The DSL for composing functions for FaaS platform

Nikita Gerasimov
Mathematics and Mechanics Faculty
Saint Petersburg University
St. Petersburg, Russia
n.gerasimov@2015.spbu.ru

***Abstract*—This article describes the problems that occur when using the Function as a Service model: the complexity of the centralized description of the separate function interaction within the whole system and the possibility of dependent components interface divergence during a process of their development. We propose a domain-specific language called Anzer as a solution to these problems which enables to describe the types of data transmitted, the composition of functions, the semantics of their interaction and logic of error handling. To check the consistency and compliance of the declared and implemented types, software has been produced making it possible to automate the creation of new functions and maintain their integration with others at all stages of system development and support. The language and software in combination with each other prevent errors associated with a mismatch between input data format expected by the function and actually transmitted one. All this enables to simplify and speed up the process of developing systems based on the concept of Function as a Service.**

***Index Terms*—serverless, faas, static typing, function composition, domain-specific language**

I. Introduction

The development of the Internet, the emergence of a large amount of data and process control automation have led to an increase in the software modularity. An earlier approach to partitioning programs into components was service-oriented architecture (SOA), which is characterized by getting particular sets of functions separated into independent modules. Each module is responsible for a certain range of tasks. The interaction is carried out either using enterprise service bus or RPC API.

The next iteration of the development of modular distributed software architectures is microservice architecture. The approaches are very similar at first glance, but the difference is in details. Microservices usually tend to perform much fewer functions and have fewer dependencies, which simplifies scaling [1]. In addition, they provide one or more API methods rather than complex RPC interfaces covering most of the subject area.

Microservice architecture imposes certain requirements on the infrastructure: deploy automation, automation of testing, infrastructure for service discovery.

Without preliminary preparation of such an environment, developing large systems in such a paradigm leads to greater costs than developing a monolithic system [2].

In 2014 Amazon introduced AWS Lambda serverless platform (serverless computing platform) within Amazon Web Services cloud platform. The main logical unit in it is a function, which, in fact, is also a microservice, but with some differences [3]:

- A microservice is a standalone application having necessary libraries. A function is code that implements only necessary logic. Function start, its initialization, connection to a database are carried out by a platform, which means a function itself is not self-sufficient.
- A microservice can be run separately, and a function can often be started only within a framework of a serverless platform.
- A microservice typically runs in daemon mode, responding to incoming requests. A function is often run only when it is requested to execute it.

In summary, a function is similar to a microservice but often is not self-sufficient, does not store the state, uses capabilities of the platform to run and communicate. Such peer-to-peer computing platform called FaaS (Function as a Service).

In addition to Amazon, cloud FaaS platform services are provided by Google (“Google CloudFunctions”), Microsoft (“Azure Functions”) and IBM (“IBM Cloud Functions” based on the “Apache OpenWhisk” platform). In addition to cloud providers, there are self-hosted solutions: Apache OpenWhisk, Fission, OpenFaaS and other.

Simple applications with a web interface, or IoT (Internet of things) systems, often use separate, independent functions to perform actions when certain events occur. However, systems in the FaaS platform can be built through the composition of functions. For example, in a subsystem that processes data, one component passes intermediate results to another in a chain. One of the obvious applications of such a composition is ETL (Extract, Transform, Load) processes. Using a FaaS platform it is easy to implement a similar data conversion scheme if each step is allocated to one or more functions. The functions work in parallel, independently, do not store the state, return result of the work or an error message.

The benefits of serverless computing:

1) No need for infrastructure support (in case of using cloud-based FaaS solution providers).
2) No need to implement supporting code, such as logging, connecting to databases etc.
3) Simple scaling of individual components rather than a system as a whole.
4) No downtime costs because functions run on demand.

5) The need for a competent division of logic into modules, based on the concept definition.

