for future work.

2 Related Work

The way of building micro-services from isolated functions upon cloud computing infrastructure, is called Function-as-a-Service (FaaS), which was first proposed by Hook.io in 2014, then followed by almost every giant cloud computing service provider, including Amazon AWS Lambda, Google Cloud

Functions, Microsoft Azure Functions, Alibaba Cloud Function Compute, etc.

However, only a few research works concerned about FaaS so far. Fritz, et al.[15] believe FaaS has generated significant interest from out-sourcing computation. They presented a general explanation of FaaS concept and implemented a FaaS architecture (S-FaaS) to provide strong security and accountability guarantees backed by Intel SGX. Michal and Ioannis[16] briefly introduced nowadays Serverless architecture and proposed Named Function as a Service (NFaaS), a framework that extends the Named Data Networking architecture to support in-network function execution. It is very interesting to see FaaS and Serverless being applied in network related research work.

Also, FaaS has not flourished in industrial practices, mostly due to "bottom-up" viewpoint from cloud computing service providers, in which FaaS is closely bound up with Serverless architectures, hindering its application in large-scale business development.

3 Motivation

Compared to "how", "why" is relatively more important when creating something new. Thanks to cloud computing, thousands of online applications bloomed rapidly in recent years, serving the everyday life of millions even billions of people, dramatically increasing social efficiency and life quality. However, in terms of large-scale application developing, the impassable gulf still exist between the capacity of giant technology companies and startup team. And no sign of narrowing the gap, even though the cloud computing service providers have made significant progress in providing much more flexibility and scalability by granulating computing resources into as small granule as possible.

From the view of service provider, the cost truly dropped down in building simple structure applications. It won't be a big deal any more to take a try at building systems for startup businesses. But from the view of developing and evolving large-scale distributed system, some critical timepoints usually remain unsustainable for startup teams, such as when the business booming too fast in a sudden, and sharply perplexing business model, which is a common phenomenon of online businesses. As from the view of software development, these fine granulated resources are scattered and require immense efforts to glue them together.

If given the ability to fully utilize cloud computing while no need to concern about its complexity, brilliant startup ideas may survive a bit easier than before, and large-scale micro-service architecture systems may also evolve more quickly with lower cost.

4 Design of Abstraction

The idea is to provide a middle level agency between the software and its runtime infrastructure. As shown in Figure 1, firstly, the cohesion of runtime and compile-time properties is decoupled by using an agency in uniform to take control the invocation of functions and services at compile-time, and the rest is to adapt the agency at runtime with various infrastructure environments (We use term "agency" to represent the instance of framework in the rest of this paper).

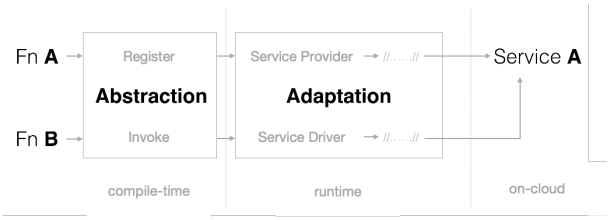

Figure 1: A middle-layer abstraction of compile-time and runtime in cloud computing

As a usual habit of engineering to think about what the realization would be beforehand, it would be a little weird to invoke a function using a third-party tool instead of invoking directly when designing or coding. But there could be plenty of ways to provide compile-time support transparently, such as compilers with this specific capability. So it would be a little rambling to concern specific implementations at the beginning.

Therefore, we will present the idea of middle level abstraction first to provide a rudimental sense, and then introduce an layered architecture based on this abstraction. We will also provide an implementation in later section.

The target of middle level abstraction is to provide an agency at the compile-time which is compatible with the runtime transforming between inner function and remote service. So, this abstraction resides in the compile-time, which is a pivotal component in the software architecture.

