YASF: A Vendor-Agnostic Framework for Serverless Computing

Mikael Giacomini and Amjad Ullah[a]
School of Computing, Engineering & the Built Environment, Edinburgh Napier University, Edinburgh, U.K.

Keywords: Serverless Computing, Function as a Service (FaaS), Vendor Lock-in, Vendor-Agnostic Framework, CDK.

Abstract: The serverless execution model allows application developers to deploy their software using tiny functions with zero administration, no handling of resource provisioning, monitoring and scaling. Due to such advantages, the serverless model emerged as a new promising paradigm, where pay as you go offerings can be found by all public cloud providers. However, these offerings encourage vendor lock-in. This paper aims to address the vendor lock-in issue using a novel framework that combines the strength of an agnostic infrastructure configuration based on the Constructs Programming Model, and the creation of abstraction layers that supports function logic by handling provider-specific integration. The proposed framework abstracts away the specificities and complications of the various underlying serverless platforms and the application developers are only required to provide their specific functions. The evaluation consists of the deployment of a benchmark application across different cloud providers to demonstrate the ease and flexibility of the framework.

1 INTRODUCTION

Cloud computing has immensely changed the provision of computing, both for individual and business users. Since its inception, adoption of cloud services has continued to grow. This is evident from the current market size of 2022, which is estimated as 483.98 billion $ and is expected to witness a compound annual growth rate of 15.7% to reach 1,554.94 billion $ by 2030. This is not surprising considering the inherent characteristics of cloud Infrastructure-as-a-Service (IaaS) model, which takes away the responsibility of managing hardware from customers, and offers economic benefits as well as operational efficiencies (Marston et al., 2011). This is further revolutionised by the serverless model, which has also taken away the responsibility of resource provisioning and tasks such as run-time management, monitoring, and scalability (Baldini et al., 2017). As a result, the serverless model emerged as a new promising paradigm, where pay as you go offerings can be found from all public cloud providers, e.g. AWS Lambda, Google Functions, Azure Functions, and Tencent SCF.

These offerings allow developers to just deploy their code for execution using tiny functions with no administration overhead. However, each of these rely on the use of their own underlying infrastructure and services, hence resulting in vendor lock-in, where application owners get restricted to the use of services from a specific provider. The lock-in issue is implicit to the design of cloud solutions by choice; however, it plays a role in slowing the adoption of cloud, as organisations are concerned with the associated technical and legal issues of lock-in (Opara-Martins et al., 2014). Hence, any system that helps loosen vendor lock-in, contribute to an additional growth of cloud and support a healthier environment where application owners can be empowered to freely move between different clouds (Kumar, 2019).

The lock-in issue is not specific to serverless offerings. It also remains a challenge in case of IaaS resource offerings. Over the years, a number of solutions such as Cloudify (Cloudify, 2022), Terraform (HashiCorp, 2022), MiCADO (Kiss et al., 2019; Ullah et al., 2021), Cloudiator (Baur and Domaschka, 2016), PrestoCloud (Verginadis et al., 2017), have been developed to improve the portability and interoperability of cloud solutions. Similarly, a number of such frameworks have also emerged to resolve the lock-in issue for the serverless model. For example, some industry initiatives include The Serverless Framework (Serverless, 2022), Spring Cloud (Tanzu, 2022), Midway Serverless(Midway.js, 2018), Up (Apex, 2021), , Wing (Ben-Israel, 2022) and some notable academic ones include Lithops (Sampe et al., 2021), Kotless (Tankov et al., 2019), Nimbus (Chatley

[a] https://orcid.org/0000-0002-2407-4480

Giacomini, M. and Ullah, A.
YASF: A Vendor-Agnostic Framework for Serverless Computing.
DOI: 10.5220/0011847700003488
In *Proceedings of the 13th International Conference on Cloud Computing and Services Science (CLOSER 2023)*, pages 46-56
ISBN: 978-989-758-650-7; ISSN: 2184-5042

and Allerton, 2020), Functionizer (Matei et al., 2021). These solutions have contributed towards achieving some aspects of agnosticism. However, they fell short in various aspects such as facilitating an overall agnostic functionality in terms of infrastructure configuration, function deployment and execution (e.g. (Serverless, 2022; Apex, 2021; Sampe et al., 2021; Midway.js, 2018)), lack of developers friendly abstraction layer (e.g. (Matei et al., 2021; Casale et al., 2020)) or lack of agnostic integration of extended components and platform configuration (e.g. (Tankov et al., 2019; Tanzu, 2022)). All these aspects are essential to be part of a framework that aims to provide a complete agnostic solution in terms of provider and language independence. This paper presents YASF (yet another serverless framework), which also aims to facilitate application developers to make use of fully-managed serverless environments without risking lock-in.

The rest of this paper is organised as follows. Section 2 consists of state-of-the-art, where a thorough review of related works in Section 2.2 is carried out in light of a novel taxonomy (Section 2.1). Section 3 reflects on the reviewed solutions and identifies a set of requirements for YASF. Section 4, presents YASF framework, where Section 5 demonstrates the applicability of YASF using a benchmark application and two public clouds. Lastly, Section 6, concludes and briefly discusses future work.

