Multivocal Survey of the Function Management Layer in the Open-Source Serverless Platforms

Petar Galic
Vrije Universiteit Amsterdam
p.galic@student.vu.nl

Erwin van Eyk
Vrije Universiteit Amsterdam
e.vaneyk@atlarge-research.com

Alexandru Iosup
Vrije Universiteit Amsterdam
a.iosup@vu.nl

April 2020

Abstract

Serverless computing is a new cloud paradigm that enables the rapid development of business cases and offers an efficient pay-as-you-use model. Function-as-a-service is an extension of the serverless model that offers the runtime environment for the user functions, along with the management of a function lifecycle (scaling, managing updates, and others) The big cloud providers are offering their FaaS platforms which allows the users to integrate the workflow with other cloud offerings. These platforms are closed source, which doesn't allow researchers to explore their architectures. The SPEC Research Group has presented their reference architecture of the serverless platforms by analyzing the open-source services. In this survey we analyze the architecture of some of the most popular open-source serverless platforms, and compare it to the reference architecture. We also discuss the similarities and differences between these platforms, based on the solutions to most common problems of serverless platforms: cold start optimization, metadata storage, and scaling.

1 Introduction

Serverless computing platforms enable the users to focus solely on their application by not only managing the resources and infrastructure, but also by providing the dynamic allocation of servers and containers. The benefits for the users are clear: they do not have to spend time and resources on server ops, but they delegate this responsibility to the serverless provider, who takes care of uptime and elastic scaling of the components when necessary. The users also do not pay for the unused resources, which provides a cost benefit as compared to IaaS model.

Most of the serverless computing providers offer compute runtimes, better known as Function-as-a-Service (FaaS), which allow writing single functions, which are executed by various triggers, such as HTTP. This model is an extension of the microservice design pattern, and requires a switch from the traditional software design approach. In the FaaS model we have many functions running independently, usually state-less, that need to communicate to accomplish more complex tasks.

The most popular FaaS platforms are offered as part of the large cloud platforms like AWS, Google Cloud and Microsoft Azure, which provide stable ground for researchers [10, 4, 11, 20, 16] and industry professionals [2] to write papers on creating applications, benchmarks and general analysis of these platforms. These platforms are, however, closed-source which limits knowledge sharing and deeper architectural analysis. On the other hand there are many open-source projects. These projects give a clear picture of how the serverless platforms are operating, and have been the subject of many papers [6, 13, 1, 12].

One of the papers, written by the SPEC Research Group [3], based on extensive analysis of the serverless platforms proposed a reference architecture for serverless platforms, which consists of three layers: *resource layer*, which manages underlying resources needed for functions to run, *function layer*, which deals with manages everything related to lifecycle of a single function, and *workflow layer*, which manages function orchestration. This paper did not fo-

cus on the interactions and data-flow between the sub-components of each layer, and it did not describe its interaction to components outside of the ones described in the architecture. We observe that in the current literature there is no in-depth analysis of differences in architecture between different FaaS projects. A finer-grained analysis of the components of each layer is also missing in the literature. In this work, we survey current, open-source platforms for serverless operations to fill in the gaps observed in the literature. To satisfy the purpose of this paper, we propose a following research question:

How to identify the fine-grained architectural components and their interactions in the function management layer of the serverless platforms? Once we establish the analysis method we use it to answer the following two questions:
How to compare, across the well-known serverless platforms, the data and control flow occurring between the components of the function management layer? and
How do the function layer components of the well-known open-source serverless platforms map to the reference architecture for the serverless platforms proposed in the SPEC-RG paper?

To answer the research questions posed we need to conduct a survey into the current state-of-the-art, by providing a deep analysis of at least 5 serverless platforms, describing the data and control flow for each of them. Using insights gained from the survey a generalized fine-grained model of the function management layer architecture will be made.

The contributions to answer our research questions are following:

1. A systematic method for analysis of papers, documentation and code of the well known open-source platforms (S2). We adapt the novel Multivocal Literature Review proposed by Garousi et al. [5] to suit this survey.

2. A novel model to understand FaaS platforms (S3.2). Our model extends the state-of-the-art model proposed by the SPEC-RG Cloud Group in 2018. The key extension is a set of finer-grained architecture elements for the Function Management layer.

3. Analysis of the data-flow and architecture of the chosen platforms (S4). We present core components along with external components for each platform, describe their function and analyze the data flow based on function invocation.

4. Comparison and analysis of the proposed architectures of the each platform to the SPEC-RG reference architecture (S5). We map identified components to the SPEC-RG architecture and discuss mapping differences.

2 Method for Finding, Selecting, and Characterizing Relevant Material

Here we present a method which was used to perform the analysis of the serverless open-source platforms. We explain what are the common methods and processes for finding, selecting, and analysing relevant material. We also provide reasoning why these processes could not be fully applied in the case of this survey, and what are the differences in our method that helped us conduct this survey. Furthermore we explain our choice of open-source serverless platforms.

2.1 Analysis of Widely Used Methods

The common processes in the field when it comes to gathering the literature and performing the survey are snowballing, unguided traversal of the material and the Systematic Literature Review method proposed by Kitchenham et al. [7].

The first two methods are unguided, and the biggest drawback is the lack of reproducibility of the process. The literature gathering process started by two different researchers, even when started with the same set of query words, will end up with two completely different literature sets. However, these processes are simple and straightforward. The Systematic Literature Survey is a method that, if performed correctly, solves the reproducibility issues of the first two method. We have followed this method when it comes to gathering relevant scientific literature.

However, the drawback of the SLR method is that it takes as input only academic peer-reviewed articles. Initial queries have indicated a lot of primary literature on the general topic of serverless, but no satisfying in-depth analysis of different platforms. This means that most of the information we need for this survey is present in grey literature (GL): blog posts, videos, online books, and especially relevant for this survey, documentation and code repositories.

We have identified a novel method, Multivocal Literature Review (MLR) [5]. The idea of this method is that in cases when there is not enough primary literature present, we expand our scope to perform a secondary study and consult sources from the grey literature. Schenuer and Lautner [14] have already performed MLR in the field of serverless bench-

#	Question	Answer	MLR-AutoTest
1	Is the subject "complex" and not solvable by considering only the formal literature?	Yes	Yes
2	Is there a lack of volume or quality of evidence, or a lack of consensus of outcome measurement in the formal literature?	Yes	Yes
3	Is the contextual information important to the subject under study?	Yes	Yes
4	Is it the goal to validate or corroborate scientific outcomes with practical experiences?	No	Yes
5	Is it the goal to challenge assumptions or falsify results from practice using academic research or vice versa?	Yes	Yes
6	Would a synthesis of insights and evidence from the industrial and academic community be useful to one or even both communities?	Yes	Yes
7	Is there a large volume of practitioner sources indicating high practitioner interest in a topic?	Yes	Yes

Table 1: MLR questionnaire with our answers and expected answers for MLR (MLR-AutoTest).

marking, justifying it by the combination of high industry and academia interest in the topic. We have included a questionnaire from the MLR guidelines in 1 which shows that our survey benefits from inclusion of GL. This questionnaire is consisted of 7 questions that check if there is a sufficient reason for consulting grey literature.

2.2 Method Description

We have first performed selected steps from SLR to analyze academic literature. In 1 we have identified the need for a review and specified the research questions. In this section we describe our review protocol 2.2, define search queries and select primary studies 2.3.2. In 4 we perform data extraction and monitoring, and in 5 we perform data synthesis.

Multivocal Literature Review method expands Systematic Literature Review method by providing guidelines on how to search, validate and select grey literature. We identify platforms to be searched and perform a secondary and tertiary study selection 2.3.3.

2.3 Selection and Identification of Literature

2.3.1 Selection Strategy

We define the following inclusion and exclusion criterias to select only literature that provides relevant information:

1. We *include* literature with at least one serverless architecture or data-flow analysis.
2. We *exclude* literature that provide only analysis or implementation of solutions on top of existing platforms (if first point is not satisfied).
3. We *exclude* literature that are concerned with closed-source platforms.

This selection criteria limits significantly the amount of academic literature, as not many academic papers have been written about serverless architecture. It also scopes grey literature as there is a much larger set of articles / blog posts / videos about implementation of solutions on serverless platforms than about inner workings of these platforms.

2.3.2 Primary Literature

For this research, we identified Google Scholar, Serverless Literature Dataset by Spillner and Al-Ameen [17] and AIP database tool developed at VU Amsterdam, as the platforms to be searched for primary literature.

Next, following standard SLR guidelines, we define a set of search strings to be combined in a search query. These strings (Table 2) have been selected on basis of trying to capture knowledge about general serverless architecture and data-flow model, and also knowledge about platforms that have been chosen for analysis in this survey (2.4). We supply these query strings with strings `open-source`, `components` or `architecture` to narrow down the resulting set.

2.3.3 Grey Literature

It is important that the information from the articles comes from trustworthy sources, because grey

literature is not peer-reviewed as the primary sources in conference papers and magazines.

We decided to judge our sources by the following criteria: *Producer authority*, *methodology*, *objectivity*, *date*, *novelty* and *outlet type*. We have identified three types of grey literature that is interesting in the context of this survey: *Articles and blog posts*, *Conference videos* and *Codebase and official documentation*.

In the first category we have identified following platforms: *Google Search*, *Hacker News Algolia*, *Reddit* and *Medium*. In the second category we identified just one platform: *YouTube*. In the case of the third category we identified GitHub as the primary platform as all the projects have their code repositories hosted by GitHub. Along with GitHub we have also identified official project websites which contain official documentation. The query parameters are the same as the ones mentioned in Table 2. The search results are presented in Table 3.

2.4 Platform selection

The scope of this paper is the analysis of platforms that provide function management layer, as defined by the SPEC-RG reference architecture. The workflow layer is out of scope, and resource layer is analysed only when describing its interaction with function management layer.

