Preprint of The Journal of Systems & Software (2022) 1–17

The Uphill Journey of FaaS in the Open-Source Community

Nafise Eskandani[a], Guido Salvaneschi[b]

[a]*Technische Universität Darmstadt, Germany*
[b]*University of St.Gallen, Switzerland*

Abstract

Since its introduction in 2014 by Amazon, the Function as a Service (FaaS) model of serverless computing has set the expectation to fulfill the promise of on-demand, pay-as-you-go, infrastructure-independent processing, originally formulated by cloud computing. Yet, serverless applications are fundamentally different than traditional service-oriented software in that they pose specific performance (e.g., cold start), design (e.g., stateless), and development challenges (e.g., debugging). A growing number of cloud solutions have been continuously attempting to address each of these challenges as a result of the increasing popularity of FaaS. Yet, the characteristics of this model have been poorly understood; therefore, the challenges are poorly tackled. In this paper, we assess the state of FaaS in open-source community with a study on almost 2K real-world serverless applications. Our results show a jeopardized ecosystem, where, despite the hype of serverless solutions in the last years, a number of challenges remain untackled, especially concerning component reuse, support for software development, and flexibility among different platforms – resulting in arguably slow adoption of the FaaS model. We believe that addressing the issues discussed in this paper may help researchers shaping the next generation of cloud computing models.

1. Introduction

Since the introduction of FaaS in 2014 by Amazon, all major cloud service providers, including Google, Microsoft, and IBM, have started supporting equivalent services. By now, this model is also available in a growing number of open-source platforms such as Apache OpenWhisk, OpenFaaS, and Kubeless. Gartner has predicted that half of the global enterprises will adopt serverless computing by 2025 [1].

The key idea is that programmers implement one or more functions that execute in isolation based on activating events such as HTTP requests, file uploads, or database triggers. The *provider* is then responsible for infrastructure management, enabling programmers to solely focus on the application logic rather than on resource provisioning. Along with securing scalability, this approach reduces costs because charges are based on the actual resources that an application consumes, spawning less functions if the load reduces. The FaaS approach, with fully automatic infrastructure management, and a truly pay-as-you-go, up and down scaling mechanism, appeared to finally fulfill the original expectations around cloud computing's transparent, on-demand provisioning [2].

A closer look indicates a more articulated reality. Despite the name suggesting that serverless functions are indeed *just functions* executed in the cloud, serverless applications have more diverse characteristics and requirements than the applications developed using traditional cloud models, leading to subtle issues such as unpredictable performance and cost [3, 4, 5], lack of tool support for testing and debugging [6, 7], and vendor lock-in [8]. These issues encouraged researchers and providers to propose a variety of solutions to address the challenges introduced by FaaS [9, 10, 11]. Yet, a recent study on serverless application development [12] reports that programmers still struggle to adopt the FaaS model due to its immature ecosystem – lacking support for common application implementation tasks such as

package integration and function invocation as well as low-level development such as provider, function, and resource configurations.

The growth of attention to FaaS and the challenges mentioned above demand for a reality check on the adoption of this model and a better understanding of the FaaS ecosystem. In this study, we inspect a snapshot [13] of about 2K open-source real-world serverless applications to investigate the main technological and architectural characteristics of these applications, and answer the following research questions regarding open-source FaaS software: RQ1: How has the popularity of FaaS been growing since its appearance? RQ2: How are applications that adopt the FaaS model implemented in practice? RQ3: How are the programmers mitigating the current limitations of FaaS? RQ4: What are the use cases currently targeted by FaaS?

Our work indicates that despite the increasing offer of FaaS services by cloud providers, this model is still struggling with fundamental issues. Among other results, we have found that (RQ1) despite the growing popularity of FaaS in recent years, few already mature projects adopt this model, due to several technological and design challenges concerning migration. Besides, component reuse across projects is alarmingly modest, depicting an ecosystem (community-driven repositories of reusable components, open marketplaces) that is still in its infancy. (RQ2) The flexibility in software development that is possible *in principle* – multi-language application, just to mention one aspect – is not exploited in practice – Node.js and Python cover 92% of serverless software. This is likely due to a mismatch between the technology commonly used in microservices (e.g., JVM, with high start up time) and the FaaS execution model (cold start for every function). (RQ3) Support for the development process is severely lacking, with developers systematically adopting proprietary plugins and external services for local integration testing, security management, and function orchestration. (RQ4) The stateless nature of FaaS, the limited amount of dedicated memory, and short-lived functions limit the use cases of FaaS to mostly non-memory-intensive web services and data processing. We conjecture that the issues above constitute a significant barrier to entry for the adoption of the Faas model, explain its limited success, and constitute a burning priority for the research community in the next years.

As of October 2022, GitHub, with more than 101 M[1] programmers and 42 M[2] public repositories, is the largest collection of open-source projects. We believe that a dataset that takes a snapshot of all available projects of a particular class in GitHub is representative of how a technology is being used in practice. In this study, we focus on open-source applications. While a generalization to close-source projects would require further investigation, we believe that the results presented here can guide future research that aims to better understand the the entire FaaS ecosystem. For example, a previous study [14] projects from GitHub indicates that the *open-source* nature of the application does not impact the size of the project and the team size. As we consider such features—among others—in our investigation of serverless applications (Section 4.1) we expect that the results provide a valuable indication for close-source projects, too.

To the best of our knowledge, this is the first study on FaaS targeting the source code of serverless applications and the largest study on real-world serverless applications to date. We provide implications of our results for researchers, programmers, and serverless providers. The software and the analysis developed for this study are publicly available[3].

2. Background

This section introduces FaaS along with an example of a serverless application in the FaaS model.

2.1. FaaS in a nutshell

Cloud computing refers to the combination of hardware and system software in the data centers and the applications that they offer as a service [15]. Depending on the abstraction model, software as a service, platform as a service, or infrastructure as a service can be offered to clients [16]. With the growth of cloud computing, an increasing number of programmers migrated from physical machines to virtual ones to scale their applications in response to the changing requirements. Even though the virtual resources drastically minimize infrastructure maintenance and the pay-per-use model significantly reduces the operational costs, infrastructure provisioning is still required to manage the virtual machines [17].

[1] `https://github.com/search`

[2] `https://github.com/search?q=is:public`

[3] Provided as supplementary materials for review

A recent newcomer in the line of cloud computing models is the FaaS model for serverless computing [18]. The term serverless computing was initially introduced as a processing or networking model that does not use an actual server to operate, such as peer-to-peer software or client-side only solutions [19, 20]. With the rise of cloud computing, the term serverless shifted to abstracting server management away from programmers, such as the case of software as a service [21]. In recent years, FaaS has rapidly become one of the most popular models of serverless computing – in a way that these two terms are often used interchangeably.

The FaaS model of serverless computing refers to server-hidden solutions that employ an event-driven paradigm to trigger stateless functions [18]. A serverless application includes services composed of functions that are managed by providers and triggered by events. The services are units of configuration where function properties – such as their triggering events, resources and security concerns – are defined. Event triggering for the functions can be due to a scheduler, to an HTTP request from an Application Programming Interface (API) gateway, or to various event sources such as database triggers or message queues – depending on the events supported by the provider.

Leaving the management of functions into the hands of the provider enables autoscaling that not only eases the development and maintenance of the serverless applications but also reduces costs by scaling idle instances to zero when they are not required. Yet, autoscaling – likely most important characteristic of FaaS – can result in a dramatic reduction of the performance and utility of serverless applications [4]. Autoscaling requires applications to be stateless. Therefore, in a serverless application, the state of a function is not preserved through multiple executions. There is also a possibility that different functions of an application do not execute on the same machine. As a result, applications require to use external shared storage systems such as Amazon Simple Storage Service (S3) for exchanging intermediate data [22]. Obviously, communicating through an external storage system is slower and costlier than point-to-point message exchange.

