The SPEC-RG Reference Architecture for FaaS: From Microservices and Containers to Serverless Platforms

Erwin van Eyk
Vrije Universiteit Amsterdam and Platform9 Systems

Johannes Grohmann
University of Wuerzburg

Simon Eismann
University of Wuerzburg

André Bauer
University of Wuerzburg

Laurens Versluis
Vrije Universiteit Amsterdam

Lucian Toader
Vrije Universiteit Amsterdam

Norbert Schmitt
University of Wuerzburg

Nikolas Herbst
University of Wuerzburg

Cristina L. Abad
Escuela Superior Politecnica del Litoral

Alexandru Iosup
Vrije Universiteit Amsterdam

***Abstract*—Microservices, containers, and serverless computing belong to a trend toward applications composed of many small, self-contained, and automatically managed components. Core to serverless computing, function-as-a-service (FaaS) platforms employ state-of-the-art container technology and microservices-based architectures to enable users to manage complex applications without the need for system-level expertise. Victim of its own success, and partially due to proprietary technology, currently**

Digital Object Identifier 10.1109/MIC.2019.2952061
Date of publication 8 November 2019; date of current version 17 January 2020.

the community has a limited overview of these platforms. To address this, we propose a reference architecture and ecosystem for FaaS platforms. Based on a year-long survey of real-world platforms conducted within the SPEC-RG Cloud Group, we highlight specific components and identify common operational patterns.

For the past three decades, many applications have been shrinking in size and exploding in number. Continuing a trend already occurring in grid computing,[1] cloud computing technology and software development practice have enabled more diverse workloads and more focused tasks. Recently, advances in container technology (such as Docker) leveraged this shrinking trend to decrease the build-time and the deployment overhead of software, enabling far more granular processes compared to traditional server-based provisioning and even to virtual machines (VMs). In parallel, aiming to improve development time, and system resiliency and scalability, development practices have promoted new architectures of fine-grained microservices over the traditional monolithic and service-oriented architectures. The emerging technology of serverless computing combines these advances,[2] taking one more step toward *computing as an Internet-based utility*. But what is the status of serverless technology and where is it heading?

Serverless computing is a set of (cloud) computing technologies that adhere to three principles:[3] 1) operational logic is abstracted away from users; 2) users only pay for the resources they need, with fine granularity; and 3) the computing model is event driven and operations are scaled elastically.

Function-as-a-Service (FaaS) is a model for serverless operations offered "as a service." Cloud users providing *functions* to the cloud provider, who, for a preagreed payment, are responsible for the full operational lifecycle—from deploying the function in the datacenter to ensuring that security patches are applied to the software stack executing the function.

FaaS offers good reasons to exist within the already rich cloud service landscape. It provides high-level abstractions of distributed computing elements, reducing the need for users to be experts in distributed systems, or to manage complex microservice-based architectures themselves. It lets users delegate operational issues, allowing organizations to focus on addressing business concerns.

The market uptake of serverless computing and FaaS is large and rapidly increasing. The market currently hovers at around $5 billion and is predicted to be worth nearly $15 billion by 2023 (https://www.marketsandmarkets.com/Market-Reports/serverless-architecture-market-64917099.html). Correspondingly, every major public cloud provider already offers a FaaS solution as well as other serverless products, such as serverless-friendly databases (AWS Aurora). In the open-source community, numerous FaaS platforms and serverless projects are actively being developed. Finally, the potential of serverless computing has triggered the academia.[4–6]

Yet, reminiscent of the early days of cloud computing, serverless computing lacks a community-wide consensus on the fundamental architectures, components, and practices. (We and others have already proposed terminology and properties.)[3] Compounding the problem, for FaaS the cloud providers lack an incentive to make their serverless platforms transparent, or even open source. This could obscure emerging solutions and hamper the development of best practices, design patterns, and knowledge of how the field evolves. For example, as we show in this article, the current approaches for managing functions and the *workflows* developers compose from them are poorly understood.

For grid and cloud computing, prominent reference architectures[7,8] were designed early, and helped guide from public discussion to system design. Thus, toward a community-wide view on serverless computing, we propose a reference architecture for FaaS platforms. Started as a project in the SPEC-RG Cloud Group (https://research.spec.org/working-groups/rg-cloud.html), whose goal is to investigate performance and operations in cloud computing settings, ours is *the first systematic approach to identify the architectural patterns and components that enable FaaS operation*. To show the usefulness of the reference

architecture, we have spent nearly a year tracking, cataloging, and investigating dozens of existing FaaS platforms and related systems.