From the agency view, it is unsure whether a function would be deployed as a micro-service sometime in the future. So, it has to assume every function will become a micro-service eventually. But it also has to leave the runtime properties as undefined at compile-time. Because these properties only belong to runtime, the agency has no way to know anything about them.

As for micro-service, some properties actually can be moved backward into compile-time. They are used to define service identity. In production system, service is identified according to the communication mechanism. It could be an URL and operation method in HTTP protocol, or just a hash string in Peer-to-Peer system. But they all represent the nature of identity, which is equivalent with function name at compile-time.

For this reason, the agency should name a function with a identity which could be translated into runtime identity, namely an function handler or a remote service address. Thus, as shown in Figure 1, it should support two actions: 1) register a function with a unique identity in the abstraction layer; and 2) invoke a function or remote service regardless of its runtime state, just according to its identity.

5 Adaptative Architecture

At runtime, the agency is responsible of identifying functions locally or finding their instances from remote services. And it's also responsible of delivering the computing tasks by invoking local functions or calling remote services. Like modern compilers, layered architectures are better suited to a wide variety of runtime environments.

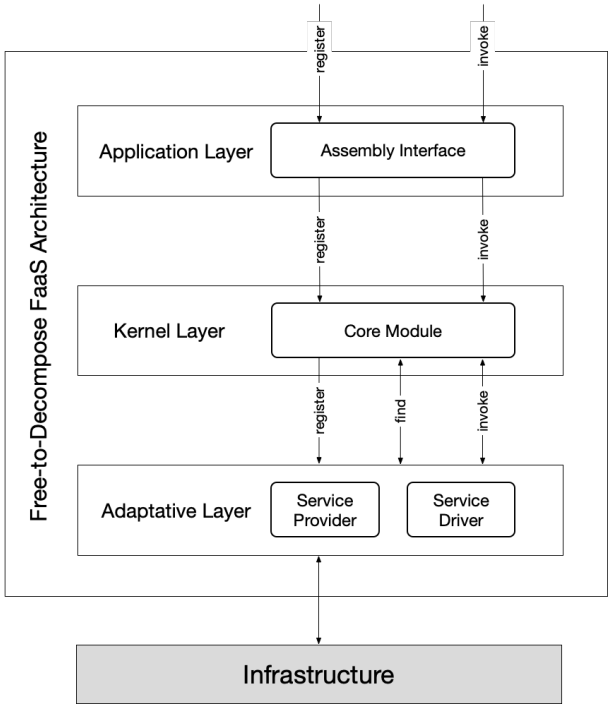

Figure 2: A free-to-decompose FaaS architecture

5.1 Architecture Layers

We introduce the architecture in a "top-down" view of developer, as shown in Figure 2. While the architecture is not necessarily contain only these three layers. Take the *Kernel Layer* as an example, it could be further separated into several sub-layers for specific purposes. Here we skip those possible extensions just to keep the idea as simple as possible.

- **Application layer:** provide compile-time interfaces, literally by a third-party tool, or transparently by the support of compiler.
- **Kernel layer:** track function or service instances by their identities, organize them with inner representations at runtime, and process the data between invokers and their target instances.
  Extra utilities may happen at this layer, for example, security, performance, load balancing, and dynamic routing could be controled or analyzed from the passing-by data. It's nice to support third-party plugins or modules by providing standard interfaces at this layer to do these jobs, but it is out of the scope of this paper right now.
- **Adaptation layer:** translate invocations into runtime requests, and parse their runtime responses into intelligible information for the kernel layer. This layer consists of two complementary components, which always appear in pairs:
  - **Service provider:** when a function is deployed as a service, this component is responsible of building that service independently, or cooporating with third-party frameworks, such as Zookeeper or some kind of message queue, to setup an online service for that function. After setting up the service, it will generate a description of that service.
  - **Service driver:** call a remote service according to its description, send data received from upper layer, and parse the response into intelligible information and return to upper layer.