Another consequence of autoscaling is that in an application with several functions, most dependencies are only available at runtime. As a result, *local* integration testing and debugging can be impossible in particular cases [8]. Even though a single function can be unit tested locally before deployment, there is still a lack of tool support for testing, debugging and continuous integration of serverless applications with more than a single function [6, 7]. Other characteristics of serverless applications, including delays caused by the time needed for the container starts (cold start latency) [5], complex triggering processes [3], and limited lifetime of functions [4] can also significantly impact the performance of serverless applications, making it harder to predict than traditional cloud solutions.

2.2. Example of a Serverless application

The overall procedure of developing a serverless application from scratch – with minor variations in implementation details for each serverless platform – consists of three main steps: (1) Configuration of the environment and resources related to FaaS using the cloud console or a definition file written in a data serialization language such as YAML, (2) Implementation of functions with a focus on the application logic, and (3) Deployment of the application to a platform [12].

```yaml
service: simple-clock

provider:
    name: aws
    runtime: python3.7
    region: us-east-1
    timeout: 3

functions:
    currentTime:
        handler: handler.getTime
        events:
            - http:
                path: ping
                method: get
```

Listing 1: Configuration of a serverless service

We present an example of a clock serverless application deployed using the Serverless Framework[4], the open-source tool used by the applications in our dataset. This application includes a single service and a single function. The configuration file of the *simple-clock* service is in Listing 1. The list of available properties for a service mainly

[4] https://www.serverless.com

depends on the features supported by the provider. These properties can range from the details of function deployment to the access management and specification of resource requirements.

The *provider* property (lines 3 – 7) specifies the cloud provider that the Serverless Framework uses to deploy the functions. The function will be deployed to Amazon Web Services (AWS) instances in the *US-east* region with a Python runtime and maximum of 3 s execution time. The AWS credentials are required so that the Serverless Framework can have access to creating and managing resources. The *functions* property (lines 9 – 15) includes declarations such as the name of the functions, their handlers, runtimes, and triggering events. This service contains only a function named *currentTime*. The handler for this function is the *getTime* method triggered by a HTTP request. The handler for the *currentTime* function is in Listing 2. The *getTime* method (in Python) simply returns a JSON object with two variables, a status code and a message with the current time.

```
1  import json
2  import datetime
3
4  def getTime(event, context):
5      clock = datetime.datetime.now().time()
6      body = {
7          "message": "Current time is :" + str(clock)
8      }
9
10     response = {
11         "statusCode": 200,
12         "body": json.dumps(body)
13     }
14
15     return response
```

Listing 2: A serverless function handler

Although the name of the function is *currentTime*, the HTTP API endpoint is exposed as *ping*; Therefore, the endpoint can be directly invoked with a HTTP request using a tool such as *curl*. The URL needed for the HTTP request is assigned and sent back to the programmer by the provider as the result of the deployment.

3. Study Design

In this section, we describe our data source in detail and we present the methodology we follow to answer them.

3.1. Data source description

This study is focused on open-source projects. We define "open-source" projects simply as the projects that their source code is publicly available, without considering the dependencies on third-party resources or issues related to licensing management. We used an existing dataset [13] of 1,877 real-world serverless applications extracted from GitHub. The dataset consists of serverless applications with at least a serverless function, a specific provider, and active contributors for a project lifetime of more than a year. The providers for these applications are AWS, Microsoft Azure, Google Cloud Platform, Apache OpenWhisk, Cloudflare, and Others, and the applications are developed using different programming languages such as Python, Java, Ruby, and JavaScript.

The fundamental difficulty of creating such a dataset is distinguishing serverless applications from others. One can search for serverless applications using keyword search. Although, the presence of a keyword does not guarantee that an application is serverless. Analogously, an application may be serverless without containing a specific keyword. Alternatively, one can search for serverless applications supported by each provider based on each provider's specific serverless configuration file. Although, the serverless configuration files are hardly distinguishable from other cloud setups. For example, one of the default serverless configuration files in AWS is index.js. In response to creating function event in Lambda console, AWS creates this file for Node.js serverless applications to generate a handler function [5]. In October 2022, there were more than 306 M index.js files on GitHub, most unrelated to Serverless applications. To mitigate these issues, the authors of [13] extracted GitHub projects developed using Serverless Framework and filtered them against the empty, toy, immature, and inactive projects. Serverless Framework requires applications to have a configuration file exclusively for serverless applications – with the same default name across different providers and platforms – making the search for these applications feasible.

[5] `https://docs.aws.amazon.com/lambda/latest/dg/lambda-nodejs.html`

The challenges of identifying serverless applications and accessing their source code led us to limit the scope of this study to open-source serverless applications deployed using Serverless Framework. Therefore, self-hosted solutions that do not require any automation deployment tool are excluded from our dataset. For the rest of this paper, when we mention serverless applications, we point to the projects publicly published in GitHub and automatically deployed using Serverless Framework.

We believe that the dataset is representative of the domain for two reasons. First, according to a study on serverless frameworks [23], Serverless Framework, with the highest number of supported providers like AWS, Microsoft Azure, or a Kubernetes-based solution like Kubeless, and with the security, deployment, testing, and monitoring offerings, is the most extensive framework. Second, Serverless Framework, with more than 43 K GitHub stars and 15 M downloads, is the most popular open-source serverless deployment automation tool to date (reportedly 90% [24]). Therefore, the applications developed using this framework are de-facto representative of the serverless applications supported by each provider. In addition, we argue that using a particular framework for deployment does not affect the implementation of these applications since the configurations of a serverless application depend on the adopted provider – not the adopted framework. Serverless Framework is only an automation tool that facilitates easier deployment of serverless applications across multiple vendors by providing a unique configuration file. This is while the configuration of the application is dependent on the properties that each vendor supports.

3.2. Code repository analysis

The dataset includes the URLs of the repositories, the cloned version of the repositories from GitHub, and their git commit history. To discuss RQ1, we used the git commit history to access the dates that different files were added to a project and the number of contributors to each project. We also extracted the meta-data of each project using the *mercy-preview* media type provided by GitHub API to know about the number of GitHub stars and licenses for each repository.

We used Python scripts to automatically extract the related information for answering RQ2 from the serverless configuration files. The configuration file of a serverless application includes deployment parameters such as the name of the provider and the programming language of the serverless services, and the function details such as the triggering events. We used *cloc*[6] to know about the files in the repositories and the lines of code based on each programming language.

To answer RQ3, we used Python scripts to extract the external resources and plugins from the configuration files. To get the runtime dependencies of Python projects, we used *pipreq*[7] that generates the pip *requirements.txt* file based on the imports of a project. We used *npm-ls*[8] to get the list of installed packages in a project for Node.js dependencies.

For answering RQ4, we used the meta-data of each repository that includes the topics and description of each project. We manually checked the *readme* files and Github repository descriptions of each repository along with a superficial inspection of the source codes to know about the potentials and purposes of serverless components in every project.

4. Results and Implications

In this section, we present the main findings of our study for each of the RQ1–RQ4 and we discuss their implications for researchers, programmers, and serverless providers.

4.1. Popularity of FaaS

Schleier-Smith et al. [2] estimated that serverless computing would evolve to dominate cloud computing. To verify this hypothesis in the open-source community, we consider the adoption rate of FaaS, the migration rate to FaaS, contributors to each project to estimate the popularity of FaaS among industrial projects, and stars and licenses to investigate the popularity of FaaS as a reused-based ecosystem.

