Journal of Software Engineering and Applications, 2020, 13, 278-287
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

Result-as-a-Service (RaaS): Persistent Helper Functions in a Serverless Offering

Arshdeep Bahga, Vijay K. Madisetti, Joel R. Corporan

Georgia Institute of Technology, Atlanta, USA
Email: arshdeepbahga@gmail.com, vkm@gatech.edu, corporan@gatech.edu

How to cite this paper: Bahga, A., Madisetti, V.K. and Corporan, J.R. (2020) Result-as-a-Service (RaaS): Persistent Helper Functions in a Serverless Offering. *Journal of Software Engineering and Applications*, **13**, 278-287.
https://doi.org/10.4236/jsea.2020.1310018

Received: September 21, 2020
Accepted: October 24, 2020
Published: October 27, 2020

Abstract

Serverless Computing or Functions-as-a-Service (FaaS) is an execution model for cloud computing environments where the cloud provider executes a piece of code (a function) by dynamically allocating resources. When a function has not been executed for a long time or is being executed for the first time, a new container has to be created, and the execution environment has to be initialized resulting in a cold start. Cold start can result in a higher latency. We propose a new computing and execution model for cloud environments called Result-as-a-Service (RaaS), which aims to reduce the computational cost and overhead while achieving high availability. In between successive calls to a function, a persistent function can help in successive calls by pre-computing the functions for different possible arguments and then distributing the results when a matching function call is found.

Keywords

Serverless Computing, Functions-as-a-Service, Lambda Functions

1. Introduction

Serverless Computing or Functions-as-a-Service (FaaS) is an execution model for cloud computing environments where the cloud provider executes a piece of code (a function) by dynamically allocating resources [1] [2]. In the serverless computing model, the code is structured into functions. The functions are triggered by events such as an HTTP request to an API gateway, a record written to a database, a new file uploaded to cloud storage, a new message inserted into a messaging queue, a monitoring alert, and a scheduled event. When a function is triggered by an event, the cloud provider launches a container and executes the function within the container. Some important concepts related to serverless computing are described as follows:

- **Push and Pull Models of Invocation**: Functions in a serverless offering are invoked by event sources, which can be a Cloud service or a custom application that publishes events. The event-based invocation has two modes: push and pull.
- **Concurrent Execution**: Concurrent execution refers to the number of executions of the functions which are happening at the same time. Cloud providers set limits on concurrent executions.
- **Execution Duration**: Cloud providers set a time-out limit under which a function execution must complete. If the function takes a long time to execute than the timeout limit, the function execution is terminated.
- **Container Reuse**: Cloud providers typically use containers for executing the functions in their serverless offerings. A container helps in isolating the execution of a function from other functions. When a function is invoked for the first time (or after a long time), a container is created, the execution environment is initialized, and the function code is loaded. The container is reused for subsequent invocations of the same function that happen within a certain period.
- **Cold and Warm Functions**: When a function has not been executed for a long time or is being executed for the first time, a new container has to be created, and the execution environment has to be initialized. This is called a cold start. Cold start can result in a higher latency as a new container has to be initialized. The cloud provider may reuse the container for subsequent invocations of the same functions within a short period. In this case, the function is said to be warm and takes much less time to execute than a cold start.

The key contributions of this work are 1) a new computing and execution model for cloud environments called Result-as-a-Service (RaaS) is proposed over FaaS, which aims to reduce the computational cost and overhead while achieving high availability, 2) an approach for optimizing FaaS offerings by introducing a library of "persistent helper functions" is proposed, 3) an analytical model and an algorithm for maximizing the performance in a serverless offering is presented, and 4) an implementation case study using persistent helper functions is presented.

2. Related Work

AWS Lambda [3], Azure Functions [4] and Google Cloud Functions [5] are examples of popular commercial FaaS offerings. Popular open source serverless frameworks include OpenFaaS [6], Kubeless [7], Fission [8] and Apache OpenWhisk [9].