2 STATE OF THE ART REVIEW

This section discusses the related vendor-agnostic serverless solutions (VaSS). However, to perform a thorough review, we initially introduce a novel taxonomy. This enables us to accumulate, analyse and synthesise the related work, using a uniform framework. This also aims to highlight the importance of why there is the need to propose yet another serverless solution. The following section briefly introduces the proposed taxonomy, where the next section presents the review.

2.1 Taxonomy

Figure 1 presents our taxonomy, which consists of attributes that are essential to the core implementation of a VaSS. These attributes are identified after a thorough review of existing solutions. These attributes are structured into following categories:

Functionalities grouped together the following key operations that a VaSS should provide to its users:

- Infrastructure configuration: A VaSS should provide a mechanism that enables developers to describe the infrastructure requirements of their serverless functions, without having to deal with the underlying vendor-specific configurations. This attribute measures whether a VaSS provides such a support in an agnostic way or not.
- Deployment: Function developers require tools to be able to deploy functions along its required configurations to a specific cloud provider without investing additional efforts especially in regards to target a different (or new) provider. The deployment can be either *Managed* — where the deployment is carried by the solution — or *Not managed* — where developers manually handle the deployment of additional assets.
- Function execution: Different cloud providers have different requirements in relation to the specification of events and context information. These requirements affect the way developers parse inputs/outputs of their functions. If not handled properly, this may result in an incompatible format. In this regard, this attribute evaluates the underlying abstraction mechanism of a VaSS, which can be either *Provider agnostic* — where a VaSS provides an abstraction layer that allows developer's code to be agnostic — or *Provider dependant* — where developers are required to produce different function assets for each cloud provider.

Design grouped together aspects that are related to the design of a serverless solution itself. These include the following aspects:

1. Integration: Serverless functions usually interact with other infrastructure components, such as with storage solutions, databases, and event queues. The VaSSs are therefore required to provide a mechanism, which facilitates developers to integrate functions with other components. Such a support can be evaluated in two aspects: 1) the underlying nature that the integration can be handled within a solution, i.e. either in a provider agnostic or dependent way, and (2) the level of provided integration, which can be either only at the *Function level* — the bare minimum requirement for a solution — or also include the *Extended level* — where agnostic support for interaction with additional components is provided.
2. Extendability: The success of a VaSS mainly relies on the following key factors: how easy a solution adapts to vendor changes, enabling support for new vendors, languages, and the ability to maintain low complexity (or no changes) from users viewpoint in case new capabilities are added. Hence, it is important for a VaSS to sup-

port the ability to incorporate new vendors, and languages.

Current support evaluate existing ability of VaSSs in terms of *Providers*, *Languages*, and their suitability towards target *Application type/s*. This evaluation helps in gauging the overall maturity of a VaSS,

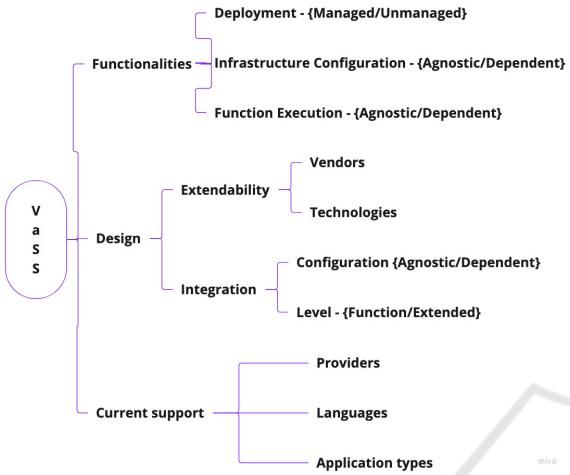

Figure 1: Taxonomy.

2.2 Review of Existing Solutions

The solutions selected for review is the result of a two step process: 1) We searched for relevant solutions using terms like *cloud (or multi-cloud or vendor) agnostic (or independent) serverless (or FaaS) frameworks (or solutions)*, 2) From the results, we further selected solutions that have mainly focused on vendor lock-in issue. During step 1, we commonly encountered solutions such as Fn, Knative, and OpenFaaS as vendor-agnostic serverless solutions. These and other such solutions followed a serverless approach. However, these solutions generally created a containerised support for the functions that are further deployable across multiple clouds. This is undoubtedly a valid approach to tackle vendor lock-in. However, it also results in giving up cloud native FaaS features such as monitoring, billing per execution, and container free management. Hence, all such solutions are considered as non-native FaaS platforms and therefore, do not fall within the scope of the review carried out in this paper.

All relevant solutions are grouped into two categories, namely industry and academic initiatives. These are discussed in following sections, where their summaries in light of the attributes from taxonomy are presented in Table 1 and 2.

2.2.1 Industry Initiatives

The Serverless Framework (TSF) (Serverless, 2022) supports a large number of cloud providers and programming languages. Furthermore, it also provides a huge number of plugins to support integration with other AWS components such as API Gateway, Step Functions, Alerts, and DynamoDB. However, the configuration needed for enabling the integration is required to be provided in an AWS specific manner, which encourages lock-in. Furthermore, the same level of support is not offered to integrate components from other vendors. The Serverless Multicloud Library (Microsoft, 2019) extends the TSF solution with the key purpose of providing a more complete provider-agnostic experience to users. This solution, in contrast to TSF, only includes support for AWS and Azure, where the support for Google cloud platform is currently in progress. However, it has achieved an extended level of agnosticism by enabling the support to integrate the main components of both Azure and AWS.