Our work complements an emerging body of research in serverless computing, from academia[9,10] and industry (https://github.com/cncf/wg-serverless/blob/master/whitepapers/serverless-overview/cncf_serverless_whitepaper_v1.0.pdf), and existing reference architectures for workflow management[11] and cloud resource management.[12] Focusing on FaaS platforms, and in contrast to this body of related work, this work makes a threefold contribution:

1) We design a reference architecture and ecosystem for FaaS platforms (see Section REFERENCE ARCHITECTURE). Our reference architecture is the first comprehensive approach to identify the common layers and operational components of FaaS platforms. It models a wide range of system designs and abstractions, from entire FaaS platforms orchestrating workflows to FaaS layers managing individual functions and components responsible for managing resources.
2) We provide a summary of the systematic analysis of existing FaaS platforms (Section SURVEY AND MAPPING), in which we identify 47 FaaS-like systems. We cover diverse open-source and closed-source platforms, from workflow composition engines (e.g., Fission Workflows and AWS Step Functions) to single-function engines (e.g., Apache OpenWhisk and AWS Lambda) and cloud resource managers (e.g., Kubernetes). The analysis in this section not only explains the architecture and components of the platforms we study, but also gives strong evidence that the reference architecture is comprehensive.
3) We identify a set of common operational patterns in FaaS platforms (Section DISCOVERING OPERATIONAL PATTERNS) and leverage the analysis results to create knowledge about common solutions adopted by FaaS platforms to common operational problems. We present three such patterns for building and executing functions, and for executing workflows. The analysis in this section also emphasizes how the reference architecture can give system designers and engineering an instrument for reaching consensus on emerging patterns.

The proposed reference architecture aims to bring many practical and long-term benefits. We have already mentioned the role reference architectures have played for grid and cloud computing; in general, this reference architecture could play a similar role. Specifically, it could guide developers of serverless functions by providing them with a better understanding of the common architectural and operational patterns of serverless platforms. It also facilitates public discussion about the serverless computing model, by providing stakeholders with a common terminology and understanding of serverless platforms. Additionally, teams looking to develop custom serverless platforms can use the reference architecture as a blueprint for innovation, against which new designs can be compared. Already, a serverless workflow platform has been codesigned and built in parallel with the reference architecture and drew guidelines from it.[13]

REFERENCE ARCHITECTURE

The key conceptual result of this work is the reference architecture. Figure 1 depicts its three hierarchical layers, each of which captures a set of core responsibilities of a generic FaaS platform. The reference architecture is the result of a number of design choices, including:

1) The current serverless ecosystems include much possible functionality, e.g., user interface, monitoring, security. We consider the entire ecosystem explicitly in Section Reference Ecosystem, but focus the reference architecture on the functionality needed to execute (workflows of) functions. This includes elements from composing workflows down to orchestrating the resources needed to run them.
2) The reference architecture omits explicit data, control, and—specific to serverless computing, see Section Function Management Layer—function-code flows between components. We found throughout our study of FaaS platforms (see Section SURVEY AND MAPPING) that, in practice, their components can form many communication topologies and

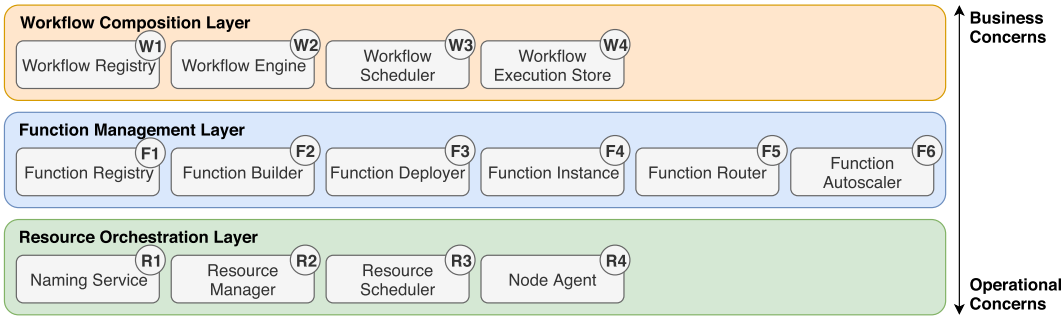

Figure 1. SPEC-RG reference architecture for FaaS platforms.

use diverse communication styles. Thus, a general reference architecture should not fix either the topology or communication style.

Resource Orchestration Layer

At the lowest level in the reference architecture, the *Resource Orchestration Layer* is responsible for the management of physical resources of a cluster of machines. The components in this layer manage the operational lifecycle of the containers or VMs, which are consolidated on physical resources.

We separate this generic resource layer from the FaaS-specific layers to indicate where the FaaS platforms fit into the existing cloud infrastructure, and to capture the common approach of FaaS platforms to delegate resource management to more mature systems (such as Kubernetes).

The Resource Orchestration Layer consists of:

1) *Naming Service*: provides cluster-wide unique and consistent naming to resources. This allows components to identify each other. In some cases, naming is extended to also store cluster-wide configurations.
2) *Resource Manager*: manages the available resources of cluster-nodes through node agents, ensuring that the state of the resources (eventually) conforms with the desired state. It also monitors resources for changes, and executes actions if needed.
3) *Resource Scheduler*: determines which actions are needed to ensure that the current state of the resources converges toward the desired state of the resources in the most efficient way. In general, the scheduler decides for each job, on *which* resources it should be deployed.
4) *Node Agent*: is deployed on each node in the cluster. It monitors the local resources, informs the resource manager, and executes instructions it receives from the Resource Manager.