Serverless offerings have limitations such as cold starts and timeout limits. Other challenges include provisioning and requesting overhead, pricing models [10] [11], and orchestration [12] [13].

In [14], Azari and Koc have presented an approach for partitioning tasks between hardware and software to improve performance. We have adapted this approach for modeling speedup from using persistent helper functions in a RaaS offering.

3. Proposed Approach

We propose a method for optimizing FaaS offerings by introducing a library of persistent helper functions that are not billed like the functions in a FaaS. The persistent helper functions can "turbo" boost the execution by prefetching data and precomputing logic. In between successive calls to a function, a persistent function can help in successive calls by precomputing the outcomes for different possible arguments and then distributing the results when a matching function call is found. This makes function calls faster and also reduces load since common computation is shared by the cloud provider across millions of calls that can share the common precomputed values. Different third parties can compete to provide helper functions that different retail users can leverage, thus creating a Persistent Functions marketplace, much like an "app store" [15].

There are two reasons why RaaS is favored over FaaS. Firstly, as a consequence of cost-savings when scaling, the proposed pricing model is detached from the computational process expected by the on-demand request and is likely to be much lower when users are incurring on the shared service rather than individual functions with the same purpose. Secondly, we demonstrate the round-trip latency is significantly reduced after the precomputation of the expected values, thereby achieving high availability on request. The new model aims to meet the requirements of low-latency applications such as smart metering, smart cities, autonomous vehicles, wearable devices, among others, to reduce the cost of compute-intensive tasks.

An app store of persistent helper functions from third parties and cloud providers can help accelerate and optimize the use of serverless applications in the cloud context. Sophisticated identification, linkage, and lifecycle licensing modules allow applications and helper functions to be scaled, priced competitively, and also allow privacy through authentication and encryption.

3.1. RaaS: FaaS Offering with Persistent Helpers

In this section, we present a new computing and execution model for cloud environments called Result-as-a-Service (RaaS). RaaS is an enhancement over FaaS as it reduces the computational cost and overhead while achieving high availability through the use of persistent helper functions. Figure 1 shows the architecture of a RaaS offering. The components in the RaaS architecture are as follows:

- **Load Balancer**: Load balancer routes events/requests to servers, which ultimately invokes the functions which are executed within containers running on the servers. If a server has a hot container for a function already running, the request is routed to that server.

- **RaaS Server**: Figure 2 shows the architecture of a RaaS server, which executes functions within containers and returns a response to clients. Functions are invoked by event sources. The event-based invocation has two modes: push and pull. The server also handles CRUD (create, read, update and delete) operations for setting up functions. When a server runs a function for the first time, it caches the function image and starts a hot container. If a container is already running, the server routes the function call to the running container. The response from a function execution is then sent back to the load balancer. The server maintains a pool of containers for persistent