Spring Cloud Function (SCF) (Tanzu, 2022) also supports a large number of vendors; however, it only supports functions in Java language. Similarly, the Midway Serverless framework (Midway.js, 2018) only supports Node.js. SCF's agnostic behaviour is limited to function level and does not facilitate agnostic integration of other components as part of the deployment phase. However, it takes 'recognising and parsing incoming events and output compatibility of the functions' into consideration. In contrast to the SCF, the Midway Serverless (Midway.js, 2018) offers some integration with different cloud's storage solutions, e.g. with Alibaba (Alibaba, 2022) and Tencent (Tencent, 2022) storage systems. However, the provided procedure requires to perform the integration is not agnostic. Moreover, similar to TSF, both these solutions also require making explicit cloud-specific changes when switching the deployment across different cloud environments.

The Up framework (Apex, 2021), despite claiming to be a platform-agnostic, its current support is only limited to AWS. Up supports multiple programming languages including Golang, Node.js, Crystal, Python, Java and Clojure. However, additional features are hidden behind a commercial plan. Lastly, Wing (Ben-Israel, 2022) distincts itself from other solutions by proposing a new programming language, which incorporates both infrastructure configuration and functions logic in a single asset. Wing language definitions are agnostic. However, their solution currently only supports AWS.

2.2.2 Academic Initiatives

The scope of Functionizer (Matei et al., 2021) is larger than just serverless agnostic behaviour across different providers. It also facilitates simultaneous cross-cloud deployment, real time monitoring, and most importantly, utility-based optimisation. Functionizer uses CAMEL modelling language (Camel, 2022), where developers write application specification as CAMEL model, which has the ability to directly define a constraint programming model. Functionizer accepts CAMEL model to solve the defined cloud optimisation problem. A similar cross-cloud approach, however, without the optimisation aspect, is also proposed by Wurster et al. (Wurster et al., 2018), where a well-known standardised modelling language for cloud orchestration named TOSCA (Lauwers and Tamburri, 2022) has been used for abstraction. Wurster et al. suggests the use of existing TOSCA constructs (e.g., Node templates, Node and Relationship types) to describe aspects of serverless application (e.g., events, functions and event source). In such a case, the interpretation and mapping of existing constructs of TOSCA rely on the modeller, hence it cannot be adopted as a generic approach.

TOSCA currently lacks native support to model serverless application (Wen et al., 2022). To address this (Casale et al., 2020) within the scope of an in-progress EU research project called RADON, aimed to extend TOSCA with new constructs for modelling serverless functions. Furthermore, they also introduce a constraint definition language (CDL) for the formal specification of serverless application requirements with a key purpose to increase automation in their design. RADON aims to support serverless functions, microservices, and data pipelines using a uniform description approach by utilising the combination of TOSCA and CDL annotation. Yussupov et al. (Yussupov et al., 2022) focused on a similar approach, where they proposed the use of business process model and notation (BPMN) and TOSCA to model orchestration of functions and their automatic deployment. This solution, however, require modellers to produce two different models. Firstly, to produce a generic BPMN-based model representing the function orchestrations, which could be transformed into a provider specific orchestration model (e.g., ASL model for AWS). Secondly, to produce a TOSCA model to define technology-agnostic function orchestration deployment model, which can be executed (i.e., deployed and orchestrated) by any TOSCA-compliant orchestrator.

Nimbus (Chatley and Allerton, 2020) makes use of annotation-based configuration in contrast to the above-mentioned approaches that are based on separate configuration (or an additional layer of abstraction). Nimbus approach allows developers to define functions and resources using internal annotations. Such an approach reduces the need for explicit configuration, however, their implementation approach is function-centric, where the extended integrations are supported, however, these are not deployed with the function but expected to have already been implemented in an external infrastructure configuration. Furthermore, though, the overall architecture of Nimbus is cloud-agnostic; their current support is limited to AWS only. Moreover, it only supports Java Functions. Similarly, Lithops (Sampe et al., 2021) is for the Python language and specifically designed to support parallel data intensive systems that follow the Map-Reduce programming model. In contrast to Nimbus, Lithops has the support for a large number of cloud providers and relies on an external YAML based configuration. However, its agnostic behaviour is limited in terms of storage services and not extended to other services of the providers. Lastly, Kotless (Tankov et al., 2019) is specifically for the Kotlin language, where the use of a client side DSL is proposed to facilitate developers to write their functions using the proposed DSL annotations to automatically generate Terraform based deployment code. The issue with Kotless is that it does not interact with cloud providers to deploy the code. Instead, it provides another Gradle based deployment plugin that is used for deployment. Furthermore, various infrastructure configuration related attributes, such as integration with storage components, is not agnostic.

3 REQUIREMENTS

The following paragraphs reflect on the review carried out in Section 2.2 and introduce key requirements for our proposed solution.

Developer's Friendly Abstraction Layer. The core function of a VaSS is to facilitate developers with an abstraction mechanism for describing their applications. The reviewed solutions address this challenge in two ways: 1) using a custom DSL or annotation-based configurations such as Kotless (Tankov et al., 2019), Nimbus (Chatley and Allerton, 2020), Lithops (Sampe et al., 2021); 2) using a standardised modelling language, such as the use of Camel by Functionizer (Matei et al., 2021) or the use of TOSCA in (Wurster et al., 2018; Casale et al., 2020; Yussupov et al., 2022). Both of these approaches facilitate agnostic behaviour. However, such approaches also expect developers to learn an-

Table 1: Key functionalities and design characteristics of the reviewed solutions.