5.2 Interfaces

Also, in order to keep the idea simple, here we just list basic interfaces necessary for explaining the idea, which are illustrated in Figure 2.

Interfaces between Application and Kernel layer, which are also exposed to developer:

- **register:** set an identity for a function in the kernel, thus the kernel can map this identity to its runtime handler or service description, which is generated by specified service provider(s) according to its runtime deployment.
- **invoke:** request a specific computing task, then the kernel will find out an appropriate instance to perform the task, and return the result to invoker accordingly.

Interfaces between Kernel and Adaptation layer, which are hidden inside of the framework:

- **register:** set up a service instance using a specified service provider, and storing the service description under the identity of that service. This could have two strategies: 1) cooperate with existing service management elements in the network; 2) build the service on its own.
- **find:** search for an appropriate service on the network with the help of available service providers and drivers. This also could have the same strategies as "register" interface. And this is a bi-direction interface, which means each layer provide this interface for another.
- **invoke:** call the selected service instance using specified service driver according to its description, which is also a bi-direction interface.

5.3 Runtime Sequences

We use sequence diagram of **invoke**, as Figure 3 has shown, to illustrate how the architecture will work at runtime to adapt various runtime settings and environments. There are 6 different roles in the diagram, which are *Program, Assembly Interface, Local Core Module, Local Node, Remote Node, Remote Core Module* respectively. In these roles, the *Assembly Interface* is the instance of *Application Layer*, *Core Module* is the instance of *Kernel Layer*, and *Node* denotes the instance of *Adaptation Layer*. We use *Local* representing the local instance, and *Remote* representing the corresponding instance resides somewhere else in the network.

S1: Invocation is triggered in *Program*, through the interface *invoke* between *Application Layer* and *Kernel Layer*.
S2: *Assembly Interface* take over the invocation, translate it into inner representation, like taking a note of incoming request, and then pass it to *Core Module*, which is consisted of bundle of inner handlers, and necessary inner data storage in-memory or outside.
S3: According to the identity of target function or service, and the specific request parameters, *Core Module* search for the target function locally first; and/or search in the network if needed.
S4: If the target function was found locally, then *Core Module* execute the computing task using the target function handler, and return the result to *Assembly Interface*.
S5: When searching the network, *Core Module* communicate with *Local Node*, which is local instances of *Adaption*

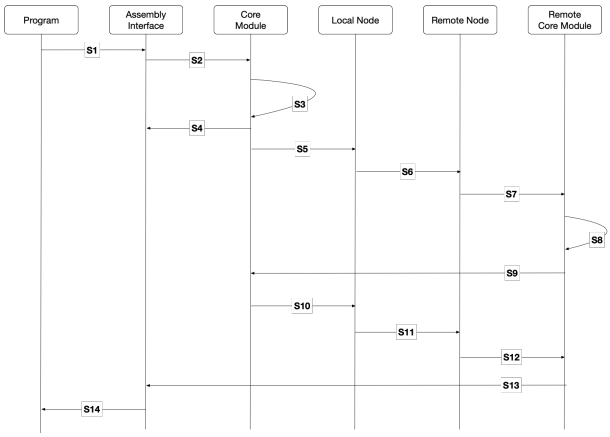

Figure 3: Sequence diagram of *invoke*

Layer, through the interface *find* between *Kernel Layer* and *Adaptation Layer*.

S6: When *Local Node* receive the search request, it may communicate with a third-party element such as a service management center, or permeate though the request through out the network by communicating with other instances in the network, which is the *Remote Node*.

S7: When *Remote Node* receive the search request, it pass the request to its *Core Module* through the bi-direction interface *find*.

S8: If *Remote Core Module* can not find the target service locally, under the control of some specific request parameters, it may continue permeating the request in rest part of the network.

S9: *Remote Core Module* send back the result of searching through the original path, to *Local Core Module*. If it found the target service successfully, the returned result should contain the description of that service.

S10: *Local Core Module* invoke the remote service according to its description, by passing the request to *Local Node*.