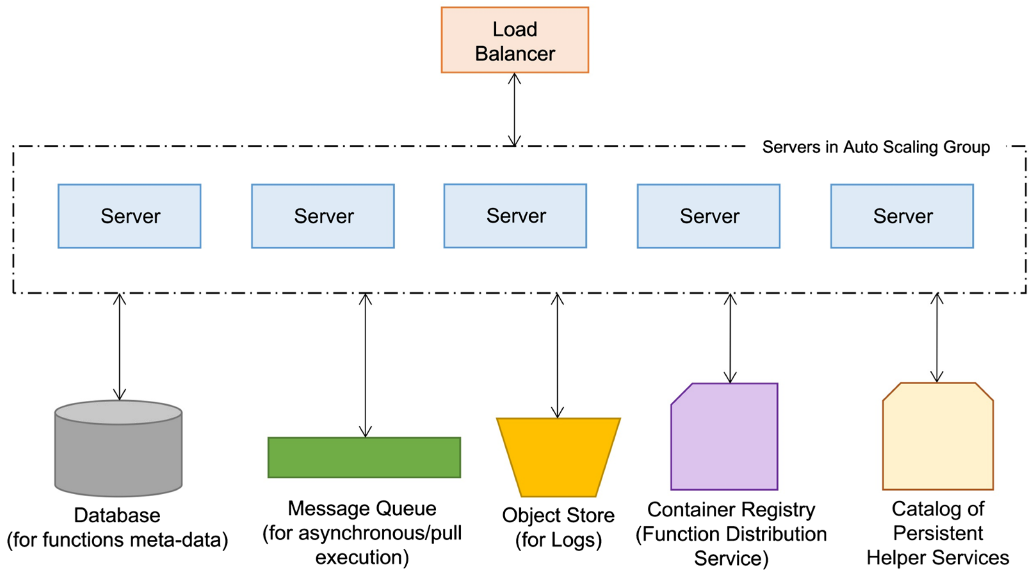

Load Balancer
Servers in Auto Scaling Group
Server
Server
Server
Server
Server
Database (for functions meta-data)
Message Queue (for asynchronous/pull execution)
Object Store (for Logs)
Container Registry (Function Distribution Service)
Catalog of Persistent Helper Services

Figure 1. Architecture of a RaaS offering.

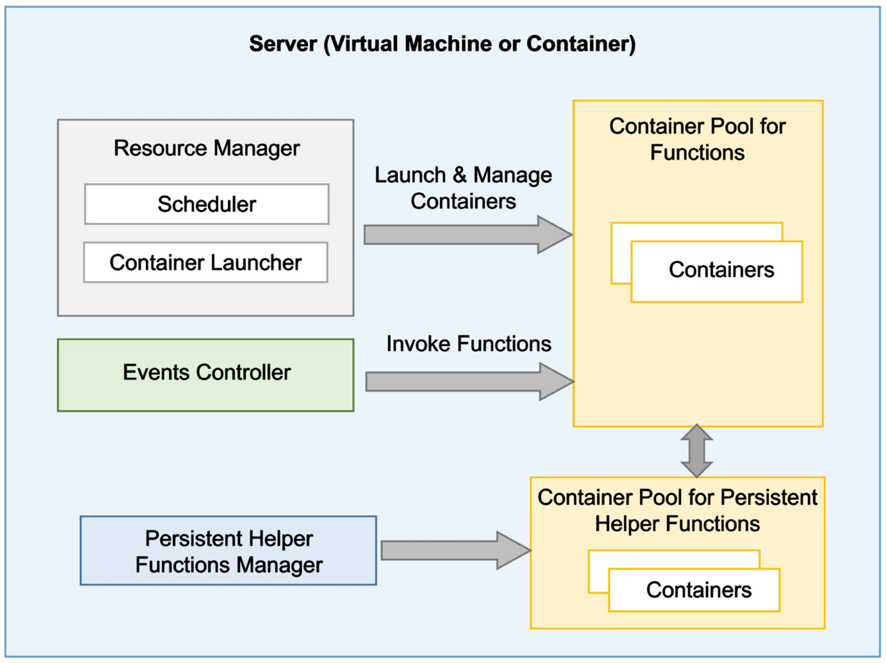

Server (Virtual Machine or Container)
Resource Manager
Scheduler
Container Launcher
Launch & Manage Containers
Container Pool for Functions
Containers
Events Controller
Invoke Functions
Persistent Helper Functions Manager
Container Pool for Persistent Helper Functions
Containers

Figure 2. RaaS Server architecture with support for persistent helper function.

helper functions, which are independent of the containers which execute the functions.

- **Packaging Functions & Persistent Helpers**: The source code of functions and persistent helpers is packaged as a container image. A container registry (or functions distribution service) maintains a record of all the functions registered with the RaaS offering. Similarly, a catalog of persistent helper functions in the RaaS offering is maintained.