Function Management Layer

The *Function Management Layer* contains the core components responsible for the (operational) lifecycle of individual FaaS functions: deploying *function instances*, executing functions triggered by events, and elastically scaling functions.

Whereas the resource orchestration layer is concerned with the management of arbitrary resources, the function management layer manages arbitrary functions. In this layer, the components rely on the lower level layer for the correct management of the resources.

At its core, the function management layer consists of:

1) *Function Registry*: serves as a (local or remote) repository of functions. In practice, this registry is often further split into a *function metadata store*, for low-latency look-ups of function metadata, and a *function store*, which contains the binaries of the function (the *function-code*).
2) *Function Builder*: turns function sources into deployable functions. Functions typically have to undergo transformations (e.g., compiling, validating, dependence resolving) before they are stored in the *function registry* or deployed by the Function Deployer.
3) *Function Deployer*: ensures an instance of an arbitrary function, a function instance (see next component), is deployed. The function

deployer combines the configuration stored in the function registry, the parameters supplied by the requester, and other factors into a decision of how the function should be deployed, including how many and what kind of resources it should receive. Though it decides *how* the function instance should be deployed, the deployment of the function instance itself is delegated to the resource orchestration layer.

4) *Function Instance*: is a self-contained worker —typically a container—capable of handling function executions. For scalability, a function can have multiple concurrent function instances.
5) *Function Router*: routes incoming requests or events to the correct function instance. If no function instance is available, the function router queues the events to await the deployment of new instances.
6) *Function Autoscaler*: monitors the demand and supply of resources, and elastically scales the number of function instances—adding or removing instances as needed.

Workflow Composition Layer

Modern applications require the orchestration of multiple functions and the management of interfunction state. We consider these two related concerns—state management and interdependent functions—to be the responsibility of the higher level *Workflow Composition Layer*.

Previous work[3] identified workflows as an appropriate abstraction for function composition. A *workflow* is a set of interdependent *tasks* (functions) that together form a larger structured computation or data-processing operation. A workflow management system (WMS) orchestrates the workflow; it is responsible for ensuring that executions (or *instances*) of this workflow progress to completion correctly, e.g., by passing the output of a task to its successors despite the failure of individual tasks. (In specialist terms, workflows cover both control flows and data flows.)

The motivation for a separate, higher level layer is twofold. First, composing functions into more complex *workflows* depends on the components in the function management layer to ensure the correct deployment and execution of *individual functions*. This allows the higher level layer to focus solely on concerns such as scheduling the correct functions, transferring state from one function to another, etc. Second, although the notion of workflows is useful, not every FaaS platform includes support for it yet; often platforms only offer the execution of individual functions and leave the complexity of managing workflows to the user.

As with the resource orchestration layer, we rely on the extensive existing work on WMSs for the components of this layer. The difference with original workflow systems is that *serverless workflows* are smaller, executed more frequently, and have more demanding performance requirements.

The workflow composition layer consists of:

1) *Workflow Registry* serves as a (local or remote) repository of workflows. To be admitted to this registry, the workflow typically requires validation and compilation of its individual tasks.
2) *Workflow Engine* is responsible for monitoring workflow executions. It takes the appropriate action based on decisions from the Workflow Scheduler, such as triggering execution of functions in the workflow whose predecessors have completed.
3) *Workflow Scheduler* decides *which* functions to execute *when*. It makes these decisions based on a number of factors, including the current state of the workflow execution and historical data.
4) *Workflow Execution Storage*ensures the persistence of data of workflow executions. To ensure reliable workflow executions, a database holds the state of workflow executions. The strictness of the persistence depends on the level of reliability guaranteed by the FaaS platform.

Reference Ecosystem

Balancing conciseness with completeness in a reference architecture is a delicate task. Too much detail makes it difficult to comprehend and remember; too little detail makes it miss fundamental components. We therefore keep the reference architecture focused, but also, to complement it, we propose a *reference ecosystem*: what components and systems are not vital or

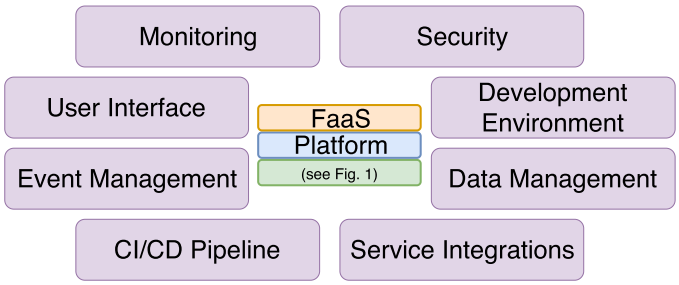

Figure 2. FaaS ecosystem with the FaaS reference architecture represented in the center.

specific to the core of executing (workflows of) functions, but are typically present in production-ready FaaS platforms?

