A Mixed-Method Empirical Study of Function-as-a-Service Software Development in Industrial Practice

Philipp Leitner[a,*], Erik Wittern[b], Josef Spillner[c], Waldemar Hummer[b]

[a] *Software Engineering Division, Chalmers | University of Gothenburg, Sweden*
[b] *IBM Research, Yorktown Heights, New York, USA*
[c] *Service Prototyping Lab, Zurich University of Applied Sciences, Switzerland*

Abstract

Function-as-a-Service (FaaS) describes cloud computing services that make infrastructure components transparent to application developers, thus falling in the larger group of “serverless” computing models. When using FaaS offerings, such as AWS Lambda, developers provide atomic and short-running code for their functions, and FaaS providers execute and horizontally scale them on-demand. Currently, there is no systematic research on how developers use serverless, what types of applications lend themselves to this model, or what architectural styles and practices FaaS-based applications are based on. We present results from a mixed-method study, combining interviews with practitioners who develop applications and systems that use FaaS, a systematic analysis of grey literature, and a Web-based survey. We find that successfully adopting FaaS requires a different mental model, where systems are primarily constructed by composing pre-existing services, with FaaS often acting as the “glue” that brings these services together. Tooling availability and maturity, especially related to testing and deployment, remains a major difficulty. Further, we find that current FaaS systems lack systematic support for function reuse, and abstractions and programming models for building non-trivial FaaS applications are limited. We conclude with a discussion of implications for FaaS providers, software developers, and researchers.

1. Introduction

Since its emergence, the programmable cloud has been a rapidly growing area of interest for application deployment. Various providers, including Amazon Web Services, Google Cloud, Microsoft Azure, and IBM Cloud (formerly Bluemix), offer services on different levels of the cloud stack, e.g., Infrastructure-as-a-Service (IaaS) or Platform-as-a-Service (PaaS). IaaS services often lend themselves to a “lift-and-shift” style migration, where entire applications can be migrated without deep changes as mostly self-contained virtual machines (VMs) or containers. However, PaaS services, which provide a higher level of abstraction, ask for more concessions in how applications are architected, built, and deployed in return for a richer development experience and more built-in features [1]. Ultimately, many applications deployed in PaaS clouds are “cloud-native”, in the sense that they are specifically built for the cloud, or even a specific combination of services, and cannot easily be operated anywhere else. This introduces a certain amount of lock-in, which practitioners are still often willing to accept in exchange for built-in elasticity and resilience, lower costs of operation, and a relief from having to manage their own infrastructure [2].

In recent years, the term “serverless computing”[1] has gained momentum to describe the pinnacle of the cloud-native model: a “serverless” cloud application is deployed to infrastructure components that are entirely transparent to the application developer. The fundamental promise of serverless is sometimes pointedly described as “NoOps”, a wordplay on the well-known DevOps movement [3] that stresses that “no operations” is, at least in theory, required to maintain a serverless application.

The most prominent implementation of the serverless model are *Function-as-a-Service* (FaaS) offerings, such as AWS Lambda, Azure Functions or IBM’s and Google’s Cloud Functions. When using FaaS offerings, application developers provide source code of relatively atomic and typically short-running functions, and define triggers for executing these functions, for example HTTP requests or system events. The FaaS provider then, on-demand, executes and bills functions as isolated instances and scales their execution horizontally as needed. Ideally, application developers using FaaS benefit from simple deployments, reduced operation efforts, and pay-as-you-go pricing, while

*Corresponding author
Email addresses: `philipp.leitner@chalmers.se` (Philipp Leitner), `witternj@us.ibm.com` (Erik Wittern), `josef.spillner@zhaw.ch` (Josef Spillner), `whummer@ibm.com` (Waldemar Hummer)

[1] `https://martinfowler.com/articles/serverless.html`

providers have the chance to efficiently serve large numbers of users and functions with relatively few resources.

Unsurprisingly, these advantages of FaaS do not come for free. FaaS-based applications need to adhere to a multitude of technical and architectural restrictions to ease their transparent management. For instance, functions deployed to AWS Lambda cannot, at the time of writing, retain state between invocations, need to finish processing in 5 minutes or less, have unpredictable execution time deviations and cannot make use of more than a single CPU thread [4]. More importantly, AWS Lambda and other FaaS services are ultimately intended to host small microservices implemented in a few hundred lines of program code. Composing larger applications from such stateless microservices, or decomposing an existing monolith into them, is still a challenging open issue [5, 3].

In this paper, we present the first systematic study of software development for FaaS-based applications. Our study addresses the following research questions:

- **RQ1:** Which types of applications is FaaS-based computing used for in today's industrial practice? Which types of use cases is this technology valuable for?
- **RQ2:** What are the key architectural patterns and best practices for building FaaS applications?
- **RQ3:** What are the major advantages and challenges of using serverless and FaaS in practice?

Given the immaturity of our study subject, we use an exploratory mixed-method empirical research design to address these questions. We initially conduct a structured review of multi-vocal ("grey" [6]) literature (e.g., online blogs) and semi-structured interviews with 12 practitioners in the field. We use grounded theory, as well as open and axial coding, to generate a number of initial findings and hypotheses related to our research questions, which we then validate and refine based on a Web-based survey with 182 valid respondents.

Our study shows that FaaS is commonly used in backend scenarios, where the technology is often used to handle batch jobs. Building user-facing FaaS applications is possible, but requires careful design to deal with slow tail latency due to container startup. Adopting serverless requires a different mental model, where systems are primarily constructed by composing pre-existing services, with FaaS often acting as the "glue" that brings these services together. FaaS offers technical and business-related advantages, but managing and predicting deployment costs is difficult for larger applications. Further, tooling availability and maturity, especially related to testing and deployment, remains a barrier to entry. Finally, limited support for function sharing and the absence of a service ecosystem is seen as a challenge.

The remainder of this paper is structured as follows. We provide details on FaaS offerings, development of cloud functions and other important concepts in Section 2. In Section 3, we discuss our study design in detail, followed by an extensive discussion of results and study outcomes in Section 4. Section 5 discusses the main implications of these results and presents the central lessons learned for FaaS providers, users, and researchers. Section 6 puts our work in context of the existing body of research. Finally, Section 7 summarizes and concludes the paper.

2. Background

This section introduces the main concepts and architecture of Function-as-a-Service (FaaS) offerings. It further introduces selected technology stacks, providers, and developer tools including their characteristics and limitations.

2.1. FaaS Characteristics

FaaS offerings allow developers to provide pieces of code that, upon being triggered, are executed in an isolated environment. This model can be considered an evolution over previous cloud computing paradigms. Initial cloud computing offerings such as Amazon EC2 or SoftLayer relied on (orchestrated) virtual machines which, along with system containers, offer a high degree of isolation and architecture-specific but language-independent encapsulation. Platform-as-a-Service offerings such as Heroku raise the abstraction by providing language-specific application runtimes, which are generally long-running. In comparison, FaaS offerings provide more fine-grained scalability and corresponding pricing.

Individual functions typically describe only parts of a larger application. For example, rather than containing a complete web application with a RESTful interface, a single function may only implement one endpoint of such an interface. Functions are expected to execute in a limited amount of time, i.e., a few minutes at most. When exceeding this threshold, the execution will time out. Depending on configuration, failed executions (due to timeout or any other reason) may be automatically retried. Hence, it is important that the logic of functions be implemented in an idempotent manner [7]. Functions can receive input data, which may be required by or influence the function execution, and produce output data. In addition, function executions may result in additional data being produced, such as logs or execution metrics.

Function executions are triggered by events, which can be diverse in nature. For example, client requests, events produced by (external) systems, data streams, or even complex rules describing a combination of the above can all be configured to trigger a function. The concrete available triggers are specific to the FaaS offerings.

The FaaS provider is responsible to horizontally scale function executions in response to the amount of incoming events. That is, function developers have no insights into how their code is actually being provisioned, but may

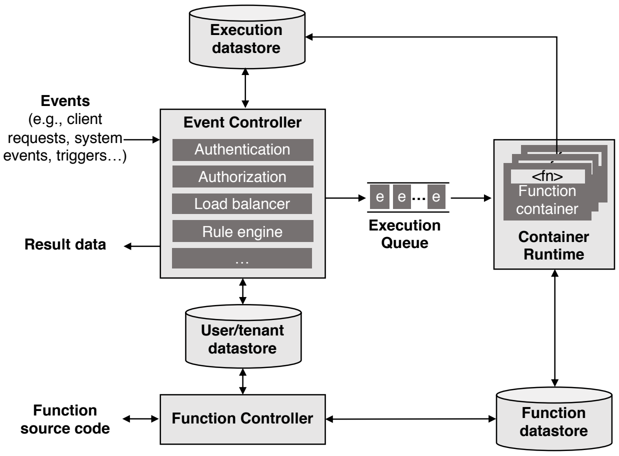

Figure 1: Exemplary FaaS system architecture

assume that sufficient resources are provided to deal with whatever workload the service experiences. These resources are typically billed using a pay-as-you-go model, e.g., per number of function executions, function execution time, or I/O operations. That is, costs generally depend directly on usage, and an idle function that is not invoked incurs no or very little charges.

Functions are assumed to be stateless. After being executed, state in memory at the end of the execution can not and should not be assumed to be available during the next function execution. Functions can be written in various programming languages, however, which concrete languagues are available depends on the specific FaaS provider. Most providers support at least basic JavaScript (Node.js), Python, and Java runtimes but expect functions to encapsulate additional third-party libraries.

2.2. Common FaaS System Architecture

Figure 1 illustrates an exemplary architecture of a FaaS system with the previously outlined characteristics. The architecture is loosely based on that of the publicly available Apache OpenWhisk FaaS system.[2]

At design time, function developers interact with a *function controller* to create, update, or delete functions. As part of this, they provide the function source code, set up event triggers, and possibly define rules for function executions. Function source code is persisted in a *function datastore*, and triggers and rules are persisted in a *user / tenant datastore*.

At runtime, incoming events are processed by the *event controller*. It determines which functions to execute based on triggers and rules present in the user / tenant datastore. The controller also ensures that events are authenticated and authorized to be processed. The event controller determines which *container runtime* is supposed to execute the function, based on current system load. Then the execution is queued to ensure it is performed even under heavy system load or in case of (partial) system failures. Next, the (designated) container runtime picks up the execution from the queue. It either starts a new *function container* and injects the required function source code from the function datastore into it, or it reuses an existing function container for execution. Importantly, the service user or developer has no control over whether the FaaS service allocates an existing container or provisions a new one. The function is executed in the selected container, and the execution results are persisted in the *execution datastore*, from where they can be returned to clients via the event controller. The container runtime may time out executions that take too long, and will eventually destroy idling function containers.

The possibility that the container runtime may reuse existing containers leads to functions not being entirely stateful: reused containers may allow to access state set on disk during past executions, but there is no guarantee for any future invocation to actually have access to this state. Container reuse also impacts the response times of functions. Starting a new container and injecting function code takes a significant amount of time (up to multiple seconds, strongly depending on the used programming language), which leads to functions experiencing high tail latency [8].

2.3. Example Function

An example function for AWS Lambda, which fetches images from an S3 datastore and runs an object detection algorithm, is provided in Listing 1. The example is written in the Python programming language. Functions typically need to implement a generic interface, taking the triggering event and the context as input, and often produce a JSON-serializable object as output. Function implementations are often rather small, such as in the provided example.