		Key operations			Design			
					Platform integration		Extendibility	
		Agnostic Infrastructure Configuration	Managed Deployment	Agnostic Function Execution	Level	Agnostic Configuration	Vendors	Languages
Industry	The Serverless Framework		✓	✓	E		✓	✓
	Serverless Multicloud Library	✓	✓	✓	E	✓	✓	
	Spring Cloud Function			✓	F	✓	✓	
	Midway Serverless			✓	E		✓	
	Up				E			✓
	Wing	✓	✓	✓	E	✓	✓	
Academic	Lithops (Sampe et al., 2021)	✓		✓	E		✓	
	Kotless (Tankov et al., 2019)	✓	✓		F		✓	
	Nimbus (Chatley and Allerton, 2020)	✓	✓	✓	E	✓		
	Functionizer (Matei et al., 2021)		✓		F	✓	✓	
	Wurster et al. (Wurster et al., 2018)	✓		✓	E	✓	✓	?
	Yussupov et al. (Yussupov et al., 2022)	✓		✓	E	✓	✓	?
	RADON (Casale et al., 2020)	✓	?	✓	E	✓	✓	✓

Function level **(F)**, Extended level **(E)**

Table 2: Support currently provided by the reviewed solutions.

		Vendors												Languages									App type
		AWS	Google	Azure	Alibaba	Cloudfare	Fn	Knative	Kubeless	IBM	OpenWhisk	spotinst	Tensent	C#	Crystal	F#	Go	Java	Kotlin	Node.js	Python	Ruby	
Industry	The Serverless Framework	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓		✓	✓			✓	✓	✓	G
	Serverless Multicloud Library	✓		✓																✓			B
	Spring Cloud Function	✓	✓	✓	✓													✓					G
	Midway Serverless	✓			✓								✓							✓			B
	Up	✓													✓		✓	✓		✓	✓		B
	Wing	✓																		✓			G
Academic	Lithops	✓	✓	✓	✓					✓											✓		BP
	Kotless	✓		✓															✓				B
	Nimbus	✓																✓					G
	Functionizer	✓		✓														✓					G
	Wurster et al. (Wurster et al., 2018)	This work only consist of a prototypical implementation where OpenTOSCA was used to orchestrate an example scenario.																					G
	Yussupov et al. (Yussupov et al., 2022)	✓		✓						✓											✓		G
	RADON (Casale et al., 2020)	✓	✓	✓			✓				✓							✓					G

Generic **(G)**, Backend **(B)**, Batch Processing **(BP)**

other tool and/or modelling language for writing abstract models. Such a functionality can be avoided by empowering developers to write the required models (and configuration) using a tool of their choice.

Overall Agnostic Functionality. A VaSS, as discussed in Section 2.1, should consider the following four agnostic aspects: infrastructure configuration, function deployment, integration of extended components, and platform configuration. It is evident from Table 1, only Serverless Multicloud Library (Microsoft, 2019) and Radon (Casale et al., 2020) support these features. However, Serverless Multicloud Library only supports Node.js and is specific for batch processing type applications, where Radon, as mentioned earlier, is based on TOSCA, i.e. it does not provide a developer friendly abstraction.

Simplicity and Extendibility. Many of the reviewed solutions, e.g., Functionizer (Matei et al., 2021) and Radon (Casale et al., 2020), facilitate agnostic deployment of functions through their custom-built orchestrators. The scope of such systems is usually much larger, where they also handle other tasks such as monitoring, and scaling; and therefore, additional components need to be deployed and executed constantly. Hence, an overhead overall. In the serverless execution model, such administration tasks are the responsibility of cloud providers. Hence, a VaSS should avoid any such unnecessary overhead. Furthermore, a VaSS should follow a modular and extendable architecture such that the support for new cloud providers and languages can be easily added.

4 YASF: YET ANOTHER SERVERLESS FRAMEWORK

YASF is our proposed solution to address the abovementioned requirements. Figure 2 depicts the high level architecture of YASF. It mainly consists of three components titled YASF Abstraction, YASF Core, and YASF Resolvers. The following subsections ex-

plain these components in detail followed by a description on the usage workflow of YASF in Section 4.4.

4.1 YASF Abstraction

YASF Abstraction is the only component that is in direct contact with the developer's provided code. YASF Abstraction allows agnostic executions of developer's code across different providers, where otherwise a change in code would have been required in relation to a specific provider. More specifically, YASF Abstraction guarantees agnostic function execution and extended platform integration by providing an interface between providers' invocation format and function logic. The abstraction interface processes the provider's input into an agnostic input. Where necessary, the YASF Abstraction also transforms the function's output, such that an agnostic response can still be provided when vendors require a specific format for the function's output. Furthermore, YASF Abstraction also provides agnostic integration with other cloud components, e.g. with a database storage. As this normally requires provider-specific SDK libraries, further attention is provided such that these libraries are installed only in the function asset being deployed against the relevant target provider.