The reference ecosystem in Figure 2 positions the FaaS platform (described by the reference architecture) as the functional core of a broader FaaS ecosystem. Production-grade FaaS platforms, such as those from major cloud providers, contain additional components to improve developer experience, maintainability, event management, and data management.

Components in the reference ecosystem include the following:

- *User interface*: Allows users to manage and interact with the FaaS platform, which range from simple command-line interfaces, to full-fledged (visual) editors.
- *Developer environment*: Reduces the difficulty of developing functions and workflows by integrating with popular editors and tools and providing local emulators of (proprietary) FaaS platforms.
- *Event management*: Manages the lifecycle of the events triggering the operation of a FaaS platform. Although the FaaS reference architecture deploys and executes functions based on the arrival of events, it does not include the lifecycle of these events.
- *Data management*: Manages the lifecycle of data in the FaaS ecosystem—how the data are stored and transferred. Ideally, a data management system stores and transfers data efficiently in-between the functions, ensuring low latency and availability for a low cost.
- *Continuous integration (CI) and deployment (CD)*: Manages the lifecycle of function sources. Depending on those processes, the CI/CD pipeline typically consists of staging environments, complex dependence resolving, multiple stages of testing, governance checking, and (manual) approval steps—before the function is submitted to the Function Builder.
- *Service integration*: Manages integrations between the FaaS and other cloud services.
- *Monitoring and logging*: Monitors and logs metrics on arbitrary layers and components for online or later visualization and analysis.
- *Security*: Ensures proper user (and function) authorization and authentication to access and change parts of the FaaS platform.

SURVEY AND MAPPING

To validate the FaaS reference architecture and gain insight into real-world FaaS platforms, we conducted a year-long process of identifying, mapping, and analyzing FaaS-like platforms, that is, platforms for resource, function, and workflow management. We describe first the process, then summarize (the full mapping can be found at https://go.uniwue.de/faasmapping) the mappings (see also Table 1), and finally present our main findings.

We have *identified* 47 relevant platforms through a systematic survey of: 1) FaaS platforms of the major public cloud providers in the U.S. and Europe, e.g., Amazon, Microsoft, IBM, and Google; 2) open-source efforts present on GitHub or under the guidance of open-source foundations, such as Apache and CNCF; 3) academic projects published (we use the procedure to identify relevant papers proposed by Papadopoulos *et al.*,[14] traversing systematically all articles published in the past 7 years in 16 top systems conferences and journals) in top systems venues cosponsored by IEEE, ACM, and USENIX.

Table 1. Mappings of platforms to reference architecture. Symbols: ● : Yes; ○ : No; ◐ : Delegated; — : Unknown.

	Open	Resource Orchestration				Function Management						Workflow Composition			
Platform	Source?	R1	R2	R3	R4	F1	F2	F3	F4	F5	F6	W1	W2	W3	W4
Kubernetes	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○
Marathon (Apache Mesos)	●	●	●	●	●	○	○	○	○	○	○	○	○	○	○
AWS Lambda	○	—	—	—	—	—	—	—	—	—	—	○	○	○	○
Fission	●	◐	◐	◐	◐	●	●	●	●	●	●	○	○	○	○
Knative Serving	●	◐	◐	◐	◐	●	◐	—	●	◐	●	○	○	○	○
Apache OpenWhisk	●	◐	◐	◐	◐	●	●	●	●	●	●	○	○	○	○
Apache Airflow	●	◐	◐	◐	◐	○	○	●	●	○	●	●	●	●	●
AWS Step Functions	○	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	—	—	—	—
Azure Durable Functions	◐[4]	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	—	●	●	●
Fission Workflows	●	◐	◐	◐	◐	◐	◐	◐	◐	◐	◐	●	●	●	●

We then *mapped* the FaaS platforms to the reference architecture, using a pair-reviewing system. Each platform was assigned to 2 out of the 10 participating reviewers. We distributed the platforms among the reviewers to minimize the number of platforms reviewed by the same pair of reviewers—each pair of reviewers analyzed the same platform at most twice. Then, the reviewers analyzed the assigned platforms independently, by surveying the corresponding scientific publications, online documentation, technical reports, presentations, and source code (when available). The result of each mapping was, for each component of each platform, a decision of whether the component was explicitly present ("Yes" in Table 1), explicitly delegated to another platform ("Delegated"), or clearly not present ("No"). We also encountered a fourth situation, "Unknown," where the reviewers were not able to find any information about whether a component exists or not; although reviewers could make educated guesses about the existence of at least some of these components, they could not find evidence to support this. Twelve (closed-source) platforms did not disclose any concrete information for any of the components. Rather than exclude these, we kept these systems in the overview to highlight the opaqueness of some serverless platforms.

When *merging the mappings across all reviewers*, for any conflict (i.e., reviewers' opinions differ), the reviewers compared their notes and tried to reach a consensus. This was possible for all conflicts, as the most frequent type of conflict was that one reviewer found additional documentation (for example, a new blog post).

