A Review of Serverless Frameworks

Kyriakos Kritikos*
ICS-FORTH, Crete, Greece
kritikos@ics.forth.gr

Paweł Skrzypek†
7Bulls, Warsaw, Poland
pskrzypek@7bulls.com

***Abstract*—Serverless computing is a new computing paradigm that promises to revolutionize the way applications are built and provisioned. In this computing kind, small pieces of software called functions are deployed in the cloud with zero administration and minimal costs for the software developer. Further, this computing kind has various applications in areas like image processing and scientific computing. Due to the above advantages, the current uptake of serverless computing is being addressed by traditional big cloud providers like Amazon, who offer serverless platforms for serverless application deployment and provisioning. However, as in the case of cloud computing, such providers attempt to lock-in their customers with the supply of complementary services which provide added-value support to serverless applications. To this end, to resolve this issue, serverless frameworks have been recently developed. Such frameworks either abstract away from serverless platform specificities, or they enable the production of a mini serverless platform on top of existing clouds. However, these frameworks differ in various features that do have an impact on the serverless application lifecycle. To this end, to assist the developers in selecting the most suitable framework, this paper attempts to review these frameworks according to a certain set of criteria that directly map to the application lifecycle. Further, based on the review results, some remaining challenges are supplied, which when confronted will make serverless frameworks highly usable and suitable for the handling of both serverless as well as mixed application kinds.**

***Index Terms*—serverless, function-as-a-service, abstraction, provisioning, framework, review**

I. Introduction

Serverless computing [1] is a new computing kind that is currently getting a momentum in the cloud world. This is due to the main advantages that it promises, which include zero administration, infinite elasticity and minimal costs [1], [2]. Indeed, serverless computing is a step above PaaS platforms [3], which enables to deploy and provision small software components called functions in the cloud. In this kind of software provisioning, the devops users do not any more worry and do not have any control over the way their software is administered. The whole burden goes to the serverless platform provider who needs to scale the software as needed in an elastic way to handle any kind of unanticipated load. Further, due to the flexible cost model in serverless computing platforms, the cost of software provisioning is minimal and depends not on certain time periods but on the number of calls [4]. This kind of flexibility enables the charging of serverless applications based on the way they are used. In contrast to IaaS services (e.g., VMs) where the charging is performed in an hourly basis.

Due to the above advantages as well as the capability to massively execute certain pieces of functionality to handle even unanticipated workloads, serverless computing is currently uptaken and finds applications in multiple areas [2], such as image and video processing and scientific computing [5], [6]. Further, it looks like a promising realisation technology for edge computing [7] especially due to the reduced computing needs that are required for hosting functions, which are sometimes called as nano-services [8].

In this respect, many traditional big cloud providers have rushed [1] to develop and offer serverless platforms which promise the rapid deployment and suitable provisioning of serverless applications. Currently, most of these platforms are in a beta version with some well-known limitations and restrictions but soon they will be moved to a production one. Due to the traditional strategy of these big cloud players, they attempt to lock-in their serverless clients through also the offering of extra services that assort the provisioning of serverless applications, which are required for triggering them or handling their state, especially as the functions that are being deployed are stateless.

To resolve this major issue, serverless frameworks have been recently developed [9]. Such frameworks have the main design goal to abstract away from the technical specificities of a serverless platform or cloud infrastructure and thus make the life of the devop easier. Such frameworks differ in two different ways: (a) based on which level they abstract from; (b) based on the level of support that they provide to the serverless application lifecycle. As such, devops are now facing yet another problem. Which serverless framework to choose based on their needs that would allow them to rapidly develop and deploy serverless applications with ease.

To assist devops in this selection task, this paper attempts to review the serverless frameworks that are currently offered in the market. This assessment is conducted according to an especially designed set of criteria which span different phases in the lifecycle of serverless applications. Thus, the main goal of this assessment is to unveil those frameworks which can become an ideal companion to software devops and would enable them to have an appropriate control over how their serverless applications are mapped to and exploit respective cloud infrastructures or specialised platforms so as to be properly provisioned.

Based on the results of this analysis, there is no framework that can be said to really prevail having the best possible performance in all the criteria considered. However, there seem to exist some promising alternatives. A true prevalence could

978-1-7281-0359-4/18/$31.00 ©2018 IEEE
DOI 10.1109/UCC-Companion.2018.00051

be only achieved through the addressing of certain challenges that still need to be met by the serverless frameworks. These challenges are thus analysed to pave the way for the future work in these frameworks.

The remaining part of this paper is structured as follows. The next section defines what is a serverless framework and provides a short overview of those frameworks that have been reviewed. Section III presents the criteria via which the evaluation has been performed. Section IV reports and discusses the assessment results. Finally, the last section shortly analyses the remaining challenges that still need to be faced by the serverless frameworks.