Listing 1: Example AWS Lambda function in Python

```python
import boto3
def detectobjects(f):pass # not defined here
def lambda_handler(event, context):
  tmppath = "/tmp/tempimages/"
  s3 = boto3.client("s3")
  objectlist = []
  for image in event["images"]:
    imgfile = tmppath + image
    s3.download_file("images", image, imgfile)
    objectlist += detectobjects(imgfile)
  return {"objects": objectlist}
```

2.4. FaaS Providers

All major cloud providers offer FaaS implementations by now. Offerings include AWS Lambda, IBM Cloud Functions (which is based on the open source project Apache OpenWhisk), Microsoft Azure Functions, and Google Cloud Functions. Typically, these FaaS offerings provide integrations with other Cloud services, such as API gateways, to

[2] For details, see `https://console.bluemix.net/docs/openwhisk/openwhisk_about.html`

monitor, control, and bill function usage, authentication and authorization services, or application services, analytics services, or development systems that may trigger function executions. Furthermore, more specialized FaaS offerings exist, like IronFunctions or Webtask.io.

2.5. FaaS Developer Tooling

Most FaaS providers offer Command Line Interfaces (CLIs), Software Development Kits (SDKs), and further resources to facilitate the design, implementation, testing, and deployment of cloud functions. AWS, for example, offers reference functions, tutorials, and webinars, an application repository[3], an API reference, and a local testing framework in addition to SDKs in various programming languages. In addition, some providers offer orchestration tools like IBM Composer, AWS Step Functions, and Fission Workflows that help developers to form larger applications from functions or to execute complex workflows. These tools allow to define sequences and/or parallel executions of functions and provide strategies to handle execution failures. Finally, many FaaS providers offer metric collection, tracing, and debugging facilities, if only at additional cost. Examples include AWS X-Ray and Google Stackdriver.

Beyond provider-offered tools, a plethora of third-party created tools have emerged. Foremost among them are FaaS deployment tools such as the Serverless framework[4], which abstracts from provider-specific aspects of FaaS offerings, allowing to build and deploy provider-independent functions.

3. Study Design

We express the goal of this study based on the template defined by the TAME project [9]. The purpose of this study is to characterize the use of FaaS. Characterization helps in understanding and guides practitioners and researchers in the evolution of FaaS offering, related tools, and related software engineering processes. This study reveals common practices, advantages, disadvantages, challenges and opportunities of using FaaS. The here presented perspective is that of practitioners from industry who develop systems or applications that use FaaS offerings as part of their job. We focus on FaaS systems offered for on-demand use by Cloud service providers specifically, and serverless computing more broadly. Given the exploratory nature of this study subject and our research questions, we decided on a mixed-method study protocol that combines qualitative and exploratory elements with a structured, quantitative survey.This follows the recommendations by Bratthall and Jørgensen [10] as well as established practice used in other, comparable studies in software engineering [11, 12, 2].

3.1. Overview

We base our research on three primary data sources: semi-structured practitioner interviews, a systematic review of multi-vocal ("grey") literature (e.g., blogs and Web articles), and a Web-based quantitative survey. Following a grounded theory approach, we combine open and axial coding in multiple rounds with an assessment of survey data. The basic structure of our research methodology is outlined in Figure 2. We conducted our research in three phases. In the first exploratory, phase we conducted interviews and analyzed grey literature. In the second phase, we turned towards validating and quantifying the qualitative results from the first phase through an online survey. In the third and final phase, we refined the results from the first two phases and constructed a final theory of serverless and FaaS usage in industry, which is the primary outcome of this study.

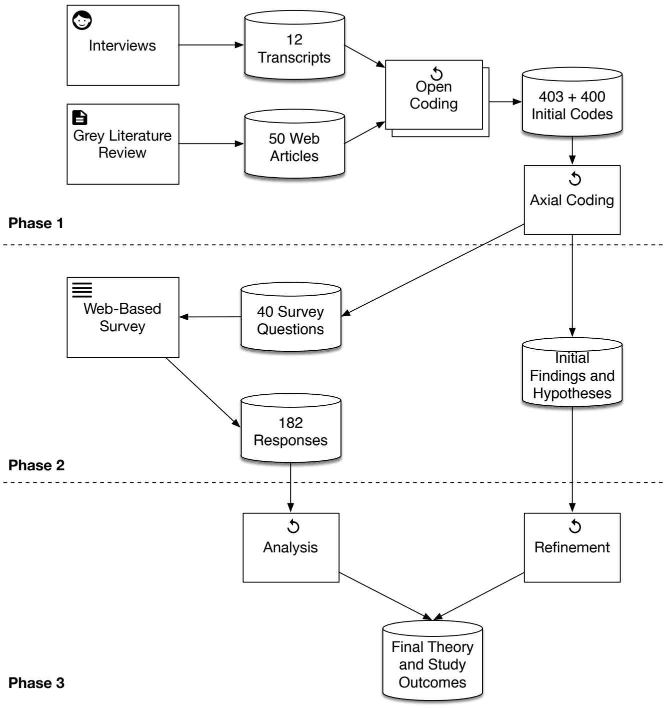

Figure 2: High-level overview of our mixed-method study methodology. Primary data sources of the study are (1) transcripts of 12 semi-structured interviews with practitioners, (2) 50 Web articles ("blogs") discussing industrial experience, and (3) 182 responses to a Web-based survey.

3.2. Semi-Structured Interviews

Interviews with practitioners are a common research method often employed in emerging fields, in which the scientific theory is not yet stable enough to compile scientific hypotheses through an analysis of peer-reviewed literature alone. The goal of our interviews was to collect and consolidate the, often disparate, lessons learned, success factors, and best practice knowledge that early adopters of serverless computing and FaaS have acquired.

[3] See `https://aws.amazon.com/serverless/serverlessrepo/`
[4] `https://serverless.com`

ID	Source	Main Provider	Main Lang.	Company Type	Country	Experience [Years]
I1	PN	IBM	various	Enterprise	US	*na.*
I2	PN	AWS	JavaScript	SME	CH	11+
I3	PN	IBM	various	Enterprise	US	11+
I4	AWSRC	AWS	JavaScript, Python	Enterprise	FR	3–5
I5	PN	AWS	Scala, JavaScript	Enterprise	DE	3–5
I6	PN	AWS	Python, Java	SME	AT	6–10
I7	WS	AWS	.NET, JavaScript	SME	IN	6–10
I8	MSAAN	MS	C#	Enterprise	PL	11+
I9	MSAAN	MS	C#	Enterprise	NL	11+
I10	WS	AWS / Google	JavaScript, Go	SME	US	*na.*
I11	MSAAN	MS	.NET	Enterprise	JP	11+
I12	MSAAN	MS	C#	Enterprise	CZ	11+

Table 1: Summary of basic interviewee information. "PN" refers to interviewees acquired through our personal network, "AWSRC" and "MSAAN" are recruited through the Amazon and Microsoft reference customer lists, and "WS" indicates interviewees found through Web search. All other data is based on self-reported information by the interviewees. The column "Experience" lists overall professional experience in the cloud domain. For I1 and I10, no data on their professional experience is available.

Participant Selection. Given these goals, our primary acceptance criterion for interview partners was *real-life production experience* with at least one FaaS technology (e.g., AWS Lambda, Azure Functions, etc.). Given that FaaS, while hyped, is still a niche topic, we used an opportunistic, multi-pronged approach to recruit suitable interview partners. Firstly, we used our personal networks to find interviewees. Secondly, we selectively contacted companies on the FaaS reference customer lists of AWS and Azure and asked for developers that we could interview. Thirdly, we actively approached individuals which are outspoken on the Web about their usage of FaaS (e.g., through blogs or Reddit discussion threads) and asked for interviews. Using this recruiting strategy, we acquired 12 interview partners out of 15 we initially contacted, which we refer to as *I1* to *I12*. Table 1 summarizes basic information about all interviewees. We list how we got in contact with them (personal network, AWS and Azure reference customer lists, and via the Web), which cloud provider their experience relates to, what programming languages they use in conjunction with FaaS, how large their company is (SME or large enterprise, where the SME category also includes startup companies), where they are located geographically, and how much experience with cloud technology this interviewee has reported. We refrain from reporting experience with FaaS specifically, as the technology is still young enough that even "experienced" practitioners have realistically not used the technology for more than a year or two.

Our interviewee population covers three continents, all major cloud providers, as well as startups, medium-sized companies, and large enterprises. Further, we have interviewed participants using a wide range of programming languages. In terms of cloud providers, there is a bias towards AWS and Microsoft Azure, which is not surprising given that these two providers are also leaders in the FaaS market at the time of our study. Finally, the majority of our interviewees are FaaS end users, but we have also interviewed two individuals who are developers working on a FaaS solution themselves (I1 and I3) and one individual who is a major contributor to the well-known open source toolkit *Serverless Framework* (I6).

Study protocol. We conducted all interviews following a semi-structured approach. We developed a coarse-grained interview guideline (see also the appendix of this paper). Specifically, we asked questions related to the projects that the interviewee used FaaS for in the past, why they chose to use it, how they architect serverless solutions, which advantages and disadvantages they see with the technology, and what they think that the future of FaaS holds. However, we did not cover all questions in the same order and in the same depth with each interviewee, but instead followed the flow of the conversation. We conducted all interviews remotely via Skype or Google Hangouts, and in English. All interviews were conducted by a combination of the first, second, and third author of the paper (for most interviews, multiple authors were interviewing together). Interviews took between 30 and 60 minutes, and were recorded with the permission of the interviewee to foster transcription and easier analysis.

Analysis. As a first step towards analysis, the first, third, and fourth authors manually produced verbatim transcripts of each interview. Then, the first and the second author independently produced a hierarchical set of codes from all transcripts using the open coding methodology. Afterwards, these initial code hierarchies were enriched with codes produced by analysing multi-vocal literature (see below), followed by another analysis step where the first and second author discussed and resolved differences in their coding and conducted axial coding together. This step led to an initial theory for FaaS usage in practice, which was then discussed with the other authors. We also used the results of the axial coding to develop the questions for the Web-based survey.

3.3. Survey of Multi-Vocal Literature

As noted by Garousi et al. [6] as well as by Barik et al. [13], much of the important discourse in software engineering does not happen through peer-reviewed, scientific articles, but rather through more informal publications, such as practitioner-oriented books, blogs, press releases, and white papers. While important to the conversation and often highly impactful in practice, these sources need to be considered with a certain skepticism, as they often lack a sound empirical and methodological basis. For our study, we have chosen to treat this body of "grey" literature as another source of qualitative information, analogously to the semi-structured interviews. This reflects the fact that individual grey literature items may be biased or not trustworthy (just like any individual interview does not necessarily deliver robust scientific results in itself), but in aggregation they still provide an accurate reflection of how the practitioner community thinks about our study subject.

Incidentally, our study also showcases one of the dangers of using grey literature as basis for academic research: article A42 (a blog post on the blogging site Medium[5]) has been removed by Medium shortly prior to the submission of this article, and is not available anymore. This shows that there is a need to consolidate the knowledge available on the Web in archived scientific publications.