S11: *Local Node* assemble runtime request according to the service description, and send the request using the specified *Service Driver* in the service description.

S12: *Remote Node* received the request of invocation, and pass it to its *Core Module* through the bi-direction interface *invoke*.

S13: *Remote Core Module* return the result of invocation through the original path, to *Local Core Module*, after extra processes if needed, finally returned to *Assembly Interface*.

S14: *Assembly Interface* parse the result of execution and return an intelligible back to *Application*.

It's obvious that the second remote request during one time invocation is unnecessary. The interfaces of *find* and *invoke* should be merged into one. Actually, we did merge them in the implementation. Here we present them separately to clarify the basic processes in detail.

6 Implementation

We implemented the framework using Javascript language, which can be compiled into a server side verion with the help of Node.js, and a client side version which can run directly within the browsers. However, it's not necessary to support everything using the same code base, and it does not necessarily mean Javascript is a more suitable programming language. The reasons we choose it are:

- Most FaaS service providers currently use Javasript to build micro-services.
- Javascript is widely used by agile product development teams.
- We want to provide an universal view of building applications on distributed computing resources, and deliver it as soon as possible.

6.1 Architecture Design

We implement the architecture design and go a bit further, as shown in Figure 4. In the *Application Layer*, we provide not only basic interfaces, but also including some special interfaces:

- **join**: we organize the network without third-party help, by building a self-orgnaized and decentralized network. While this interface may literally suggests at least a central element in the network, it does not mean the entrance should be the central element. Actually the entrance could be any element running in the network, and is not necessarily remain functional after this action.
- **configure**: for simplicity, we leave the identity responsibility to developer, to name their application code base and services in a hierarchical style, together with their versions. What's more, we provide the ability to deploy specific services by reading from configurations through this interface.

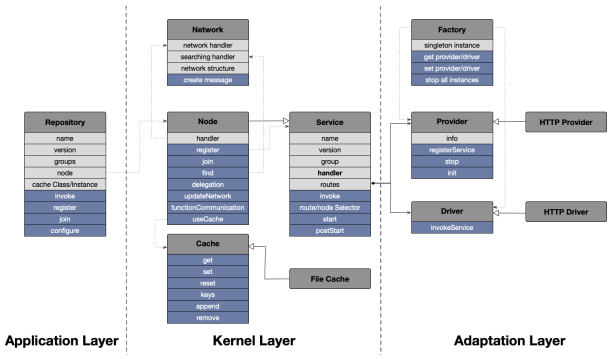

Figure 4: Architecture of BND

In the **Kernel Layer**, we provide:

- inner representations: *Network, Node, and Service.*
- in-memory data structure: *Cache* and its extension *File Cache* as outside storage, to cope with the communication and data sharing problems in multi-thread or multi-process scenarios.

In the **Adaptation Layer**,

- we use a factory paradigm to provide singleton instances of *Service Provider* and *Service Driver*.
- also implement a pair of *Service Provider* and *Service Driver* of HTTP protocol, which are in charge of building and calling HTTP Services.

6.2 Function Style Communication

This work also provide an ability of **Function Style Communication**, which has been applied as a patent, as illustrated in Figure 5. Which means the runtime settings of a function instance are transmitted together with the invocation parameters, enabling:

- an executable instance of a function can be transmitted through the network, which could also be a return value of previous service of a function.
- hiding details of communication.
- dynamic chain of communication.