The use of the solution mentioned above also has its drawbacks:

1) Lower transparency of the system compared to a monolithic application.
2) Difficulties with debugging of functions and a system as a whole.
3) There is no commercial solutions or standards in the field of systems testing that built in the FaaS platforms.
4) An unresolved issue of resources caching that should be initialized each time the function runs (for example, database connections).

Another problem that may arise during the development and support of such a system is the control of interaction and compatibility of individual functions within the whole system. To solve this problem some FaaS cloud service providers make it possible to describe the order of functions run or their compositions. However, existing platforms do not provide proper type checking. There is a situation when a monolithic application is divided into separate logical blocks smaller even than microservices, and there is no way to guarantee these blocks will work together consistently and correctly. Condition may occur in which some of the components of the system provide the updated interfaces, and dependent components wait for outdated interfaces; that situation will result in an error in the process of operation. This fact is a disadvantage of serverless solutions in comparison with traditional methods of systems development: monolithic and service-oriented approaches.

In the article, we consider existing solutions for function composition in the FaaS platforms (sec. II), suggest a way to describe the composition of functions and automatically check their compatibility by means of DSL (sec. III) and a software complex that extends opportunities of the FaaS platform "Apache OpenWhisk" [4]. The use of such an extension is assumed to reduce number of errors associated with mismatch between function types during developing.

II. Alternative solutions

Serverless computing is a young approach, not yet widely known and not widely used. As a result, there are only a few solutions providing the composition of functions.

Amazon provides AWS State Machine with its own language, Amazon States Language describing a sequence of functions to run (AWS Step Functions) within a specific task [5]. Language and platform capabilities enable:

1) Define the order which functions should be started in.
2) Handle errors.
3) Set the number of data processing retries in case of error.
4) Run multiple processes in parallel.
5) Set the conditions for the launch of certain functions based on transmitted data.

Sequences of invoking functions are not functions themselves, they are state descriptions made by an external environment.

In addition to Amazon, IBM provides the ability to describe the composition of functions using a developed JavaScript library and "IBM Composer" functionality built in IBM CloudFunctions [5]. The solution also makes it possible to describe conditions, number of function retries and some other features. An important difference between IBM Composer and AWS State Machine is that composition in the former is also a function that can participate in composition.

Microsoft provides 2 mechanisms for the composition of functions in its "Azure Functions" platform: "Azure App Logic" and "Azure Durable Functions". Just as "IBM Composer", the mechanisms are built in the platform and provide an opportunity to describe conditions, cycles, number of retries, etc. [5].

If choosing among serverless self-hosted platforms, Fission and Fn enable to describe processes as the composition of functions using "Fission Workflows" and "Fn Flow" respectively [6]. These mechanisms also make it possible to describe the composition using conditional statements.

Project StdLib with FaaSlang provides a completely different way of using serverless technology. StdLib is platform-agnostic API gateway and serverless framework for FaaS enabling user to easily change serverless provider. FaaSlang provides an approach to specify a function's input and output types. However, StdLib with FaaSlang does not ensure that the realized function fulfils its interface declarations. Moreover, the current state of the project does not support any language except JavaScript.

None of the solutions found enables to make sure in the minimal form that the function will start in the scheme of operation with parameters that are passed to it, i.e. to check in advantage the compatibility of the data types of the interacting components. Therefore, we need a new alternative solution to describe the composition of functions which should provide the following features:

1) To describe the composition of functions, namely:
   - To describe the composition of one function with another, when the result of the first is passed to the second.
   - To describe error handling mechanism.
2) Make it possible to define the types of arguments and results for the functions involved in the composition and check their compatibility.

To meet paragraph 2 from the list of requirements, it is also necessary to have an extension for the FaaS platform which will manage functions building and deploying to ensure the type safety of the entire project because:

- If a function implements an interface different from the one the function declares, its compilation will be impossible.
- If the composition of a new version of the function is impossible with operating ones, its deployment will not happen.