Figure 1 shows the creation date of each repository and the release date of the major serverless providers. Since all the applications in our dataset are developed using the Serverless Framework, we consider the release date of this

[6] `https://github.com/AlDanial/cloc`
[7] `https://github.com/bndr/pipreqs`
[8] `https://docs.npmjs.com/cli/v7/commands/npm-ls`

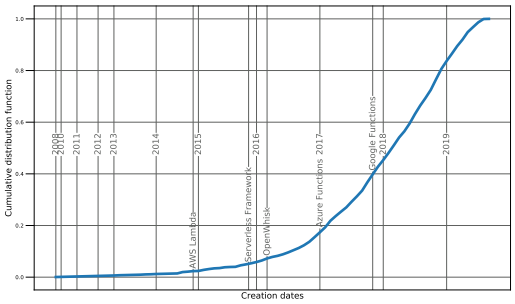

Figure 1: Creation dates of repositories

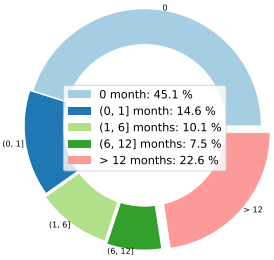

Figure 2: Get serverless.

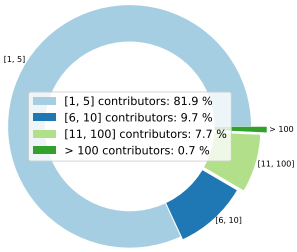

Figure 3: Contributors

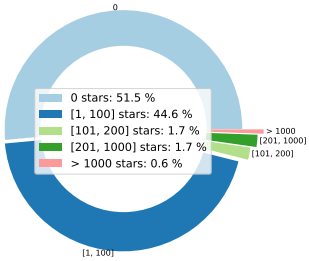

Figure 4: GitHub stars

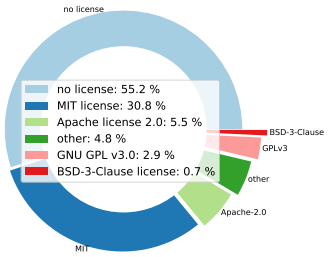

Figure 5: Licenses

framework, October 2015, as the starting point for our analysis. More than 50% of serverless applications are created since 2018, indicating the rapid growth of this model in recent years. Instead, less than 15% of the repositories were created before October 2015. Interestingly, such projects have started as non-serverless projects and adopted the FaaS model later on, after the introduction of the Serverless Framework. We extracted the time required for each project to add the first serverless feature after the first commit to find the projects that migrated to or adopted the FaaS model sometime after creation (Figure 2). Almost 45% of the projects are born as serverless applications. Although nearly 40% of the projects migrated to or adopted the model more than one month after creation, the number of already mature projects that adopted this model is modest, despite the claims about potential improvement in cost and maintenance/operation effort. The low number of migrations is most likely due to the multiple design and technological challenges even when migrating simple use cases to a FaaS model [12, 25].

To evaluate the team size of the projects developing serverless applications, we extracted the number of contributors to each project (Figure 3). About 82% of the projects are developed within small groups of less than six people, and only less than 1% of the projects belong to large groups – more than a hundred programmers. Inevitably, these numbers are affected by the fact that not many companies publish their projects on GitHub. A previous study [26] shows that the average team size for developing software in industry-type projects is 8.16 members – almost three members more than the average team size of more than 91% of the projects in our dataset, which is 5.5 members. Compared to individual programmers and small groups, large organizations are even more concerned about migration challenges due to the high cost, long duration, and involved organizational restructurings [27]. These challenges can be a major factor for the relatively lower interest of teams with larger size to this model.

To accelerate the adoption of FaaS, researchers may further investigate the migration barriers to characterize the architectural changes required for converting a monolithic or microservice-based system to a serverless application. In addition, serverless applications typically contain multiple components that require deployment into distinct environments, testing in combination with each other, and clear identification when an error arises. Serverless providers can help with the adoption and migration process by automating the continuous integration and continuous deployment pipelines, supporting integration testing and debugging tools, and improving monitoring services [28, 6, 29].

We investigated GitHub stars to estimate the maturity of serverless applications (Figure 4). Almost half of the serverless applications do not have any GitHub star, and only less than 5% of the projects have more than a hundred stars. A former study [30] reveals that three out of four programmers consider the number of GitHub stars a criterion

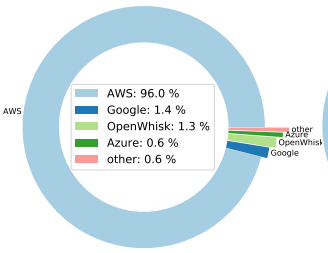

Figure 6: Providers

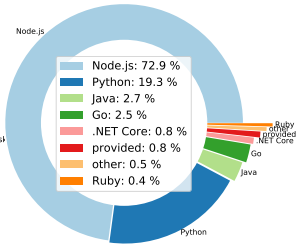

Figure 7: Runtimes

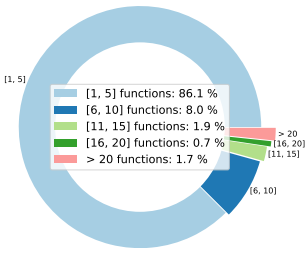

Figure 8: #Functions

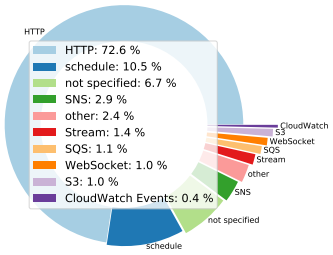

Figure 9: Triggers

for deciding to reuse a GitHub project or to contribute to it. This result, together with the low number of GitHub stars for the serverless projects, indicates the low potential for code reuse in the current state of the serverless ecosystem.

From a legal perspective [31], the open-source community depends on licensing for redistribution, integration, and reuse of existing code. To know more about the potential for component reuse enabled by project licensing in serverless ecosystem, we extracted the licenses used by each project (Figure 5). More than half of the projects come with no license, meaning that all rights are reserved, and the projects are not open-source or available for reuse [32]. Yet, 37% of the projects are under the MIT, Apache-2.0, and BSD-3-Clause licenses, three of the most permissive ones [32].

Although FaaS has been introduced as a general-purpose abstraction [33], the low potential and legal possibilities of component reuse in this ecosystem is a surprising finding that systematically leads to application-specific solutions. This characteristic indicates that the state of FaaS is still a long way from defining a reuse-based ecosystem in the style of, e.g., Maven Java packages. The efforts to address these issues are in their early stages: the AWS Serverless Application Repository (SAR)[9] has been recently proposed as a managed repository to store and share reusable serverless components. In addition to contributing the best practices to develop serverless applications, we believe the attempt to share these components goes in the right direction by motivating programmers to develop applications with use cases that will significantly contribute to the growth of the serverless community. Yet, we argue that the ultimate solution for this issue should be community-driven repositories managed by independent parties.

4.2. Implementation of serverless applications

Designing a serverless application is challenging due to the several technological and architectural choices such as the serverless provider, the programming language, the workflow and granularity of functions, and the selection of triggering events [12]. We study the most popular approaches to develop a serverless application and investigate the factors affecting the decision-makings.

Starting from the most popular technologies (Figure 6), AWS, which provides the FaaS offerings of 96% of the applications in our dataset, is by far the most popular serverless provider. The predominance of AWS is also reported in a previous study [34]. We believe that this dominance is influenced by the maturity of the platform and by the variety of the supported services [35]. Other popular providers that, however, see much lower adoption are the Google Cloud Platform followed by Apache OpenWhisk – an open-source platform that has been a pioneer of FaaS.

According to our results 4.1, the cases of migration of already mature projects to the FaaS model are quite limited. One of the main issues in this context, is the vendor lock-in, introduced by the adoption of serverless providers, that affects the applications in various dimensions such as portability, implementation requirements, and architectural designs [25]. Our data clearly indicate that AWS remains the technological leader of the FaaS ecosystem. We argue that the dominance of a single technology worsens the lock-in problems of by limiting the programmers to specific services and offerings. In addition, the risk and uncertainty of depending on a third party as a serverless platform are high, particularly in industrial projects. [36]. These problems can be tackled by adopting comprehensive open-source serverless platforms that enable programmers to have more flexibility in their decisions. Although a few mature open-source platforms such as OpenFaaS with more than 22 K GitHub stars and Apache OpenWhisk with more than 5 K GitHub stars exist, they either introduce infrastructure provisioning challenges to the system due to the self-hosted

[9] `https://aws.amazon.com/serverless/serverlessrepo/`

nature of these platforms, or they force the usage of a cluster manager such as Kubernetes, that introduces yet another platform to the project and adds up to the cognitive overhead [10].