The list of open-source FaaS platforms that we found thanks to the relevant data-set from the SPEC-RG paper [19] and also analysing new serverless projects that came out after the SPEC-RG paper was published is presented in Table 2. We characterize them by properties which highlight community recognition and project activity. It is important to note that a lot of these projects are backed by large companies (for example IBM, Oracle), so there is likely a divergence between the open-source code and the code that is running in the production. That is why some of these projects seem inactive, while the actual usage and development effort is hard to estimate.

As we see from the Table 3, there is a lot of activity in most of the projects. By far the most recognised project is OpenFaas, which is leading in the number of stars. It is also one of the most used open-source platforms in practice. We pinpoint OpenWhisk, Knative serving, Fission, Kubeless, and nuclio as projects that are active and have a large community recognition. Azure Functions Core Tools and riff have been published as open-source projects relatively recently, and they have not caught attention of the community, but they are active nonetheless. On the other hand, VMware dispatch, OpenLambda and fn are stale projects. VMware dispatch and fn are projects backed by VMware and Oracle respectively, so we assume that the development on these projects is mostly closed-source, with open-source lagging behind. OpenLambda is a project created by academia, and it's activity should not be compared to the commercial projects.

We have decided to analyze **OpenWhisk**, **Knative serving**, **OpenFaas**, **Fission**, and **OpenLambda**. We have chosen these projects because of their relevance and activity, and in the case of OpenLambda due to it's different nature which provides useful information about different perception of serverless in industry and academia. We would have preferred to analyze more projects, but we have decided to scope it due to the time limit of this survey.

2.5 Threats to Validity

Although the MLR method shows a lot of promise, it is still a novel and untested method. As we mentioned before, using grey literature will produce errors because the sources are not peer reviewed as the academic papers. We tried judging the grey literature sources based on aforementioned attributes, but we cannot judge every source as reviewers do with every paper posted to the journals.

Another possible shortcoming of this survey is based on the selection of the query strings. Notably, the term *lambda* is used to describe serverless functions, and it is used by the wider community. However, both academic papers and grey literature is mostly revolved around AWS Lambda (which is excluded based on our selection criteria), so finding the relevant literature was a very extensive task that we could not perform due to the scope of this survey.

3 General Architecture Model

In this section we present an existing reference architecture proposed by the SPEC Research Group, we also provide a conceptual contribution by further expanding the serverless reference architecture proposed by the SPEC research group. In this section we answer the research question RQ1: *How to identify the fine-grained architectural components and their interactions in the function management layer of the serverless platforms?*

3.1 Reference Architecture Mapping

The SPEC-RG reference architecture contains following components:

Query	# Selected	# Analysed
serverless	8	8
FaaS (function-as-a-service)	3	1
OpenLambda	2	2
OpenWhisk	5	4
OpenFaas	3	2
Knative	4	3
Fission	3	1
Total:	28	21

Table 2: Primary literature identification and study, third column indicates deduplicated literature.

Query	CG1	CG2	CG3
serverless	5	1	-
FaaS (function-as-a-service)	1	0	-
OpenLambda	2	0	1
OpenWhisk	7	1	2
OpenFaas	5	2	4
Knative	5	3	2
Fission	5	2	2
Total:	30	9	11

Table 3: Number of Analysed grey literature resources: Articles and blogposts (CG1), Conference videos (CG2), and Codebase and official documentation (CG3).

Project	Latest commit	Stars	# Commits	Section
OpenFaas	Mar 19th, 2020	17.2k	1,858	4.3
Kubeless	Mar 17th, 2020	5.5k	1,005	-
Fission	Mar 20th, 2020	5k	1,134	4.5
OpenWhisk	Mar 20th, 2020	4.6k	2,760	4.1
fn	Dec 19th, 2019	4.5k	3,393	-
nuclio	Mar 19th, 2020	3.2k	1,277	-
Knative serving	Mar 21st, 2020	2.8k	3,961	4.4
riff	Mar 2nd, 2020	776	1,268	-
OpenLambda	Dec 6th, 2019	669	811	4.2
Azure Functions Core Tools	Mar 14th, 2020	531	1,142	-
VMware Dispatch	Jan 8th, 2019	518	538	-

Table 4: Analysis of GitHub repositories (master branch) of open-source serverless projects on 22nd of March, 2020.

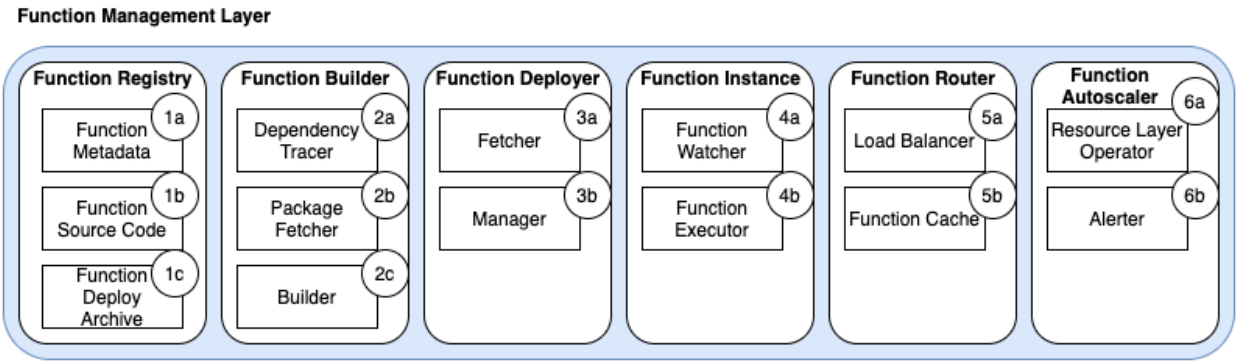

Figure 1: Reference architecture of subcomponents in the Function Management Layer.

1. **Function Registry** (F1): Store which contains source code, deploy archive and function metadata. Used for building, deploying and searching for functions.

2. **Function Builder** (F2): Responsible for building source code into deployable archive that can be run inside containers. Part of this component are usually also dependency tracer and package fetcher.

3. **Function Deployer** (F3): Ensures that the instance of a function is deployed. Combines function metadata and deploy archive and interacts with the underlying resource management layer to create one or more function instances.

4. **Function Instance** (F4): Representation of a function, able to execute based on various triggers. Function code is attached to a container and that represents a Function Instance, but typically there are also other containers or processes injected together with the function container.

5. **Function Router** (F5): Has overview of all Function Instances, and acts as the entry point in the function management layer. It has a load balancing role when there is multiple Function Instances. Also queues requests when there is no Function Instances available.

6. **Function Autoscaler** (F6): Responsible for autoscaling of Function Instances, based upon number of requests. Usually combined with other components that reports usage statistics based on which autoscaler increases or decreases number of Function Instances.

3.2 Going Beyond Reference Architecture

Here we present our extension to the reference architecture, which goes one level deeper into subcomponents of the reference architecture. It is based on the analysis of the platforms presented in S4.

It is important to note that the mapping of these components to their parent components varies substantially between different platforms. However, dropping the boundaries between the reference architecture components and presenting the architecture as a combination of these subcomponents carries a risk of overfitting and mispredicting what is currently used in practice.

1. **Function Registry**: We describe different data that is handled by the Function Store.

   (a) **Function Metadata**: Contains information about size, programming language, amount of resources requested, minimum scaling and other information useful for build, scaling and deployment.

   (b) **Function Source Code**: User uploaded code, used by Function Builder to build executable.

   (c) **Function Deploy Archive**: Stored in the Function Store by the Function Builder after building the function. Used by Function Deployer to attach the executable to the container.

2. **Function Builder**:

   (a) **Dependency Tracer**: Traces changes in dependencies, usually as a part of a function invocation or as a scheduled job.

(b) **Package Fetcher**: Responsible for fetching new packages as part of the build process.

(c) **Builder**: Responsible for fetching and compiling source archive.

3. **Function Deployer**:

   (a) **Fetcher**: Responsible for fetching deploy archive from the store.

   (b) **Manager**: Responsible for keeping track of free containers. Usually the place where cold start optimizations are implemented.

4. **Function Instance**:

   (a) **Function Watcher**: Entrypoint of the request in the function instance. Unmarshals requests and forwards them to the Executor. Also can report requests to Autoscaler.

   (b) **Function Executor**: Container or a process inside container that is responsible for executing function.

5. **Function Router**:

   (a) **Load Balancer**: Component that decides on which function instance to send the request to.

   (b) **Function Cache**: Used for looking up function addresses to check whether there is a function instance running.

6. **Function Autoscaler**:

   (a) **Resource Layer Operator**: Interacts with the underlying resource layer to increase or decrease number of instances. Does this by automatically changing configuration of the resource layer, therefore automating this process.

   (b) **Alerter**: Component that receives alerts, or reports itself on the load, and notifies Resource Layer Operator to increase or decrease number of instances.