Summary of results across all platforms: The majority (62%) of the platforms we analyzed are open source. This includes all platforms focusing on the resource orchestration layer (10), and all the WMSs that are actually not specific to FaaS (9). However, only roughly half of the serverless platforms were open source. From the platforms for which we could establish a mapping, those focusing on the Resource Orchestration layer include components R1–R4; a *de facto* standard. In the function management layer, all platforms include components F1 and F6; the other components are identifiable in some but not all platforms. Finally, in the Workflow Composition layer, over 80% of the platforms contain components W1–W4; additionally, the platforms that do not contain W1–W4, are not specifically designed for the serverless domain.

Summary of results for the platforms in Table 1: To illustrate the recurring themes we found in our analysis, we highlight a representative subset of the mapped systems in Table 1, and describe them in the remainder of this section. This includes seven well-known projects originating from major cloud providers (e.g., Google, Amazon, Microsoft) and/or present in the CNCF and Apache open-source ecosystem, plus three full-featured projects that operate independently and release open-source artifacts (i.e., Knative Serving, Fission, and Fission Workflows).

Kubernetes (K8S) is currently a de facto standard for container-based resource orchestration—many of the FaaS platforms in our study depend on K8S. K8S maps to the entire resource orchestration layer, confirming its representativeness.

Marathon is part of the Apache Mesos project, for which it offers resource management functionality. Although not often used to support FaaS platforms, it offers similar functionality to

K8S, indicating the Resource Orchestration Layer is approaching standardization.

AWS Lambda is a representative for other managed FaaS platforms (e.g., Azure Functions, Google Cloud Functions). AWS Lambda highlights a key issue in this domain: the proprietary nature of these systems. As argued in Section INTRODUCTION, this has significant drawbacks, e.g., it limits the understanding of system internals.

Fission is a representative for the wide ecosystem of open-source alternatives evolved in response to the closed-source, managed FaaS platforms. Fission has a wide range of function management components, and delegates resource management to K8S.

Knative Serving is the part closely resembling the core of a FaaS platform, from the Knative set of projects aiming to provide mature tooling for common (serverless) use cases running on K8S. It is positioned to be an extension to K8S itself, to which it delegates most functionality. Knative also highlights a recurring issue in open-source FaaS platforms: it lacked detailed documentation, such as architecture diagrams, which made the mapping difficult.

Apache OpenWhisk is one of the most production-ready, open-source FaaS platforms. OpenWhisk maps to the entire function management layer. It is one of few open-source FaaS platforms that (to our knowledge) is deployed in a large cloud (IBM Cloud), hinting that other proprietary FaaS platforms do not differ much architecturally from the open-source alternatives.

Apache Airflow is a representative of workflow engines repurposed from their original workloads to also serve FaaS workloads. As reflected by the lack of mapped components (corresponding cells marked "No" in Table 1), running this as a full-fledged FaaS WMS requires additional engineering. As with OpenWhisk, Airflow is available both open-source and as a managed service offered by a major cloud provider—Google Cloud Composer is based on Airflow.

AWS Step Functions popularized in the cloud community, together with AWS Lambda, the notion of composing functions into workflows. Like Lambda, step functions is closed source, making it difficult to analyze. For example, although we believe it relies on other components for the function and resource management layers, the actual architecture and details remain unclear.

Azure durable functions operates in the workflow composition layer, but employs a different programming model than the other FaaS platforms: it requires users to write their own simple workflow scheduler, instead of writing declarative workflow definitions. This allows users to express orchestrations without requiring a special workflow language, but requires a different skill set. In part due to its alternative take on workflows, it contains comprehensive documentation on the key concepts and on some of its internal details.

Fission Workflows (full disclosure: the lead author, Erwin van Eyk, leads the development of this platform.) is one of few workflow engines built to specifically target serverless workflows. It focuses on the fast and frequent execution of serverless workflows. As the name suggests, it integrates well with the Fission FaaS platform.

Overall analysis: From Table 1, we make three key observations: First, the different FaaS platforms map well to the layers of the introduced reference architecture. The platforms split functionally as predicted by the reference architecture, as indicated by the separation between AWS Lambda and AWS step functions, and fission and fission workflows. They also rarely cross the layer responsibilities defined in the reference architecture. Instead, they delegate responsibilities of other layers to other platforms.

Second, even for the subset of systems in Table 1, we identified significant differences between the systems with regard to the functionality provided by the same type of component (e.g., the way the Function Router handles cold starts is system dependent). Thus, including finer-granularity components in the reference architecture would have the potential to mispredict what is used in practice and possibly stifle innovation due to overfitting to current conditions.

Third, we have identified a significant number of FaaS-like platforms, over 15, that are closed source and thus difficult to map and analyze. We ask the developers of these platforms to open source, both to help the community develop and as an incentive to attract *lock-in-adverse clients*.