II. Serverless Frameworks

A. Definition

Before starting to analysing the serverless frameworks, a proper definition of them should be supplied which would also better formulate the scope of our analysis. In this respect, we consider a serverless framework as a software middleware which enables to abstract away from the specificities of a certain serverless platform or cloud infrastructure and thus ease the deployment and provisioning of multi- or cross-cloud (serverless) applications, i.e., applications which could be deployed via different platforms or could exploit simultaneously two different cloud infrastructures. Our vision embraces applications which take the most out of the cloud market and make the best choices with respect to what cloud services are selected and integrated.

Based on the above definition, we can clearly distinguish between the following types of serverless frameworks: (a) *abstraction frameworks* which abstract away from the technicalities of two or more serverless platforms. For example, serverless[1]) is surely an abstraction framework supporting deployment in at least five serverless platforms. Please consider that frameworks, like AWS Chalice[2], which support only one serverless platform are considered out of scope of our review; (b) *provisioning frameworks* that enable the operation of a mini-serverless platform, on behalf of the devop, over existing cloud infrastructures so as to support the provisioning of serverless applications. An example of such a framework is Fission[3], which exploits Kubernetes as the medium via which a (mini) serverless platform can be constructed and operated.

B. Framework Identification

By having in mind the above definition and classification, we have searched the internet and attempted to find multiple pointers towards serverless frameworks. We followed a multi-source search process which involved the use of well-known search engines, like Google, as well as scholarly repositories, like Web of Science (WoS) and Scopus. The main outcome was our ability to find: (a) a very good pointer to multiple frameworks (actually serverless products in general) in https://landscape.cncf.io/grouping=landscape&landscape=serverless offered by Cloud Native Computing Foundation (CNCF); (b) interesting articles either pointing to such frameworks or actually proposing them.

In the sequel, we performed a respective filtering which focused on selecting only those frameworks which both match the above definition and are offered in an open-source form which is publicly available. The main rationale for the latter is that such frameworks should come for free as the devop already pays the provisioning cost of serverless applications over respective cloud platforms or infrastructures. This resulted in the following list of shortly analysed serverless frameworks.

C. Framework Overview

1) Provisioning Frameworks: Fission: exploits Kubernetes to operate a mini-serverless platform over existing cloud infrastructures. Its main features are that it supports any programming language as well as it maintains a pool of warm containers to resolve the well-known cold start problem [1].

Kubeless[4] is another provisioning framework relying on Kubernetes. It supports multiple programming languages plus custom runtimes. It also provides its own event triggering mechanism via Kafka. Finally, it offers a user interface (UI) via which functions can be called and monitored.

IronFunctions[5] supports any programming language and can rely on Kubernetes, Docker Swarm and Mesosphere for resource management. It is also able to import Lambda functions. Finally, it offers a UI for updating function routes and calling functions while it addresses the cold start problem by using hot functions mapped to hot containers.

Sparta[6] is a provisioning framework in the sense that it can provision functions over AWS Lambda or the Amazon cloud. So, it operates on top of two different cloud services. Sparta has the interesting feature that all configuration and infrastructure elements can be specified as types in Go, i.e., its actual development language. It also offers a UI via Grafana for monitoring the status of functions.

Fn[7] is another Kubernetes-based provisioning framework. It has similar special features to IronFunctions, it enables the grouping of functions into the concept of an app, supports the configuration of variables in three levels (application, function and route), while it also offers support for data binding.

Snafu[8] [10] is denoted as a serverless host process and tool for Python. It supports currently only Python and Java as programming languages. It exhibits the nice feature of supporting the importing from serverless platforms, like AWS Lambda and IBM OpenWhisk, and the exporting to other serverless frameworks, like Fission and Kubeless. Finally, it offers a multiple list of own connectors/triggering mechanisms to functions while it offers the important functionality of faasification [11] (see details in next section).

[1]https://serverless.com/framework/
[2]https://chalice.readthedocs.io/en/latest/
[3]https://fission.io/

[4]https://kubeless.io/
[5]https://github.com/iron-io/functions
[6]http://gosparta.io/
[7]http://fnproject.io/
[8]https://github.com/serviceprototypinglab/snafu

2) Abstraction Frameworks: Serverless[9] is an abstraction framework over the following serverless platforms: AWS Lambda, Azure Functions, IBM OpenWhisk, Google Cloud Functions and SpotInst. In addition, it can be integrated with other serverless frameworks like Kubeless and Fn. It adopts the CloudFormation modelling language for configuring the deployment and provisioning of the serverless functions. Finally, it enables to monitor the health of the platform while it provides support for WSGI apps through the frameworks of Flask, Django and Pyramid.