3.2. Features of Persistent Helper Functions

- **Stateful**: A key differentiating factor of persistent helper functions from existing FaaS offerings is that the persistent helper functions can be stateful, whereas functions are stateless and any state information has to be separately maintained in a state database.
- **Billing & Subscription Models**: Persistent helper functions can have different billing and subscription models. For example, the number of requests or events processed, duration or time period, amount of computing and memory resources used, and amount of data processed can be independently configured. The third parties providing persistent helper functions can share royalty with the cloud provider that provides the serverless offering.
- **Continuous Training**: Persistent helper functions can be continuously trained and optimized independently of the functions which use the persistent helper functions.
- **Distribution and Management**: Persistent helper functions are made available through a functions store (like an app store). Developers can choose persistent helper functions from the functions store and select among various subscription, billing and licensing models. Each instance of a persistent helper function is identified by a unique ID and may be used by one or more functions. The user is provided a dashboard that shows the status of persistent helper functions instantiated by the user, their cost and other runtime expenses and workload utilization.
- **Scaling**: Persistent helper functions are scaled elastically. There is a load balancer frontend to the persistent helper functions manager. It spawns new helper instances and goes through a lifecycle approach to support functions.
- **Execution**: The persistent helper functions could be executed on GPU or ASICs to speed up the execution.
- **Sharing**: The persistent helper functions can be shared across multiple functions.
- **Configuration and Customization**: The persistent helper functions can be configured or customized to be used in different functions.
- **Third Party Libraries**: The persistent helper functions may use a third-party library or may be developed by the user.

3.3. Modeling Speed-Up from Persistent Helper

A function in a serverless offering is represented as a Control Data Flow Graph

(CDFG), as shown in **Figure 3**. There are two types of nodes in a CDFG: data flow nodes and decision nodes. A data flow node is a piece of code that has a single entry point, single exit point and no condition, whereas, a decision node is a piece of code which has at least one condition. Nodes can be persisted and the profitvalue determines the benefit from persistence in memory or database. For each node in the CDFG, the actual execution time (Ti') and the execution time of a persisted version (Ti) is determined. The profit value for each node is the difference ($Ti' - Ti$).

We present an algorithm to partition portions of a function (nodes in CDFG representation of a function) into two sets—persisted and not-persisted, as follows:

Set of nodes which are not persisted:

$$NPset = \{n_1, n_2, n_3, \cdots\} \tag{1}$$

Set of nodes which are persisted:

$$PHset = \{\ \} \tag{2}$$

Set of Profit Values for nodes:

$$PVset = \{\ \} \tag{3}$$

Memory Used:

$$M_U = 0 \tag{4}$$

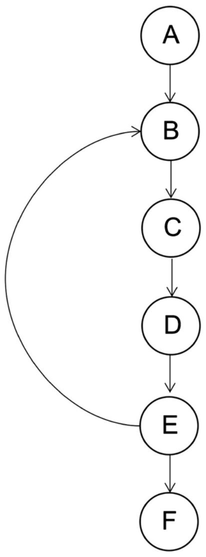
$$\text{Total Memory available: } M_T \tag{5}$$

$$\text{for } n_i \text{ in } NPset: \tag{6}$$

$$Pi = Ti' - Ti \tag{7}$$

$$PVset = PVset + \{(Pi, n_i)\} \tag{8}$$

$$PVset = Sort(PVset) \text{ by } Pi \tag{9}$$



Constraints ↓ Benefit from persistence ↓

Node	Time Actual (ms)	Time Persisted (ms)	Memory Used (MB)	Profit (ms)
A	38	30	60	8
B	42	28	52	14
C	98	40	35	58
D	55	38	23	17
E	67	41	65	26
F	72	59	44	13

Constraints can be for instance memory used, database read/write capacity used or database size

Figure 3. Example of a Control Data Flow Graph for modeling speed up in a RaaS offering.