The implementation of YASF Abstraction is designed such that very minimal developer's effort is required to adapt their existing function code with YASF Abstraction. Figure 3 presents a Python based function as an example, where the developer only requires to add two objects from YASF abstraction including the *entrypoint* decorator and *GenericContext*. The *entrypoint* decorator is responsible for providing abstraction for the event trigger input and function response output, where *GenericContext* provides type definitions of the *context* variable, which contains execution metadata. These additions to user code are neither substantial nor specific to any cloud provider. In the case of switching from one cloud provider to another, not a single line of code change is required. Our abstraction approach is similar to the internal annotation-based approach provided by Nimbus (Chatley and Allerton, 2020). However, though Nimbus supports extended integration of additional components, these are not deployed with the function but expected to have already been implemented in an external infrastructure configuration. In contrast, YASF infrastructure configuration aims to be non-function specific and it is designed to support the provisioning of other cloud resources as well.

4.2 YASF Core

YASF Core exposes agnostic constructs to developers that can be used to build a generic infrastructure configuration. These constructs are based on the concept of Constructs Programming Model (CPM) (AWS, 2022), or also known as Cloud Development Kit (CDK), which developers used to write Infrastructure as Code (IaC) without having to learn a purpose-specific configuration language such as TOSCA (Lauwers and Tamburri, 2022), CAMEL (Camel, 2022), or Terraform HCL (Hashicorp, 2022). CPM empowers developers to write cloud configurations in the language of their choice and with a variety of features inherent from the use of programming languages such as loops, conditions and objects re-usability. The aforementioned reasons highly favor the choice of this technology in regard to the developer friendly abstraction. For example, consider the following code snippets:

The snippet (A) is the definition of a generic cloud provider with a requirement for region, where user function aims to be deployed. The specific details such as the choice of cloud providers and user credentials will be provided at deployment time. More details on this is provided in Section 4.4. Snippet (B) on the other hand, define the function asset by indicating the underlying language runtime — Python in this case — and other related attributes including: 1) *handler*: name of the source file, 2) *entrypoint*: name of the entrypoint function within source code, and 3) *functionAssets*: file system reference to the bundled function.

CPM is an emerging young concept for defining IaC using familiar programming languages and rich object-oriented APIs (AWS, 2020). It was initially introduced by AWS. However, it is not AWS specific and is largely adopted by the open-source community such as Terraform (Howard, 2022), Kubernetes (cdk8s, 2021) and Projen (Projen, 2020). The inherent features of CPM like the free choice of any programming language and use of common programming constructs as we seen in Figure 4 make YASF Core more developer friendly in comparison to other domain specific languages such as TOSCA, CAMEL that demands learning of additional modelling constructs.

4.3 YASF Resolvers

YASF provides a repository of Resolvers, where a specific Resolver is responsible for mapping the agnostic infrastructure configuration, written using the YASF Core component, and deploying it to a tar-

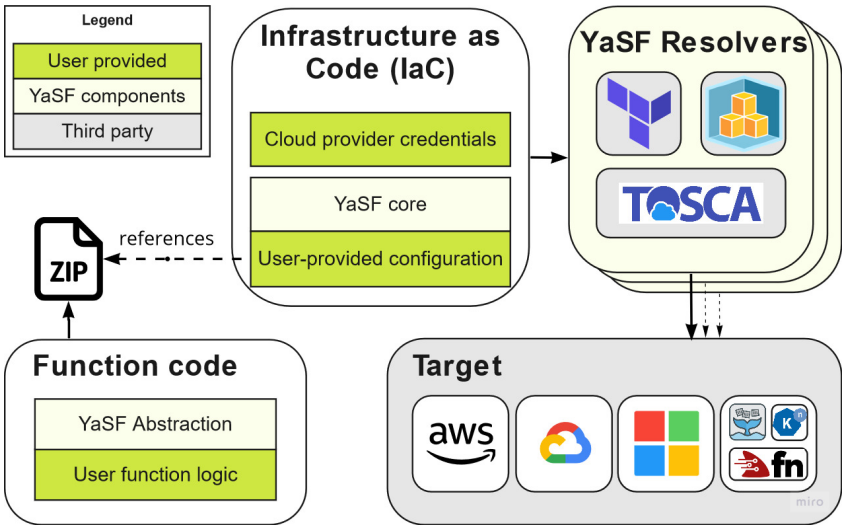

Figure 2: YASF architecture.

```
from yasf import entrypoint, GenericContext

# Handles abstraction process and inputs/outputs
@entrypoint
def handler(event: dict, context: GenericContext):
    # event/context received in same format
        ↪ independently of cloud provider
    # Developer provided code
    # ...
```

Figure 3: Usage demonstration of abstraction decorator.

```
new GenericProvider(this,
{
    continent: CONTINENTS.Europe,
    coordinates: COORDINATES.West,
    location: "2",
});
```

(a) Cloud requirements.

```
new GenericFunction(this,"demo-fun",{
    runtime: RUNTIMES.PYTHON39,
    handler: "demo_handler",
    entrypoint: "main",
    functionAssets: demoAssets,
});
```

(b) Function definition.

Figure 4: YASF Core snippets.

get cloud provider. A single Resolver supports one or multiple cloud providers. Overall, YASF follows a generic and modular approach, where each of the three components work independently from each other. YASF facilitates easy extendability, where a custom Resolver can be developed to target a new infrastructure (either private or public), or containerised serverless solutions such as Knative, Fn and OpenFaas. Adding a new Resolver will not have any impact on existing support.