III. Assessment Criteria

The classical lifecycle of a software application can be seen in Fig. 1, where different colours are used to denote the respective flows involved (normal plus runtime and testing adaptation). As it can be seen, the testing flow enables to move back until the design phase of the application while the runtime adaptation flow can even reach the initial phase of requirements analysis. Due to the current focus of most cloud-based application management frameworks, the first initial phase in the lifecycle is ignored. As such, the focus of serverless frameworks should be at least in the 6 remaining phases. Such a focus is examined in this paper through the use of a set of criteria organised into 7 categories, 6 according to the lifecycle phases plus the cross-cutting category of security. In the following, these criteria are analysed according to the category in which they belong.

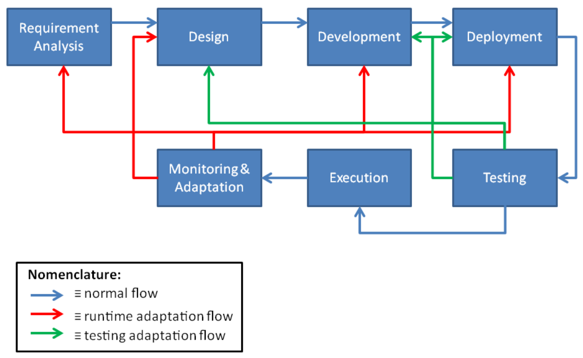

Fig. 1. The classical software application lifecycle.

A. Design

As expected, this phase is not very well supported. To this end, we have identified the following two main criteria which are related to this phase and are supported in a certain degree by the respective frameworks.

a) Composition: As an application can comprise multiple functions, it is essential to also explicate the way in which these functions can be executed. As such, a certain composition flow needs to be defined and executed at runtime.

[9]https://serverless.com/framework/docs/getting-started/

b) FaaSification: This maps to a process [11] that can be followed for migrating existing code into a set of functions. This process is essential as not all application components can be faasified and for those that can, it is not always possible to support this in a fully automated manner. Thus, such a process can provide assistance to devops in appropriately identifying those application parts that can be fully faasified so as to focus their development and manual faasification effort on the rest.

B. Development

This is a process that has concentrated most of the work in serverless frameworks. As such, we supply the following set of criteria to assist in the proper identification of those frameworks that prevail according to which development aspects.

a) Language: It is essential that a framework can support the development of functions in multiple languages to satisfy the diverse needs of multiple developers.

b) Function Development Kits – FDKs: The supply of FDKs can be a nice add-on to support a more focused, specialised and rapid development of functions in different programming languages.

c) Integration: Different frameworks and platforms might exhibit different features. In this respect, it can be beneficial to have the ability to integrate them together or to enable switching from one to the other.

C. Deployment

This is another phase of core focus in serverless frameworks. It represents the most crucial step for having a function up and running. Thus, the next criteria have been considered.

a) Continuous Integration & Deployment – CI/CD: The application code can undergo constant changes. In this respect, it is essential if there is support in the automation of code downloading (e.g., from code repositories), packaging and deployment to enable executing the most recent and up-to-date version of a function.

b) Versioning: Functions could be upgraded into different versions while some versions could be development / testing ones. As such, it is essential to support versioned function deployment to enable the exploitation of such different function versions by user application or external clients.

D. Testing

For this phase, we have utilised a single criterion to investigate the support of a framework to different types of testing, such as unit and integration testing.

E. Execution

Executing a function could be straightforward most of the times but it might require using different means to achieve this. In the respect, the following two criteria have been considered.

a) Event Coverage: Serverless computing is a kind of event programming paradigm [2]. In this sense, at its core is the notion of events which can be used to trigger functions. As such, with this criterion we evaluate the kind of events that can be supported by a framework.

b) Execution Support: Here we investigate the existence of a CLI or UI through which functions can be called. A UI might be more intuitive in this case as it could enable also to instantly check the execution status of the function call.

F. Monitoring & Adaptation

All serverless frameworks do not support function adaptation. As such, the focus logically leans towards function monitoring and the following criteria have been considered.

a) Logging: Many frameworks provide support for a simple or advanced form of logging which enables to observe the status of function calls. In some cases, the logs could also unveil the details of some metric measurements.

b) Metric Support: To support the full monitoring of an application, there is a need for realising a right set of metrics which are directly related to the application non-functional requirements. With this criterion, we examine the richness of the metric set supported by the respective framework.

c) Monitoring UI: The use of a monitoring UI or dashboard can enable the continuous monitoring of the functions deployed by the devops which can enable the quick detection of issues and their possible manual remedy.