Serverless platforms support several programming languages by providing *runtimes*. Each runtime supports a single version of a programming language, multiple versions, or even multiple programming languages. Figure 7 shows the distribution of language runtimes for the serverless services. Node.js and Python, with more than 92% of the services, are the most popular, followed by Java (2.7%). Some cloud providers, such as AWS, allow programmers to use custom runtimes. The *provided* runtime in less than 1% of the services refers to this case. The significant difference among the values can be explained through the results of previous studies [37, 38] that show the choice of the language ultimately affects cold start latency; For example, in AWS, JavaScript and Python have much lower start latencies than other programming languages such as Java. This issue has been tackled by recent approaches [39, 40, 41] to mitigate the cold start problem – these soluations may eventually lead to a higher adoption of the other programming languages. Besides, there is a considerable difference between the numbers of packages related to serverless computing provided by different language runtimes. As of July 2021, there are 4,473 JavaScript packages on Node Package Manager (npm)[11] tagged with "serverless" that ease the development of serverless applications, from enabling programmers to run AWS functions locally[12] to solutions for speeding up cold starts[13]. With the same tag, we found 546 packages on Python Package Index (PyPi)[14] and 134 artifacts on Maven repositories[15], explaining the predominance of Node.js among popular language runtimes.

Serverless applications can provide independent services, and each of these services can include several functions. Programmers can implement every single service and function using the language from the range of the supported runtimes that better targets the purposes of the application as well as the required throughput [42]. Yet, about 10% of the applications in our dataset provide more than a single service, and more than 85% of these multi-services applications are developed using the same language for all services. Similarly, only 1.5% of the serverless services in the applications contain runtime separate specifications for each function – the remaining are single-runtime services.

Even though FaaS theoretically enables individual programmers to contribute to collaborative projects with the programming language in which they are most productive, most serverless applications in our dataset are developed using a single language runtime. This result might be affected by the fact that we considered each application in our dataset as a monorepo. Therefore, we cannot generalize the result to the applications that span over several repositories. Even though the team expertise and required integration effort have significant effect on the choice of programming language, we still believe that programmers will have more flexibility to choose the programming language once they do not have to worry about cold-start latency. Various techniques already exist to speed up function start-up. Researchers can help this process with further studies on reducing start-up latency, e.g., to reuse existing execution environments for running several functions or to pre-allocate and pre-warm such environments [41, 39, 43]. Serverless providers can help by improving the start-up time of the less popular (in FaaS) language runtimes, and programmers can accelerate this process by improving the support for the FaaS model in different language runtimes via developing serverless related packages and adding them to the package manager of each runtime.

Project size heavily influences the effort to develop software, as well as maintainability after development [44]. Despite the limitations, Lines of Code (LoC) is still a widely used metric for project size [45][16]. The LoC metric typically counts every physical source line of code in a program, excluding blank lines and comments. A major limitation of LoC as a metric is the case of different programming languages. To mitigate this issue, we selected the serverless applications developed using the four most popular language runtimes (Node.js, Python, Java, and Go) in the FaaS ecosystem to investigate their sizes individually. We considered the language runtimes specified in the serverless services as the runtime of the application. For applications with multiple services in different languages, we considered each service an individual serverless application.

Figure 10(a) shows the cumulative distribution of LoC per serverless application based on language runtime. More

[10] `https://winder.ai/a-comparison-of-serverless-frameworks-for-kubernetes-openfaas-openwhisk-fission-kubeless-and-more/`

[11] `https://www.npmjs.com/search?q=serverless`

[12] `https://www.npmjs.com/package/lambda-local`

[13] `https://www.npmjs.com/package/serverless-plugin-warmup`

[14] `https://pypi.org/search/?q=serverless`

[15] `https://mvnrepository.com/search?q=serverless`

[16] *"The LoC measure is a terrible way to measure software size, except that all the other ways to measure size are worse [44]."*

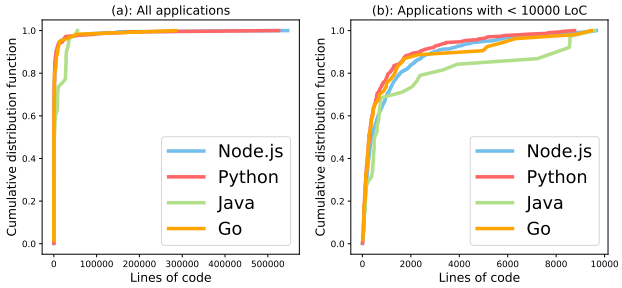

Figure 10: Applications' sizes

than 95% of the applications developed using Node.js, Python, and Go consist of less than 100K LoC. For Java, this is the case in all applications in the dataset. Based on the project size classification reported by Aguilar et al. [46], these numbers indicate that serverless applications typically have a medium to small size; consequently, they are expected to require less than 60 months and fewer than 10 team members for development. Figure 10(b) considers applications with less than 10K LoC. More than 90% of these applications developed using Node.js, Python and Go have less than 3K LoC, while this number is 70% for Java applictions. A former study [47] concluded that more LoC are required for a Java function compared to JavaScript or Python. Our results confirm a similar pattern in the FaaS ecosystem. Often, more LoC are required for a serverless function in Java compared to Python and Node.js – possibly another factor that leads programmers to choose these language runtimes in the serverless configuration.

Figure 8 shows the number of functions per application. More than 94% contain ten functions or less, a value consistent with previous findings [8, 34]. We also investigated the number of functions per service of a serverless application. About 58% of these services define a single function. More than 99% contain twenty functions or less.

We believe that the low number of functions per service and per application stems from the difficulties programmers encounter when designing the workflow of serverless software [48, 49]. The workflow specifies the order in which a pool of functions should be executed (chaining, branching, or parallel), and it can be defined via functions that orchestrate other functions or via external schedulers [50]. In addition to challenges like failure handling and process monitoring, designing the workflow for complex applications where functions require access to various parts of a shared state can be particularly cumbersome due to the stateless nature of FaaS [48]. Therefore, programmers are forced to introduce orchestration services to coordinate serverless components [50, 48]. Typically, each serverless provider has its own serverless orchestration service (e.g., Step Functions for AWS) with a specific programming model (e.g., Amazon States Language). In addition to further exacerbating lock-in, these platform-dependent coordination services add the cognitive overhead of learning new technologies to the adoption and migration process. These issues suggest the necessity of further research in the direction of an open-source, platform-independent programming model and orchestration service that can be used across platforms.

The correct granularity of serverless functions is still disputed. The current views cover a wide range from considering single methods as a serverless function to wrapping a full microservice into a serverless function [51, 34]. Following the results from a previous study based on programmers interviews [8], we also observed two common patterns for implementing functions based on their triggering events (Figure 9). The most common case is that functions are fine-grained as one function per API endpoint such as HTTP and WebSocket. Alternatively, there is one function per external resource such as Amazon Simple Queue Service (SQS). Figure 11 demonstrates the methods used by the HTTP API endpoints. The results show that more than 97% of these endpoints are as fine-grained as individual HTTP requests – about half of them being the *get* request.