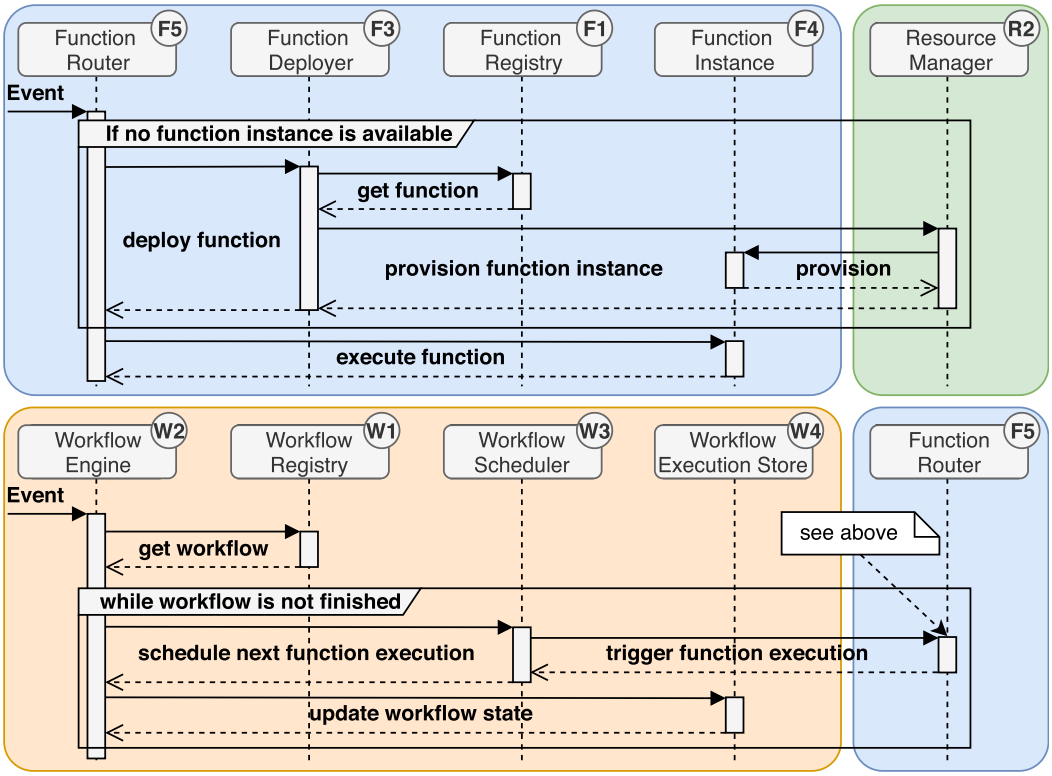

Figure 3. Two operational patterns: (top) function execution. (bottom) workflow execution.

DISCOVERING OPERATIONAL PATTERNS

We focus in this section on identifying *operational patterns*: recurring solutions that FaaS developers have included in their FaaS architectures to address prevalent problems. The patterns we identify show how the components of the reference architecture can interact with each other to facilitate common functionality, and appear in the FaaS platforms we have studied for this work (Section SURVEY AND MAPPING). We address in the following three key operational patterns: building functions, executing functions, and executing workflows. These patterns are not binding, meaning other approaches are possible, but they are widespread in current FaaS platforms.

Function Building

A common operational pattern is that of a user submitting functions to the FaaS platform, as source code. The source code traverses the CI/CD pipeline defined by the user (see Figure 2) before arriving at the FaaS platform. After initial validation, the source code is sent to the Function Builder, which transforms it into a deployable function.

This process depends on the implementation of the FaaS platform—it can be nearly nonexistent—but typically involves build steps that are language-specific, such as compiling Java to bytecode or resolving Python dependencies. Once the building process has completed, the resulting artifacts are stored in the Function Registry. From there, the function is available to the FaaS platform as a deployable function.

Function Execution

Function execution in FaaS [see Figure 3 (top)] inherently involves the function deployment, as function instances scale up from zero and scaled back to zero after inactivity. First, an event, such as a HTTP request, arrives at a Function Router, which triggers the deployment of a function instance if none is available—a *cold start*. Then, the Function Deployer fetches the function (metadata) from the Function Registry to decide the appropriate configuration of the function instance. It then tasks the resource manager with ensuring that a function instance with the

appropriate configuration is deployed. Once the function instance is fully deployed, the event can be passed to it to start the function execution.

If a function instance is already available for handling the execution, the expensive *cold start* process can be bypassed. The function router directs the event to the existing function instance—be it directly over RPC, or using a message queue.

Workflow Execution

The workflow execution pattern in Figure 3 (bottom) builds on the pattern for individual function execution by maintaining inter-function state and transferring data between functions. An event arrives at the Workflow Engine, either proxied by the function router or directly from connected event sources. It contains a workflow identifier, which is used to fetch the related workflow definition from the Workflow Registry. Then, the Workflow Engine queries the Workflow Scheduler for decisions on which functions to execute next. Based on these decisions, the Workflow Engine triggers the execution of the individual functions, and persists the operations and data to the Workflow Execution Store. This process continues until the workflow execution is completed.