Article selection. We particularly focus on articles, blogs, and discussions in comment threads on the Hackernews[6] news portal. We used the Hackernews search engine Algolia[7], and executed the search terms *serverless*, *aws lambda*, *azure functions*, and *openwhisk* to discover relevant articles. These search terms were generated in an interative, exploratory manner. Initially, we experimented with different variations of our main study subject (*serverless*, *FaaS*, *cloud functions*), but found that *FaaS* as a term is not sufficiently well-established to lead to interesting hits, while *cloud functions* was too general (many hits were false positives dealing with some arbitrary functionality of a cloud). We added *aws lambda*, *azure functions*, and *openwhisk* as simply searching for *serverless* resulted in a biased data set discussing AWS Lambda almost exclusively, and searching for specific services resulted in a broader coverage. To keep the size of the study managable, we focused on the services that were most prominently discussed in our interviews (Lambda, Azure, Openwhisk), and decided to skip the many alternatives that exist in the market (e.g., Google Functions, OpenFaaS, etc.).

Given the vast amount of blogs etc. covering our study subject, a complete survey was deemed infeasible. Hence, prior to starting our article collection, we set a goal of collecting 50 relevant articles. More concretely, we decided to first select 20 results for the general search term *serverless*, and enrich this data set with 10 additional results for each of the other terms. We ranked all search results by popularity (based on voting through the Hackernews platform), and went through the lists in order of popularity, checking for each article (1) whether it matches our inclusion/exclusion criteria (see below), and (2) whether it was not already contained in the data set. For each search term, we stopped when we reached the a priori set number of hits. Searches have been executed on September 9th, 2017 (*serverless*) and October 23rd, 2017 (other search terms).

Our inclusion/exclusion criteria were as follows. We accepted articles that describe reference architectures, case studies, or experience reports, but rejected tool announcements or pure marketing communications. We have accepted articles that advertise specific tools if the tool itself was built on top of FaaS (rather than being a FaaS service itself), and the article talked about how the tool made use of FaaS. For each article, we also skimmed the Hackernews comment threads, and included them in our article analysis (see below) if salient additional comments on the topic were raised in the comments. We refer to our total data set as articles *A1* to *A50*. A full list of articles including links is available in the appendix.

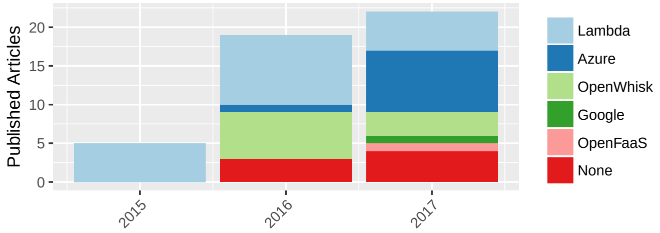

Figure 3: Distribution of Hackernews articles on serverless topics over years and cloud providers.

Figure 3 depicts when these articles have been published (if a publishing date is specified in the article), and what cloud provider they primarily discuss. The earliest article in our data set is from May 2015, the most recent one from October 2017. Most articles have been published in 2016 and 2017. Further, it is evident that the majority of articles in our set discuss AWS Lambda. Other cloud providers (e.g., Microsoft Azure and IBM's OpenWhisk) primarily come up when we explicitly searched for them, i.e., through the search terms *openwhisk* and *azure functions*. Two articles discuss the Google Cloud platform and the open source FaaS implementation OpenFaaS, which we do not explicitly cover in this work. Seven articles in the set are generic and do not discuss a specific provider.

Analysis. After open coding of the interview transcripts, the first author read all articles in our data set and updated the previous hierarchy of codes with any new codes emerging from the articles. That is, the article texts were treated as another source of qualitative evidence, coded, and integrated with the opinions collected through the interviews. In addition to the articles themselves, we have also read the comment sections on Hackernews related to the article. We treat these comments in the same way as the articles themselves. If new codes emerged from the discussion in the article comments, we have taken them up in the code hierarchy. However, we have excluded comments that were (1) downvoted (had a negative total rating on Hackernews) or (2) out of scope (i.e., they discussed an

[5] `https://medium.com`
[6] `https://news.ycombinator.com`
[7] `https://hn.algolia.com/`

aspect not directly related to our study subject).

Relationship to Existing Guidelines. Garousi et al. [14] present valuable guidelines for conducting multi-vocal literature surveys. Our research has been conducted in parallel to the development of these guidelines. Hence, we do not follow the guidelines exactly. However, we argue that they are largely compatible in goal and spirit.

Our design differs from these guidelines primarily in how we have built up the pool of candidate articles. Garousi et al. suggest to use a general search engine (e.g., Google) and extend the pool through snowballing. Following older suggestions by Barik et al. [13], we have instead queried a much more specific database (namely Hackernews), and did not make use of snowballing. We argue that our approach has both, advantages and disadvantages over the approach suggested by Garousi et al. Namely, our approach has a higher danger of missing relevant articles. However, it is presumably easier to replicate, and Hackernews provides a reasonable article quality indicator through community rating. Further, another advantage of using Hackernews is that the, often extensive, article comment threads provide another interesting data source besides the articles themselves.

3.4. Web-Based Survey

The main goal of the survey was to validate our qualitative findings on a larger sample of practitioners. We distributed an anonymous Web-based survey using the survey tool Typeform[8] consisting of, in total, 40 questions in 7 categories. An excerpt is available in the appendix and the raw form in the accompanying open research data repository [15].

Survey dissemination. Given that we do not have access to a baseline demography of cloud developers that we could send the survey to, we used a convenience sampling methodology and distributed the survey through a range of different channels. The survey was publicly available from January 11th to February 13rd 2018, and yielded 182 responses, or 5.52 per day. The survey took respondents on average 11:42 minutes to complete, and had a completion rate of 33.8%. Given our convenience sampling strategy, we are unable to provide an estimation of the survey response rate.

All responses were tagged with a source attribute in order to distinguish the survey dissemination channels and to prevent a domination of results from a single channel. Most responses were acquired through a topically relevant open source project that the last author is involved with[9] (49.4% of responses), advertisements on social media (15.9% of responses), and a mention in the newsletter of the well-known German IT news site heise.de[10] (15.3% of responses). The remaining 19.4% of responses came in through personal contacts of the authors, or other sources.

Participants. Our survey attracted responses from a wide range of IT professionals. Most respondents reported to be working in software developer or architect roles. However, we have also received responses from product owners, IT managers, site reliability engineers, DevOps engineers, researchers, and C-level executives.

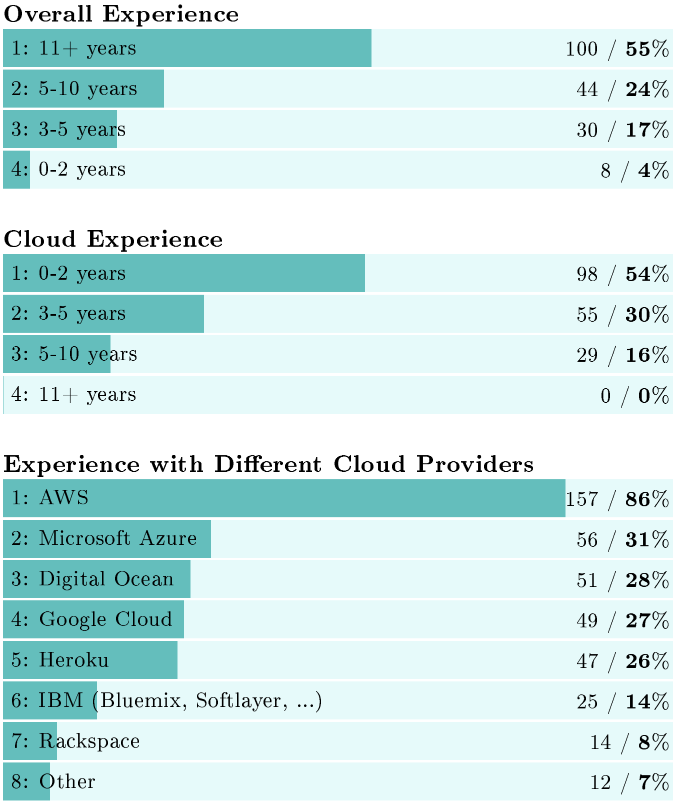

Figure 4: Overview of self-reported relevant experience of survey respondents. 54.9% of respondents report more than 10 years of IT experience. However, 53.8% report less than 3 years of experience with cloud technologies. AWS is by far the most commonly used cloud provider among our respondents.

As indicated in Figure 4, most respondents were experienced IT professionals (54.9% report more than 10 years of professional experience), but most respondents (53.8%) are not very experienced with cloud technology. This may be due to the fairly young age of the field in general. Most of our survey respondents report having worked with AWS in the past (86.3%). However, Microsoft Azure, Digital Ocean, Google Cloud, and Heroku are also commonly used (by 30.8%, 28%, 26.9%, and 25.8% of respondents respectively).

Survey questions. The survey consisted of seven groups of questions and a total of 40 questions of mixed type, including multiple-choice questions, numeric range choices, Likert scales, boolean yes/no questions, and free text questions. In addition to demographics (5 questions), we asked

[8] `https://typeform.com/`

[9] `https://github.com/localstack/localstack`

[10] `https://www.heise.de/newsletter/`

questions related to terminology (a single question), application architecture (14 questions), the FaaS development practices and patterns (10 questions), the FaaS mental model (5 questions), advantages and challenges when using FaaS (3 questions), and the future of FaaS (2 questions).

3.5. Limitations and Threats to Validity

While we have designed our research as a mixed-method study and based on grounded theory as a strong theoretical framework, there are still some limitations to our research design.

External validity. In terms of external validity, the question arises to what extent our interview participants are representative of serverless and FaaS developers, or of cloud developers in general. This threat is amplified, as 53% of survey respondents reported that they are not very experienced with cloud technology (see Section 4). We have mitigated this threat by also taking into account multivocal literature as a second qualitative data source, and by validating our findings through a survey. However, all three data sources in our study are potentially biased towards developers that are positive towards FaaS due to self-selection bias. That is, developers that are skeptical about or are not interested in FaaS are less likely to blog about the topic, agree to be interviewed about it, or fill out a Web survey. However, given that the goal of our research is to identify practices more than establish to what extent FaaS is used, we assume the impact of this bias to be small. Further, we have focused on AWS Lambda, Azure Functions, and IBM OpenWhisk as FaaS technologies in our study. While these appear to be the most widely used providers at the time of our study, it is not clear to what extent our results also generalize to other FaaS providers, particularly as we have in fact observed significant differences between providers.

Internal validity. In terms of internal validity, it is possible that we have biased the interviewees through the preselection of questions and topics in our interview guide. Consequently, we may have missed interesting codes because they were not discussed during the interviews. Our analysis of multi-vocal literature again served as a fail-safe against this threat, as we expect that any major missing discussion items would have emerged during this analysis. However, this has not been the case. Hence, we judge the risk that we have missed important aspects entirely to be low. Another threat to internal validity of our study is that we need to trust that interviewees, survey respondents, and article authors report truthfully on their usage of FaaS, i.e., we report on what participants *say*, but we do not have insights into what they actually *do*. This threat is inherent to our choice of research method.