We selected the serverless applications developed using Node.js and Python with less than 1K LoC to study the number of functions in each and investigate patterns in the size of each function (Figure 15). While there are several applications with less than 200 LoC and more than four functions, most have more than 800 LoC and only a single function. Although a previous investigation [47] reports that the average LoC for a function in JavaScript is 47, serverless programmers do not seem to stick to a specific size spectrum – our samples cover a broad range from a few LoC to hundreds. Yet, we should emphasize that this result is affected by the programming style and the usage of dependencies in each application, which are not considered in this study.

In FaaS, the infrastructure is hidden from the programmers. Yet, programmers can specify the maximum memory

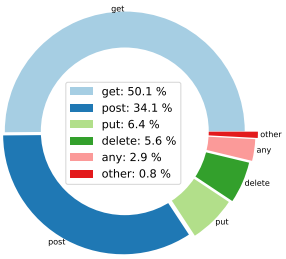

Figure 11: Methods

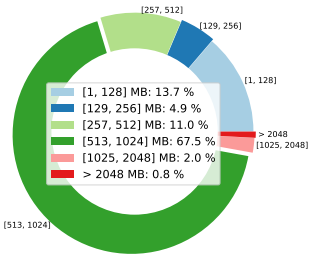

Figure 12: Memory sizes

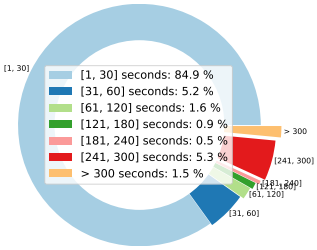

Figure 13: Timeouts

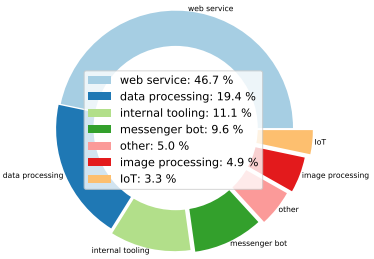

Figure 14: Use cases

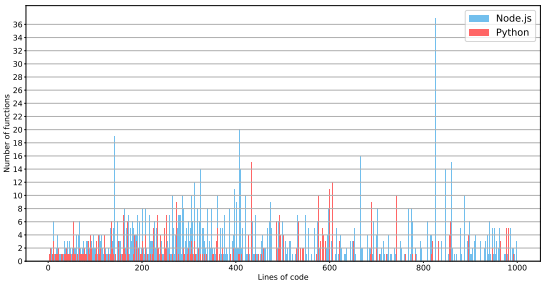

Figure 15: Applications' sizes based on #functions

size required to deploy their functions within a specific range supported by the provider as well as the maximum time function is allowed to run before getting stopped (timeout). The distribution of memory sizes and timeouts required by the functions are in Figure 12 and Figure 13. We considered the default size proposed by the provider for the functions that do not specify a memory size or timeout. For example, the default memory size and timeout for functions are 1024 MB and 6 s in AWS, 256 MB and 60 s for Google Cloud Platform, and 512 MB and 60 s for OpenWhisk. Roughly 97% of the functions require 1 GB of memory or less for deployment. 1 GB of memory is enough for simple web servers such as those with 1K or less visits per day and a couple of websites, or simple word processing tasks. The limited memory required by the majority of functions suggests that serverless functions are typically expected to perform non-memory-intensive operations. Yet, serverless applications can still perform memory-intensive operations by parallel execution of lightweight functions. Similarly, the maximum execution time required by about 85% of the functions is 30 s, with more than 65% requiring 6 s or less, showing that serverless functions are typically used for relatively short-running tasks.

The limited amount of time allowed for executing serverless functions and the relatively small amount of memory for deploying them raise the need for decomposing functions and using external storage resources in data-intensive and computation-intensive scenarios [12]. Besides increasing the complexity of the application, function decomposition leads to a higher number of functions that complicates workflow management. Also, external storage negatively affects the cost and performance of the application, forcing programmers to avoid the FaaS model when cost and performance are hard constraints. Although the bigger memory size increases the cold start latency [7] and the longer timeout makes scaling and cost management more difficult [48], serverless providers can still consider loosing the restrictions on execution time and memory size by improving start-up time [41, 39, 43] and using fine monitoring of servers to increase the applicability of FaaS, ultimately broadening its adoption. Along with eliminating the restrictions on the upper bounds, Google Cloud Platform and Apache OpenWhisk can reduce their default timeouts to a lower number due to the high demands for short-lived functions (6s).

4.3. Mitigating the limitations

We observed that one common solution in our dataset to deal with the challenges of FaaS such as state management, security, and vendor lock-in is the use of external resources and plugins. Despite the popular belief that serverless applications are mostly used in conjunction with backend as a service resources [52, 8], several studies reveal that this is not the case in practice, most likely due to the latency introduced by an external service [53, 54]. Only

Table 1: Popular external resources

Resource	Use case	#Applications	Occurrence%
DynamoDB	database	311	16.6%
S3	storage	168	9.0%
IAM	authorization	84	4.5%
API-Gateway	event source	72	3.8%
SNS	event source	53	2.8%
SQS	event source	46	2.5%

Table 2: Popular serverless plugins

Plugin	#Applications	Occurrence%
serverless-offline	502	26.7%
serverless-webpack	320	17.0%
serverless-python-requirements	204	10.9%
serverless-domain-manager	131	7.0%
serverless-dynamodb-local	94	5.0%
serverless-dotenv-plugin	86	4.6%

28% of the applications in our dataset define bindings to a backend as a service resource. Table 1 shows the most popular external resources used in the dataset. These values show that programmers handle the stateless nature of serverless applications by sharing state via Amazon DynamoDB for relational data and Amazon S3 for Binary Large OBjects (BLOB). Less than 10% of the applications use Amazon API-gateway, SNS, or SQS to manage communication between components and function invocations. Although eliminating the need for infrastructure management shifts the burden of handling security updates for operating systems and language runtimes from programmers to serverless providers, the necessity of using external storage systems for sharing state and the potential to receive function invocations from various types of event sources still expose serverless applications to security risks [2, 53]. Even though basic Amazon Identity and Access Management (IAM) utilities is by default included in AWS supported applications to manage resources, specific add-ons need to be added to the configuration file to enable the custom management of the roles and policies for each function. These resources are used in less than 5% of the applications to create custom control for accessing AWS services, resources and API endpoints. While these external resources help to tackle state management, communication, and security issues, they also introduce a significant cognitive overhead, performance latency (discussed in Section 2.1), and cost that can be among the reasons for the low adoption of such services.

Plugin software architectures are often used to compensate missing features in the underlying framework [55]. More than 62% of the applications in our dataset use at least one plugin as a *local* solution developed by the open-source community for extending the functionalities of the framework (Serverless Framework in our dataset). Table 2 presents the distribution of widely-used plugins. The most popular is *serverless-offline* – more than 26% of the occurrences – that enables programmers to develop a serverless application locally using the AWS FaaS emulation and API-Gateway. Programmers often use this plugin to speed up the development cycle, mitigating the limitations of the FaaS model, such as lack of tooling for integration testing and debugging. The *serverless-webpack* is another popular plugin that bundles the serverless functions with Webpack[17], an open-source static module bundler and dependency manager for Node.js applications. When the plugin is included in a serverless service, it reduces the size of the function packages by bundling the code with the dependencies needed to run each individual function, resulting in faster deployments that use less memory. The *serverless-python-requirements* plugin does a similar task for the

[17] `https://webpack.github.io`

Table 3: Popular runtime dependencies

Runtime	Dependency	Use case	Occurrence%
Node.js	aws-sdk	AWS toolkit	17.1%
Python	boto3	AWS toolkit	13.7%
Node.js	lodash	utility for FP	9.7%
Node.js	express	web framework	8.4%
Node.js	requests	HTTP library	7.7%
Python	requests	HTTP library	6.7%
Python	Flask	web framework	2.9%

Python applications. The popularity of these plugins suggests the lack of tooling for managing the dependencies of serverless services with multiple functions. A total 7% applications use the *serverless-domain-manager* plugin to manage custom domains with API gateways and reduce the challenges caused by the change of hostnames in case of redeployment. The *serverless-dynamodb-local* is used by 5% of the applications to run Amazon DynamoDB locally, lower latency, as well as help with integration testing and debugging. To reduce the configuration effort, more than 4% of the applications use *serverless-dotenv-plugin*, a plugin that preloads environment variables into the serverless configuration file.