$$\text{while}\quad M_U < M_T: \tag{10}$$

$$\text{for}\quad (Pi, n_i)\quad \text{in}\quad PVset: \tag{11}$$

$$\text{if}\quad (M_U + M_{ni}) < M_T: \tag{12}$$

$$PHset = PHset + \{n_i\} \tag{13}$$

$$NPset = NPset - \{n_i\} \tag{14}$$

$$M_U = M_U + M_{ni} \tag{15}$$

The goal of this algorithm is to maximize the performance by using persistent helper functions given constraints such as memory used, database read/write capacity used or database size. The speedup from using persistent helpers can then be computed as follows:

$$Speedup = \frac{(A-(B+C))}{A}$$

where, A = Total time for execution of all nodes if no persistence is used; B = Total time for execution of nodes in the persisted set; C = Total time for execution of nodes in the non-persisted set;

4. Implementation Case Study

To evaluate the proposed approach, we developed a reference application for sentiment analysis of social media posts such as tweets from Twitter as shown in Figure 4. A custom listener component fetches tweets using the Twitter API and posts the tweets to an API gateway endpoint which triggers a function in a serverless offering to compute sentiment of each tweet. The computed sentiments are stored in a database. A web application presents the sentiment analysis results.

Different approaches can be used to compute sentiment of tweets such as a sentiment analysis function that uses a sentiment lexicon, a third-party library such as Python TextBlob, or a web-based NLP service such as AWS Comprehend. In the FaaS version of the function where no persistence is used, one of the above three approaches is used to analyze each tweet. Whereas in the RaaS version, a persistent helper service is set up, which stores the computed sentiments in memory or a database, and the function which processes the tweets uses this service. Whenever there is a request from the function to persistent helper to compute sentiment, the persistent helper service checks if the tweet has been evaluated before. If the sentiment is not found in memory or database, it is

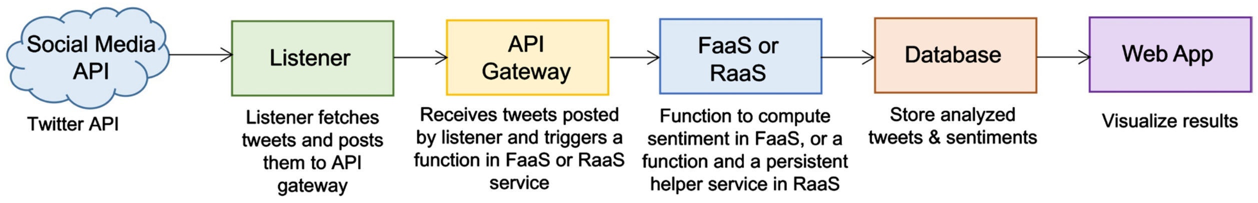

Social Media API
Twitter API
Listener
Listener fetches tweets and posts them to API gateway
API Gateway
Receives tweets posted by listener and triggers a function in FaaS or RaaS service
FaaS or RaaS
Function to compute sentiment in FaaS, or a function and a persistent helper service in RaaS
Database
Store analyzed tweets & sentiments
Web App
Visualize results

Figure 4. Reference application for social media sentiment analysis.

computed and stored. Otherwise, if the sentiment had previously been computed, the stored results are returned, thus saving time by avoiding a redundant repeated computation.

We developed and deployed a series of functions into AWS Lambda and tested two different conditions: 1) single tweet text and 2) random tweet text. The functions ran in a cold and warm state, each with different memory sizes. We used AWS Comprehend to analyze the text and derive the sentiment and used AWS API Gateway as the RESTful API to handle incoming GET request from the client.

5. Experimental Results

To evaluate the performance of RaaS approach over FaaS we measured the run times of the functions in RaaS and FaaS versions of the reference application shown in Figure 4.