4. Study Results

We now discuss the main outcomes of our study based on the raw results published as open research data [15]. As not unusual for current trends in Web development, we have experienced that concepts and terminology around serverless are less than well-defined. Particularly, the very name "serverless" is a source of confusion, as there certainly *are* servers running any serverless application – they are simply invisible to the application developer, as also mentioned in A39.

> "*'Serverless Computing' doesn't really mean there's no server. Serverless means there's no server you need to worry about.*" -Scott Hanselman, quoted in A39

A consequence of this comparatively broad definition of "serverless" is that the term does not only include FaaS, but encompasses various other kinds of hosted cloud services, many of which predate FaaS substantially (including hosted database technologies, such as DynamoDB [16]). In fact, many early definitions of cloud computing used the same principle of "servers as utility" as the defining feature of clouds [17]. According to some of our interviewees (e.g., I1), FaaS should really be understood as "serverless for computing", whereas "serverless for data storage" has been available since the early days of cloud computing.

One problem with this broad definition is that the delineation to PaaS is not clear-cut. In fact, a minority of interviewees actually consider PaaS services such as Heroku or Google's Appengine to be an early version of serverless. However, our survey results indicate that the majority of respondents (58%) still largely equate the terms FaaS and serverless (see Figure 5).

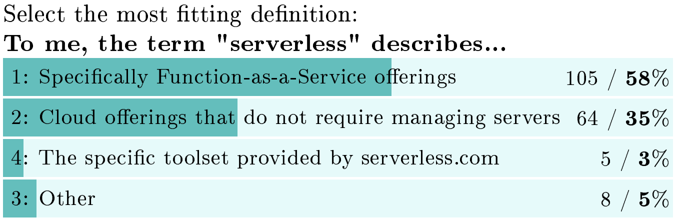

Figure 5: Survey respondent's definition of the term "serverless"

In the remainder of this paper, we will use the term "serverless" to represent the more general development model, and FaaS to specifically refer to cloud services such as AWS Lambda (but not, for instance, to DynamoDB or Appengine). Further, we use "Serverless framework" to refer to the popular open-source toolkit of the same name.

4.1. Mental Model

In our study, we observed that successful adopters of serverless have a different mental model of their systems or applications than developers of traditional Web-based applications. In traditional projects, developers often value being "in control" of all components in their application and re-use existing functionality (e.g., through external APIs) only when there are clear advantages or concrete needs for doing so. Serverless developers, on the other hand, inherently think of their application as a composition of small, stand-alone components, only some of which they (or their team, or even company) build themselves. Using external components becomes the default, and serverless developers assume from the get-go that most of their development will be dedicated to the integration of existing services rather than writing “new” code. Consequently, serverless applications by nature are more microservices-oriented than monolithic [18]. I6 goes as far as calling serverless *“microservices on steroids”*, i.e., the idea of microservices taken to the extreme. In many ways, serverless can be seen as a resurgence of Service-Oriented Architecture (SOA) concepts, such as service composition [19].

Further, given the tight coupling of serverless applications with other services in the cloud provider’s ecosystem, a tight coupling and considerable vendor lock-in is unavoidable. In our survey, a third of respondents consider vendor lock-ins to be a significant challenge when using FaaS, making it the third most named challenge (see also Section 4.5).

> "*AWS API Gateway, S3, Kinesis, SNS, DynamoDB, Step-Functions, or their Azure and GCP siblings — are at play with any serverless solution*" -A9

A common theme in our interviews was that referring to serverless applications as “applications” is even misleading, given the fundamentally different nature.

> "*I think the term ’application’ is oftentimes not really that applicable anymore (...) it‘s really hard to say, like, what is the application anymore [and what is part of the cloud or infrastructure.]*" -I6

Our survey confirms that building a serverless application using FaaS requires a different mental model than using more traditional cloud technologies, such as IaaS or Docker (Figure 6). 76% of respondents agree or strongly agreed with the sentiment that a different mental model or mindset is required to successfully build FaaS applications. None of the respondents strongly disagreed with this notion.

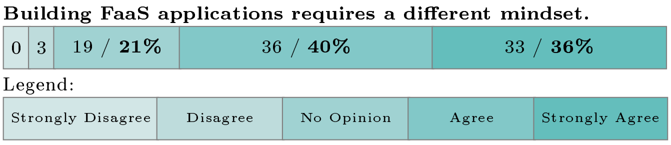

Figure 6: Mental model for developing FaaS applications. Values without percentage sign refer to absolute numbers of responses.

We further collected data which other types of (serverless or other) cloud services repondents commonly use in conjunction with FaaS (Figure 7). Unsurprisingly, hosted database services (77%) as well as API gateways (69%) are most commonly used in conjunction with FaaS. Especially using an API gateway is a de facto technical necessity when building a pure serverless, end-user facing application. It is interesting to note that 52% of respondents use FaaS in conjunction with IaaS following a hybrid model, but only 16% combine it with PaaS services, such as Heroku or CloudFoundry.

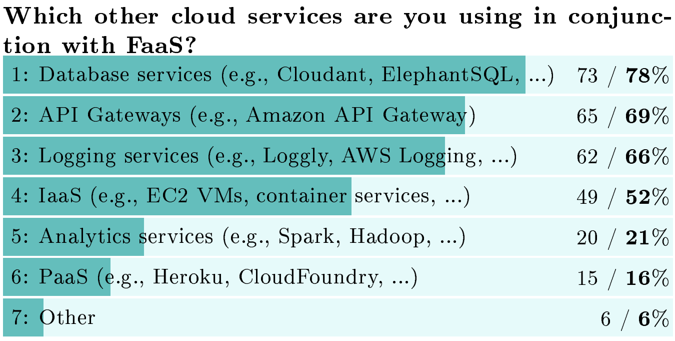

Figure 7: Cloud services used in conjunction with FaaS

In theory, this composition-focused mental model would enable high reuse of functions. However, in practice, serverless applications, at the time of our study, are largely compositions of services provided by the cloud, well-known external APIs, and self-written functions. A more peer-to-peer exchange of end user functions sounds promising, but is not a reality at the time of study. Some providers maintain a marketplace for functions (e.g., the Amazon Serverless Application Repository). However, so far structured reuse of existing functions from this marketplaces does not appear to be a common practice, although our interviewees found the general idea to be rather intriguing.

> "*(...) the idea of sharing code amongst various functions is an attractive one.*" -I10

Further, interviewees have also expressed that the different mental model of serverless is also a challenge for newcomers and can lead to a steep learning curve.

> "*People are very comfortable with things that they spent years learning, and this is different.*" -I3

Interestingly, this has not been confirmed in our survey. 56% have argued that the mental model behind FaaS is *not* difficult to grasp (Figure 8).

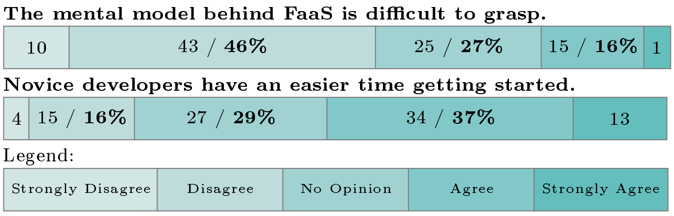

Figure 8: Difficulty of grasping the FaaS mental model. Values without percentage sign refer to absolute numbers of responses.

We speculate that one reason behind this surprising result may be that some existing development practices may train developers well for adopting a serverless mindset. Indeed, our survey confirmed that specifically experiences in functional programming (70%) and programming with scale-out, immutable infrastructure [2] (60%) are considered an asset when diving into FaaS (Figure 9).

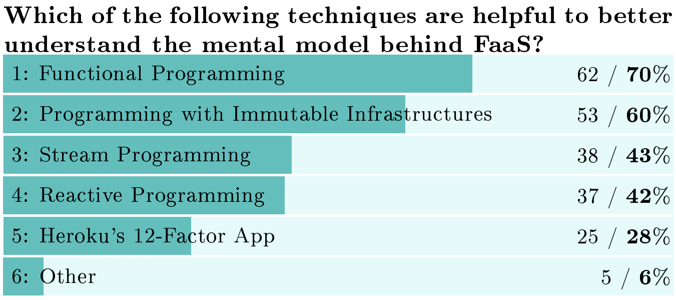

Figure 9: Helpful techniques to better understand FaaS

However, given that some developers still appear to be struggling with adopting to the serverless mental model, A20 speculates that hiring inexperienced developers may sometimes actually be beneficial as these newcomers may have an easier time adopting to the FaaS mental model.

> "*With Serverless hiring less experienced developers can work out better than hiring experienced cloud developers*" -Paul Johnston, quoted in A20

Our survey respondents moderately agreed with this notion, although the idea is not uncontested (Figure 8) – 51% of respondents agree or strongly agree with this notion, while 20% disagreed or strongly disagreed.

Summary: Building Serverless and FaaS applications requires a different mental model that emphasizes "plugging together" microservices. Currently, most services are either self-written or provided by the cloud infrastructure. Adopting this different mental model may be different, but experience with functional programming and the immutable infrastructure paradigm helps.

4.2. Types of Serverless Applications

In our interviews, two important dimensions to classify FaaS applications have emerged: whether the application is part of an end-user facing request cycle (e.g., a REST service that is invoked to serve a user request) or a backend application (e.g., a function that consolidates server logs), and whether the application is built entirely from serverless components ("pure serverless") or in conjunction with traditional cloud technology, such as virtual machines or Docker containers ("hybrid serverless"). We refer to the former as the *serverless use case type* (user-facing or backend service), and to the latter as the application's *purity* (pure or hybrid).

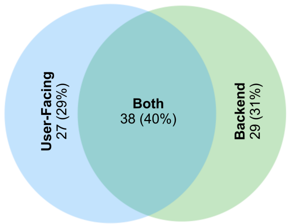

Figure 10: Percentage of survey respondents who have built only user-facing, only backend, or both types of FaaS applications.

As Figure 10 visualizes, both application types are commonly used among our survey respondents. 40% of respondents actually have built FaaS applications of both types in the past.

Serverless use case type. While the survey has shown that many practitioners actually build serverless user-facing applications, many of our interviewees were not convinced about this particular use case. An often-voiced concern was that the response time and latency of functions (with a tail latency in the range of multiple seconds [20], when a new container instance is started by the provider) does not lend itself to usage directly in an end user facing request cycle. These interviewees have argued that the primary use case for FaaS should be in backend applications, such as for transforming images, processing logs or telemetry data, scheduling and executing backups, sending out notification emails, and similar tasks.

> "*It's not in the request cycle directly what we do with Lambda, but it's more operations which also involve shipping, business information, business data as well.*" -I5

Another common backend use case was to use FaaS to write small "glue code" functions which connect other services or applications, e.g., take a file from S3, apply a small transformation to it, and forward it to a Big Data service for further processing. It can be argued that this is the true power of FaaS – to act as the glue that allows developers to bring together a multitude of existing hosted or self-developed cloud services. Such applications tend to not run continuously, but in a batched or event-based fashion, making FaaS a natural fit.

> "*How Lambda plays into infrastructure automation and management, and how that will change the way we build infrastructure, and how we actually get to this infrastructure as software kind of world, that was always a big thing for me.*" -I6

This was confirmed in our survey (Figure 11). Most backend FaaS usage is for processing application data (76%) or running scheduled jobs (64%). About a third of respondents uses FaaS for processing monitoring or telemetry data.