We propose a domain-specific language called "Anzer"[1]

[1]https://github.com/tariel-x/anzer

as well as software that includes the language analyzer, the system of function building, interaction with the FaaS platform and the user interface as a solution to the problem mentioned above.

In addition, it is assumed that the language should not be highly specialized for use with a single FaaS platform. This means that the software package to be developed should be sufficiently versatile and modular to be able to adapt it to the new framework with minimal modifications.

III. The language of functions composition

Functions in the FaaS platforms are triggered in case of need, perform the programmed action and pass the result forward, not keeping the state. This feature partly creates an affinity between the functions mentioned above and the concept of functions from some functional programming languages.

In addition, if we consider a set of functions as a single monolithic program, then, in the absence of a global runtime environment and variable changes, drawing an analogy with imperative programming languages is impossible. Functional approach, on the contrary, is characterized by the composition of independent functions, which means the result of the calculation of the previous function is applied to the next one. Also, programs written in a pure functional style do not contain mutable variables, and functions can be easily moved from one program to another.

Listed properties create an affinity between a functional programming style and systems built using the FaaS platforms. In this regard, the concepts of such functional programming languages as Haskell and PureScript were taken as a basis for the proposed language.

A. Type system

The implemented language supports both basic types (string, boolean etc.) and custom user-defined record types, as shown in listing 1.

Listing 1. User-defined types

```
type Address = {
   house :: Integer
   street :: Maybe String
   city :: MinLength 10 String
   country :: String
}
type Addresses = List Address
```

Also, the language supports extension of basic and user-defined types with the help of type constructors. `List` is the type constructor, that is, the function that converts `Address` type into `List Address` which is an array of addresses. `MinLength` cconstructor defines a string with minimum length of 10. One can use `Maybe` constructor to determine that a field may not be present in the data being passed. The type with the applied constructors is the new type. There are more type constructors defined by the language.

Anzer language type system supports subtype polymorphism. Let us assume there is a function a waiting for input data of type A, but it is transmitted data B. If type A and provided type B are actually different, and the function will handle B correctly, it can be concluded that B can be subtype or type equivalent to the type A.

For example, listing 2 describes type A, which contains a string type field named `f1`. It also describes type B, which contains the same string field `f1` and, optionally, an integer field `f2`. Since type B contains the same fields of the same type as A, we can say that $A <: B$.

Listing 2. A and B subtype

```
type A = {
   f1 :: String
}
type B = {
   f1 :: String
   f2 :: Integer
}
```

That is, B is subtype of A if every term B can be safely used in the context where A is expected (1) [7].

$$\frac{\Gamma \vdash x : A \quad A <: B}{\Gamma \vdash x : B} \tag{1}$$

Inheritance is reflective: $A <: A$ and transitive. For instance, listing 3 shows an example of A, B and C, for which the following equation is true: $A <: B$, $B <: C$ and $A <: C$.

Listing 3. Transitivity

```
type A = {
   f1 :: String
}
type B = {
   f1 :: String
   f2 :: String
}
type C = {
   f1 :: String
   f2 :: String
   f3 :: String
}
```

Moreover, depth subtyping is true, i.e. the types of each corresponding field of a composite type may vary, but should be in terms of inheritance, as in the listings 4 and 5.

Listing 4. Inheritance in depth

```
type A = {
   f1 :: String
}
type B = {
   f1 :: MinLength 10 String
}
```

Listing 5. Inheritance in depth

```
type A = {
   f1 :: {
      sf1 :: String
   }
}
type B = {
   f1 :: {
      sf1 :: String
```

```
        sf2 :: String
    }
}
```

Rearranging of fields in the description of the user-defined type does not affect subtyping.

If there are two types A and B for which $A <: B$ and $B <: A$ are true, such types should be considered equivalent.

It should be noted that the application of some constructors to any type forms its subtype. For example, let A be a base string type, that is `type A = String`, and type B be a base string type with the applied string maximum length constraint constructor `type A = MaxLength 10 String`. Then, $A <: B$ is true. B. Language defines constructors `List`, `Maybe` and `Either`, which do not form a subtype due to their higher complexity.