4 Platform Analysis

We structure the analysis in the following way: for each platform, we first introduce the platform. Then we present the architectural overview of the system, and describe the functionalities of the components and how they interact. Then, we describe function invocation flow as a way of describing control flow between components identified in the architectural overview. In the next step we explain the data flow among the different components. By doing these two steps we answer the research question RQ2: *How to compare, across the well-known serverless platforms, the data and control flow occurring between the components of the function management layer?* Lastly, we map system components to the SPEC-RG reference architecture and discuss this mapping. With this mapping and the discussion we answer the final research question RQ3: *How do the function layer components of the well-known open-source serverless platforms map to the reference architecture for the serverless platforms proposed in the SPEC-RG paper?*

4.1 Apache OpenWhisk

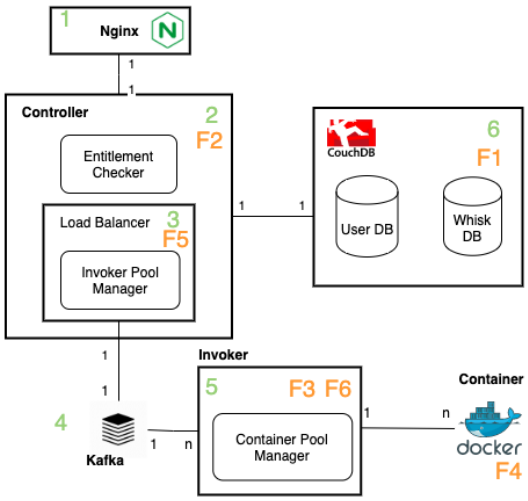

Figure 2: OpenWhisk architecture diagram. Existing open source technologies are marked with an icon. We map components to the SPEC RG reference architecture, and label the main component-mappings with F1-6.

Apache OpenWhisk[1] is an open-source serverless platform under the Apache foundation. It is implemented as the FaaS platform in IBM Cloud. Considering its strong support from industry, the project is well documented and covered in many articles, blog posts and books [15]. OpenWhisk supports Go, NodeJS, Java and Swift runtimes, and also supports execution of Docker containers which adds support for virtually any language. It is the only analysed platform that is built using Scala.

[1] github.com/apache/openwhisk

4.1.1 System Overview

OpenWhisk consists of the following core components:

1. **Nginx**: HTTP and reverse proxy server, used for SSL termination and forwarding http requests
2. **Controller**: core component that receives an HTTP request from user, and also other non user-initiated triggers. Interacts with the database for authentication and checking **entitlements** (rights), and also for CRUD operations on functions.
3. **Load Balancer**: component that keeps track of all the registered executors (Invokers) that can execute the action. Forwards function activations from Controller to Invokers. It can be centralized or distributed.
4. **Kafka**: distributed messaging queue used for communication between Load Balancers and Invokers. It provides fault-tolerance and buffering of activations.
5. **Invoker**: component responsible for managing Docker containers and deploying functions into the containers.
6. **CouchDB**: database used for storing function code and entitlements.

Figure 2 shows how these components are organized in the architecture. OpenWhisk has been built using existing open source technologies that are highlighted in this diagram. Core components are drawn with square boxes and icons, and internal components are marked with round boxes. In orange we can see the mapping to the SPEC-RG reference architecture components, which is analysed in 4.1.4.

4.1.2 Function Invocation Flow

Figure 3 depicts the invocation flow of OpenWhisk. The activation request reaches Nginx as a first contact with the system and it is forwarded to the Controller. Once Controller receives an activation, it checks user and action (function in OpenWhisk) entitlements in CouchDB to ensure the authenticated user has the rights to invoke action. If the user has the right entitlement for the action, the action metadata is sent to the Load Balancer.

The Load Balancer chooses an Invoker from its pool based on computing a hash for a user namespace and action, ensuring that for the same action and namespace Load Balancer will always try scheduling the action on the same Invoker. If the Invoker is full and cannot accept any more requests, the algorithm is used again to find an empty Invoker which can execute actions.

All communication between Invokers and Load Balancers goes through Kafka, which allows services to subscribe to new messages. Invokers can create and destroy containers based on the configuration of the maximum number of containers. Invokers also keep a pool of prewarmed containers that are used to optimize cold start times. Once an Invoker receives action metadata from Kafka, it retrieves the actione executable from the Whisk Store, which is also a part of the CouchDB. If the action executable is missing, it means that the user concurrently deleted the action, which returns an error.

If the executable is present, the Invoker tries to find a warm container (container that is running and its in use for the specific action and namespace), and if it finds it it executes the action and returns the result. If it does not find the warm container, it checks the prewarmed container pool (preconfigured running containers) to find a suitable running container. If there is no suitable container, only then the cold start is used. Since the number of containers in the pool is limited, in case a new container needs to be initialized, possibly an unused container will be removed.

Once the result is obtained, it is stored in results database in CouchDB. Load Balancer and Invokers are communicating using acknowledgments. When a Load Balancer receives a result ack, if the activation was blocking, it retrieves the result from the results database and sends it to the user. If the activation was non-blocking, the Load Balancer already returned activation id once it got confirmation from the Invoker that the action execution has started. Users can query the results and activation status by using the activation id. Once Load Balancer receives completion ack, it releases the Invoker activation slot.

4.1.3 Data Flow

In this section we describe the data flow between the high level architecture components identified in the Figure 2.

The system entrypoint, *Nginx* receives and forwards HTTP requests to the *Controller*. The HTTP requests include credentials, which the Controller sends as queries to the database to check whether the user has rights to perform the action specified in the HTTP request. The database sends the query responses to the Controller which are used to allow or

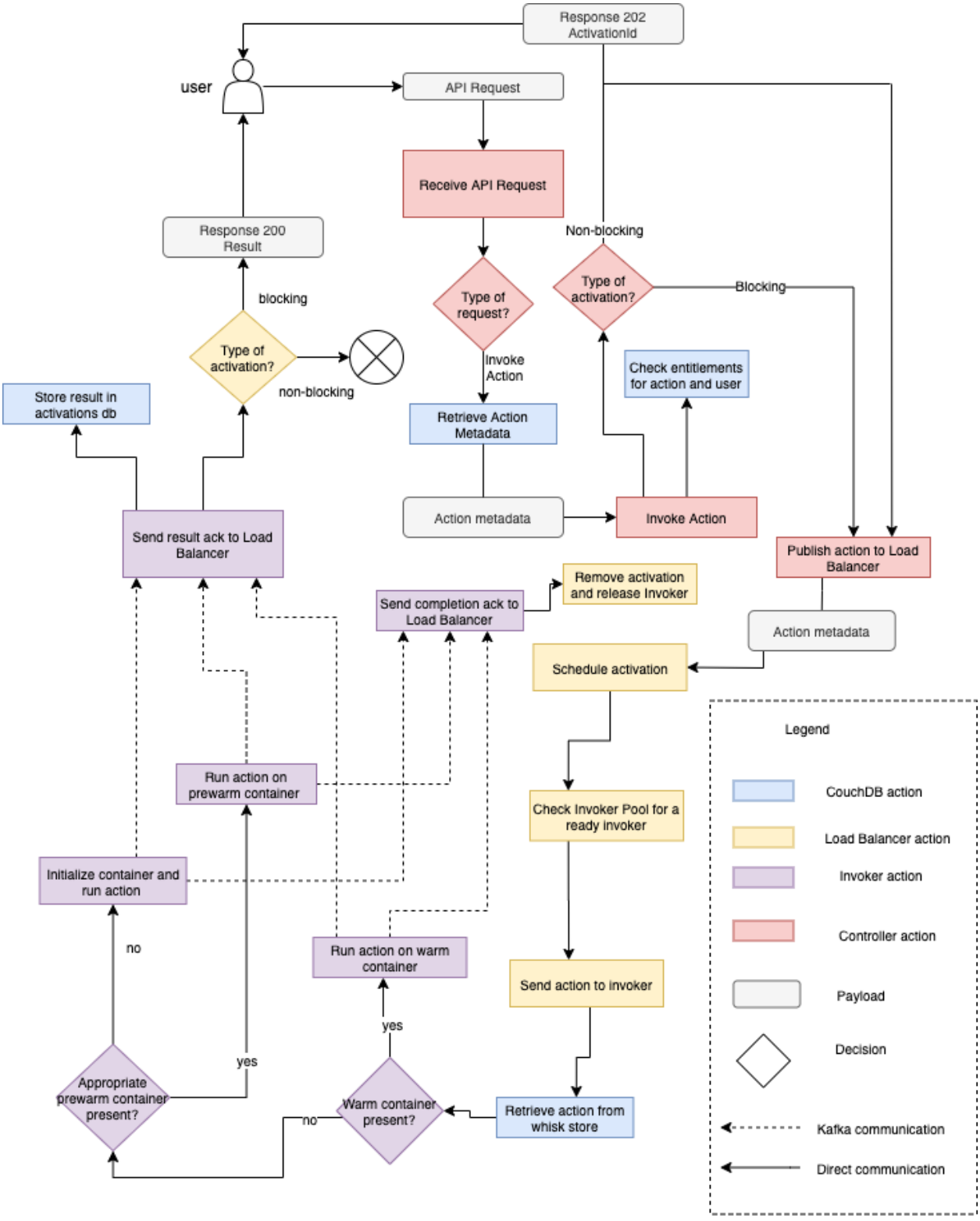

Figure 3: OpenWhisk function invocation flow.

deny the user action. The Controller again interacts with the database, this time the Whisk database.

In case of the new function creation, the Controller sends function code, default parameters, and

resource restrictions to the Whisk database where new entry is created. In case of the function invocation, the Controller sends a query to the Whisk database to fetch the function. Then the Whisk database sends the function code, default parameters, and resource restrictions back to the Controller. The Controller sends the function metadata needed for running the function to the Load Balancer.

This data is the data retrieved from the Whisk database merged with the function parameters supplied in the HTTP request. The function metadata, together with the Invoker address selected by the Load Balancer is then forwarded through Kafka to the right Invoker. Invoker either sends the function parameters to the running function container, or injects the function executable into a new container. In case a new container is needed, the Invoker sends the request with container configuration to the Container pool to retrieve the container address. The system returns the user the HTTP response with the activation id of their invocation.