For the FaaS version, we used a Lambda function set up in the AWS Lambda service, which computes sentiments using the AWS Comprehend service. Whereas in the RaaS version, we used a Lambda function set up in the AWS Lambda service along with a persistent helper service that computes and stores sentiments in memory.

We evaluated the cold run and warm run performance of the functions in the FaaS and RaaS versions. The cold runs measure the behavior of functions when provisioned for the very first time. We took a number of measurements of function run times by varying the container memory size. Figure 5 shows the cold and warm run performance for FaaS and RaaS versions. For reference, we also show the predicted performance with persistence, which is estimated using the model described in Section 3.3. As seen from the cold and warm run charts, the predicted performance closely matches the actual performance.

Figure 6 shows the results for an alternative implementation of persistent helper service that computes and stores sentiments in a NoSQL database instead of memory. For the single tweet text condition, we extracted a single tweet from a training dataset that contains 5000 tweets. Our first test consisted of sentiment analysis on the text without persisting the data, and as performed in both a cold

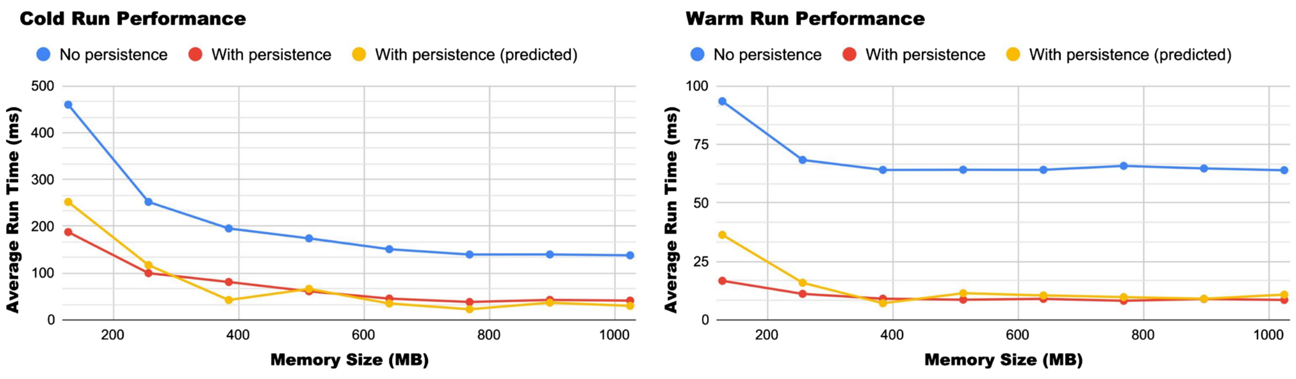


Figure 5. Cold and Warm run performance: no persistence vs persistence in memory.