From a technical viewpoint, YASF Resolvers reference YASF Core. The agnostic infrastructure constructs provided through YASF Core become input into YASF Resolver such that the agnostic infrastructure can be translated to provider-specific requirements. When utilised in a project, YASF Resolvers are dynamically loaded by YASF Core, meaning that a change of Resolver does not require a change of imports or definitions in the infrastructure configuration. Additionally, a simple CLI tool is offered that facilitates developers to select a Resolver of their choice in case multiple Resolvers are available.

4.4 YASF Usage Workflow

The usage workflow of YASF can be seen in Figure 5. The following paragraphs explain the involved steps.

1. To start, developers can simply apply YASF Abstraction into their functions' code as explained

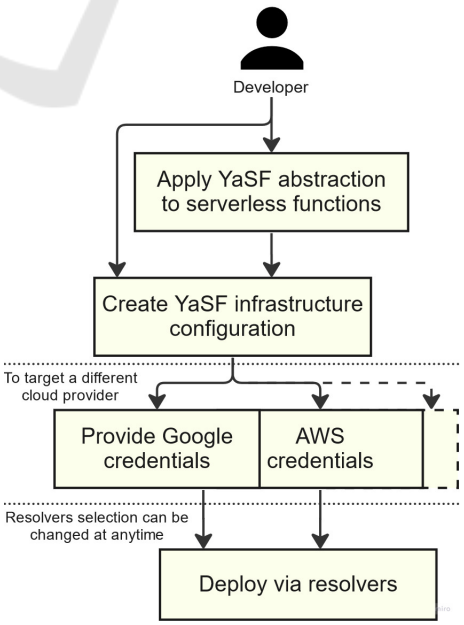

Figure 5: Developer usage steps of the solution.

earlier in Section 4.1. Developers can also skip this step, if the aim is to target a specific provider, or the required infrastructure does not include any function, or the target code already includes some abstraction that can manage function execution on a given provider.

2. This step is compulsory, where a CDK-based configuration using YASF Core needs to be produced. In this step, the developer specifies their functions, the code bundle to be deployed, the runtime, and other details like the configuration related to memory, location, environment variables, etc. Furthermore, if required, integration details with other services (e.g. NoSQL database) should be provided. It is important to note that YASF Core is not necessarily function-centric; for example, a configuration without any function only consisting of a deployed database for external purposes is also possible.
3. Prior to deployment, the developer is required to provide their credentials separately to the CDK-based configuration previously produced. Different cloud providers may require different information, e.g. Google Cloud Platform requires a ProjectID to be specified in addition to credentials. Providing all such details is the responsibility of developers. For both practicality and security, such details are not included as part of the CDK-based configuration. Furthermore, YASF does not handle account creation, hence any account setup is external to YASF.
4. Finally, a YASF Resolver can be used to deploy function/s and other infrastructure to a specific cloud. YASF Resolvers fetch credentials from the host system's environment variables. This approach also allows that the deployment process can be carried out locally and also in CI/CD setup, where credentials could be set as CI/CD secrets.

It is important to note, once steps 1 and 2 are completed, a developer can switch between cloud providers by repeating only steps 3 and 4, where users can deploy their functions by simply providing credentials while using the same or a different supported YASF Resolver. Once a function is deployed (see Figure 6), YASF Abstraction is the only persisted component in the final state, which handles the interaction between user code and provider execution. As mentioned earlier, developers can also skip the use of YASF Abstraction. In such a case, user code is responsible for this mediation itself. This decoupled nature of YASF components is one key difference between our solution and other related works such as (Chatley and Allerton, 2020; Serverless, 2022; Ben-Israel, 2022). Such decoupling empowers developers to freely choose to deploy their functions by applying the YASF Abstraction or simply use YASF framework for deployment.

5 USE CASE DEMONSTRATION

YASF[1] is fully agnostic. Currently, we have implemented two YASF Resolvers to support AWS and Google environments. Both these Resolvers are based on a well-known IaC orchestrator called Terraform (HashiCorp, 2022), which supports around 1000 providers (HashiCorp, 2022). The implemented YASF Resolvers provide a mapping between the developer-provided agnostic configuration and corresponding Terraform based provider specific constructs. While Terraform was chosen for the development of the first two Resolvers, this does not prevent any other technology to be used for new Resolvers. The remainder of the section further discusses the use of a benchmark application for demonstrating the applicability of YASF by deploying it to two different clouds without making any changes.

The benchmark application (Tai Nguyen Bui, 2019) is a prototype for a basic inventory system that includes CRUD operations on products using a NoSQL database. Each of these operations is encapsulated in a different function. This application was originally developed for AWS and therefore it uses AWS specific input/output formatting, and vendor-specific calls to DynamoDB, which is an Amazon specific NoSQL database.

To make the benchmark application executable across different cloud environments, it needs to decouple from the specificities of AWS, especially

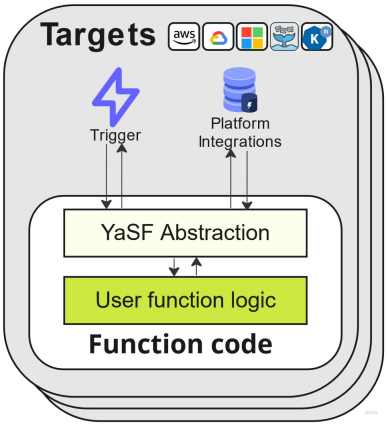

Figure 6: Function post deployment architecture.