While programmers commonly attempt to mitigate the FaaS challenges by means of external resources and plug-ins, these approaches are platform-dependent, and they inevitably introduce their own lock-in, cost, and performance challenges. These issues demand further research about generic and open solutions such as stateful functions [11] for handling shared states, defining coverage criteria [56] in integration testing, for the deployment of within secured units such as Intel's SGX [57, 58] to run data-sensitive risks, and for automating dependency management [59] of dependencies.

In FaaS, programmers can define packages for frequently used functionalities. Serverless applications can then list such packages as runtime dependencies. Table 3 shows the most popular dependencies of Node.js and Python in serverless applications. In more than 30%, developers use the *aws-sdk* or *boto3* packages from the AWS Software Development Kit (SDK) to programmatically create, configure, and manage the AWS backend as a service resources. AWS already includes *boto3* by default in Python runtimes. The popularity of *aws-sdk* suggests that AWS can also include it by default in their supported language runtimes to reduce the size of runtime dependencies and improve start-up latency during the deployment of serverless applications. Almost 10% of the applications use *lodash* that enables programmers to develop more concise and maintainable applications in the functional paradigm. This number confirms the results from a previous study [52] that functional programming is a relevant programming model for serverless functions. More than 11% of the applications use *express* or *Flask* to build web applications and API endpoints, and more than 14% use *requests* for HTTP requests. These results indicate that web services and development of the API endpoints are among the most popular use cases of serverless applications. We discuss these use cases in more detail in the following section.

4.4. Use cases of FaaS

Since FaaS is relatively new, we still lack a complete overview of its application targets. In particular, it is unclear whether the model can be adopted in conjunction with other models that require explicit resource provisioning, whether it is widely applicable, like other cloud models, or it serves a narrower scope, and whether it is adopted in the presentation layer (frontend) or in the data access layer (backend).

We randomly selected 1K applications in which we manually investigated the role of the serverless services to gain a better understanding of the areas serverless applications can serve. In roughly 64% of the applications, all components follow the FaaS model. Yet, in about 35%, serverless services are developed in conjunction with the services that use traditional cloud-based or server-centric infrastructure. In more than 50% of the applications, the serverless part concerns the backend services, and there are only about 5% of the applications in which the FaaS model is adopted only for the frontends. Cold start latency hinders the use of serverless functions for interactive user

interfaces and can be one of the factors affecting the low number of frontend adoption. Another factor can be the restricted execution time for running large dynamic frontend applications. Yet, serving static web pages potentially can be a use case for serverless applications which was not observed as a popular choice in our dataset.

We classified serverless applications into seven categories, and we assigned each of the applications to only one class of: web service, messenger bot, data processing, image processing, internal tooling, Internet of Things (IoT), or *others*. We considered the applications related to developing a website for the Internet or developing an API for a web server or a web browser as a *web service*. The *messenger bot* category refers to the applications developed as a bot for the services such as Slack and Telegram. We counted the applications providing an API for a database or translating the format of data to another format as a *data processing* application. The applications that perform a task such as image resizing are in the *image processing* category. The *internal tooling* category refers to applications for personal or organizational purposes such as reminders. The applications related to smart homes are in the *IoT* category. The *other* category covers the rest.

The use cases of FaaS are in Figure 14. Interestingly, roughly half of the applications are web services with a serverless service as their backend. The resource requirements of web services vary a lot in different situations. For example, in a user-based web service such as a commercial website, the number of user requests can significantly increase because of a newly released product, or it can majorly decrease at specific times of a day. Autoscaling in FaaS helps with handling peak traffic and reducing costs by optimizing resource usage when the number of requests is low. Also, FaaS can reduce the latency and improve the performance of a web service by deploying the services geographically close to the region of users. Even though serverless functions similar to traditional cloud offerings are still geographically localized and they need to be deployed in specific regions – typically with different pricing – they can still support optimal redundancy and replication by specifying different deployment models. The other popular use case of FaaS, with an adoption rate of almost 20%, is data processing. FaaS features both autoscaling and event-based activation, that makes it a good fit for processing and analyzing real-time streaming data.

Our results indicate that adopting serverless computing does not have to be a binary decision. Depending on the workload and use case, the FaaS model can be integrated with or connected to traditional architectures to complement them. Serverless functions are ideal for short-running and non-memory-intensive tasks. The combination of traditional models and FaaS can be particularly beneficial when programmers choose resource provisioning and potentially higher cost over the burden of coordinating several serverless functions created from decomposing long-running operations. Researchers can support serverless programmers to make such decisions with further studies that improve the predictability of serverless applications' performance and cost [60].

We believe that the most common use cases of serverless applications are another significant factor, in addition to the ones mentioned in Section 4.2, affecting the popularity of Node.js and Python. According to our results, web services and data processing are two of the most popular use cases of serverless applications. It is clear that Nodej.js is a full-stack technology of choice for web services development, and Python is among the preferred languages for data science [61, 62]. We argue that improvements of serverless related supported packages by the Python language runtime can increase the use cases of FaaS for data processing.

5. Threats to Validity

The validity of this study concerns two different aspects: the validity of the data source and the validity of the analysis.

Despite being extensive in size and variety, the data source used for this study has two main limitations [13]. First, the dataset consists of projects developed using the Serverless Framework. We discussed in Section 3.1 that, based on GitHub stars and the results of a previous studies [23, 24], the Serverless Framework is the most popular open-source tool for serverless applications. Therefore, we believe that limiting the data source to projects developed with this framework does not hinder generalizing the results of our study. Second, AWS plays a major role in serverless technology, which reflects into the fact that, in the dataset, AWS projects are predominant; consequently, the analysis can be biased towards AWS models and offerings. We discussed the dominance of AWS in the FaaS ecosystem in Section 4.2. Therefore, we believe this leaning towards AWS is the nature of any real-world dataset containing multiple providers and platforms.

The limitation of our analysis comes down to the results related to the RQ4, which required the author(s) of this study to manually inspect the documentation of the projects to know more about their use cases, and to classify the use

cases across different categories (Section 4.4). Since this activity is extremely time consuming, we decided to consider only about half of the dataset, with randomly selected projects. While using a subset of a dataset for manual inspection is a common practice in several surveys and empirical studies, it is worth mentioning that these classifications may be subjective based on the understanding of the author(s).

6. Related Work

For an overview of the current research trends in this model, we refer to the ACM's contributed article on the next phase of cloud computing [2], to the summary of existing issues compiled during the first international workshop on serverless computing [21], and to the *Berkeley view* on this topic [33].

Some works qualitatively classify and review popular FaaS platforms combining various research methods (e.g., literature reviews, interviews, code inspection). Yussupov et al. [35] present a FaaS platform classification framework obtained by reviewing academic literature and by analyzing platforms' documentations. They used this framework to conduct a technology review of ten general-purpose and actively-maintained FaaS platforms from both a business view and a technical viewpoint. From the business perspective, they consider criteria such as licensing, release status, community, and documentation while they analyze development experience, testing and debugging, code reuse, etc. for the technical perspective. Kritikos and Skrzypek [23] also evaluate and compare several serverless frameworks based on criteria that map to different phases of a serverless application lifecycle.