If our interviewees use FaaS functions in the request cycle, they are often used as backend implementation of

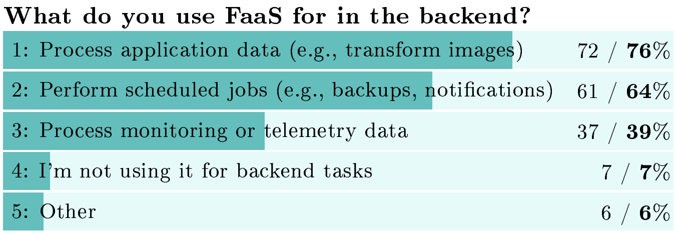

Figure 11: Usage of FaaS in the backend

a REST API. Two architectural styles have emerged in our study how to implement a REST API or service using FaaS: either using one function per service or REST resource, or using one function per HTTP operation. The former style leads to significantly larger functions which need to do considerable dispatching internally, while the latter style allows for fairly small and highly granular functions. Of course, this also means that the number of functions that need to be managed can potentially explode in the latter style.

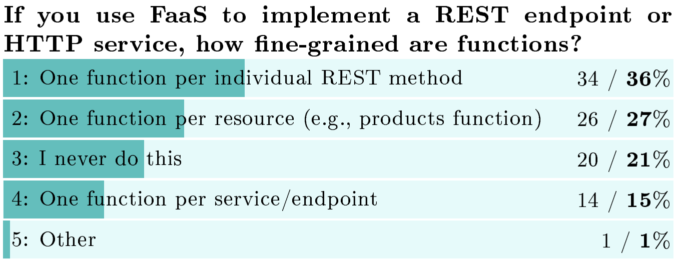

Figure 12: Granularity of FaaS functions

In our survey, using one FaaS function per REST method was reported as the most common architectural style (Figure 12). Presumably, this is because this style optimally fits the FaaS paradigm of tiny, stand-alone functions. Interestingly, 15% of survey respondents implement an entire service or endpoint using just one function, an architectural style that has emerged neither in our interviews nor in our analysis of grey literature.

One article in our literature study describes an interesting middle ground approach to use FaaS for user-facing applications. They use FaaS functions to generate static HTML, which can then be delivered using a CDN.

> "*Combining serverless APIs with static file hosting for site resources, e.g. HTML, JavaScript and CSS, means we can build entire serverless web applications.*" -A49

In general, we have observed that use cases which exhibit some of the following characteristics tend to be particularly suitable for FaaS:

- Applications which are *predominantly idle*. The strict pay-per-use model of FaaS makes for a particularly compelling cost case for such applications.
- Applications that face *bursty workloads* with stringent requirements regarding scalability and elasticity, and particularly applications that need to provide consistent Quality-of-Service in spite of intermittent slashdotting (i.e., short, unpredictable periods of orders of magnitude increased load).
- Applications that are *data or event stream driven*. Our interviews have shown that many interviewees considered, for example, IoT scenarios to be a natural fit for FaaS.
- Early application *prototyping*, where “getting it to run” in the shortest time possible is a primary concern.

Contrary, the following characteristics describe use cases for which FaaS may not be the right choice. Many of these characteristics are tightly linked to the inherent limitations and restrictions of current FaaS platforms:

- Applications that are inherently heavily *stateful*, such as database systems.
- Applications that comprise *long-running tasks*.
- Applications that have *high performance or real-time requirements*, where the performance impact through virtualization and, more importantly, Docker startup latency cannot be tolerated.
- Applications that have requirements with regards to *data locality*, i.e., which need to store data to the file system.

Within these general constraints, an interesting special case is parallel and high-performance computing. On the one hand, the high latency of FaaS is a problem for such performance-critical systems. On the other hand, the fact that FaaS gives a developer essentially unlimited cores to do distributed computation on makes the abstraction powerful for such applications. This has also been observed in A2.

> "*Parallelization with Lambda is as easy as executing as many functions as you need to cover the full depth and breadth of your dataset, in real time as it grows. It's like having a CPU with virtually infinite cores.*" -A2

Initial scientific computing frameworks that build on FaaS are already starting to gain traction. One example is the PyWren framework [21], which provides a parallel computing framework on top of various FaaS providers, most importantly AWS Lambda.

Purity. The second important distinction is between pure serverless, where all parts of the application are either externally hosted services (such as S3 or DynamoDB) or implemented on top of FaaS, and hybrid serverless, where this is only true for parts of the application. I2 was actually able to build an entire startup company without having to manage servers at all: the entire application is a pure serverless application, and all development tools (e.g., CI, bug tracker) are hosted external applications. One challenge of pure serverless applications is that authentication gets more difficult without a stateful component to hold user-authenticated sessions. This problem has also been observed by Adzic and Chatley [22].

In our interview study, most participants opted for a hybrid model. This was especially true for user-facing applications, which are often built using a stateful, end-user facing entry component (e.g., an nginx Docker container) and one or more stateless request handlers built using FaaS. However, others have chosen to just implement small parts of their application using FaaS:

> "*We are doing a lot of hybrid – for example you can just have your authentication on Lambda, and the rest of your code is on standard EC2.*" -I4

One reason for this may be that a common pattern for migrating web applications to serverless is to gradually "cut out" functionality from a monolith and re-implement or move them to serverless. As one interviewee puts it:

> "*I think a good rule is to start decomposing your application into smaller and smaller functions so that it's easier for you to essentially make cuts in the graph, and move the boundary of what belongs to one [function] and what to another.*" -I1

However, our survey responses do not support that many FaaS applications are are actually built in that fashion (Figure 13). 58% of respondents argued that they rarely or never gradually migrate existing appllications.

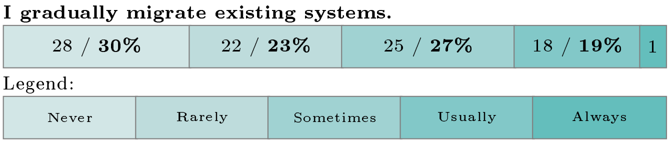

Figure 13: Migration to FaaS. Values without percentage sign refer to absolute numbers of responses.

Summary: FaaS is used for both, user-facing applications and backend utilities. However, if used within the user-facing request cycle, some technical challenges, most importantly slow tail latency, need to be overcome. FaaS is particularly suitable for event-driven applications or applications facing bursty workload that are idle a significant fraction of the time. This type of service is less suitable for applications comprising tasks that are long-running or have strong performance requirements. It is common to use FaaS in a hybrid model, i.e., in conjunction with traditional cloud services, such as virtual machines or containers.

4.3. Application Patterns

We now discuss common application patterns we have observed in our research.

Application size. Regarding the number and sizes of functions used in applications and systems, the interviewees to a large extent paint a similar picture. Typically, between 5 and 15 functions are combined to form or complement an application or system. Only two interviewees report larger systems containing multiple tens of functions. The survey respondents' answers support this finding, as can be seen in Figure 14. Nearly two thirds of respondents use between one to ten functions per application or system.

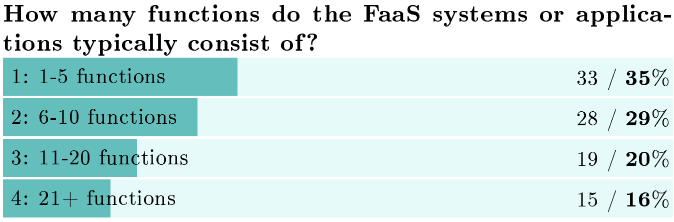

Figure 14: Number of functions per system or application

The functionality provided by individual functions is typically chosen to be relatively atomic. According to interviewees, functions are scoped to reflect a specific business need, or to correspond to one operation of a REST API.

Common patterns. One particularly interesting outcome that emerged from our interviews and literature review was that there are a number of recurring patterns in industrial FaaS solutions to deal with technical or conceptual limitations, such as too short maximum timeouts or the complexity of dealing with API gateways. We dub the patterns we observed *externalized state* (all state is stored in an external database), *routing function* (a central function is configured to receive all requests and dispatches them), *function chain* (one function calls another to increase timeouts), *function pinging* (functions are periodically "pinged" with artifical payload to prevent the FaaS platform from discarding all containers), and *oversized function* (functions are configured with excessive memory requirements purely to get deployed to a faster physical machine). We provide an overview of these patterns in Table 2. It is interesting to observe that, with the exception of *externalized state*, all of these (anti-)patterns can be seen as developers struggling with the inherent limitations of FaaS and working around them.

In Figure 15, we summarize how prevalently these patterns were used among our survey respondents. *Externalized state* is by far the most common, with two thirds of participants reporting that they at least sometimes use it. 17 respondents (18%) always make use of *externalized state* when building FaaS applications. The remaining patterns are less commonly used, which is unsurprising given that

Pattern Name	Addressed Problem	Description	Advantages	Disadvantages
Externalized State	Functions need to be stateless; any state saved in a function is not guaranteed to be available in subsequent function calls.	Developers externalize all function state in key/value data storage, such as Redis.	State is persisted reliably between function calls.	Persisting state to key/value stores induces latency overhead and requires additional programming effort; unless a hosted database service is used, an additional application component needs to be managed.
Routing Function	API gateways and routes are cumbersome to configure.	A central routing function is used to receive requests and forward them via function chaining (see below) to the appropriate other functions, based for example on the received payload.	API routes only need to be configured for one function, not for each of them.	Routing information is hidden in a function implementation rather than in the configuration.
Function Chain	Functions are restricted to a maximum call duration.	When the normal (i.e., non-erroneous) execution time of a function sometimes exceeds the maximum configurable timeout threshold (e.g., 5 minutes on AWS Lambda), developers may split the function into multiple parts which are then chained to effectively prolong the allowed call duration.	Allows to circumvent platform timeouts.	Essentially creates two deployment units for one logical service; splitting the function may be difficult for some applications; introduces strong coupling between the chained functions.
Function Pinging	Container cold start times lead to high latency for some requests (tail latency), especially after an idle period that causes the platform to stop all containers for a function.	Developers put functionality in place that periodically "pings" (triggers) the function even if no production workload is to be handled, to avoid containers being discarded by the platform.	Timeouts are avoided.	Periodic pings induce unnecessary costs; additional code needs to be developed, tested, and maintained to manage pinging.
Oversized Function	Current FaaS platforms do not provide mechanisms to directly select what type of CPU to execute the function on.	In some platforms, the only way to get a function deployed to a stronger physical machine is to increase the memory requirements for the function, even if the function does not actually require more memory.	Functions with higher memory requirements get deployed to physical machines with faster CPUs.	The function is billed significantly more for the higher memory allowance without actually using it.

Table 2: Common application patterns that developers use to address FaaS limitations.

these patterns are workarounds around somewhat niche constraints and problems rather than actual best practices. However, all four remaining patterns are reported to be used at least occassionally, indicating that the limitations imposed by the FaaS model are constraining developers at least sometimes.