G. Security

For this phase, we consider the ability of a framework to support both the authentication and authorisation tasks to regulate a controlled access to the deployed functions.

IV. Evaluation Results

We summarise the evaluation results produced in Table I in which on the left side we supply the criteria and on the top the frameworks examined such that each column covers the performance of one framework over all the criteria considered.

The analysis of the above results follows two directions: (a) examining which framework seems to be the best or most promising; (b) checking how well the software application lifecycle is supported by the serverless frameworks.

The best framework is certainly Serverless, as it seems to cover almost all criteria set apart from the FaaSification one. This indicates that this framework has been indeed designed with the main goal to support as much as possible the devops in serverless application development and operation. In our view, this framework's main strengths concern its ability to support a plethora of languages, unit testing as well as a great set of metrics (invocation, error, duration, count and throttle-based) powered by CloudWatch[10]. However, we should mention that the latter means that this framework integrates well with AWS Lambda but it is not certain whether it has the same level of support for other serverless platforms.

Sparta is placed second with 3 unsupported criteria and having as its main strengths the ability to compose functions as well as offer an advanced form of logging. Finally, Fn is placed third exhibiting the main advantages of function composition, unit testing and rich set of metrics being supported.

The lifecycle support level is analysed in the following paragraphs.

[10]https://aws.amazon.com/cloudwatch

a) Design: In overall, we can see that this phase is not very well supported, with most work being concentrated in function composition. For the latter, we see that more than half of the frameworks are able to support it. However, two of these frameworks seem to rather support AWS Step, i.e., a technology that works only on AWS Lambda. In this respect, we could consider Fission and Fn as the best performers for this criterion. Fission offers the Workflows technology enabling to specify function workflows which can be implicitly sequentially executed (where the output of one function is the input of the other) while other well-known workflow/programming constructs are supplied like if-the-else, for / while loops and switch. Fn Flow, on the other hand, currently supports only Java function composition with the philosophy of using the constructs of an existing language to define the right control flow. Bindings for other languages are expected to be supplied in the near future. As such, due to this latter restriction, we could nominate Fission Workflows as the most suitable function composition technology for the current moment.

Concerning FaaSification, only one framework is able to support it, Snafu, which is a rather disappointing result. Nevertheless, Snafu supplies tools for both Java [11] and Python [12] with nice results and high quality. However, as the Snafu developers point out, especially for Java, the main difficulties in FaaSification stem from the fact that original code is not written for single function access while other issues relate to interfacing with the JVM and the command line. As such, the handling of dynamic classloading and server-side state manipulation are left out as future work directions.

b) Development: Concerning the development phase, the results are better. Apart from the fact that all frameworks provide baseline support for this phase due to the offering of a CLI for code packaging, we can observe that two out of the three criteria that have been set seem well supported by many frameworks.

At the field of the programming language, almost all frameworks apart from Sparta support at least two programming languages. The best performers are surely Fission and Fn which rely on container technology to package code in any language. In the second place, we would place Serverless with its ability to support 11 languages. However, we should mention that Serverless claims this support as it abstracts from 5 main serverless platforms, where each one provides support for a different subset of the languages claimed. This can mean that the need for an unpopular language would increase the risk of getting locked to a certain platform, which seems as a quite natural result.

Fn can be distinguished as the sole framework supplying FDKs to devops with respect to particular programming languages. Such FDKs comprise a well curated image, the ability to access configuration information via an API, data binding facilities as well as capabilities for unit testing. As such, this seems a rather added-value feature which has been greatly neglected by the rest of the frameworks.

Finally, concerning integration, we can definitely see that

TABLE I
THE SERVERLESS FRAMEWORK EVALUATION RESULTS

		Serverless Frameworks						
Phase	**Criterion**	Fission	Kubeless	IronFunctions	Sparta	Fn	Snafu	Serverless
Design	Compos.	Workflows	–	–	AWS Step	Flow	–	AWS Step
	Faasif.	–	–	–	–	–	Yes	–
Develop.	Language	NodeJS, Python, Ruby, Go, PHP, Bash, any linux executable	NodeJS, Python, Ruby, PHP, Go, Ballerina	Go, .NET, Javascript, Java, Lambda, Python, Ruby, Rust	Go	Any	Python, Java	Javascript, C#, F#, Scala, Python, Java, Golang, Groovy, Kotlin, PHP & Swift
	FDKs	–	–	–	–	Yes	–	–
	Integr.	–	–	Picasso	AWS Lambda	AWS Lambda	AWS Lambda, OpenWhisk, Fission, Kubeless	AWS Lambda, Azure Functions Google Functions, OpenWhisk SpotInst, kubeless, Fn
Deploy.	CI/CD	–	–	–	Yes	–	–	Yes
	Vers.	–	–	–	Yes	–	–	Yes
Testing		–	–	–	–	Unit	–	Unit
Exec.	Event Cov.	HTTP, Cron, MQ	HTTP, Cron, MQ, Stream	HTTP, MQ, Alarm	HTTP, MQ, Stream	HTTP, MQ	HTTP, MQ FS, Cron	HTTP, Cron, MQ, Stream
	Support	CLI	Both	Both	CLI	Both	CLI	CLI
Monit.	Logging	Simple	Adv.	Simple	Simple	Adv.	Simple	Adv.
	Metrics	Resource	(Succ.) Call Num Exec. Time	Not Def.	Custom	Count, Duration, Resource	Exec. Time	Cloudwatch*
	UI	–	Yes	–	Yes	Yes	–	Yes
Sec.		–	U/RBAC Adv. Auth.	2-Level Auth	RBAC	–	UBAC Basic Auth	u/RBAC Adv. Auth