Applying different constructors or the same constructor with a different parameter to any type leads to the appearance of two new different types. For example, `type A = MaxLength 10 String` and `type A = MaxLength 20 String` can not be considered in terms of inheritance or equivalence.

B. Functions

The types in Anzer are used to describe function's arguments and results of its operation. A function is either a reference to a repository with its source code or a synonym for the composition of other functions. Therefore, Anzer does not provide writing application logic, making it possible only the type-safe composition of functions having been implemented in other languages. An example of the system description is shown in the listing 6.

Listing 6. An example of the system description in Anzer

```
type RawAddress = MinLength 10 String
type Address = {
    street :: Maybe MinLength 10 String
    city :: MaxLength 20 String
    country :: String
}

github.com/u/parse[go]::
    RawAddress -> Address
isp github.com/u/isprovider[go]::
    Address -> Bool

detect :: RawAddress -> Bool
detect = isp . parse

invoke (
    detect,
)
```

The record `github.com/user/parse` in the example is the reference to the repository with the function's source code, and `RawAddress -> Address` is the description of input or output data. The record `isp . parse` defines the composition of `isp` and `parse` functions, and `detect` is the function defined as the composition of the other two.

The `invoke` keyword determines, which functions will be deployed in the FaaS platform. In this case, `detect` is synonymous with the composition of other functions: `isp` and `parse`, hence the latter will be deployed. At the end of the `isp` operation there will be an event created in the system and containing result of its operation, by means of which `parse` function will be launched and will receive this result.

C. Error handling

To handle errors you can use the type constructor `Either a b = Left a | Right b` which specifies that the function returns data of either type `a` or type `b`. For example, `Either Error Result` defines an algebraic data type which means that the result can be either an `Error` type or the `Result` type. There is no predefined error type.

The composition of functions which returns the result `Either Error Result` with functions expecting only the `Result` type is performed using `Either` monad. Its definition and use are similar to that of the Haskell programming language [8]. Construction `Either` in Anzer defines 2 operations: $>>=$ and `return`.

Operation $>>=$ (bind) is the higher-order function of Anzer, takes some data and another function as arguments and is defined as follows:

```
Right a >>= f = f a
Left a >>= f = Left a
```

Given example shows that if the first function argument $>>=$ is of type `Right a`, where `a` is a user-defined data type, then $>>=$ converts data of type `Right a` into type `a`. Then, the function `f` passed by the second argument is applied to the given data, and the result of this application is the result of the operation $>>=$.

If the first binding argument is of `Left a` type, the operation returns the data passed to it unchanged.

Thus, using the $>>=$ operation it is possible to bind functions returning an error message instead of operation result with functions expecting only correct data, not an error. At the same time, once generated, the error will reach composition's end without changes.

Another operation of `Either` monad is `return` function, which can be defined as follows: `return a = Right a`. As you can see from the definition, `return` casts user-defined data of type `a` to type `Right a` by means of the `Right` constructor. With the help of this operation you can bind functions that return type `a` instead of `Right a`.

Since Anzer is the domain-specific language, not a general-purpose language, unlike Haskell or other similar functional languages, there is no provision for creating custom type constructors, monads, or higher-order functions.

IV. Anzer Platform

One composition language would not be enough to achieve the goal, so the proposed solution also includes a specialized platform, the tasks of which include:

- Type checking in the description of the function composition in Anzer.
- Functions building using a specialized library to ensure compliance of the declared and implemented types.

- Deployment of new versions of functions in the selected FaaS platform.

A. General organization

To ensure compliance of the declared and implemented types, a specialized library is used that encapsulates all interaction with the FaaS platform. In addition, the proposed solution has a built-in code generator that prepares the basic structures or interfaces based on the data types described in the Anzer language. After generating a function basis, developer only needs to implement the business logic of an application.