OPEN CHALLENGES

The FaaS reference architecture is merely a starting point for further research around the serverless model. In previous work, we identified over 20 challenges in software, systems, and performance engineering.[3,15] We link a subset of these to the new insights offered by the reference architecture:

1) By identifying common architectural and operational patterns, our reference architecture provides a needed conceptual step for defining a community-wide benchmark for serverless platforms. The SPEC-RG Cloud Group is currently developing such a benchmark, aiming to peer into each of the layers and components identified in the reference architecture. But benchmarks are, by their broad nature, a limited tool for performance analysis; they can answer the question "how well?" but not "why?" for different platforms, and even for different *configurations* of the same platform or component. To answer the "why?," the reference architecture could help peer meaningfully into each component, by emphasizing the interplay between the components and the full-platform (cost of) operation.
2) The cold-start impact on the performance of FaaS is still a key obstacle to serverless adoption, especially for latency-sensitive workloads. As Figure 3 shows, the *cold-start process* involves multiple steps and components, which ensure correct operation but also slow it down. Promising directions in this space include *reducing overhead* through better engineering, introducing more sophisticated *scheduling policies*, and *profiling* function performance more accurately.
3) Within the FaaS ecosystem, many concepts remain relatively unexplored. What kind of *programming models* are intuitive for developing functions and workflows at scale? How do event- and data-management systems benefit from the characteristics of serverless *computational models*?
4) We foresee that the use of the *reference architecture* can guide the development and tuning of serverless systems, e.g., a team developing a new serverless platform could, from their first design session, already have a set of target components for their new architecture. *How can the reference architecture guide these processes? How can we leverage the architecture and patterns to improve the likelihood of good designs and fast time-to-market of serverless systems?* Additionally, a common reference architecture could foster designs of reusable, open-source solutions for the individual components of serverless platforms. *How to design interfaces of the individual components of a serverless platform be designed, to enable reuse?*

CONCLUSION

Serverless computing, and its cloud variant of Function-as-a-Service (FaaS) deployed on containers, promise to enhance existing microservice architectures by running them as more centrally managed services. This would allow

software developers to focus on business concerns and defer operational logic to specialized (internal) cloud providers. Already, tens of serverless and FaaS platforms exist. To achieve their full potential, and to prevent that improving them or selecting between them become unnecessarily daunting problems, we ask: What is the current FaaS technology? Where is it heading?

Addressing these questions, in this work we design a reference architecture and ecosystem for FaaS platforms. Concluding a long-term, systematic investigation, we map to the reference architecture 47 FaaS-like platforms. We find that real-world platforms map well to the architecture, and match the layered structure. We further use the reference architecture to discover operational patterns, and to present directions for future research.

The FaaS reference architecture offers a much needed analytic framework to industry and academia. It provides representative layers and components that are common to dozens of serverless and FaaS platforms in the field. By providing a higher level abstraction, it can help conquer or limit the inherent complexity required to design, tune, and even to compare such systems. Concretely, it has already guided the design of a serverless workflow platform, and is at the core of a new serverless benchmark within the SPEC-RG Cloud Group.

We conclude with a call to action for providers of closed-source platforms to reveal their designs or even their entire source code. This will enable the community to develop, and increase the level of trust that potential users have in these FaaS platforms.

Data availability: The authors of this study follow open science processes. All the data presented in this study are published on Zenodo: https://go.uniwue.de/faasmapping.

REFERENCES

1. A. Iosup and D. H. J. Epema, "Grid computing workloads," *IEEE Internet Comput.*, vol. 15, no. 2, pp. 19–26, Mar. 2011.
2. E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uta, and A. Iosup, "Serverless is More: From PaaS to present cloud computing," *IEEE Internet Comput.*, vol. 22, no. 5, pp. 8–17, Sep. 2018.
3. E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC Cloud Group's research vision on FaaS and serverless architectures," in *Proc. 2nd Int. Workshop Serverless Comput.*, 2017, pp. 1–4.
4. S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with OpenLambda," in *Proc. 8th USENIX Workshop Hot Topics Cloud Comput.*, 2016, pp. 33–39.
5. E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, 2017, pp. 445–451.
6. A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic ephemeral storage for serverless analytics," in *Proc. 13th USENIX Symp. Oper. Syst. Design Implement.*, 2018, pp. 427–444.
7. F. Liu *et al.*, "NIST cloud computing reference architecture," *NIST Special Publication*, vol. 500, no. 2011, pp. 1–28, 2011.
8. I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure.* Amsterdam, The Netherlands: Elsevier, 2003.
9. J. Spillner and M. Al-Ameen, "Serverless literature dataset," 2nd Generation Dataset, ODP converted to JSON files," 2018. [Online]. Available: https://doi.org/10.5281/zenodo.1436432
10. P. G. López, M. S. Artigas, G. París, D. B. Pons, Á. R. Ollobarren, and D. A. Pinto, "Comparison of production serverless function orchestration systems," *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion*, 2018, pp. 148–153.
11. D. Hollingsworth and U. Hampshire, "Workflow management coalition: The workflow reference model," Document No. TC00-1003, vol. 19, 1995.
12. R. B. Bohn, J. Messina, F. Liu, J. Tong, and J. Mao, "NIST cloud computing reference architecture," in *Proc. IEEE World Congr. Services*, 2011, pp. 594–596.
13. E. van Eyk, "The design, productization, and evaluation of a serverless workflow-management system," 2019.
14. A. V. Papadopoulos *et al.*, "Methodological Principles for Reproducible Performance Evaluation in Cloud Computing - A SPEC Research Technical Report," SPEC Research Group—Cloud Working Group, Standard Performance Evaluation Corporation (SPEC), Tech. Rep. SPEC-RG-2019-04, April 2019. [Online]. Available: https://research.spec.org/fileadmin/user_upload/documents/rg_cloud/endo rsed_publications/SPEC_RG_2019_Methodological_Principles_for_Reproducible_Perf ormance_Evaluation_in_Cloud_Computing.pdf