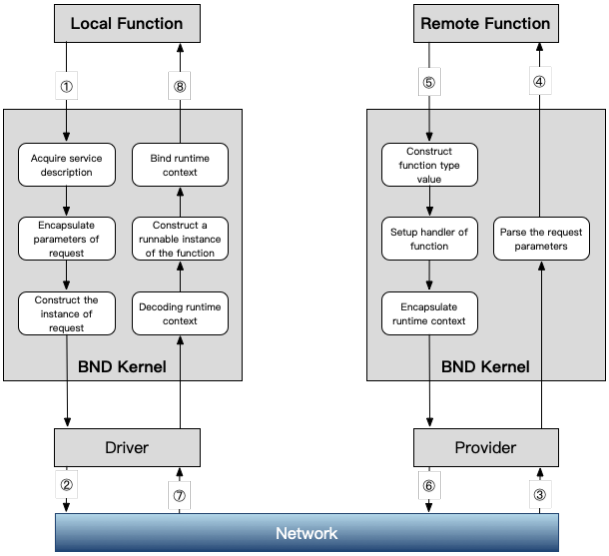

Figure 5: Function style communication

6.3 Universal Application View (MVC)

We provide an universal application view in the MVC style, as shown in Figure 6, with the help of some extra components.

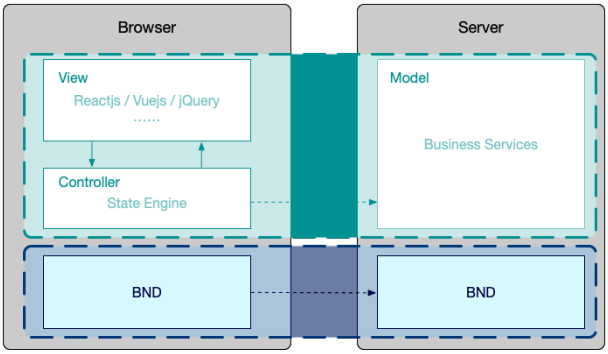

Figure 6: A universal application view (MVC)

In this way, developer can focus on building business logic just as building a single monolithic application, without concerning their runtime complexity.

6.4 Service Decomposing & Deployment

Services can be splited at function level, in any arbitrary way just according to the configuration files in the runtime environment, as Figure 7 shown. In practice, the service can be decomposed as wish just using *continuous integration (CI)* tools, while CI has become one of the basic services of cloud computing infrastructures in recent years.

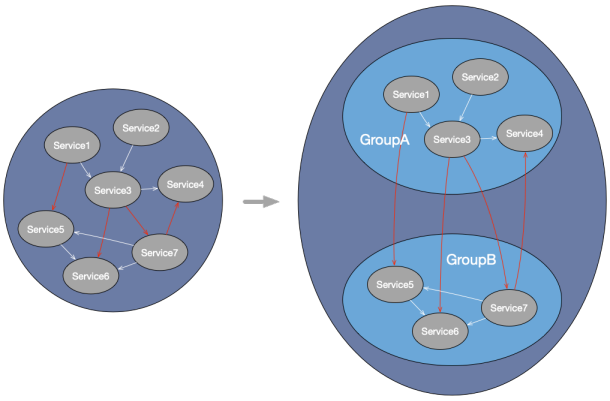

Figure 7: Decomposing services according to configuration

7 Conclusion & Future Work

While due to lacking standard evaluation of architecture, it's not cogent to simply say this architecture or framework is better than someone else. And we know this implementation is still at the prototype stage, it's too early to compare with existing distributed system architecture in the facade of the stability, availability, security, performance and so on. By far, the implementation can only demonstrates the idea does work, that we have successfully decoupled the compile-time and runtime properties by hiding the communication details and the runtime heterogeneity and dispersity, and making it as simple as building monolithic applications when building FaaS microservices.

We hope to model the efficiency of software architecture, however, in practice, the overall effect usually looks complicated. Because software architecture is only one part among numerous influence factors, such as:

- hardware architecture, infrastructure service, network topology;
- the model, frequency and scale of data, transactions, businesses;
- labor quality, organization and pattern of cooperation.

We can only claim, if given a mature cloud computing infrastructure support, it will provide the possibility for startup teams to catch up with giant technology companies on building large-scale distributed online systems, which is still remained to be confirmed in future practices.