When you deploy a function in the platform, generating additional function code based on the description in the Anzer language comes first. Additional code is required to confirm that the function implementation matches the description. If the implementation does not match the description declared in the Anzer file, the implementation language compiler would report an error.

The schematic representation of the system consisting of Anzer, the FaaS platform and functions is shown in figure 1.

The square with the caption "Anzer-λ" in the scheme shows a component implementing the logic of the application directly. This part is implemented by developer by means of the selected programming language. Arguments for launching in the function's code and the return of the work result are available using the Anzer library. The library, in turn, interacts with the selected FaaS platform.

The square with the caption "λ" directly shows a container with an executable file launched by the FaaS platform. The data bus is the part of the FaaS platform.

The rectangle labelled "Anzer Platform" indicates that the functions are managed through the appropriate user interface. Despite this, it remains possible to use standard tools of the selected FaaS platform.

The typical process of creating a new system in the FaaS platform using the proposed solution is as follows:

1) Describing required data types.
2) Generating a necessary function's basis in terms of the described types by means of Anzer user interface.
3) Implementing the function's operating logic.
4) Deploying a function in the version control system and describing functions in Anzer according to the same scheme as that of types.
5) Deploying the implemented functions via the Anzer user interface. What happens alongside:
    a) Verification of function composition availability based on the specified types.
    b) Based on the described types generation of missing code to work within the target FaaS platform.
    c) Compilation (if possible) of the implemented function and the code generated in step 5.b.
    d) In case of the successful container's compilation here comes the deployment of the functions in the FaaS platform.

The languages for which there is a compiler or static analyser with the possibility of static type checking in the code can be used as the languages of the function implementation. Such languages can be, for example, C++, Go, Java, TypeScript or PHP.

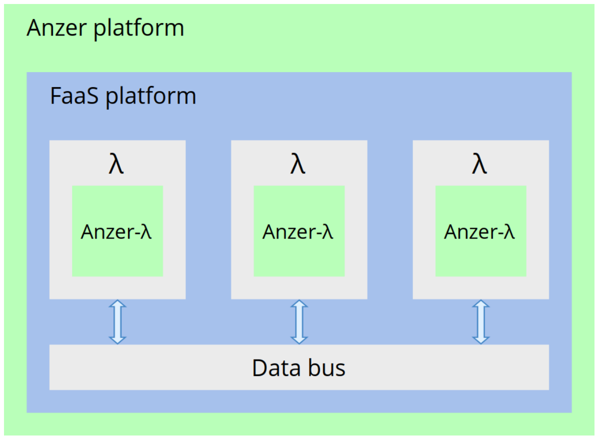

Figure 1. Schematic representation of the system.

B. User interface

The user interface of the solution is a set of utilities with CLI (Command line interface) interface. The utilities use a document in Anzer language to generate the basis of a function, to build it and to deploy it in the FaaS platform.

For instance, calling `anzer g –i scheme.anz –o ~/go/src/a/a.go –f parse` generates a basis for the function `parse` which is shown in the listing 6.

Calling `anzercli build ––anz listing2.anz parse` builds a container from source code of the `parse` function.

V. Implementation and testing

A. Implementation

Project Apache OpenWhisk [4] has been selected as the first supported FaaS platform due to ease of deployment, adequate performance [10] and the availability of the required functionality to implement the proposed solutions [9]. The proposed solution is implemented by means of Golang programming language which was chosen due to having necessary competence. Having Golang supported by main suppliers of serverless cloud services (AWS Lambda Functions Azure, Google CloudFunctions) enabled to use it as the first language supported by the Anzer platform.

The developed software includes:

- CLI user interface;
- Anzer parser and interpreter;
- A component to work with OpenWhisk:
    - A client for the OpenWhisk HTTP API;
    - A library for Golang that encapsulates the work with the FaaS platform.