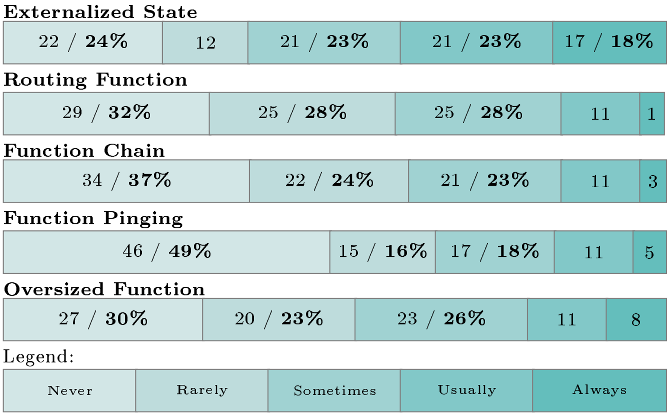

Figure 15: Prevalence of FaaS application patterns in practice. Values without percentage sign refer to absolute numbers of responses.

Summary: Current FaaS applications are commonly small, and often consist of 10 functions or less. Developers use various application patterns and workarounds to deal with the inherent limitations of current FaaS platforms.

4.4. Development Languages and Practices

We observe that there are a small number of programming languages that are commonly used to implement serverless solutions. In most cloud platforms, JavaScript, Python, and, to a lesser degree, Java are dominant. In Azure, C# / .NET unsurprisingly is of great importance. This can also be seen in our survey: from 96 respondents that answered this question, 74% (71) include either JavaScript (with or without Node.js), Python, or Java. Figure 16 breaks down the answers provided in the survey in detail. These results indicate on the one hand developers' preferences for writing functions, but are also determined by what languages are made available. For example, of the 8 responses marked "Other" in Figure 16, 5 include "Go", which became available in Google's FaaS offering only when our survey was already live.

Development challenges. Given the relative immaturity of the technology, it is unsurprising that we have observed

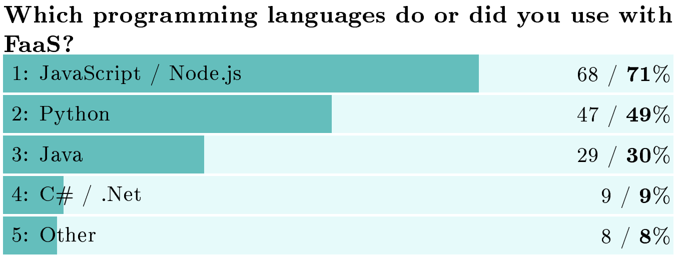

Figure 16: Programming languages used in FaaS applications

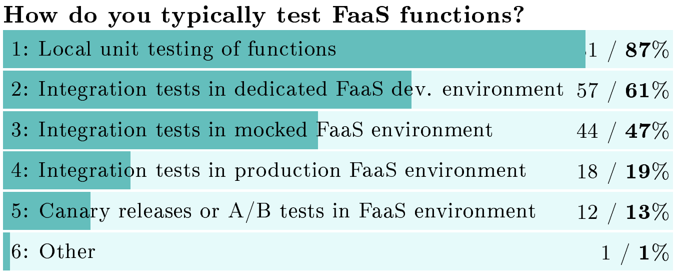

Figure 17: Testing approaches for FaaS functions

some challenges and grievances that even advanced practitioners currently struggle with. A major challenge is how to test functions. Due to the relatively small size and often low complexity of individual functions, they lend themselves well for unit tests, which can be performed locally. However, testing the integration of multiple functions or external services is harder, as local replication of the entire system is often not possible or hard to achieve.

> "*[...] it is not possible to replicate a serverless or cloud system on your local machine.*" -I6

One possible solution is to test functions directly in production, or in a dedicated development environment that is also hosted in the cloud. One common way to implement the latter is to have multiple separate accounts with the cloud provider, one for production and one for development and testing. Both approaches have the obvious disadvantage that they require developers to pay for test invocations the same as for production workload. In addition, we have observed that testing in actual production environments can have (negative) side-effects on production systems in some cases. One approach to deal with this issue is to perform canary releases or A/B testing, so that possible side-effects can be assessed for a small number of requests.

The testing practices used by survey respondents are illustrated in Figure 17. As expected, unit tests are commonly performed locally. When it comes to integration tests, dedicated development environments and mocked environments are more commonly used for testing than production environments (in general, or via canary releases or A/B tests). 23.7% (22) of respondents to this question perform tests in both, dedicated FaaS development environments and mocked FaaS environments, while only 16.1% (15) respondents test both in a dedicated environment (dev or mocked) and in a production environment.

Another core challenge is a lack of tooling and insufficient documentation. Tooling is especially desirable for the interviewees when it comes to deploying (sets of) functions, mapping events to functions (using, for example, API gateways to make functions accessible to HTTP requests), and monitoring and logging. At the same time, only a few of the available tools are actually used. With 79.7% of survey respondents using it, the Serverless framework is by far the most common among them. Contrary, the next frequently named library, Chalice, was only named by 11.6% of respondents. This indicates that existing tooling, with the exception of the Serverless framework, appear to not address the core challenges that developers currently face, or their existance is not yet widely known.

Further, interviewees perceive the available documentation to be insufficient, especially as best practices around FaaS still evolve. With regard to deploying functions, one interviewee remarked.

> "*(...) there are really not a whole lot of accepted patterns or state of the art solutions [how to deploy functions].*" -I1

Acknowledging these shortcomings, I1 and I3, who work for a major FaaS provider, emphasized that one of their current priorities is to improve documentation of their offerings.

> **Summary:** JavaScript, Python, Java, and C# are currently the dominant implementation languages for FaaS services. Current main development challenges for building FaaS solutions include (integration) testing applications, as well as a lack of good tooling and documentation.

4.5. Advantages and Challenges

We now discuss the major advantages and challenges when adopting FaaS. For the latter, we focus on more architectural and strategic difficulties rather than the more technical development challenges discussed in the previous section.

Advantages. We observe that there are three classes of advantages that motivate developers to use FaaS offerings, namely business-related, technical, and security-related advantages (see Figure 18). This is in line with previously reported results for cloud computing in general [2].

In terms of business advantages, the interviewees consider the pay-as-you-go pricing model typically used for FaaS as a big factor. It guarantees that costs correspond to usage volume, and especially avoids any cost when no usage occurs. In consequence, functions do not need to be

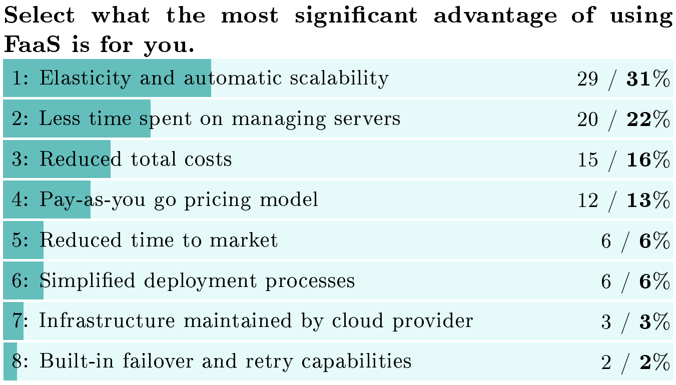

Figure 18: Significant advantages when working with FaaS services

"unprovisioned" or "undeployed" for cost reasons when little or no usage is expected. Further, as FaaS liberates developers from managing servers themselves, they can spend more time focusing on business features than would be possible when running applications on their own hardware or VMs. This observation holds even as FaaS requires developers to write some "management code", for example to deploy related sets of functions. Deployment processes for FaaS are generally considered to be simpler than for other types of cloud services, freeing up additional time.

> "*(...) a lot of integrations we have were built in let's say one week. And almost every integration with external services like SAP or CRM or service bus would have cost you a couple of sprints with a senior development team.*" -I9

Another cost-related aspect of using FaaS is that the per-request billing model of FaaS enables straight-forward deployment cost optimization. Essentially, every millisecond that a FaaS function executes faster directly translates into cost savings. This makes reasoning over whether certain code-level optimizations are "worth it" easier from a deployment cost point of view.

As for technical advantages, the interviewees consider the elastic scalability of functions to be a major technical advantage. If demand rises, FaaS providers horizontally scale up functions. Because functions are short running, they automatically scale down upon completing execution. Thus, FaaS provides elastic scalability without any setup efforts for users. The interviewees consider this to be a significant advantage, even though no interviewee yet reports to use FaaS in large-scale setups. Further, interviewees welcome FaaS' failover capabilities. If function executions fail, they can automatically be retried without explicit configuration or coding.

A somewhat less discussed advantage of serverless is that it can also increase the security of applications. FaaS shifts the burden of managing and maintaining machines to cloud providers, which are more likely to keep machines up-to-date with patches:

> "*Serverless practically eliminates the main source for successful exploits today — unpatched servers. Such servers are using binaries with known vulnerabilities, as they did not apply the latest security updates of those dependencies.*" -A8

Another aspect of this is that (Distributed) Denial-of-Service (DDoS) attacks become a billing rather than an availability issue. Where a traditional system may become unavailable under a DDoS attack, a FaaS-based solution scales up to deal with the load, incurring potentially significant additional costs. Whether this is preferable to a downtime is of course context- and application-dependent.

Challenges. In contrast to these advantages, the interviewees also raise challenges with FaaS or serverless computing in general. Some of these may be attested to the relative immaturity of FaaS offerings, while others are the result of the concepts and practices underlying FaaS. Figure 19 summarizes what our survey respondents consider to be significant challenges when using FaaS in order of how often the respective challenge has been selected (multiple selections were possible).

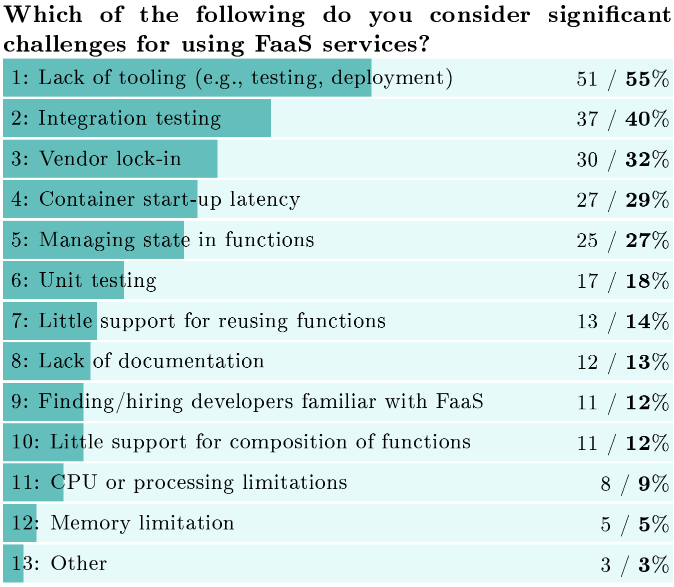

Figure 19: Significant challenges when working with FaaS services

Besides lack of tooling and difficulties of integration testing, vendor lock-in is also named as a pressing issue. All prominent FaaS providers offer custom feature sets and APIs. Further, due to the tight integration of FaaS with other cloud services, migrating a FaaS application to a different provider is rather difficult. Tail latency and handling state are further commonly named challenges. It is interesting to observe that hiring challenges, as well as lacking support for function composition (two aspects that have emerged prominently in our interviews and analysis of grey literature) are only considered to be a main challenge by 12% of survey respondents. We attribute this to the fact that many applications built using FaaS today appear to be fairly small and of young age, so both, function composition and hiring may not have become a pressing issue in today's industrial practice, which may of course change in the coming years.