4.1.4 Reference Architecture Mapping

In OpenWhisk, Controller has the central role, therefore it encompasses multiple roles by the components defined in the reference architecture. Since Controller receives both requests for creating and invoking actions, its role is both to build the function executable if necessary, and to route incoming requests. It is important to note that Load Balancer is the part of controller. Load Balancer's role is to route the requests to the correct Invoker. Invoker, as a container pool manager, has a role of elastically scaling containers, which are Function Instances, and to deploy and monitor them. The following mapping is produced:

- **F1**: CouchDB
- **F2**: Controller
- **F3**: Invoker
- **F4**: Container
- **F5**: Load Balancer
- **F6**: Invoker

4.2 OpenLambda

OpenLambda[2] was created as a research project from University of Wisconsin. They defined the FaaS computing model as Lambda model, as inspiration from Amazon Lambda. It was built as a platform for researchers to evaluate new approaches to serverless computing [6], create benchmarks and develop applications. The project is written in Go and at the moment it's stale, although we assume the project is still widely used and developed under different research groups.

4.2.1 System Overview

The architecture of OpenLambda is shown in Figure 4.2.1. We highlighted external open-source technologies that are part of the platform (CouchDB, Nginx, Docker). Also, we marked the Worker component with dashed line to highlight that it is not a functional component, but a distribution unit that encompasses core components of OpenLambda. The round boxes are internal components while square boxes are the core components.

The core components are following:

1. **Nginx**: HTTP and reverse proxy server, used in OpenLambda as a scheduler. It provides limited scheduling techniques. A more sophisticated scheduling technique that takes into account locality (which is useful for optimizing cold start) is presented in [18].
2. **Worker**: Worker is the unit of distribution in OpenLambda. It contains core components necessary for function management.
   (a) **Lambda Server**: Entry point of the request. It disambiguates the requests and forwards them via Lambda Manager to the Lambda Function
   (b) **Lambda Manager**: Lambda Manager is the component that manages function life cycle. It keeps a map of existing functions, function instances and also registers tools such as *Dependency Tracer*, *Handler Puller* and *Package Puller* to track and install required packages and dependencies for functions. It contains a sandbox pool used for creating instances.
   (c) **Lambda Function**: Component representing the function registered in the system. It keeps track of Function Instances, updates to the function, code directory and metadata needed to create a sandbox. Lambda Function also employs auto scaling techniques, based on number of requests and time needed for function execution.

[2] github.com/open-lambda/open-lambda

(d) **Lambda Instance**: Instance of the function, it is a snapshot of the function at the time of the instance creation. It is a virtual instance, backed by a sandbox, if there is one available.

(e) **Sandbox**: Abstraction of the resource management layer, it is a container instance that is used by the Lambda Instance.

3. **Function Registry**: Database option used for retrieving function metadata and code. Due to change feed abstraction, CouchDB or RethinkDB are preferred options.

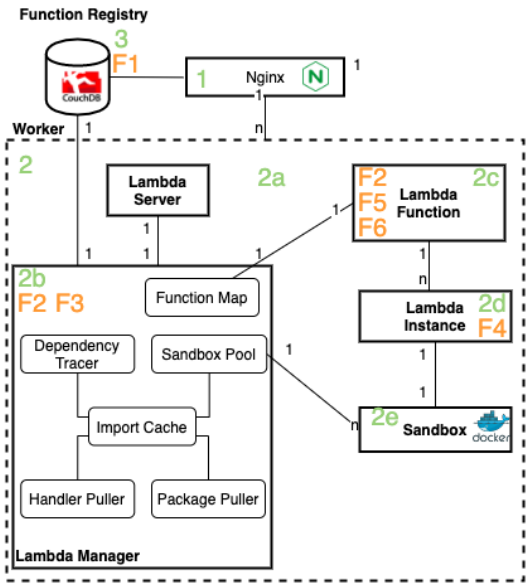

Figure 4: OpenLambda architecture diagram. Existing open source technologies are marked with an icon. We map components to the SPEC RG reference architecture, and label the main component-mappings with F1-6.

4.2.2 Function Invocation Flow

Invocation flow is highlighted in Figure 4.2.2. The function enters the system via Nginx proxy, and it gets forwarded to one of the workers. Request is received by Lambda Server, which retrieves the function based on the url. Lambda Manager either returns a function that is already registered, or creates a new Lambda Function. The task is forwarded to one of the Lambda Instances via a queue. Once the task is received by the Lambda instance, it checks if there is a sandbox mapped to it already. If there is one, the Lambda Instance checks its status and based on the response it either executes function in the existing sandbox, or creates a new sandbox and executes the function (cold start).

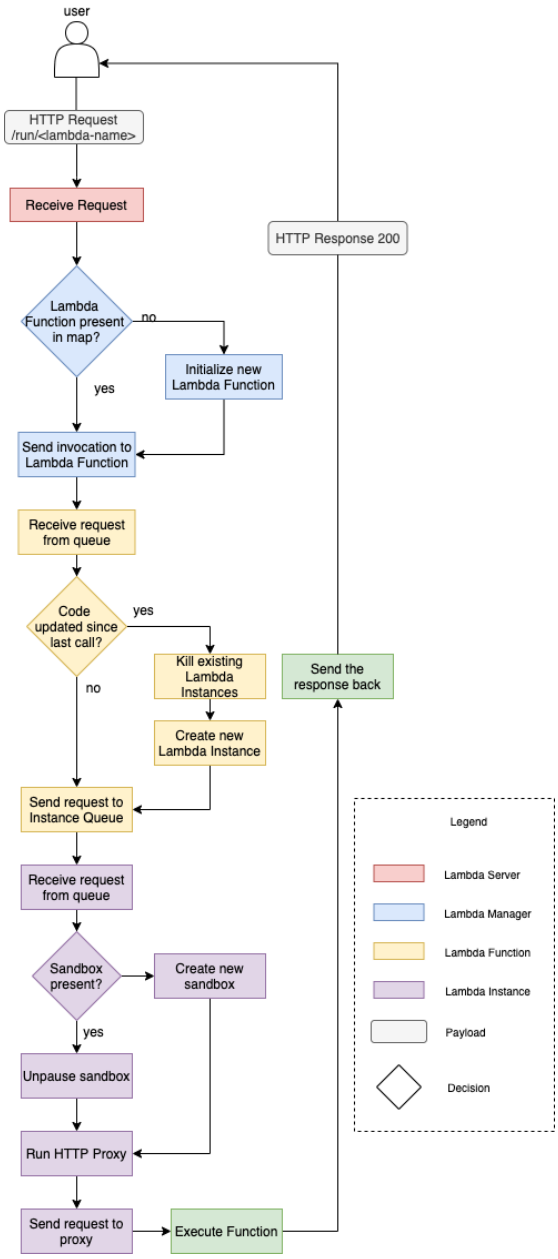

Figure 5: OpenLambda invocation flow.

4.2.3 Data Flow

The HTTP request is received and forwarded by the Nginx proxy to the Lambda Server. The Lambda Server sends the name of the function to the Lambda Manager which returns the Lambda function metadata back to the Lambda Server. The Server then passes the HTTP request to the channel that the

Lambda Function task is listening to. The Lambda task forwards the HTTP request to the Lambda Instances that wait for the requests. The Lambda instance forwards the HTTP request to the sandbox proxy. The sandbox sends an HTTP response directly to the client that sent the request.

4.2.4 Reference Architecture Mapping

In OpenLambda, the Lambda Function is the heart of the system. It has the responsibility of scaling Lambda Instances, and also keeps track of the function updates to deploy or update Lambda instances. However, most of the functionalities for updating and tracing dependencies are logically mapped to the Lambda Manager component, so their difference is not very clear if judged only by the reference architecture. Even though in the SPEC-RG reference architecture mapping the Function Router (F4) component is mapped as missing, we do think that the Lambda Function resembles this behavior, as it has the channel in which it places the request for the available Lambda Instance.

- **F1**: Function Registry
- **F2, F3**: Lambda Manager
- **F4**: Lambda Instance
- **F5,F6**: Lambda Function

4.3 OpenFaas

OpenFaas[3] is the most community recognized FaaS platform, written in Go as most of the platforms in this survey. It is a platform that delegates resource management layer responsibilities. It supports any resource management tools such as Kubernetes or Docker-swarm, and it also supports forwarding functions through Amazon Lambda or Fargate, as long as the operator (point of interaction betweeen OpenFaas API gateway and the rest of the system) implements the interface provided by OpenFaas. The recommended container orchestration engine is Kubernetes, which is supported with an OpenFaas provider for Kubernetes called `faas-netes`.

The Kubernetes provider works in two modes: either by directly using Kubernetes resources to manage functions, or by defining custom resources using Kubernetes Custom Resource Definition **Function**. Function is composed of basic Kubernetes resources - Secret, Deployment and Service. The function metadata is stored in the manifest file for a Function object, and upon creation and updates, OpenFaas Kubernetes Operator is creating or updating the related basic Kubernetes resources.

OpenFaas is a part of the PLONK stack, which consists of Prometheus (error reporting), Linkerd (service mesh), OpenFaas, NATS (asynchronous message bus) and Kubernetes. OpenFaas offers a commercial solution OpenFaas Cloud, which is a multitenant instance of OpenFaas with integrated CI/CD pipeline and external authentication system.

4.3.1 System Overview

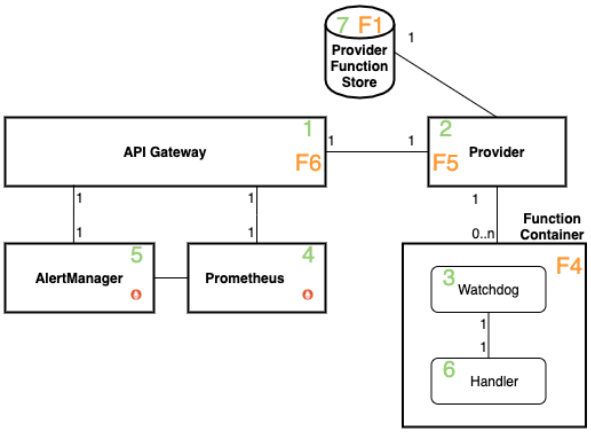

Figure 6: OpenFaas architecture diagram. We map components to the SPEC RG reference architecture, and label the main component-mappings with F1-6.