5 out of the 7 framework provide some sort of integration with another framework or platform. IronFunctions integrates with the Picasso abstraction framework. Snafu has import capabilities from two serverless platforms and export ones to two other frameworks. In addition, four frameworks support AWS Lambda but the 3 of them only via the ability to deploy functions in this platform. We could consider as a clear winner here Serverless due also to its ability to integrate with provisioning frameworks which enables to construct a full abstraction and provisioning solution for serverless applications.

c) Deployment: By observing Table I, someone would misjudge that the frameworks do not support function deployment at all. On the contrary, they do via the offering of appropriate CLIs. However, due to the modern way via applications are built with a rather agile development and provisioning lifecycle, we needed to examine if the frameworks could provide good or excellent support to this overall pipeline.

In this respect, we can clearly see this support is really bad in overall, with only two frameworks offering a certain solution to the considered criteria. With respect to CI/CD, Sparta offers two respective solutions. The first relies on AWS CodePipeline[11] while the second on a custom solution based on other AWS services. On the other hand, Serverless relies only on CodePipeline. In both frameworks, through the use of AWS CodeDeploy[12], it is now possible to also gradually shift the traffic from an old to a new function version.

Concerning versioning, again only Sparta and Serverless offer support for it mainly in the context of AWS Lambda. Which means that someone would have to lock-in to this platform to be able to realise this feature. We hope that this situation changes in the near future.

[11]https://aws.amazon.com/codepipeline/
[12]https://aws.amazon.com/codedeploy/

d) Testing: Concerning testing, we do see some basic support being supplied by almost all frameworks as they enable to either locally execute a certain function or to execute it once it is deployed in a remote platform or infrastructure. Only two frameworks go beyond this to support some form of unit testing, i.e., Fn and Serverless. Fn offers some FDKs which enable the use of unit testing facilities for functions developed in particular languages (currently only for Java via JUnit and Javascript via the combination of tape/wire/mock). On the other hand, the Mocha unit testing framework for Javascript is supported by Serverless. Thus, in overall, we do observe some unit testing support for a small number of languages while integration testing, in any form, is not supported at all.

e) Execution: This is a phase very well-supported by the frameworks. In terms of event coverage, we see that in average at least three different kinds of events are supported by a framework. HTTP and Message Queue(MQ)-based events are commonly supported by all frameworks. Stream-based events are supported via the use of the AWS Kinesis technology. File-system based events are uniquely handled by Snafu which can be considered as file change based alarms.

The frameworks that can be distinguished for this criterion are Kubeless, Snafu and Serverless as they support 4 kinds of events. As expected, Serverless relies mostly on AWS technologies to support the different event kinds. However, in this case, it is advertising that it exhibits its own Event Gateway which can support different event kinds across different platforms.

Concerning execution support, all frameworks, as expected, offer a CLI to execute serverless functions (either locally or remotely). However, three can be distinguished as they also offer a respective UI via which not only calls can be performed in a parameterised way but also the routes to the respective functions can be appropriately modified/configured.

f) Monitoring: This phase seems to be well covered with all frameworks supplying logging mechanisms and a majority of them also a certain set of common metrics.

With respect to logging, the frameworks that can be discerned are Kubeless, Fn and Serverless. Kubeless offers HTTP request logging via Morgan[13] as well as its own logging mechanism with filtering capabilities over functions, pods and runtime containers. Fn recommends the use of logspout[14] for forwarding logs to an aggregator while exploits docker syslog driver for remote logging. It also enables tracing via either Zipkin[15] or Jaeger[16]. Finally, Serverless relies on CloudWatch also for logging and supplies a CLI enabling to filter the logs via region, state, start time and interval for one or all functions.