Package of functions, i.e. a collection of bound functions and triggers, is created for each document in Anzer language with the help of the OpenWhisk client. A trigger, by means of which a function can be called, is created for each function. In the case of function composition, the code required to generate an event for calling the next function is generated automatically. Rules for calling functions for HTTP events, database events etc. can be configured by OpenWhisk built-in tools.

A partial example of a template that could be generated by the command from Chapter IV-B is shown in the listing 7.

Listing 7. Generated Go-template
```
type TypeIn struct {
   Price float64 `json:"price"`
   Text string `json:"text"`
}
type TypeOut struct {
   Desc string `json:"desc"`
   Name string `json:"name"`
   Price float64 `json:"price"`
}
func Handle(input TypeIn) TypeOut {
   var out TypeOut
   return out
}
```

It is notable that there is no line length check in the given sample code. The specified data type requirement is checked by the Anzer interpreter, but checking in the function code is yet to be implemented. After generating the template, you should implement business logic in the `Handle` function.

The code required to work in OpenWhisk will be generated at the stage of function deployment. Due to the use of interfaces in it, it is impossible to change the format of the transmitted data without editing the Anzer document.

B. Comparison with analogues

Comparison with analogues was made using criteria from [5]:

1) ST-safeness [11]: the solution meet the criterion as:
   - The function composition is a new function.
   - The composition of functions does not incur additional costs of computing power and, as a consequence, financial costs.
   - In the proposed solution, the following function is called asynchronously, that is, the execution time of all functions in the composition is not summed.
2) Programming model: functional-like DSL.
3) Parallel execution support: not supported at the moment.
4) State management: the presented solution uses the OpenWhisk platform that, in turn, uses the Apache Kafka message broker in the data transmission channel between functions. Therefore, the maximum size of the transmitted state is equal to the maximum size of the message in Apache Kafka.
5) Software packaging and repositories: the source code of functions is stored in Git repositories.
6) Architecture: uses an OpenWhisk architecture consisting of a controller and a message queue.
7) Overhead: absent as Anzer is not a component of the FaaS platform and responsible only for configuring the interaction between functions.
8) Billing model: not applicable.

There is no type safety in the list of criteria because no alternative solutions matching this criterion have been found.

According to the criteria of the article [5], the proposed solution is not inferior to the alternative. However, if we take into account possibilities presented by Anzer, it is less functional, which means it is impossible to use it to build complex systems yet.

It should also be noted that the Anzer platform does not increase the system's consumption of machine resources and the same goes for the operating time of functions, as in fact it only adjusts the connection between them.

C. Testing

The proposed solution is being in the process of testing. Currently, Anzer is used in several simple systems, one of which is used to simplify the process of passing code-review. The system consists of 3 functions. Two functions perform specific actions and return a result by a repository management system event. The third one receives an action result and sends a text message to a chat.

An example of the scheme used in testing is shown in the listing 8. The `Hook` type describes a small part of the query that is automatically sent by the project management system Gitlab when certain events occur. The `Event` type contains validated information about edits in the code in the repository. The `Assignment` type is used to transfer information about the person assigned to control the current edits in the code.

The `validate` function, in accordance with its name, checks the incoming query and, using Gitlab API, determines the programming language used in the repository. The `assign` function appoints the person responsible for checking code edits based on the programming language of the repository. The `notify` function sends an appointment notification to the chat.

Listing 8. The example of using Anzer
```
type Hook = {
   user :: {
   username :: String
   }
   repository :: {
   name :: String
   homepage :: String
   }
}
type Event = {
   author :: String
   repository :: String
   language :: String
}
type Assignment = {
   reviewer :: String
   repository :: String
}
```

```
github.com/u/validate[go]::
   Hook -> Event
github.com/u/assign[go]::
   Event -> Assignment
github.com/u/notify[go]::
   Assignment -> Bool

assign_mr = validate . assign . notify
invoke (
   assign_mr,
)
```

The given example is the very simple system not using even a possibility to handle errors. Nevertheless, its construction made it possible to verify the viability of the approach at the minimum level.