OpenFaas system consists of the following components (highlighted in Figure 6).

1. **API Gateway**: Provides access to the system from external sources, used for creating and invoking functions. Also used for scaling functions according to demand. Contains UI for performing these functionalities.
2. **Provider**: Interface for the OpenFaas backend which enables functionalities that are performed by API Gateway. OpenFaas supports various backends, container orchestrators like Kubernetes and Docker-swarm, or other FaaS and CaaS platforms like AWS Lambda and AWS Fargate. The recommended provider is `faas-netes`, which enables Kubernetes as a backend for OpenFaas.
3. **Watchdog** (Function Container): Small Golang web server that receives function requests, and forwards them to the function process. It is the init process for the container, and there is two versions. The deprecated one,

[3] github.com/openfaas/faas

`watchdog`, provides only one mode which forks one process per request. The current watchdog, `of-watchdog` provides five modes, with http mode being the default one. It forks the process when the watchdog is created, and it forwards the requests to the http port of the function container. The cold start is optimized by keeping the container running (warm) throughout the watchdog lifetime. The other supported modes are *serializing fork* (for backwards compatibility with original watchdog, *streaming fork*, which forks a process per request and deals with a request body larger than the memory capacity, *afterburn*, which uses a single process for all requests, and thefinal static mode, which is used for serving static content. Modes 2-4 are considered experimental.

4. **Prometheus**: Prometheus is an open-source alerting and monitoring toolkit. In OpenFaas it is used in combination with AlertManager to collect metrics on the function load which are used by the OpenFaas gateway to elastically scale function containers.

5. **AlertManager**: Reads usage statistics (requests per second) from Prometheus and based on configured thresholds fires alerts to OpenFaas Gateway.

6. **Handler**(Function Container): User-defined process running inside Function Container

7. **Provider Function Store**: Function metadata store, it is related to the type of backend used. In `faas-netes` function metadata is stored as a manifest file for a Function CRD.

4.3.2 Function Invocation Flow

A large portion of the flow is handled by the orchestration backend that depends on the choice of the provider, we describe a high-level function management side on Kubernetes with `faas-netes` as the backend provider. The function invocation is received through API gateway, which is the entrypoint of the system. The function is forwarded via the message queue (queue worker enables the asynchronous processing) to the orchestration provider. The functions are deployed in Kubernetes as containers which consist of the Watchdog and the Handler. Handler is the process which represents user created function. Watchdog receives the http request, and forwards it to the Handler. Once the Handler is done, the Watchdog forwards the response to the API Gateway which sends the response back to the user. The asynchronous function invocation is supported by callback urls, and NATS queue.

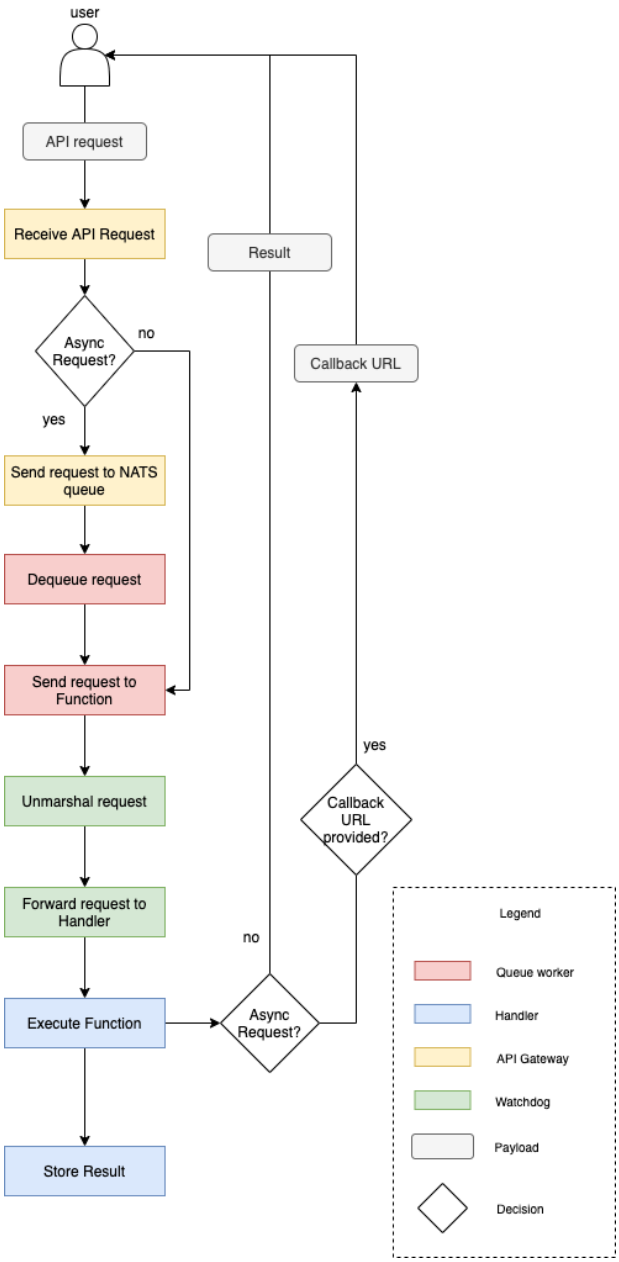

Figure 7: OpenFaas invocation flow.

4.3.3 Data flow

The API Gateway receives the request from the client and interacts with the provider by sending the HTTP requests to the REST API. The request is then forwarded to the underlying resource orchestrator which the provider is based on. The Function Container receives the HTTP requests which are for-

warded from the Watchdog to the Handler. The Handler sends the response back to the Watchdog which sends the HTTP response back to the user.

4.3.4 Reference architecture mapping

The OpenFaas reference architecture mapping done by the SPEC-RG correctly identified that the resource management layer is delegated. OpenFaas delegates the resource management to the aforementioned tools and platforms and supports providers which interact with the function management layer.

This allows OpenFaas to use some functionalities of these tools even in the function management layer. For example, autoscaling functionality is built upon the underlying scaling functionality of Kubernetes.

OpenFaas Gateway role is to scale the function containers based on the alerts received by AlertManager. OpenFaas is delegating the responsibility of building and deploying the functions to the user, either through using the command-line interface or CI/CD pipeline. Therefore, the function is built into an immutable Docker image and the deployment is handled by providing a manifest file for the Function CRD (in case Kubernetes is used as backend provider). The deployment is then handled by resource orchestration layer and OpenFaas does not explicitly define a component on function layer for this task.

OpenFaas supports function creation and deployment by providing function templates for variety of programming languages. Function routing is another functionality that is delegated to the underlying load balancing mechanisms by the backend provider.

- **F1**:Provider Function Store
- **F2**: `faas-netes` (delegated)
- **F3**: missing
- **F4**: Handler
- **F5**: Provider (delegated)
- **F6**: API Gateway

4.4 Knative Serving

Knative Serving[4] is a tool for rapid deployment and automated scaling of containers in Kubernetes. It has been developed by Google. It is part of a Knative offering, which also contains Knative Build, which is a tool that enables continuous integration, building and running containers in Kubernetes cluster, and Knative Eventing, which is a event triggering framework within a cloud-native environment. Knative Serving is not a platform, as the other projects Analysed here, it is rather a collection of Kubernetes extensions that simplify management of serverless workloads. That means that it does not fully abstract the resource layer, and the developers still have to make changes to Knative custom objects to roll out a deployment. Knative Serving defines a set of objects that are defined as Kubernetes Custom Resource Definitions. It combines this with custom controllers into the Kubernetes Operator pattern, which allows for the extended behavior on top of Kubernetes. Knative Serving uses Istio Virtual Service to split traffic between different deployments.

4.4.1 System Overview

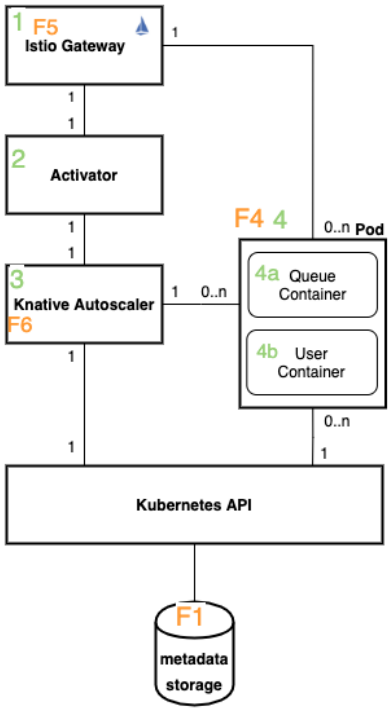

Figure 8: Knative Serving architecture diagram. We map components to the SPEC RG reference architecture, and label the main component-mappings with F1-6.

Knative Serving architecture consists of the following components (highlighted in Figure 8):

1. **Gateway**: Entrypoint of a request to the system. It maintains a view of all running services

[4] github.com/knative/serving

and routes requests to the already running services. It is an ingress gateway, and Knative is using the Istio Gateway by default. It is a load balancer operating at the edge of service mesh that receives incoming HTTP/TCP connections.

2. **Activator**: Buffer that is used in case of cold start.

3. **Autoscaler**: Component that interacts with underlying Kubernetes layer. It receives reports from function containers and instructs Kubernetes to create new pods (sets of containers).

4. **Pod**: A function instance, consisting of:

   (a) **Queue Container**: Container created automatically by Knative configuration that acts as a reverse proxy. It interacts with the Autoscaler component to elastically scale pods.

   (b) **User Container**: Container defined by the user-specified image.