Concerning the metrics being supported, it seems that *execution time* is the one mostly measured with *invocation count* coming second. However, we should note that not all frameworks support the right metrics. Fission only supports raw metrics, Sparta indicates that only custom metrics can be realised by the devops while IronFunctions does not explicate at all which metrics it can measure. From the rest of the frameworks, we can discern Fn and Serverless. Fn supports three different kinds of metric types. i.e., *count*, *duration* and *resource* ones. The former two kinds involve, apart from basic metrics, extra ones like *function build time* in containers. The latter kinds maps to container-based resource usage metrics. Finally, Serverless relies on CloudWatch to support 5 different kinds of metrics, as indicated previously.

As a monitoring UI is quite useful for the proper observation of the non-functional behaviour of serverless applications and their functions, 4 frameworks do offer it. As Kubeless and Fn rely on Prometheus[17] for the function monitoring, they exploit Grafana[18] on top of Prometheus for this reason. Sparta indicates that custom metrics can be stored in InfluxDB[19] and then different monitoring UIs like Grafana can be used for the visualisation of the respective measurements. Finally, Serverless relies on the CloudWatch UI for this purpose.

g) Security: Security and access control in particular over functions seem not to be well supported by the frameworks. In fact, 2 frameworks do not offer any kind of access control support to devops. Further, other two frameworks concentrate either on authentication or authorisation alone which is of course quite limiting. In this respect, only 3 frameworks do provide a complete access control solution. In terms of the authorisation model enforced, Role-Based Access Control (RBAC) is mostly utilised.

Concerning authentication, two frameworks can be discerned that offer advanced capabilities. Kubeless, apart from basic authentication, allows the switching to more advanced forms, including Key, OAuth 2.0, JWT, ACL, HMAC, and LDAP. On the other hand, Serverless also supports multiple authentication providers as well as offers an authentication/authorisation API via a plugin/extension while enables custom authorisation support at the AWS API Gateway level.

[13]https://github.com/expressjs/morgan
[14]https://github.com/gliderlabs/logspout
[15]https://zipkin.io/
[16]https://www.jaegertracing.io
[17]https://prometheus.io/
[18]https://grafana.com
[19]https://www.influxdata.com

V. Challenges

We envision an abstraction framework enabling users to provision mixed multi-cloud applications across different cloud platforms and infrastructures. As such, we believe that this can be achieved via two different directions based on the current available tooling. First, serverless frameworks must be integrated with multi-cloud application management frameworks to enable the handling of mixed applications. Second, serverless frameworks must get improved to truly become suitable serverless application management facilities.

The framework that currently prevails rightly moves towards the latter direction. However, Serverless needs to become more independent from AWS Lambda and the rest of Amazon AWS technologies. The main issue concerns supporting a set of features that can be exploited across all serverless platforms or at least non-singular sets of those. Only then Serverless can be a true abstraction framework. Serverless should also adopt a higher-level modelling language that abstracts away from AWS technicalities, in contrast to the current CloudFormation-like language adopted, and focuses more on how a serverless application can be configured independently of any cloud provider or platform. Model-driven application management is indeed the right way to go but it needs to be assorted with the right abstraction mechanisms and languages to achieve the most suitable automation level while catering for usability and reduced modelling complexity.

Focusing now in general on the main issues identified in the serverless framework evaluation and having our above vision in mind, we supply a set of challenges that need to be appropriately addressed across all phases of the mixed application lifecycle.

A. Design

a) Challenge C_1: A mixed application can include both normal as well as serverless components (i.e., functions), where normal are components meant to be provisioned by VMs or containers. In this respect, there is a need to appropriately design an application by taking into account the component heterogeneity, the diverse application requirements as well as the expected workload to be addressed. This rises the challenge to develop novel design methods and techniques for mixed cross- & multi-cloud applications.

b) Challenge C_2: Part of such design methods could be mechanisms which check how and when whole applications or their parts can be FaaSified. This then raises the need for: (a) devising FaaS-readiness tools which check which applications can be partially or fully FaaSified; (b) advancing current FaaSification tools to overcome the current difficulties as well as expand over other programming languages.

c) Challenge C_3: To realise mixed business applications and processes, there is currently the need to endorse the paradigm of micro- and nano-services. However, current state-of-the-art workflow languages, like BPMN, do not yet support even RESTful services which are at the core of both programming kinds. However, we do believe that the advent of serverless workflow languages and technologies is the right way to go forward, at least for the nano-service computing paradigm, but they need to be equipped with the vast experience and knowledge in workflow modelling so as to evolve into fully-fledged workflow languages that can support additional types of control and data flow constructs. Further, they need to be better integrated with the different types of events that can be supported by serverless platforms or frameworks. Possibly some lessons learnt from the scientific computing domain could also be adopted in order to further evolve these languages towards data awareness and maximum parallelisation of different kinds.