15. E. van Eyk, A. Iosup, C. L. Abad, J. Grohmann, and S. Eismann, "A SPEC RG Cloud Group's Vision on the performance challenges of FaaS cloud architectures," in *Proc. Companion ACM/SPEC Int. Conf. Perform. Eng.*, 2018, pp. 21–24.

Erwin van Eyk leads the research effort into serverless and FaaS architectures within the SPEC RG Cloud Group. He is part of the AtLarge Research Group, Vrije Universiteit and Delft University of Technology in the Netherlands and works as a Software Engineer at Platform9 Systems. His research interests include resource management and scheduling in distributed systems, specifically serverless computing. Contact him at e.vaneyk@atlarge-research.com.

Johannes Grohmann is currently working toward the Ph.D. degree in software engineering, University of Würzburg and serves as an election manager of the SPEC Research Group. His research topics include machine learning, performance modeling, and model learning, self-aware computing as well as cloud and serverless computing. Contact him at Johannes.grohmann@uni-wuerzburg.de.

Simon Eismann is currently working toward the Ph.D. degree in software engineering, University of Würzburg. His research topics include cloud computing, serverless, devops as well as performance modeling. Contact him at simon.eismann @uni-wuerzburg.de.

André Bauer is currently working toward the Ph.D. degree in software engineering, University of Würzburg. He serves as elected newsletter editor of the SPEC Research Group. His research topics include elasticity in cloud computing, auto-scaling and resource management, autonomic and self-aware computing, and forecasting. Contact him at andré.bauer@uni-wuerzburg.de.

Laurens Versluis is currently working toward the PhD degree at Vrije Universiteit Amsterdam, The Netherlands. He works on workflow scheduling in datacenters with functional and nonfunctional requirements. His research interests include distributed systems, privacy and security, gaming, and image processing. He is part of the AtLarge Research Group, where he is the tech lead of the MagnaData Project. Contact him at l.f.d.versluis@vu.nl.

Lucian Toader is currently toward the MSc degree at Vrije Universiteit Amsterdam, The Netherlands, where he studies modern distributed systems. He is part of the AtLarge Research Group, Vrije Universiteit and Delft University of Technology in The Netherlands. He received the MSc degree from University Politehnica of Bucharest in 2018. His work on massivizing computer systems led him to serverless computing. Contact him at lucian.toader93@gmail.com.

Norbert Schmitt is a doctoral researcher at the Chair of Software Engineering, University of Würzburg. His research interests include the energy efficiency of cloud and edge computing as well as interconnected IoT devices. Making the system aware of its energy consumption and allowing for autonomous decision making. Contact him at Norbert.schmitt@uni-wuerzburg.de

Nikolas Herbst is a research group leader at the Chair of software engineering, University of Würzburg. His research topics include elasticity in cloud computing, auto-scaling and resource management, performance evaluation of virtualized environments, autonomic and self-aware computing. He received the Ph.D. degree from the University of Würzburg in 2018 and serves as elected vice-chair of the SPEC Research Cloud Group. Contact him at nikolas.herbst@uni-wuerzburg.de

Cristina L. Abad is a professor at Escuela Superior Politecnica del Litoral, ESPOL, Guayaquil, Ecuador. She was a recipient of the Computer Science Excellence Fellowship and a Fulbright Scholarship from University of Illinois at Urbana-Champaign. From 2011 to 2014, she was a member of the Hadoop Core team at Yahoo, Inc. At ESPOL. She has been the recipient of two Google Faculty Research Awards. Her main research interests include the area of distributed systems. She received the M.S. and Ph.D. degrees from the Computer Science Department, University of Illinois at Urbana-Champaign. Contact her at cabad@fiec.espol.edu.ec.

Alexandru Iosup is a full tenured professor and the University Research Chair at Vrije Universiteit, Amsterdam, The Netherlands. He is the Chair of the SPEC RG Cloud Group. His work in distributed systems and ecosystems has received prestigious recognition, including the yearly Netherlands ICT Researcher of the Year in 2016, the yearly Netherlands Higher-Education Teacher of the Year in 2015, and the SPEC community award SPECtacular (last in 2017). He received the Ph.D. degree in computer science from TU Delft, The Netherlands. Contact him at a.Iosup@vu.nl.