4.4.2 Kubernetes Resource Description

As we mentioned in the Knative Serving introduction, Knative Serving defines custom Kubernetes resources to provide serverless lifecycle management on top of Kubernetes. The following resources are defined:

1. **Service**: Main Knative resource. Service object is initialized with workload specification and traffic specification. Former will create a Configuration object from which a Revision object is created and the latter will create a Route object. Updating the Service object will subsequently update the Route object and Configuration object, and create a new Revision object.

2. **Route**: Route is a Knative representation of Istio Virtual Service. It controls the traffic split between different Revision objects.

3. **Configuration**: Configuration object creates new Revision object based on the updated workload specification.

4. **Revision**: Revision object consists of Kubernetes Service, Kubernetes Deployment and Knative Pod Autoscaler (which is an extension of Kubernetes Horizontal Pod autoscaler). Revisions are snapshots of serverless functions at the time of the configuration update.

4.4.3 Function Invocation Flow

Knative Serving invocation flow is highlighted in Figure 9. Once the function request is received by the Gateway component, if the Gateway does not have a direct route to the Revision object, it means the Revision object does not contain any active pods, and the pod needs to cold start. The request is then sent to the Activator component, which buffers the request and forwards it to the Autoscaler component. Autoscaler component requests through Kubernetes API the creation of a new pod. Once the new pod for this revision is created, Autoscaler sends a signal back to Activator, which notifies the Gateway component which creates a direct route to the pod. The Activator sends the buffered request to the newly created pod. Once the request is received by the Queue container, it sends a signal to the Autoscaler component to update its inflight request count. The Autoscaler component increases and decreases number of replicated pods based on configured thresholds of inflight requests count. Once the task is finished by the User Container, it is returned back to the requestor, and the Queue Container sends a signal to the Autoscaler to decrease its inflight request count.

4.4.4 Data flow

The data exchanged between different components in Knative Serving is mostly user initiated HTTP requests at the high level. Gateway sends HTTP requests to Activator component, and directly to the function pods. Activator sends requests for creating pods to the Autoscaler. Autoscaler sends addresses of new pods to Activator, which Activator forwards to Gateway for direct connection later. Autoscaler also interacts with underlying resource layer by sending requests for creating and destroying pods. Activator and Gateway send the HTTP requests to the running pod, where they are received by the Queue container. Queue container sends requests for increasing and decreasing request count to the Autoscaler. The request is forwarded from Queue container to User container, and the response is returned from the User container to the Queue container. Queue container returns the response to the Gateway.

4.4.5 Reference architecture mapping

As we already mentioned, Knative Serving is not a true serverless platform, meaning that it does not completely alleviate the operational burden from the developers. Developers still need to have the knowledge of the underlying Kubernetes mechanisms and

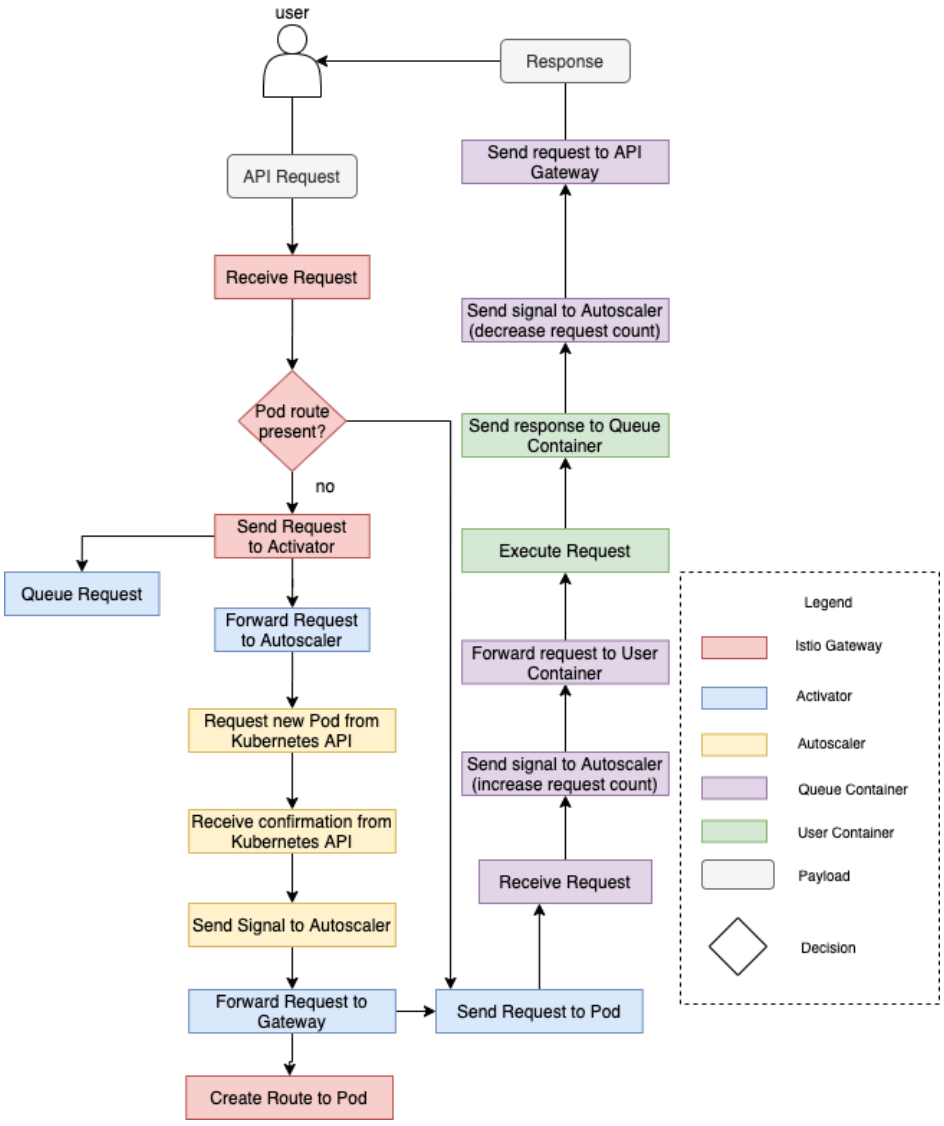

Figure 9: Knative Serving invocation flow.

need to change the Knative Serving CRD configuration to create a new update.

In that sense, a lot of components identified in the SPEC-RG reference architecture paper are missing or delegated, as it is correctly identified in the mapping. Knative Serving doesn't have a dedicated Function Registry, but since it is built on Kubernetes and relies on containers, integration of any kind of container registry such as Docker Hub or Google Container Registry is possible. YAML files for the Service CRD, which are function metadata, are stored inside the Kubernetes cluster as the other YAML files. Even though this component is marked as present in the SPEC-RG mapping, we would rather identify it as delegated.

Building of functions is a function of aforementioned Knative Builder, and it is marked as delegated in the SPEC-RG mapping. We agree with this as it follows the same logic as in OpenFaas Builder and Deployer delegation. Deploying functions is, as with OpenFaas, delegated to the resource management layer. That means that Knative Serving does not have a separate function deployer component in the function management layer. As we already discussed, in Knative Serving, a instance of a function is the Kubernetes pod that contains User Container and Queue Container. When the function is scaled, the Kubernetes pod is replicated or destroyed.

The Function Router in Knative Serving is the Istio Gateway, which has the overview of all pods

related to a Service, therefore this functionality is marked as delegated. Knative Pods Autoscaler represents the Function Autoscaler component present in the SPEC-RG architecture mapping.

- **F1**: Kubernetes CRD and Container Registry (delegated)
- **F2**: Knative Builder (delegated)
- **F3**: Missing
- **F4**: Pod (User Container and Queue Container)
- **F5**: Istio Gateway (delegated)
- **F6**: Knative Autoscaler

4.5 Fission

Fission[5] is a Kubernetes based serverless platform that is written in Go. It is created by the Platform9 company It relies on Kubernetes CRD to fully manage serverless function behavior in Kubernetes. It also extends the functionalities by providing automated deployment and function building, which makes it a complete platform. It supports environments for NodeJS, Python, Ruby, Go, PHP, Bash, and any Linux executable.

4.5.1 System Overview

Fission consists of following core components which are highlighted in Figure 10:

1. **Controller**: Entry point to the system, it accepts CRUD requests and triggers for functions. It also forwards the requests in case there are other internal services in the Kubernetes cluster than Fission.
2. **Executor**: Receives requests for creating new function instances from the Router. Retrieves function metadata (Kubernetes CRD) and invokes one of executor types to deploy the function. Contains following executor types:
   (a) **PoolManager**: Used for short-living functions that require low cold start times. Maintains pools of warm containers, which can then be attached with function code upon Executor invocation.
   (b) **NewDeploy**: Used for functions that need to handle large amount of traffic. Creates Kubernetes Horizontal Pod Autoscaler, Service and Deployment. It can maintain minimum number of function instances, or scale to and from 0 if the minimum number is not defined.
3. **Router**: Has overview of function instances and routes requests to them. If the function instance is not present, it requests Executor to create a new instance. It is stateless.
4. **Function Pod**: Function instance that is created by Executor. It contains two containers and a shared volume.
   (a) **Fetcher**: Receives CRD for a function and the URL to the deployment archive which it downloads, verifies integrity and stores in the shared volume.
   (b) **Environment Container**: Language specific container that runs user-defined functions. Contains an HTTP Server that serves HTTP requests from the clients. Also contains a loader which loads functions from shared volume.
5. **Builder Manager**: Responsible for managing function builders, Builder Pods. Updates Builder Service (that subsequently updates Builder Pods) based on updates to environment and packages, represented by Environment and Package Kubernetes CRD.
6. **Builder Pod**: Pod used for building function code. Has the same structure as Function Pod: two containers and a shared volume.
   (a) **Fetcher**: Receives requests from Builder Manager to pull and verify the source archive from StorageSvc. Saves it to the shared volume.
   (b) **Builder Container**: Language specific container that loads the source code from the shared volume and compiles function source code into executable.
7. **StorageSvc**: Storage for all function source and deploy archives.