Several studies [37, 63, 64, 65, 66, 67, 9, 51] have evaluated the performance of FaaS solutions by running benchmark functions on different platforms. The metrics adopted in these studies cover a wide range, including computing time, memory footprint, network usage, and I/O consumption. Another class of studies investigates the cost of FaaS platforms based on utilization parameters. Work in [17] presents how adopting the Lambda deployment architecture reduces hosting costs by studying the migration of two industrial cases of early adopters. Jackson and Clynch [42] study the impact of language runtime on the cost of serverless functions in AWS Lambda and Azure Functions. Bortolini and Obelheiro [68] investigate the cost variations within and across FaaS platforms based on the choice of memory allocation, FaaS provider, and programming language. The research work by Shahrad et al. [69] characterizes the FaaS development workload of Azure Functions, from the perspective of the provider. Our study is orthogonal to these works, as we focus on the implementation characteristics of serverless applications, regardless of the provider's performance and cost.

A few works have studied the common practices and challenges of implementing a serverless application from the perspective of serverless providers and programmers. Spillner [52] investigates the evolution of Lambda functions through the AWS SAR providing high-level implementation statistics on its current use. These statistics report how functions are defined, offered, and stored by investigating the evolution of function-level metadata, code-level metadata, and code-level implementation of Lambda functions using continuous observation, extraction, mining and conflation of repositories in AWS SAR. In contrast to our paper, which targets various dimensions of serverless computing and different providers, the work by Spillner focuses only on the evolution of AWS SAR. Eismann et al. [34, 70] provide a picture of serverless computing, including company adoption, suitable application context, and implementation of serverless applications. In these studies, the authors analyze 32 open-source serverless projects along with 57 serverless sources including industrial sources, academic literature, and scientific computing. Compared to the works by Eismann et al., we inspect a much larger dataset which we believe is representative of the current state of open-source serverless applications.

Wen et al. [12] analyze 22,731 relevant questions from Stack Overflow (a question and answer website for programmers) to report on the popularity and difficulty level of FaaS for programmers. Leitner et al. [8] present the results of a mixed-method study consisting of interviews, a literature review, and a Web-based survey to identify best practices for building serverless applications. They collect the data based on the programmer's experience while developing an application. Our work is methodologically different from these studies in that we analyze the source code of open-source projects to characterize serverless applications.

7. Conclusion

This paper provides insights into the FaaS ecosystem by studying almost 2K real-world serverless applications – the largest dataset to date. We consider various aspects that characterize FaaS, like growth rate, architectural design,

and common use cases. The results of our study provide the picture of an ecosystem that, despite the hype around the FaaS model in the last few years, is still struggling with a number of basic challenges, including lack of open component repositories, limited support for the development process, and fundamental tights to vendor-specific solutions that inevitably introduce barriers between alternative platforms.

8. Acknowledgments

This work has been cofunded by the Swiss National Science Foundation (SNSF, No. 200429), by the German Research Foundation (DFG, SFB 1119), by the Hessian LOEWE initiative (emergenCITY and Software-Factory 4.0), and by the University of St. Gallen (IPF, No. 1031569).

References

[1] K. Costello, The cio's guide to serverless computing, https://www.gartner.com/smarterwithgartner/the-cios-guide-to-serverless-computing/, Accessed: 2021-07-28 (2020).

[2] J. Schleier-Smith, V. Sreekanti, A. Khandelwal, J. Carreira, N. J. Yadwadkar, R. A. Popa, J. E. Gonzalez, I. Stoica, D. A. Patterson, What serverless computing is and should become: The next phase of cloud computing, Commun. ACM 64 (5) (2021) 76–84. doi:10.1145/3406011.
URL https://doi.org/10.1145/3406011

[3] I. Pelle, J. Czentye, J. Dóka, B. Sonkoly, Towards latency sensitive cloud native applications: A performance study on AWS, in: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), IEEE, 2019, pp. 272–280.

[4] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, C. Wu, Serverless computing: One step forward, two steps back, arXiv preprint arXiv:1812.03651.

[5] J. Manner, M. Endreß, T. Heckel, G. Wirtz, Cold start influencing factors in function as a service, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 181–188.

[6] V. Lenarduzzi, A. Panichella, Serverless testing: Tool vendors and experts point of view, IEEE Software.

[7] J. Nupponen, D. Taibi, Serverless: What it is, what to do and what not to do, in: 2020 IEEE International Conference on Software Architecture Companion (ICSA-C), IEEE, 2020, pp. 49–50.

[8] P. Leitner, E. Wittern, J. Spillner, W. Hummer, A mixed-method empirical study of function-as-a-service software development in industrial practice, Journal of Systems and Software 149 (2019) 340–359.

[9] S. K. Mohanty, G. Premsankar, M. Di Francesco, et al., An evaluation of open source serverless computing frameworks., in: CloudCom, 2018, pp. 115–120.

[10] A. Eivy, J. Weinman, Be wary of the economics of" serverless" cloud computing, IEEE Cloud Computing 4 (2) (2017) 6–12.

[11] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. M. Faleiro, J. E. Gonzalez, J. M. Hellerstein, A. Tumanov, Cloudburst: Stateful functions-as-a-service, arXiv preprint arXiv:2001.04592.

[12] J. Wen, Z. Chen, Y. Liu, Y. Lou, Y. Ma, G. Huang, X. Jin, X. Liu, An empirical study on challenges of application development in serverless computing, in: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), 2021.

[13] N. Eskandani, G. Salvaneschi, The wonderless dataset for serverless computing, in: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR), IEEE, 2021, pp. 565–569.

[14] I. Scholtes, P. Mavrodiev, F. Schweitzer, From aristotle to ringelmann: a large-scale analysis of team productivity and coordination in open source software projects, Empirical Software Engineering 21 (2) (2016) 642–683.

[15] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al., A view of cloud computing, Communications of the ACM 53 (4) (2010) 50–58.

[16] P. Mell, T. Grance, et al., The nist definition of cloud computing.

[17] G. Adzic, R. Chatley, Serverless computing: economic and architectural impact, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 884–889.

[18] V. Yussupov, U. Breitenbücher, F. Leymann, M. Wurster, A systematic mapping study on engineering function-as-a-service platforms and tools, in: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, 2019, pp. 229–240.

[19] D. A. Bryan, B. B. Lowekamp, C. Jennings, Sosimple: A serverless, standards-based, p2p sip communication system, in: First International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA'05), IEEE, 2005, pp. 42–49.

[20] W. Ye, A. I. Khan, E. A. Kendall, Distributed network file storage for a serverless (p2p) network, in: The 11th IEEE International Conference on Networks, 2003. ICON2003., IEEE, 2003, pp. 343–347.

[21] G. C. Fox, V. Ishakian, V. Muthusamy, A. Slominski, Status of serverless computing and function-as-a-service (faas) in industry and research, arXiv preprint arXiv:1708.08028.

[22] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, A. Trivedi, Understanding ephemeral storage for serverless analytics, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), USENIX Association, Boston, MA, 2018, pp. 789–794.
URL https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[23] K. Kritikos, P. Skrzypek, A review of serverless frameworks, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 161–168.

[24] Datadog, The state of serverless, https://www.datadoghq.com/state-of-serverless/, Accessed: 2020-03-16 (2020).