Summary: FaaS offers advantages related to business (reduced costs and increased developer productivity), technical advantages (transparent elasticity and automatic failover), and advantages related to security (managed servers and, to some degree, resistance against DDoS attacks). The main challenges that developers face include a lack of good tooling, difficulties in integration testing, vendor lock-in, and performance problems.

4.6. Deployment Costs

While reduced costs are often named as a core driver underlying FaaS adoption, a deeper analysis revealed that this is indeed a hotly debated topic. While all of our interviewees have argued that FaaS is extremely cost-efficient, or even entirely free for many use cases, our survey of grey literature has drawn a different picture. In online articles or discussions on HackerNews, many practitioners have made negative experiences with unexpectedly high cloud bills. Partially this may be due to the highly intransparent billing model, which leaves users unclear about the "real" costs of the different parts of their deployment.

> "*(...) all of this is super hard to read on the bill (which function costs me the most and when? Gotta do your own advanced bill graphing for that) (...) Personally, I think it's all <retracted> ridiculous the amount of effort you have to spend into reading your own bill.*" -Commenter on HackerNews, A25

In addition, some FaaS users have observed that there are non-obvious "traps" that can lead to expensive mistakes. For instance, FaaS services foster fault tolerance by automatically retrying if processing a given trigger or event has failed [23]. However, in some cases, users have experienced endless loops where functions kept trying to process the same malformed input until manually terminated.

> "*Retries can crazily increase your bill if something goes wrong.*" -Commenter on HackerNews, A25

We speculate that these issues have not emerged from our interview study, as our interviewees are all experienced cloud developers who are unlikely to fall into such traps. In general, we have identified two clear fault lines between practitioners that tend to find FaaS cheap and those that do not.

Firstly, developers at startup companies tend to find FaaS cheap, while developers at companies with significant user base do not. This is due to extensive free tiers, which skew the observed costs for startup companies. However, for sustained, large-scale usage, many users have observed that price tags can become quite significant beyond the free tier, especially as users pay separately for (almost) every ecosystem service in use, including, for instance, API Gateway, Lambda, S3, and Step Functions in the case of AWS. Given that serverless applications are, as described in Section 4.1, by nature compositional, this can add up easily.

> "*I feel like the "Serverless is cheaper" thing here is being driven largely by the sorts of companies who are experimenting with it the most - small startups prematurely designing for scale.*" -Commenter on HackerNews, A9

Secondly, developers using FaaS as "glue code" tend to find FaaS cheap, while developers who use it for user-facing applications do not. In backend usage, absolute request counts are generally low, and this makes the per-invocation pricing model much cheaper than paying for an, even small, container or virtual machine. However, as soon as FaaS functions are invoked for each user request, invocation counts increase, and so do costs. It is not surprising that, once FaaS functions are actually executing continuously, simply paying for a VM or container upfront will be cheaper than paying for computation on a per-request basis. An additional factor to consider here is that particularly API Gateway, an AWS service that is virtually mandatory to use in conjunction with Lambda for user-facing applications, is reputed to be particularly expensive.

> "*The power of serverless is that it really allows you when you don't have traffic or your system is not busy that you don't consume many resources.*" -I11

In our survey, a majority of 93% of respondents has argued that they find FaaS cheap or do not care about costs at all (Figure 20). However, we caution the reader that this may be due to the self-selection bias: developers who find FaaS expensive are less likely to voluntarily participate in an online survey on the topic.

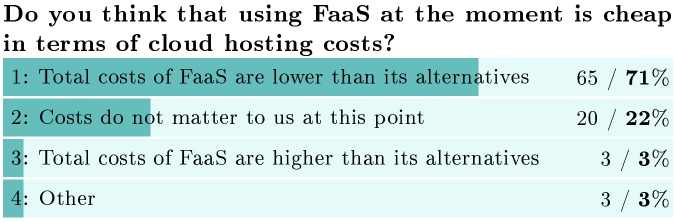

Figure 20: Perceived costs for using FaaS versus alternatives

Summary: While FaaS services are indeed often cost-effective, this may be due to generous free tiers and low-utilization use cases. Using additional services can easily increase the total deployment costs for an application. Highly intransparent billing models complicate cost planning.

5. Implications

The results of our study reveal implications for FaaS providers, consumers (i.e., application or system developers), and researchers, which we address in the following subsections.

5.1. What's next for providers

FaaS offerings are still relatively young, with initial releases of AWS Lambda in November 2014, IBM Cloud Functions / OpenWhisk in February 2016, Azure Functions in March 2016, and (the beta of) Google Cloud Functions in March 2017. Thus, as of now, significant new capabilities are routinely added to the offerings, including support for new languages, integration with other Cloud services, or tooling for testing or composition.[11] Our study reveals the need for various further improvements. Interestingly, we observe that interviewees that represent platform providers (e.g., I1 or I3) expect future serverless solutions to look quite radically different from what we have today, while interviewees that represent serverless users largely expect "faster horses", i.e., incremental improvements to existing solutions. These more incremental improvements fall into two categories, *better tooling* and *lifted restrictions.*

Provide better tools. Given the severely limited state of current FaaS tooling, it is unsurprising that most interviewees hope and expect that better and more sophisticated tool chains for developing serverless solutions will become available in the future. Most importantly, this relates to better means for testing and debugging, such as tools for record-and-replay testing of cloud events, published cloud images that allow users to more accurately reproduce cloud behavior locally, and debuggers that can connect directly to functions executing in the cloud. Azure already provides many of these features through their integration with Visual Studio. Our study participants largely expect other providers to follow suit.

Lift restrictions. Further, many interviewees expect that providers will slowly lift the current fundamental restrictions of the model (e.g., related to state, execution time limits, etc.), or at least provide better technology to work around them. Most importantly, many interviewees expect that providers will develop support for stateful functions, or provide an integrated way for handling specific types of state through the platform. In the wider ecosystem, there are already new database designs such as FaunaDB which address especially the latency issue of external state binding. While selected statefulness within functions will impact the performance and providers would adapt the pricing accordingly, the offer may still be compelling to developers due to being able to deploy more conventionally designed code as functions. I3 recognizes the need to support connection pooling or caching in end-user facing Web applications:

> "*A lot of Web systems use in-memory caches, there are things like Redis, you can use those in conjunction with serverless, but as far as in-memory caches go, given that you are not guaranteed a persistent memory ...*" -I3

From our interviewees representing providers and tool builders (I1, I3, and I6), a different set of future developments has emerged. Those interviewees expect that future updates will quite fundamentally change the serverless landscape, particularly related to *function reuse and ecosystems*, *higher-level abstractions*, *serverless for memory*, and *serverless on the edge.*

Foster function reuse and ecosystems. As indicated in Section 4.1, there is currently little infrastructure in place to enable a structured reuse of functions between cloud tenants, or even between individual applications of the same tenant. I3 has argued that future FaaS solutions will naturally include a function ecosystem, where existing external functions, even for specialized tasks, can be discovered through search engines, catalogues, or function packages. Existing function marketplaces such as the Amazon Serverless Application Repository are clear steps into this direction, but currently seem to have problems gaining traction. Better marketplaces will increase the reuse of functions, and make serverless application development even more compositional in nature. These function ecosystems may operate similarly to current-day open source ecosystems, such as NPM [24].

Provide higher-level abstractions. Both, FaaS users and developers of FaaS platforms, consider the development model provided by today's platforms, languages, and services to be quite rudimentary. While initial steps to provide something akin to a "programming language" for serverless applications have already been taken (e.g., the AWS Step Functions service, which allows to describe applications as workflows of FaaS functions), some of our interviewees see these merely as the first step:

> "*(...) format for Step functions is essentially like an assembly language (...) Nobody wants to write JSON format for a state machine that has hundreds of states.*" -I1

There is an expectation that, in the future, higher-level abstractions will be developed and current technologies, such as Lambda and Step Functions, will become mere deployment platforms, which are not intended to be programmed directly. Instead, we may see the rise of general-purpose or domain-specific languages that compile into a format that is deployable on serverless platforms:

> "*We will have languages that compile something that you can execute in a serverless platform.*" -I1

[11] See change logs for AWS Lambda (`https://docs.aws.amazon.com/lambda/latest/dg/history.html`) or Google Cloud Functions (`https://cloud.google.com/functions/docs/release-notes`) as well as individual feature announcements, for example `https://www.ibm.com/blogs/bluemix/2017/10/serverless-composition-ibm-cloud-functions/`.

Provide “serverless for memory”. As discussed, we have by now on-demand serverless technologies for all common computing resources, with the exception of memory. Memory can still only be acquired in conjunction with compute time, either through IaaS services such as EC2 or ECS, or through FaaS services such as Lambda. I3 expects that we will also see the development of analogous services for memory. Realistically, providers could exploit dynamic allocation (memory ballooning) techniques for vertical scaling in hypervisors and container engines to address this perceived need if it fits their business model.

Provide “serverless at the edge”. An already ongoing development is to provide support for function execution not only in the cloud, but also directly on edge devices. I5 sees this as the future “killer app” for serverless and FaaS:

> "*Actually being able to execute code in CloudFront on the edge, this is pretty hot.*" -I5

From a business perspective, established edge and content delivery networks being upgraded with function processing capabilities may become a game-changer given that their installation base is still vastly larger than the one of major FaaS providers, but so far no concrete plans in this direction are known to the authors.

5.2. What’s next for developers

For developers, the current state of FaaS appears to be a mixture of blessing and curse: on the one hand, FaaS offers tremendous potential for enabling automatic scaling and elasticity, reducing efforts for deployments and server management, and (often) lowering deployment and development costs. On the other hand, current limitations of FaaS offerings and related tooling as well as the need to adopt a corresponding mental model present challenges for developers to overcome.

Question whether your use case suits FaaS. Our survey finds FaaS being used both for backend applications as well as for end-user facing ones - both on their own or in combination with other services. However, many interviewees are skeptical of the second use-case, pointing for example to response time and tail latency limitations and challenges in dealing with state across requests (for example to persist sessions). When considering the use of FaaS, developers should carefully assess their requirements for handling state, expected workloads and runtimes, as well as non-functional requirements regarding, for example, memory, processing power, response times, and cost - both for using FaaS as well as for alternative solutions.

Embrace the mental model for FaaS. This model fosters the composition of applications and systems from small services implemented as functions and integration with other cloud services. Developers should be encouraged by our study’s finding that training experienced developers may be easy because of the closeness of FaaS to existing technologies and methods, and by the hope that junior developers may easily adopt this mental model as a default one (see Section 4.1).

Choose carefully which provider to use. When selecting the concrete FaaS provider to use, developers should consider on the one hand possibilities and required efforts for integrating with other cloud services, which is a common pattern for FaaS usage. If they already rely on many services from one provider, integrating FaaS may be easier and familiar tools may be (re-)used. On the other hand, developers need to carefully asses the vendor lock-in that is prevalent in virtually all existing offerings.

Anticipate your tooling needs - especially when it comes to testing. Both, interviewees and survey respondents, note that the current lack of tooling around FaaS is a central challenge. One area where this is especially obvious is testing, where most respondents rely on local unit tests. Integration testing, on the other hand, is hard to achieve, as emulation platforms are lacking and tests in production may have side-effects and cost money. Developers should consider likely testing needs before developing FaaS-based applications and systems, taking into account the specific support their FaaS provider offers.