[5] github.com/fission/fission

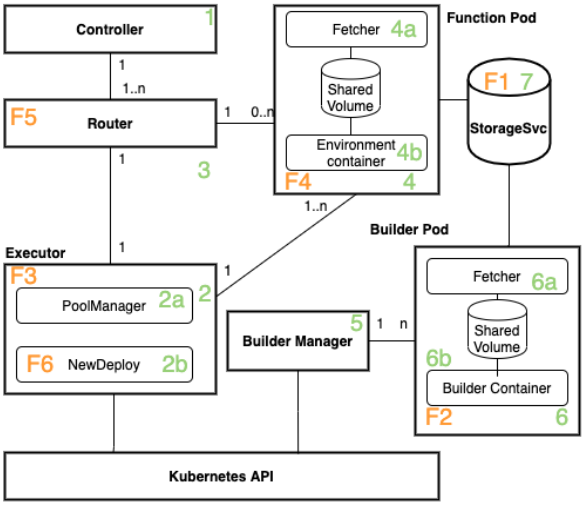

Figure 10: Fission architecture diagram. We map components to the SPEC RG reference architecture, and label the main component-mappings with F1-6.

4.5.2 Function Invocation Flow

Function invocation is highlighted in Figure 11. The entrypoint of the system is the Controller, which forwards the request to the Fission Router. The Router checks if there exists a trigger for a function in API request. If not, it returns a 404. Otherwise, the Router checks its cache for address of the Function Pod.

If there is no address present in the cache, it means there is no Function Pod running so it needs to be cold started. The Router sends the request for a new address for a specific function to the Executor. The Executor fetches the function metadata from Kubernetes CRD and invokes one of the Executor types to create a new Function Pod.

PoolManager has a pool of warm generic pods which can quickly be specialized to reduce the cold start times. PoolManager invokes the fetcher container inside the generic pod, which then pulls the deploy archive from StorageSvc, stores it in the shared volume for Environment Container to load it. PoolManager creates only one pod per function.

On the other hand, NewDeploy creates Kubernetes Deployment, Service and Horizontal Pod Autoscaler, which enables scaling of the Function Pods.

Once the Pod is created, Executor sends a new address to the Router which then sends the Request to this address. Environment Container inside Function Pod receives the HTTP request via HTTP server, executes the function and returns the response to the Router, which then propagates it to user via Controller.

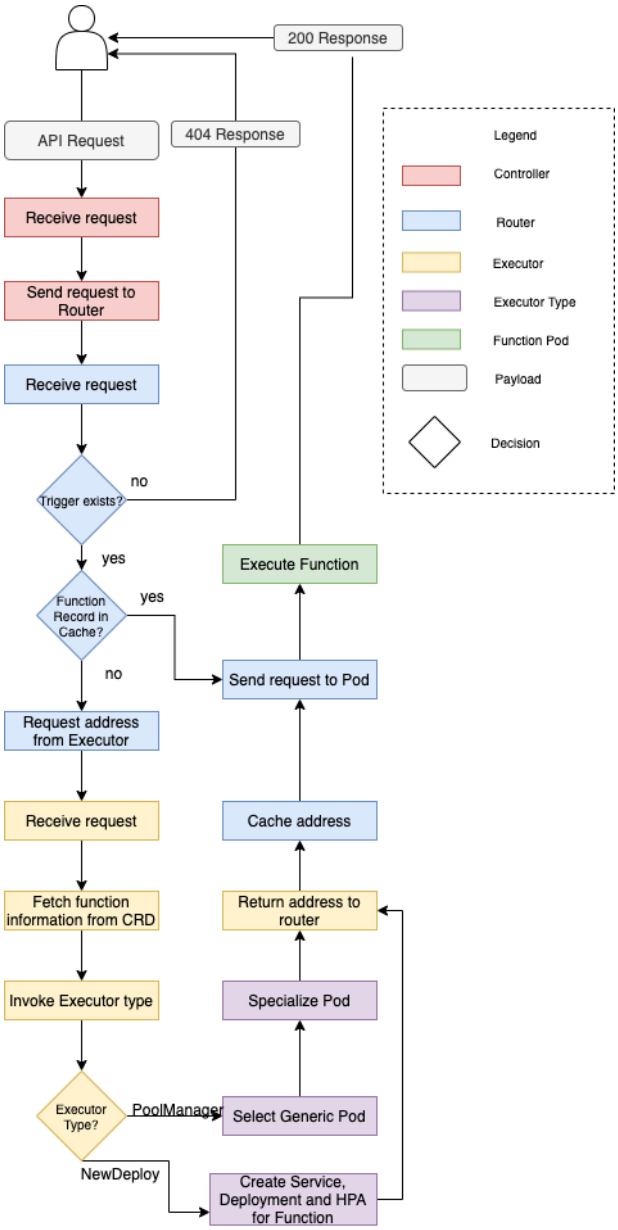

Figure 11: Fission invocation flow.

Each subsequent request for the same function is sent directly from Router to Function Pod, as long as the Function Pod is not idle for a while to get destroyed and removed from the cache.

4.5.3 Data flow

4.5.4 Reference architecture mapping

Fission is a complete serverless platform, meaning it handles all the responsibilities of the function management layer mentioned in the reference architecture.

StorageSvc stores deploy and source archives for

the functions. Function metadata is stored in Kubernetes CRD. Together they fulfill the role of the Function Registry in the reference architecture mapping. Builder Pod has the role of the Function Builder, by pulling source archive from StorageSvc, and along with the function metadata in CRD creates and stores the executable back to StorageSvc.

The role of a Function Deployer is taken by the Executor, along with its Executor types. Its role is to fetch the deploy archive from the StorageSvc and ensure there is at least one Pod running with the function specific Environment Container.

Function Instance is the Function Pod. It is a composition of two containers and shared volume through which these two containers communicate. Fission Router is the Function Router in the reference architecture mapping. It keeps track of all the addresses of Pods, and based on the presence of the address it either sends the request directly or it queues the request and waits for deployment of the function.

Executor, or more precisely NewDeploy Executor type is the component that represents the Function Autoscaler in the reference architecture. Since PoolManager is used for functions with low traffic to minimize cold start times that might occur more frequently but it does not scale Function Pods, NewDeploy on the other hand is more suitable for functions with high traffic load. It uses Kubernetes Service and HorizontalPodAutoscaler to scale Pods up and down.

The mapping is also highlighted in orange in the architecture diagram in Figure 10.

- **F1**: StorageSvc
- **F2**: Builder Pod
- **F3**: Executor
- **F4**: Function Pod
- **F5**: Router
- **F6**: Executor (NewDeployment)

5 Discussion and Comparison

The analyzed platforms are designed to handle different use cases and are aimed at the different types of users. The difference in focus causes the performance in the key aspects of serverless to vary. We look at some of the key serverless aspects in this section.

We discuss and compare the analysed platforms in the following topics: *General analysis*, *Routing*, and *Cold start optimizations*. We also discuss the relevance of the reference architecture for these platforms.

5.1 General Comparison

We identify two types of platforms while analyzing the platforms: *zero-ops platforms* and *serverless toolings*.Zero-ops platforms handle the whole function lifecycle management from building to deploying and running the service. Their aim is to provide a developer friendly platform where developers immediately start writing functions while hiding all the resource and function management away. In case of the platforms we have analyzed, the zero-ops platforms would be OpenWhisk, (partially) OpenLambda and Fission.

On the other hand, serverless toolings are extensions that enable management of serverless workloads on underlying resource management systems. Prominent resource management tool, used in some way by all of the analyzed platforms is Kubernetes. Kubernetes is a modular tool, with a possibility of defining custom objects that automate Kubernetes workflows.

The serverless toolings that we have identified are OpenFaas and Knative Serving. OpenFaas provides to some extent the full platform interaction as the zero-ops platforms, but some of the functionalities, like function building and function storage is delegated to other tools and platforms. Also in case of deploying the function, there is no function level behavior meaning the whole process is handled at the resource management layer. The advantage of OpenFaas is that it is not tied to Kubernetes and can work with any resource management tool as long as it is conforming to the `faas-provider` interface. OpenFaas provides a proprietary zero-ops platform OpenFaas Cloud, but the zero-ops flow is achieved by providing CI/CD integration, so these extensions are mostly delegated.

Knative Serving on the other hand does not provide a GUI and does not try to provide the full serverless platform experience. It is tied to Kubernetes and requires operations support. When defining new updates, the Kubernetes CRD needs to be updated. Knative is not aimed at the end-user, it is more of a platform for creating, deploying and managing serverless workloads.

5.2 Routing

Something that reference architecture by SPEC-RG did not catch, is the routing types in the serverless platforms. There is higher-level routing which is the action of routing the request to the right function instance manager. Function instance manager is the component that has the overview of all function instances, and it is effectively a representation of an active function in the system.

Once the request is received by the right function instance manager, it needs to be routed to the right function instance. This is the lower level routing that is handled by the Function Router component in the reference architecture. Many platforms, such as Knative Serving and OpenFaas bypass the higher level routing, meaning that the Function Router component is the Gateway itself. Gateway has the overview of all Function Instance addresses and therefore routes the requests directly to the Function Instance. In the other platforms, namely OpenWhisk and Fission, there are components responsible for routing the requests to the instance managers.

5.3 Cold Start Optimizations

Apart from OpenLambda, which has a basic check for paused containers, we have observed cold start optimizations in each platform. The usual approach to optimizing cold start is a pooling technique. In OpenWhisk, this technique is called pre-warming. There is a pool of pre-warmed containers that already contain language runtime environment. This is very similar to the technique implemented by Fission in the PoolManager, where a pool of generic containers is maintained per each language environment. Containers are then "specialized" by injecting function code. The main idea of this technique is avoiding the expensive `Docker run` call.