[25] V. Yussupov, U. Breitenbücher, F. Leymann, C. Müller, Facing the unplanned migration of serverless applications: A study on portability problems, solutions, and dead ends, in: Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing, 2019, pp. 273–283.
[26] P. C. Pendharkar, J. A. Rodger, An empirical study of the impact of team size on software development effort, Information Technology and Management 8 (4) (2007) 253–262.
[27] J. Fritzsch, J. Bogner, S. Wagner, A. Zimmermann, Microservices migration in industry: intentions, strategies, and challenges, in: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2019, pp. 481–490.
[28] V. Ivanov, K. Smolander, Implementation of a devops pipeline for serverless applications, in: International conference on product-focused software process improvement, Springer, 2018, pp. 48–64.
[29] J. Manner, S. Kolb, G. Wirtz, Troubleshooting serverless functions: a combined monitoring and debugging approach, SICS Software-Intensive Cyber-Physical Systems 34 (2) (2019) 99–104.
[30] H. Borges, M. T. Valente, What's in a github star? understanding repository starring practices in a social coding platform, Journal of Systems and Software 146 (2018) 112–129.
[31] C. Vendome, M. Linares-Vásquez, G. Bavota, M. Di Penta, D. German, D. Poshyvanyk, License usage and changes: a large-scale study of java projects on GitHub, in: 2015 IEEE 23rd International Conference on Program Comprehension, IEEE, 2015, pp. 218–228.
[32] Y. Golubev, M. Eliseeva, N. Povarov, T. Bryksin, A study of potential code borrowing and license violations in java projects on GitHub, in: Proceedings of the 17th International Conference on Mining Software Repositories, 2020, pp. 54–64.
[33] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, D. A. Patterson, Cloud programming simplified: A Berkeley view on serverless computing, CoRR abs/1902.03383. arXiv:1902.03383.
URL http://arxiv.org/abs/1902.03383
[34] S. Eismann, J. Scheuner, E. van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, A. Iosup, Serverless applications: Why, when, and how?, IEEE Software.
[35] V. Yussupov, J. Soldani, U. Breitenbücher, A. Brogi, F. Leymann, Faasten your decisions: A classification framework and technology review of function-as-a-service platforms, Journal of Systems and Software (2021) 110906.
[36] C. Ayala, Ø. Hauge, R. Conradi, X. Franch, J. Li, Selection of third party software in off-the-shelf-based software development—an interview study with industrial practitioners, Journal of Systems and Software 84 (4) (2011) 620–637.
[37] L. Wang, M. Li, Y. Zhang, T. Ristenpart, M. Swift, Peeking behind the curtains of serverless platforms, in: 2018 USENIX Annual Technical Conference (USENIX ATC 18), USENIX Association, Boston, MA, 2018, pp. 133–146.
URL https://www.usenix.org/conference/atc18/presentation/wang-liang
[38] T. N. Bui, Benchmarking AWS Lambda runtimes in 2019, https://medium.com/the-theam-journey/benchmarking-aws-lambda-runtimes-in-2019-part-i-b1ee459a293d, Accessed: 2021-04-23 (2019).
[39] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, V. Sukhomlinov, Agile cold starts for scalable serverless, in: 11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19), 2019.
[40] P. Silva, D. Fireman, T. E. Pereira, Prebaking functions to warm the serverless cold start, in: Proceedings of the 21st International Middleware Conference, 2020, pp. 1–13.
[41] P.-M. Lin, A. Glikson, Mitigating cold starts in serverless platforms: A pool-based approach, arXiv preprint arXiv:1903.12221.
[42] D. Jackson, G. Clynch, An investigation of the impact of language runtime on the performance and cost of serverless functions, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 154–160.
[43] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Serverless computation with openlambda, in: 8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16), 2016.
[44] S. McConnell, Software estimation: demystifying the black art, Microsoft press, 2006.
[45] S. Bhatia, J. Malhotra, A survey on impact of lines of code on software complexity, in: 2014 International Conference on Advances in Engineering & Technology Research (ICAETR-2014), IEEE, 2014, pp. 1–4.
[46] J. Aguilar, M. Sanchez, C. Fernandez-y Fernandez, E. Rocha, D. Martinez, J. Figueroa, The size of software projects developed by Mexican companies, arXiv preprint arXiv:1408.1068.
[47] K. Costantini, Updated function point gearing factor table, https://www.qsm.com/blog/2013/updated-function-point-gearing-factor-table, Accessed: 2021-08-01 (2013).
[48] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu, The serverless trilemma: Function composition for serverless computing, in: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, 2017, pp. 89–103.
[49] S. Kotni, A. Nayak, V. Ganapathy, A. Basu, Faastlane: Accelerating function-as-a-service workflows, in: 2021 {USENIX} Annual Technical Conference ({USENIX}{ATC} 21), 2021, pp. 805–820.
[50] P. G. López, M. Sánchez-Artigas, G. París, D. B. Pons, Á. R. Ollobarren, D. A. Pinto, Comparison of faas orchestration systems, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), IEEE, 2018, pp. 148–153.
[51] J. Spillner, C. Mateos, D. A. Monge, Faaster, better, cheaper: The prospect of serverless scientific computing and hpc, in: Latin American High Performance Computing Conference, Springer, 2017, pp. 154–168.
[52] J. Spillner, Quantitative analysis of cloud function evolution in the AWS serverless application repository, arXiv preprint arXiv:1905.04800.
[53] S. Hong, A. Srivastava, W. Shambrook, T. Dumitras, Go serverless: Securing cloud via serverless design patterns, in: 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18), USENIX Association, Boston, MA, 2018.
URL https://www.usenix.org/conference/hotcloud18/presentation/hong
[54] D. Bardsley, L. Ryan, J. Howard, Serverless performance and optimization strategies, in: 2018 IEEE International Conference on Smart Cloud (SmartCloud), IEEE, 2018, pp. 19–26.
[55] R. Chatley, S. Eisenbach, J. Magee, Painless plugins, London: Imperial College.
[56] S. Winzinger, G. Wirtz, Coverage criteria for integration testing of serverless applications, in: 13th Symposium and Summer School On

Service-Oriented Computing, 2019.
[57] S. Brenner, R. Kapitza, Trust more, serverless, in: Proceedings of the 12th ACM International Conference on Systems and Storage, SYSTOR '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 33–43. doi:10.1145/3319647.3325825.
URL https://doi.org/10.1145/3319647.3325825
[58] W. Qiang, Z. Dong, H. Jin, Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave, in: International Conference on Security and Privacy in Communication Systems, Springer, 2018, pp. 451–470.
[59] H. Cervantes, R. S. Hall, Automating service dependency management in a service-oriented component model, in: ICSE CBSE Workshop, Citeseer, 2003.
[60] R. Cordingly, W. Shu, W. J. Lloyd, Predicting performance and cost of serverless computing functions with saaf, in: 2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech), IEEE, 2020, pp. 640–649.
[61] A. Boschetti, L. Massaron, Python data science essentials, Packt Publishing Ltd, 2015.
[62] K. Lei, Y. Ma, Z. Tan, Performance comparison and evaluation of web development technologies in php, python, and node. js, in: 2014 IEEE 17th international conference on computational science and engineering, IEEE, 2014, pp. 661–668.
[63] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, S. Pallickara, Serverless computing: An investigation of factors influencing microservice performance, in: 2018 IEEE International Conference on Cloud Engineering (IC2E), IEEE, 2018, pp. 159–169.
[64] K. Figiela, A. Gajek, A. Zima, B. Obrok, M. Malawski, Performance evaluation of heterogeneous cloud functions, Concurrency and Computation: Practice and Experience 30 (23) (2018) e4792.
[65] H. Lee, K. Satyam, G. Fox, Evaluation of production serverless computing environments, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), IEEE, 2018, pp. 442–450.
[66] G. McGrath, P. R. Brenner, Serverless computing: Design, implementation, and performance, in: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), IEEE, 2017, pp. 405–410.
[67] T. Back, V. Andrikopoulos, Using a microbenchmark to compare function as a service solutions, in: European Conference on Service-Oriented and Cloud Computing, Springer, 2018, pp. 146–160.
[68] D. Bortolini, R. R. Obelheiro, Investigating performance and cost in function-as-a-service platforms, in: International Conference on P2P, Parallel, Grid, Cloud and Internet Computing, Springer, 2019, pp. 174–185.
[69] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, R. Bianchini, Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, in: 2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20), 2020, pp. 205–218.
[70] S. Eismann, J. Scheuner, E. Van Eyk, M. Schwinger, J. Grohmann, N. Herbst, C. Abad, A. Iosup, The state of serverless applications: Collection, characterization, and community consensus, IEEE Transactions on Software Engineering.