B. Development

a) Challenge C_4: As language independence is already there, serverless frameworks should not perform the mistake to attempt to attract devops for developing their applications in their own environment. On the contrary, devops should still stay with their favourite development environments. Thus, serverless frameworks should be integrated as plugins in such environments to offer different kinds of facilities, like FaaSification, packaging and deployment. We believe that this will certainly make both serverless frameworks really usable as well as enable a better uptake of serverless computing.

b) Challenge C_5: FDKs are the right way to go forward. As they can provide the right facilities via which functions can be developed. However, apart from the fact that FDKs are currently restricted within one framework, they are applicable only for a small set of programming languages with offered capabilities getting more restrained depending on the popularity of the respective language. We also believe that there is a need for incorporating additional constructs in FDKs enabling to support the following: (a) enhanced error handling with customised error messages and well-established status sets (e.g., originating from HTTP); (b) proper data binding and capabilities to extend it; (c) capabilities to enable functions to make arbitrary calls to any kind of function or component.

C. Deployment

a) Challenge C_6: While an abstraction framework enables a devops user to specify configuration details for functions or whole applications in a platform/cloud-independent way, there is a point where deployment decisions need to be made for both an application's normal components and functions. Such decisions need to take into account both global and local application requirements as well as certain optimisation objectives. In this respect, an appropriate deployment reasoning process should be in place which could perform the following two tasks: (a) $C_{6.1}$: matchmake component requirements with cloud/platform capabilities. Certainly we have seen frameworks supporting this for normal components but this is definitely not the case for functions, while it is more than certain that serverless platforms come with certain restrictions which might be undesirable with respect to the function requirements; (b) $C_{6.2}$ solve an optimisation problem which attempts to map, in the best possible way, each normal application component to the most appropriate VM/container offering as well as each function to the most appropriate serverless platform so as to optimise the objectives that have been set.

b) Challenge C_7: To support component matchmaking and deployment reasoning, there is also a need for two essential elements: (a) $C_{7.1}$: an appropriate modelling language able to specify all suitable details of a mixed application, spanning all possible aspects, including the deployment one, in a provider/platform-independent way. Some good candidates like TOSCA [13] or CAMEL [14] could be used to realise this; (b) $C_{7.2}$: supply a language for the modelling of provider/platform capabilities along with a respective matchmaking framework, encapsulating the offer descriptions in the form of a repository, and thus enabling the matching of function/normal component requirements with the corresponding platform/infrastructure capabilities.

c) Challenge C_8: Provisioning frameworks enable to provision a whole serverless platform in a form of a cluster over a certain cloud (or even across clouds). However, they do impose the definition of the dynamics of such platform, with respect to how it should scale, to the devop. Thus, this is certainly a place where more automation could be achieved. For instance, the user could supply some high-level parameter values and constraints over the desired application and component performance and then the provisioning frameworks, along with additional configuration details concerning the components themselves (e.g., expected size), could derive the best possible way the platform could evolve. This could be achieved, for example, via learning or reasoning over the platform's scaling rules. In our opinion, custom serverless platforms do enable to bypass current restrictions of commercial serverless platforms. Thus, they could be the right means to still adopt serverless computing even when serverless components might have a greater size or time of execution with respect to what is allowed by commercial serverless platforms. In essence, the devops users might still face another deployment option: instead of just choosing a public serverless platform, they can deploy specific application functions in their private one. This could then raise the complexity of the deployment reasoning as designated in Challenge C_6.

D. Testing

a) Challenge C_9: Unit testing is currently only supported sporadically for the Java and Javascript languages. As such, we believe that unit testing should evolve to cover additional programming languages. This would then guarantee that any function, written in any language, functions as expected when executed in isolation. By taking into account Challenge C_4, we could expect that either unit testing is offered in the form of FDKs, like in Fn, or via integrating testing methods and tools within an existing development environment (in addition to the integration of a serverless framework).

b) Challenge C_{10}: As an application can comprise multiple components, it is essential that various forms of integration testing need to be performed to guarantee the right operation and behaviour for this application. The inherent difficulty is not only related to the fact that there is heterogeneity with respect to component types but also with respect to the languages that they have been developed. Thus, we expect to see an application of appropriately extended existing methods of integration testing as well as the production of customised ones which are well-suited for the serverless computing paradigm.

E. Execution

a) Challenge C_{11}: We believe that the realisation of a concept of an *Event Gateway*, as done in Serverless, constitutes a right direction for platform independence. The main issue is whether there is a need to provide the right abstraction methods and concepts, taken from event programming, which would constitute the high level over which the mapping to the platform-specific technicalities could be performed. This is definitely worth investigating in conjunction with the evolution of the right function composition language (see Challenge C_3).