[1]YASF implementation: https://github.com/YaSF-serverless

the use of AWS specific input/outputs formatting, and calls to DynamoDB. This is performed using the YASF Abstraction step discussed earlier in Section 4.4. The difference before and after the use of NoSQL integration can be seen in Figure 7.

Following the above abstraction step, a vendor-agnostic infrastructure configuration (see Figure 8) is prepared. The lines L:4-7 are responsible for defining the region for execution, L:8-9 provision a NoSQL Database with the specified primary key, L:10-28 define the provision of 5 serverless functions. This also demonstrates how the advantage of a programming language enables code reusability in configuration. Finally, L:29-30 provide each function access rights to the provisioned NoSQL database.

Following the above two steps, the benchmark application can be deployed into any cloud environment without changing code and also without re-writing the configuration. Figure 9 depicts a post-deployment architecture of the benchmark application on the AWS and Google platforms, where the same configuration has been used across both environments. When underlying cloud-specific components have different behaviours, it is the responsibility of YASF Abstraction to provide a standard interface. Lastly, it is important to note that YASF neither deploy an additional component nor rely on any orchestration tool that requires to stay on at all time. Hence, no additional overhead in operating cost. The only minor computational overhead is the inclusion of YASF Abstraction — responsible for all the required abstraction — into the developer's function code. Hence, it directly become part of the user code and runs in the function space. The revised implementation of benchmark application is available here[2].

```
dynamoDB=boto3.resource('dynamodb'
    ↪ )
class ProductService:
  def __init__(self):
    self.dynamoDBClient =
    dynamoDB.Table(
      os.environ['TABLE']
    )
  def getItem(self, sku):
    return
      self.dynamoDBClient.get_item(
        Key={'sku': sku})
```

(a) Original implementation.

```
from yasf.nosqldb import client
class ProductService:
  def __init__(self):
    self.table_name =
      os.environ['TABLE']
  def getItem(self, sku):
    return client.get_item(
      table_name=self.table_name,
      key_name='sku',
      key_value=sku )
```

(b) YASF Abstraction.

Figure 7: YASF Abstraction integration code snippets.

[2]Benchmark application: https://github.com/YaSF-serverless/yasf/tree/main/example/aws-lambda-benchmark

```
1  class ProductsStack extends GenericStack {
2      constructor(scope: Construct, name: string) {
3          super(scope, name);
4          new GenericProvider(this, {
5              continent: CONTINENTS.Europe,
6              coordinates: COORDINATES.West,
7              location: "2", })
8          const productsDb = new GenericNoSQLDatabase(this, "products-
               ↪ db", {
9              primaryKey: "sku", })
10         const functionAssets = new FunctionAssets({
11             path: "build",
12             archiveName: "src.zip",
13             bundlingCommands: [
14                 "mkdir -p build && rm -rf build/* && cp -r src/app/*
                       ↪ build/ && cp src/requirements.txt build/",
15                 "pip install -r src/requirements-local.txt -t build/",
16                 "cd build && zip -r src.zip ." ], })
17         const createFunction = (id: string, name: string): IGenericFunction
               ↪ => new GenericFunction(this, id, {
18             runtime: RUNTIMES.PYTHON39,
19             handler: "handler",
20             functionAssets: functionAssets,
21             entrypoint: name,
22             envVariables: {
23                 "PRODUCTS_TABLE": productsDb.name, } })
24         const createProductFunction = createFunction("create-product", "
               ↪ createProduct")
25         const getProductFunction = createFunction("get-product", "
               ↪ getProduct")
26         const updateProductFunction = createFunction("update-product", "
               ↪ updateProduct")
27         const deleteProductFunction = createFunction("delete-product", "
               ↪ deleteProduct")
28         const listProductsFunction = createFunction("list-products", "
               ↪ listProducts");
29         [createProductFunction, getProductFunction, updateProductFunction
               ↪ , deleteProductFunction, listProductsFunction].forEach((
               ↪ functionInstance) => {
30             db.grantReadWriteAccess(functionInstance) }) }
31 }
```

Figure 8: YASF infrastructure configuration for the benchmark inventory application.

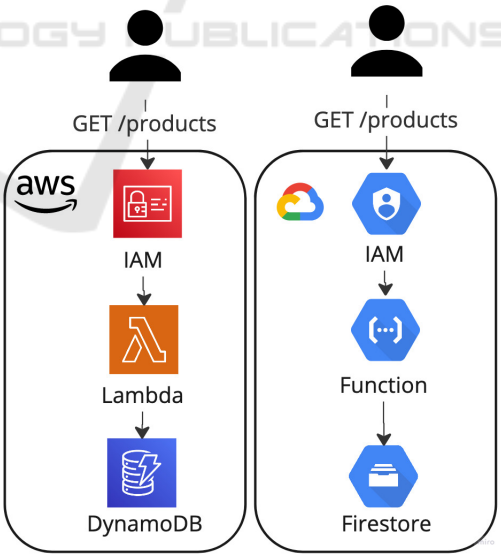

Figure 9: Use case deployment architecture.