To apply the platform and Anzer language to the real-world problems, for example, building ETL systems, a number of language and platform improvements are required, for instance:

1) The support of the conditional operator present in alternative means of composition is required. In the Anzer language, it could be a pattern-matching analogue from functional programming languages.
2) Using the project as a tool to create full-fledged commercial products requires the development of specialized tools for function debugging and testing.

In general, the use of Anzer together with the accompanying software facilitates creation of new systems and support of existing ones, while not affecting the system requirements for hardware resources and performance.

VI. Conclusion

The article considers the problem of function interaction interface divergence in the FaaS platforms. When you develop a system using this approach, you may experience a situation where some components wait or return data in an updated format that is incompatible with outdated components. One of the reasons for this problem is the lack of tools to describe and verify the type checking of interacting functions.

As a solution, this article proposes domain-specific language Anzer created to easily describe the types of data transmitted, the type-safe composition of functions within the whole system, the semantics of their interaction and the logic of error handling. The developed software enables to automate the creation of new functions and maintain their integration with others at all stages of system development and support. Together language and software prevent errors due to mismatch between input data format expected by a function and actually transmitted one. All this makes it possible to simplify and speed up the process of developing systems based on the concept of Function as a Service.

The extensive use of the type system in describing the functions interaction distinguishes the proposed solution from the alternatives, but there are several unresolved problems:

1) A possibility to select a function for a composition based on the actually transmitted data type is needed. In the current implementation, in case of using an algebraic type function, for instance, `Left a | Right b`, the final function is obliged to process both variants. For full use it is necessary to implement an analogue of the pattern matching operation in functional languages.
2) In addition to pattern matching, “if-then” construction working with data will simplify the description of complex interaction schemes.
3) Solving problems that are more complex than those described in the “Testing” section requires the possibility of local functions testing and debugging.

An important point is to develop the possibility of using Anzer language along with one of the cloud FaaS platforms, such as IBM CloudFunctions, AWS Lambda or others, as it is them whom the most mature alternative composition solutions are created for.

References

[1] N. Kratzke, “A Brief History of Cloud Application Architecturesm” Applied Sciences, vol. 8, 2018, pp. 1368-1368.

[2] R. Rodger, The tao of microservices. New York: Manning publications, 2018, pp. 17-19.

[3] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță and A. Iosup “Serverless is More: From PaaS to Present Cloud Computing,” IEEE Internet Computing, vol. 22, no. 5, 2018, pp. 8-17.

[4] (2019 Jan.) OpenWhisk. [Online]. Available: http://openwhisk.apache.org

[5] P. Garcia Lopez, M. Sanchez-Artigas, G. Paris, D. Barcelona Pons, A. Ruiz Ollobarren, and D. Arroyo Pinto, “Comparison of FaaS Orchestration Systems,” 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, Switzerland, 2018, pp. 148-153.

[6] K. Kritikos and P. Skrzypek “A Review of Serverless Frameworks,” 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, Switzerland, 2018, pp. 161-168.

[7] B. Pierce, Types and Programming Languages. London: MIT Press, 2002, pp. 251–254.

[8] B. Milewski. (2019 Jan.) Basics of Haskell - Error handling. [Online]. Available: https://www.schoolofhaskell.com/user/bartosz/basics-of-haskell/10_Error_Handling

[9] S. Mohanty, “Evaluation of Serverless Computing Frameworks Based on Kubernetes,” Aalto University, 2018.

[10] S. Shillaker (2019 Jan.) A provider-friendly serverless framework for latency-critical applications. [Online]. Available: http://conferences.inf.ed.ac.uk/EuroDW2018/papers/eurodw18-Shillaker.pdf

[11] I. Baldini et al. “The serverless trilemma: function composition for serverless computing,” in Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017), New York, NY, USA, 2017, pp. 89-103.