F. Monitoring & Adaptation

a) Challenge C_{12}: The monitoring of functions looks like a facility well realised by the serverless frameworks. However, we believe that there are still missing opportunities and capabilities via which true monitoring of mixed applications can be supported. In particular, the capability to support custom metrics in most of the serverless frameworks is missing. Further, to accompany this capability, there is a need for aggregation capabilities over the raw metrics computed. Especially as aggregated metrics are most suitable for checking the satisfaction of respective conditions mapping to application or component requirements. While they constitute the right mechanisms to have the ability to compute the utility of objective functions. Finally, mechanisms for advanced event (pattern) detection, coupling the previous capability, are actually needed to be able to respond to problematic situations. For instance, event patterns could designate an over-usage situation of a serverless platform, dictating its respective scaling.

b) Challenge C_{13}: No notion of adaptation is currently built into a serverless platform. This is logical as the original idea is that functions can scale on demand to anticipate any kind of workload. However, there are two main issues with this consideration: (a) functions can also permanently fail. Hence, scaling a failed function is not only meaningless but also unnecessarily costly; (b) functions are part of (mixed) applications. In this sense, various kind of issues can occur which can hinder the integration between normal components and functions. For instance, network problems might require redeploying a function near a normal component. Thus, to confront both issues, there is a need for equipping serverless frameworks and platforms with the right mechanisms via which serverless applications can be adapted on demand once faulty or problematic situations occur. This includes both mechanisms to sense such situations (see previous challenge) as well as to adapt the serverless application in a cross-layer manner, an essential feature especially for privately provisioning serverless platforms.

The aim of the Functionizer project is to extend the Melodic (*melodic.cloud*) multicloud deployment platform with serverless capabilities. It will allow to optimize and deploy in a cloud provider agnostic way both normal application components and functions. As such, this project will investigate and potentially address some of the aforementioned challenges.

Acknowledgment

This work has received funding from the Functionizer Eurostars project.

References

[1] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*. Singapore: Springer Singapore, 2017, pp. 1–20.

[2] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski, "Status of serverless computing and function-as-a-service(faas) in industry and research," *CoRR*, vol. abs/1708.08028, 2017. [Online]. Available: http://arxiv.org/abs/1708.08028

[3] L. F. Albuquerque Jr, F. S. Ferraz, S. M. L. Galdino, and R. F. A. P. Oliveira, "Function-as-a-Service X Platform-as-a-Service: Towards a Comparative Study on FaaS and PaaS," in *ICSEA*. Athens, Greece: IARIA, 2017, pp. 206–212.

[4] G. Adzic and R. Chatley, "Serverless Computing: Economic and Architectural Impact," in *ESEC/FSE*. Paderborn, Germany: ACM, 2017, pp. 884–889.

[5] R. Chard, K. Chard, J. Alt, D. Y. Parkinson, S. Tuecke, and I. Foster, "Ripple: Home automation for research data management," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, June 2017, pp. 389–394.

[6] M. Malawski, "Towards Serverless Execution of Scientific Workflows – HyperFlow Case Study," in *Proceedings of the 11th Workshop on Workflows in Support of Large-Scale Science co-located with The International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2016)*, Salt Lake City, Utah, USA, 2016.

[7] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless Edge Computing: Extending Serverless Computing to the Edge of the Network," in *SYSTOR*. Haifa, Israel: ACM, 2017, pp. 28:1–28:1.

[8] E. Wolff, *Microservices: Flexible Software Architecture*. Addison-Wesley, 2017.

[9] J. Spillner, "Practical Tooling for Serverless Computing," in *UCC*. Austin, Texas, USA: ACM, 2017, pp. 185–186.

[10] ——, "Snafu: Function-as-a-service (faas) runtime design and implementation," *CoRR*, vol. abs/1703.07562, 2017.

[11] J. Spillner and S. Dorodko, "Java code analysis and transformation into AWS lambda functions," *CoRR*, vol. abs/1702.05510, 2017.

[12] J. Spillner, "Transformation of python applications into function-as-a-service deployments," *CoRR*, vol. abs/1705.08169, 2017.

[13] D. Palma and T. Spatzier, "Topology and Orchestration Specification for Cloud Applications (TOSCA)," Organization for the Advancement of Structured Information Standards (OASIS), Tech. Rep., June 2013. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cos01/TOSCA-v1.0-cos01.pdf

[14] A. Rossini, K. Kritikos, N. Nikolov, J. Domaschka, F. Griesinger, D. Seybold, and D. Romero, "D2.1.3—CAMEL Documentation," PaaSage project deliverable, October 2015.