6 CONCLUSIONS

This paper thoroughly explored existing serverless solutions that aimed to resolve the vendor lock-in is-

sue. We highlighted that the lock-in issue is still a challenge. For this purpose, we proposed YASF — a simple and generic vendor-agnostic serverless framework. YASF follows a loosely coupled, modular and extendable architecture that empowers developers to enable their serverless applications to be deployable to different cloud environments without risking vendor lock-in. YASF, in addition to the function level agnosticism, also facilitates an extended level of integrating provider services in an agnostic manner. As an initial prototypical implementation, YASF includes support for Amazon and Google cloud environments; and Python-based functions. Future work will focus on two key directions: On one hand, we aim to extend our taxonomy to evaluate the serverless solutions with respect to existing standards regarding FaaS with a particular focus on the inclusion of aspects like Function call, Return interfaces, and Triggers. On the other hand, we aim to extend support for other providers and languages; support for custom overrides and escape hatches; and finally the facilitation of YASF deployments in CI/CD environments.

REFERENCES

Alibaba (2022). Object storage service (oss)-alibabacloud.

Apex (2021). Apex up — serverless applications and apis in seconds.

AWS (2020). Overview of deployment options on aws.

AWS (2022). What is the aws cdk? - aws cloud development kit (aws cdk) v2.

Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al. (2017). Serverless computing: Current trends and open problems. *Research advances in cloud computing*, pages 1–20.

Baur, D. and Domaschka, J. (2016). Experiences from building a cross-cloud orchestration tool. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms*, pages 1–6.

Ben-Israel (2022). Wing programming language.

Camel (2022). Camel: Cloud Application Modelling and Execution Language.

Casale, G., Artač, M., Van Den Heuvel, W.-J., van Hoorn, A., Jakovits, P., Leymann, F., Long, M., Papanikolaou, V., Presenza, D., Russo, A., et al. (2020). Radon: rational decomposition and orchestration for serverless computing. *SICS Software-Intensive Cyber-Physical Systems*, 35(1):77–87.

cdk8s (2021). cdk8s.

Chatley, R. and Allerton, T. (2020). Nimbus: Improving the developer experience for serverless applications. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, pages 85–88.

Cloudify (2022). Cloudify devops automation & orchestration platform, multi cloud.

HashiCorp (2022). Browse providers — terraform registry.

Hashicorp (2022). HCL Native Syntax Specification.

HashiCorp (2022). Terraform by hashicorp.

Howard, M. (2022). Terraform–automating infrastructure as a service. *arXiv preprint arXiv:2205.10676*.

Kiss, T., Kacsuk, P., Kovács, J., Rakoczi, B., Hajnal, Á., Farkas, A., Gesmier, G., and Terstyanszky, G. (2019). Micado—microservice-based cloud application-level dynamic orchestrator. *Future Generation Computer Systems*, 94:937–946.

Kumar, M. (2019). Serverless architectures review, future trend and the solutions to open problems. *American Journal of Software Engineering*, 6(1):1–10.

Lauwers, C. and Tamburri, D. (2022). OASIS Topology and Orchestration Specification for Cloud Applications.

Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., and Ghalsasi, A. (2011). Cloud computing—the business perspective. *Decision support systems*, 51(1):176–189.

Matei, O., Materka, K., Skyscraper, P., and Erdei, R. (2021). Functionizer-a cloud agnostic platform for serverless computing. In *International Conference on Advanced Information Networking and Applications*, pages 541–550. Springer.

Microsoft (2019). Multicloud solutions with the serverless framework.

Midway.js (2018). Midway serverless.

Opara-Martins, J., Sahandi, R., and Tian, F. (2014). Critical review of vendor lock-in and its impact on adoption of cloud computing. In *International Conference on Information Society (i-Society 2014)*, pages 92–97. IEEE.

Projen (2020). projen/projen: A new generation of project generators.

Sampe, J., Sanchez-Artigas, M., Vernik, G., Yehekzel, I., and Garcia-Lopez, P. (2021). Outsourcing data processing jobs with lithops. *IEEE Transactions on Cloud Computing*.

Serverless, I. (2022). Serverless: Develop & monitor apps on aws lambda.

Tai Nguyen Bui, Álvaro López Espinosa, A. B. (2019). theam/aws-lambda-benchmark: A project that contains aws lambda function implementations for several runtimes e.g. nodejs, haskell, python, go, rust, java, etc.

Tankov, V., Golubev, Y., and Bryksin, T. (2019). Kotless: A serverless framework for kotlin. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1110–1113. IEEE.

Tanzu, V. (2022). Spring cloud function.

Tencent (2022). Cloud object storage — tencent cloud.

Ullah, A., Dagdeviren, H., Ariyattu, R. C., DesLauriers, J., Kiss, T., and Bowden, J. (2021). Micado-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum. *Journal of Grid Computing*, 19:1–28.

Verginadis, Y., Alshabani, I., Mentzas, G., and Stojanovic, N. (2017). Prestocloud: Proactive cloud resources management at the edge for efficient real-time big data processing. In *CLOSER*, pages 583–589.

Wen, J., Chen, Z., and Liu, X. (2022). A literature review on serverless computing. *arXiv preprint arXiv:2206.12275*.

Wurster, M., Breitenbücher, U., Képes, K., Leymann, F., and Yussupov, V. (2018). Modeling and automated deployment of serverless applications using tosca. In *2018 IEEE 11th conference on service-oriented computing and applications (SOCA)*, pages 73–80. IEEE.

Yussupov, V., Soldani, J., Breitenbücher, U., and Leymann, F. (2022). Standards-based modeling and deployment of serverless function orchestrations using bpmn and tosca. *Software: Practice and Experience*.