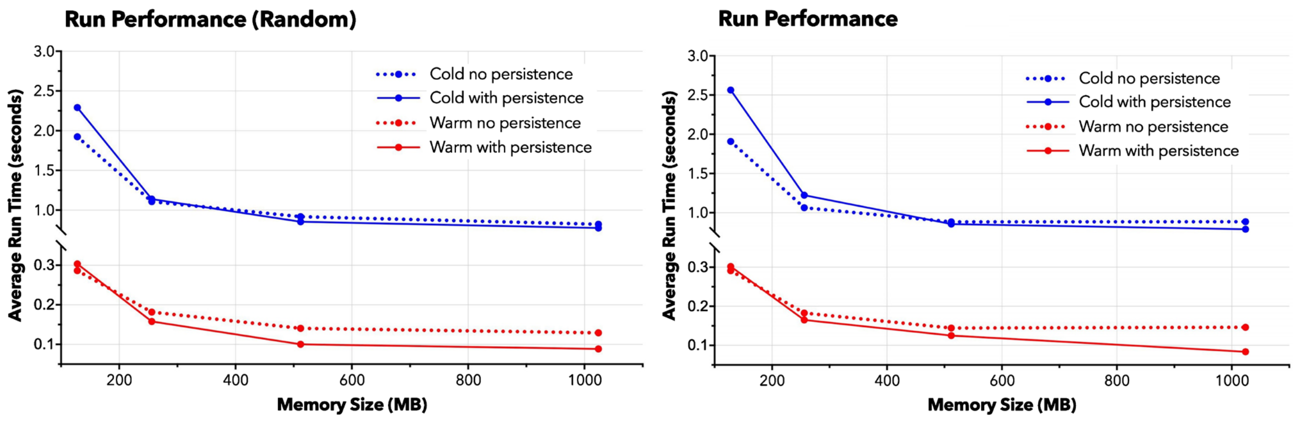


Figure 6. Cold and Warm run performance: no persistence vs persistence on database.

and warm state. For the second test, the text was persisted in the database for both cold and warm states. In the random test condition, we used the training dataset to randomly sample the sentiment analysis in both states.

In both the cold and warm run experiments (with persistence in memory and in database), we observed that the average run time improves by increasing the memory allocated. This happens because the CPU capacity allocated to containers executing the functions also increases as the memory allocated is increased. AWS Lambda states that every time memory is doubled, the CPU capacity is also doubled. Further, we observed that the RaaS approach (with persistence) outperforms the FaaS approach (no persistence).

6. Conclusion and Future Work

We presented an approach for optimizing FaaS offerings by introducing persistent helper functions, which can boost the execution by prefetching data and precomputing logic. Future work will focus on extending an open-source serverless offering such as OpenFaaS to support persistent helper functions and creating a dashboard to display the status of persistent helper functions instantiated by the user, their cost along with other runtime expenses and workload utilization.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

[1] Bahga, A. and Madisetti, V. (2019) Cloud Computing Solutions Architect: A Hands-On Approach. VPT, ISBN: 9780996025591.

[2] Leitner, P., Wittern, E., Spillner, J. and Hummer, W. (2019) A Mixed-Method Empirical Study of Function-as-a-Service Software Development in Industrial Practice. *Journal of Systems and Software*, **149**, 340-359

[3] AWS Lambda. https://aws.amazon.com/lambda/

[4] Azure Functions. https://azure.microsoft.com/en-us/services/functions/

[5] Google Cloud Functions. https://cloud.google.com/functions/

[6] OpenFaas. https://github.com/openfaas/faas

[7] Kubeless. https://github.com/kubeless/kubeless

[8] Fission. https://github.com/fission/fission

[9] Apache OpenWhisk. https://github.com/apache/openwhisk

[10] Baldini, I., *et al.* (2017) Serverless Computing: Current Trends and Open Problems. In: Chaudhary, S., Somani, G. and Buyya, R., Eds, *Research Advances in Cloud Computing*, Springer, Singapore, 1-20.

[11] van Eyk, E., *et al.* (2018) A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. *Companion of the* 2018 *ACM/SPEC International Conference on Performance Engineering*, Berlin, 9-13 April 2018, 21-24.

[12] Tosatto, A., Ruiu, P. and Attanasio, A. (2015) Container-Based Orchestration in Cloud: State of the Art and Challenges. 2015 *Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, Santa Catarina, 8-10 July 2015, 70-75.

[13] Peinl, R., Holzschuher, F. and Pfitzer, F. (2016) Docker Cluster Management for the Cloud-Survey Results and Own Solution. *Journal of Grid Computing*, **14**, 265-282. https://doi.org/10.1007/s10723-016-9366-y

[14] Azari, E. and Koc, H. (2015) Improving Performance through Path-Based Hardware/Software Partitioning. *Fifth International Conference on Digital Information Processing and Communications* (*ICDIPC*), Sierre, 7-9 October 2015, 54-59.

[15] Madisetti, V. and Bahga, A. (2019) Persistent Helpers for Functions as a Service (FaaS) in Cloud Computing Environments. US Provisional Patent Application No. 62884690.