In case of the cold start in OpenWhisk the function code needs to be compiled (in case the language is compiled). Since Fission stores the deploy archive in the database, it avoids the compilation step which gives it better cold start times. OpenWhisk partly solves this issue with aggressive caching.

In case of OpenFaas and Knative Serving [8] there is no build process involved in cold start. In case there is no running functions, it requests Kubernetes or other resource management tool to run the container. Therefore, scaling from zero is achieved by persisting the function metadata and creating a new pod with Function Instance. The same approach is used in Fission NewDeployment. Some attempts of implementing pooling technique in Knative are present in [9].

5.4 Function Instance comparison

In terms of Function Instance organization, we have noted differences across different platforms. We identified two different designs present: *single container design* and *double container design*. The serverless platforms attach components to user function container to easily manage container lifecycle and report relevant metrics.

Single container design is the Function Instance architecture where a watcher process is attached to the user process inside the same container. This process is usually an HTTP server which allows modification of requests and overview of the amount of requests the Function Instance is receiving. This approach has a performance advantage because there is no overhead of communication between two containers using HTTP. On the other hand, there are some security concerns as the function code is not isolated inside a container.

We have observed this design in OpenFaas and OpenWhisk. OpenFaas runs the process `of-watchdog` which is a server written in Go. It also controls behavior of the user process, by either forking and destroying a user process every time a request is received (which increases latency) or running one user process for the Function Instance lifetime. In OpenWhisk the process is HTTP server as well. This component accepts two calls: *init* which compiles the source code and *run* which runs the user process.

Double container design is the Function Instance architecture where the watcher process is in the container of its own. This increases the overhead because of the communication between processes, but also allows for isolation of user container and increased security.

The platforms that implement double container function instance are Fission and Knative. Since both platforms are Kubernetes based, they are grouping these containers in a single pod. However, there is a difference in responsibilities between watchers in these 2 platforms. In case of Knative, the responsibility of the watcher container (Queue Container) is to report the request metrics to the Autoscaler component to provide elastic scaling. In Fission, the role of the second container is to watch the changes in the Kubernetes CRD and fetch the deploy archive if there is a new one available. In case of Fission there is also a third component present inside a pod, the shared volume, through which user container can use the deploy archive fetched by the Fission fetcher.

5.5 Reference Architecture Mapping

We have noticed a lot of similarities between the reference architecture and the architecture of the analyzed platforms. The components that seem to be always present are Function Store, Function Instance, Function Router and Function Autoscaler. This seems to be the core of every platform offering. We have observed that Function Builder and Function Deployer are missing from some platforms (or

delegated to external toolings provided by the same project). This is mainly connected to the serverless toolings, Knative Serving and OpenFaas, whose core offering is revolved around extensions to the resource management layer, mainly Kubernetes.

We have identified some inconsistencies in the reference architecture mapping that we already mentioned in the platform analysis. For example, Function Builder is marked as missing in OpenFaas, while it is marked as delegated in Knative Serving. However, both platforms are using toolings that are external to this platform but inside the larger offering, namely `faas-netes` and Knative Build.

We have also identified two different routing behaviors in these platforms as mentioned in the Routing subsection. Function Router definition in the reference architecture section is not clear enough. Function Router's responsibility is to route the request to the right Function Instance and to queue the request if there is no Function Instance available. However most of the platforms we have observed (OpenWhisk, Fission, OpenFaas) have routers that behave differently, or share functionality between two components. For example, Fission Router keeps address of the Function Service in cache. It does not keep the address of each Function Instance, it only knows there is at least one running. The task of routing the request to the Function Instance is handled by Function Service. However, the Fission Router has the queuing behavior mentioned. We think that the Function Router is not concerned with routing to the right Function Instance, but to the Function itself, however the Function is represented in the system (Invoker, Kubernetes CRD, Lambda Manager etc.)

6 Conclusion

Serverless, and more precisely Function-as-a-Service continue to grow as more and more businesses and developers realize the advantages that function-as-a-service provides. However, current market is revolved around serverless platforms offered by big cloud providers. AWS Lambda, Google Cloud Functions and Azure Functions are still market leaders by a large margin. Their projects are unfortunately largely closed-source which stiffles the innovation and causes the unavoidable vendor lock-in.

However, there are many promising open-source projects, with a lot of community recognition as we presented here. These projects are an invaluable source of information on the current innovations in the serverless industry. By analyzing the data flow and architecture of these platforms we can hypothesize about what is happening in the closed source projects.

In this work we utilize the Multivocal Literature Review (MLR) method to gain an insight into advances in the field of serverless platforms. By using this method we also expand the reference architecture of the serverless function management layer with a granular overview of each of the components. With this we answered the research questions posed in the Section 1.

RQ1. How to identify the fine-grained architectural components and their interactions in the function management layer of the serverless platforms? To answer this question we utilized the MLR method to analyze the primary and grey literature for 5 of the most well-known serverless platforms. From there we identified the similarities in the functionalities of different components in the function management layer. Using this insight in the Section 3.2 we present the fine-grained architecture.

RQ2. How to compare, across the well-known serverless platforms, the data and control flow occurring between the components of the function management layer? To answer this question in the Section 4 we presented an extensive analysis of the data and control flows in the well known serverless platforms. Along with the textual analysis we presented the diagrams which explain the interactions between the architecture components.

RQ3. How do the function layer components of the well-known open-source serverless platforms map to the reference architecture for the serverless platforms proposed in the SPEC-RG paper? To answer this question in the Section 4 we created the mapping of the components identified in the serverless platforms to the SPEC-RG serverless reference architecture. We pointed out the inconsistencies between the mapping we produced and the mapping produced in the serverless reference architecture paper, and provided possible reasons for the differences.

We see that a lot of the projects are utilizing other cloud native open-source projects, many of them incubated by the Cloud Native Computing Foundation (CNCF). Interconnecting these projects creates a stronger ecosystem that everyone benefits from.

We conclude that there is still a lot of work to be done in the field of serverless, and serverless is not

suited for every use case. However, by examining the projects we see that the community is getting increasingly interested. We call for the large cloud providers to open-source their serverless platform code to speed up the innovation.

References

[1] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: towards high-performance serverless computing. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 923–935. USENIX Association, 2018.

[2] S. Allen, C. Aniszczyk, C. Arimura, et al. Cncf serverless whitepaper, 2018.

[3] E. V. Eyk, A. Iosup, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, and C. L. Abad. The SPEC-RG reference architecture for faas: From microservices and containers to serverless platforms. *IEEE Internet Comput.*, 23(6):7–18, 2019.

[4] M. Gabbrielli, S. Giallorenzo, I. Lanese, F. Montesi, M. Peressotti, and S. P. Zingaro. No more, no less - A formal model for serverless computing. In H. R. Nielson and E. Tuosto, editors, *Coordination Models and Languages - 21st IFIP WG 6.1 International Conference, COORDINATION 2019, Held as Part of the 14th International Federated Conference on Distributed Computing Techniques, DisCoTec 2019, Kongens Lyngby, Denmark, June 17-21, 2019, Proceedings*, volume 11533 of *Lecture Notes in Computer Science*, pages 148–157. Springer, 2019.

[5] V. Garousi, M. Felderer, and M. V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology*, 106:101–121, 2019.

[6] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In A. Clements and T. Condie, editors, *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.

[7] B. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

[8] J. Li, S. G. Kulkarni, K. K. Ramakrishnan, and D. Li. Understanding open source serverless platforms. *Proceedings of the 5th International Workshop on Serverless Computing - WOSC '19*, 2019.

[9] P.-M. Lin and A. Glikson. Mitigating cold starts in serverless platforms: A pool-based approach, 2019.

[10] T. Lynn, P. Rosati, A. Lejeune, and V. C. Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017*, pages 162–169. IEEE Computer Society, 2017.

[11] M. G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In A. Musaev, J. E. Ferreira, and T. Higashino, editors, *37th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2017, Atlanta, GA, USA, June 5-8, 2017*, pages 405–410. IEEE Computer Society, 2017.

[12] S. K. Mohanty, G. Premsankar, and M. D. Francesco. An evaluation of open source serverless computing frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2018, Nicosia, Cyprus, December 10-13, 2018*, pages 115–120. IEEE Computer Society, 2018.

[13] A. Pérez, G. Moltó, M. Caballer, and A. Calatrava. Serverless computing for container-based architectures. *Future Gener. Comput. Syst.*, 83:50–59, 2018.

[14] J. Scheuner and P. Leitner. The state of research on function-as-a-service performance evaluation: A multivocal literature review, 2020.

[15] M. Sciabarrà. *Learning Apache OpenWhisk: Developing Open Serverless Solutions*. O'Reilly Media, 2019.

[16] J. Spillner. Snafu: Function-as-a-service (faas) runtime design and implementation. *CoRR*, abs/1703.07562, 2017.

[17] J. Spillner and M. Al-Ameen. Serverless literature dataset, Apr. 2019. 3rd generation dataset, described in blog post: https://blog.zhaw.ch/splab/2019/04/22/the-5th- year-of-serverless-computing-research-coverage/ + website http://serverless.research-output.org/.

[18] G. Totoy, E. F. Boza, and C. L. Abad. An extensible scheduler for the openlambda faas platform. In *Min-Move workshop (co-located with ASPLOS)*, 2018.

[19] E. van Eyk, J. Grohmann, S. Eismann, A. Bauer, L. Versluis, L. Toader, N. Schmitt, N. Herbst, C. L. Abad, and A. Iosup. Mapping of the faas platforms, June 2019.

[20] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. M. Swift. Peeking behind the curtains of serverless platforms. In H. S. Gunawi and B. Reed, editors, *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 133–146. USENIX Association, 2018.