The implementation of this architecture is fully runnable within browsers, so we also tried to build services on the client side, namely, inside the browser. However, these services need proxy from the server where the browser is accessing, and it require a lot more work to be done to enable Peer-to-Peer routing among client-side services. In this paper, we mainly focus on the view of developer, while the compiler relative work should be considered as a way of improvement of efficiency. And most importantly, as a framework of distributed system targeting large-scale online businesses, We need to fully test all of its quality aspects, build up a baseline compared with contemporary frameworks and improve gradually based on it.

REFERENCES

[1] W. Hasselbring and G. Steinacker, *"Microservice Architectures for Scalability, Agility and Reliability in E-Commerce"*, in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden: IEEE, Apr. 2017, pp. 243–246.

[2] Alexandr Krylovskiy, Marco Jahn, and Edoardo Patti, *"Designing a smart city internet of things platform with microservice architecture"*, in 2015 3rd International Conference on Future Internet of Things and Cloud (FiCloud), IEEE, 2015, pp 25–30.

[3] Dewayne E. Perry, Alexander L. Wolf *"Foundations for the Study of Software Architecture"*, ACM SIGSOFT Software Engineering Notes, Volume 17 Issue 4, Oct. 1992, pp. 40-52.

[4] S. Zhang, X. Chen, S. Zhang and X. Huo, *"Cloud Computing Research and Development Trend"*, Future Networks, International Conference on(ICFN), Sanya, Hainan, China, 2010, pp. 93-97.

[5] N. Dragoni, S. Giallorenzo, A. Lluch Lafuente, M. Mazzara et al., *"Microservices: yesterday, today, and tomorrow"*, Present and Ulterior Software Engineering, pp. 195-216.

[6] Z. Xiao, I. Wijegunaratne and X. Qiang, *"Reflections on SOA and Microservices"*, 2016 4th International Conference on Enterprise Systems (ES), Melbourne, Australia, 2017, pp. 60-67.

[7] Claus Pahl, Pooyan Jamshidi, *"Microservices: A Systematic Mapping Study"*, Proceedings of the 6th International Conference on Cloud Computing and Services Science, Volume 1 and 2, pp. 137-146.

[8] Dmitry Namiot, Manfred Sneps-Sneppe, *"On micro-services architecture"*, International Journal of Open Information Technologies, vol. 2, no. 9, 2014.

[9] Tomas Cerny, Michael J. Donahoo, Michal Trnka, *"Contextual Understanding of Microservice Architecture: Current and Future Directions"*, ACM SIGAPP Applied Computing Review archive, Volume 17 Issue 4, December 2017, pp. 29-45.

[10] Davide Taibi, Valentina Lenarduzzi, Claus Pahl, Andrea Janes, *"Microservices in Agile Software Development: a Workshop- Based Study into Issues, Advantages, and Disadvantages"*, Proceedings of the XP2017 Scientific Workshops, Article No. 23.

[11] Mario Villamizar, Oscar Garces, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil, *"Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud"*, in Computing Colombian Conference (10CCC), 2015 10th, IEEE, 2015, pp 583–590.

[12] D. Taibi, V. Lenarduzzi and C. Pahl, *"Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation"*, in 2018 IEEE Cloud Computing, vol. 4, no. 5, pp. 22-32.

[13] M. Fazio et al., *"Open Issues in Scheduling Microservices in the Cloud"*, in 2016 IEEE Cloud Computing, vol. 3, no. 5, pp. 81–88.

[14] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, *"Multi-objective scheduling of micro-services for optimal service function chains"*, in 2017 IEEE International Conference on Communications (ICC), May 2017, pp. 1–6.

[15] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, Michael Steiner, *"S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX"*, arXiv:1810.06080, Submitted on 14 Oct 2018.

[16] Michał Król, Ioannis Psaras, *"NFaaS: named function as a service"*, Proceedings of the 4th ACM Conference on Information-Centric Networking, pp. 134-144.