5.3. What’s next for researchers

For researchers, three main implications result from our work: first, systems and software engineering research can address the identified challenges for providers and developers, which we discussed in the previous subsections. Second, empirical research, like this paper, is required to assess how FaaS is used and how provided services evolve. Third, as the amount of research about FaaS stacks up, meta research is required to contextualize, relate, and summarize findings.

Improve and extend FaaS-related software. The impact of research aiming to improve FaaS systems increases if the targeted systems are actually used by developers in practice. According to our findings, FaaS is now at the core of a rapidly evolving serverless computing ecosystem. Yet, many of the tools are first-generation prototypes with a lot of potential for optimization. Thus, our recommendation to researchers is to go beyond isolated contributions and instead consider holistic serverless application use cases involving both runtime environments and developer tooling as well as hybrid serverless/conventional application development.

Study the use of FaaS offerings. As this study aims to do, empirical research can highlight various aspects of current FaaS offerings, including strengths, weaknesses, and challenges from a software engineering point of view, economic incentives or barriers to adoption, or implications on system and application designs. While we see our study as a starting point, we have focused on breadth rather than a deep study of, for instance, the feature set development and the affected software developer behaviour over time. We see such follow-up studies as highly valuable contributions to guide software developers with building serverless solutions.

Establish the "real" expenses of using FaaS. As we found in our study, the question of whether FaaS is more cost-effective than other cloud services, and for which applications, is not easy to answer. Researchers should conduct case studies with companies to get a holistic view of the various costs and benefits involved, including development costs, increased business agility, and pay-per-use versus pay-per-time. As it currently stands, developers struggle to make an informed decision about these aspects, and are unable to distinguish cloud provider marketing from objective facts.

Align with industrial practice. Furthermore, our study approach directly leads to follow-up research possibilities. Of particular interest from a perspective of academic impact would be the matching of technological choices taken by developers with those taken by researchers. Anecdotally, we observed a mismatch, especially concerning advanced topics such as composition and testing of functions, both of which are sparsely covered by current scientific literature but appear to be immensely important in practice according. We propose to use our results as a an indication of which areas are actually in need for future conceptual improvements.

6. Related Work

FaaS platforms have been subject to a growing body of published findings. The covered fields include cloud functions in scientific computing [25, 26] and edge computing, further application domains such as data analytics, and economic aspects. For a broad overview on general serverless computing research activities and literature, we refer to the summaries by the participants of the International Workshop on Serverless Computing [27] and the Serverless Literature Dataset [28]. A more general survey of software development practices for the cloud has been presented by Cito et al. [2]. The general themes reported therein (e.g., adoption being driven by a combination of business and technical factors) have similarly emerged from our study.

There is a strong relationship between our study subject and the related research area of development of software based on microservices [29]. An older systematic mapping study by Pahl and Jamshidi gives a good overview over this related area [30]. Balalaie et al. report on a case study of migrating to microservices [18]. They argue that incremental migration and a strong emphasis on hosted services and re-use is critical to the success of microservices projects, which is in line with our findings. Mazlami et al. study various automated metrics for slicing out microservices from a monolith, which can potentially also be applied to serverless and FaaS migration [5]. Workflow and orchestration systems appear for cloud functions, as introduced in Section 2.5, but also for other microservice architectures, such as Beethoven for Spring Cloud components [31]. We are not aware of a formal elaboration on the degree of matching between microservices and predominant cloud functions models, but the selected works, our study findings and first academic works dedicated to this matching [32] suggest that most FaaS services are widely seen as microservice implementation technology.

Considering the published works related to serverless and FaaS, few studies focus on the development perspective compared to articles about runtime and more technological questions. The related work can thus be divided into three categories: platform-level development support, methods and tools, and software architecture.

Development support has been analysed for the visualisation of serverless application logs with OpenWhisk by Chang and Fink [33]. The authors note that their approach, called Witt, facilitates program understanding and automated documentation generation. Baldini et al. point out flaws in function composition which can be seen as advice for developers [8].

Development methods and tools have been explored related to sample applications such as chatbots [34] and related to code decomposition and transformation [35]. The results show that automated conversion of legacy code to cloud functions is not practical beyond toy examples and therefore the need to craft functions manually on the code level remains.

The impact on software architectures has been studied by Adzic and Chatley with AWS Lambda [22] and by Sampé et al. with Zion, an environment to execute data-driven functions [36]. These topics, which partially touch on the runtime, receive slightly more attention by fellow researchers, as also evidenced by domain-specific architecture analysis by Crane and Lin [37].

No peer-reviewed empirical works about software development in this domain are known to us. The development perspective is more prominent in industry surveys, the first of which have been conducted during our study period. SlashData, for example, has found out from more than 21000 developers that serverless adoption is increasing and that AWS Lambda is the leading platform with 44% share [38]. A CNCF survey among 550 community developers arrives at different conclusions with 70% Lambda adoption [39]. Yet these statistics are single-method only and focus on market numbers rather than development processes and requirements, and are furthermore openly disputed in developer media such as The Register and The New Stack [40]. These limitations show that there is a need for a more profound study design, which we contribute in our research.

7. Conclusions

In this paper, we presented results from the first systematic study of serverless and FaaS development. We conducted a mixed-method study that combined qualitative and exploratory elements with a structured, quantitative survey. Our results are based on the analysis of 12 interviews with professional developers who use, or have used, FaaS in the past, 50 frequently-discussed online articles or blogs related to the topic, and 182 survey responses.

Our main findings are that successfully adopting serverless requires a mental model that is different, where systems are primarily constructed by composing externally provided, pre-existing services. FaaS often acts as the "glue" that allows these services to interact. Many developers use FaaS particularly for backend tasks, but building end user facing applications is possible as long as developers remain acutely aware of the high tail latency of FaaS and other technical restrictions. We have observed five prevalent application patterns (e.g., pinging functions to keep containers "warm", chaining functions to circumvent maximum execution time limitations), indicating that some developers struggle with the technical restrictions inherent in the model. FaaS offers advantages related to both, business factors (e.g., potentially reduced costs and increased development speed) and technical factors (e.g., auto-scaling, automated failover, potentially increased security). However, the maturity and availability of tooling, particularly related to testing and deployment, remains a barrier to entry. Finally, limited support for function sharing and the absence of a service ecosysystem is seen as a challenge.

Our results have implications for service providers, developers, and researchers. For providers, our results indicate that many developers struggle the lack of good tooling and the limitations inherent in the platform. Further, our study shows a need for higher-level abstractions as well as more sophisticated means to foster re-use of functions (potentially across tenants). For developers, our study has shown that adopting the right mind set is crucial for successfully adopting FaaS. Further, developers need to carefully evaluate cloud providers up-front, also because all current offerings imply significant lock-in. For researchers, our study opens up the potential, as well as the need, for follow-up research, particularly as current research does not appear to be well-aligned with existing woes of practitioners.

Acknowledgements

First of all, we would like the thank the twelve interviewees, for kindly taking the time to talk to us and share their knowledge, and the many respondents of our online survey. Furthermore, we would like to thank Jim Laredo for providing feedback on our questionnaire.

References

[1] V. S. Sharma, S. Sengupta, S. Nagasamudram, MAT: A migration assessment toolkit for paas clouds, in: 2013 IEEE Sixth International Conference on Cloud Computing, Santa Clara, CA, USA, June 28 - July 3, 2013, 2013, pp. 794–801. doi:10.1109/CLOUD.2013.92.
URL https://doi.org/10.1109/CLOUD.2013.92

[2] J. Cito, P. Leitner, T. Fritz, H. C. Gall, The making of cloud applications: An empirical study on software development for the cloud, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 393–403. doi:10.1145/2786805.2786826.
URL http://doi.acm.org/10.1145/2786805.2786826

[3] L. Bass, I. Weber, L. Zhu, DevOps: A Software Architect's Perspective, 1st Edition, Addison-Wesley Professional, 2015.

[4] M. Malawski, K. Figiela, A. Gajek, A. Zima, Benchmarking heterogeneous cloud functions, in: Euro-Par 2017: Parallel Processing Workshops - Euro-Par 2017 International Workshops, Santiago de Compostela, Spain, August 28-29, 2017, Revised Selected Papers, 2017, pp. 415–426. doi:10.1007/978-3-319-75178-8_34.
URL https://doi.org/10.1007/978-3-319-75178-8_34

[5] G. Mazlami, J. Cito, P. Leitner, Extraction of microservices from monolithic software architectures, in: Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), IEEE, 2017. doi:2017WebServices(ICWS),.

[6] V. Garousi, M. Felderer, M. V. Mäntylä, The need for multivocal literature reviews in software engineering: Complementing systematic literature reviews with grey literature, in: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering, EASE '16, ACM, New York, NY, USA, 2016, pp. 26:1–26:6. doi:10.1145/2915970.2916008.
URL http://doi.acm.org/10.1145/2915970.2916008

[7] W. Hummer, F. Rosenberg, F. Oliveira, T. Eilam, Testing idempotence for infrastructure as code, in: ACM/IFIP/USENIX Middleware Conference, Springer, 2013, pp. 368–388.

[8] 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, Onward! 2017, Vancouver, BC, Canada, October 23 - 27, 2017, 2017, pp. 89–103. doi:10.1145/3133850.3133855.
URL http://doi.acm.org/10.1145/3133850.3133855

[9] V. R. Basili, H. D. Rombach, The TAME Project: Towards Improvement-Oriented Software Environments, IEEE Transactions on Software Engineering 14 (6) (1988) 758–773. doi:10.1109/32.6156.

[10] L. Bratthall, M. Jørgensen, Can you trust a single data source exploratory software engineering case study?, Empirical Software Engineering 7 (1) (2002) 9–26. doi:10.1023/A:1014866909191.
URL https://doi.org/10.1023/A:1014866909191

[11] G. Schermann, J. Cito, P. Leitner, U. Zdun, H. C. Gall, We're doing it live: A multi-method empirical study on continuous experimentation, Journal of Information and Software Technology nn (2018) nn, to appear.
URL http://www.ifi.uzh.ch/dam/jcr:01d34060-29fb-472e-a116-bd26c3b49f67/IST_preprint.pdf

[12] L. Singer, F. Figueira Filho, M.-A. Storey, Software engineering at the speed of light: How developers stay current using twitter, in: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, ACM, New York, NY, USA, 2014, pp. 211–221. doi:10.1145/2568225.2568305.
URL http://doi.acm.org/10.1145/2568225.2568305

[13] T. Barik, B. Johnson, E. Murphy-Hill, I heart hacker news: Expanding qualitative research findings by analyzing social news websites, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York, NY, USA, 2015, pp. 882–885. doi:10.1145/2786805.2803200.
URL http://doi.acm.org/10.1145/2786805.2803200

[14] V. Garousi, M. Felderer, M. V. M√§ntyl√§, Guidelines for including grey literature and conducting multivocal literature reviews in software engineering, Information and Software Technologydoi:https://doi.org/10.1016/j.infsof.2018.09.006.
URL http://www.sciencedirect.com/science/article/pii/S0950584918301939

[15] P. Leitner, E. Wittern, J. Spillner, W. Hummer, Survey and interview data from mixed-method survey of serverless computing and function-as-a-service software development in industrial practice, the study is currently under review and not yet published. (May 2018). doi:10.5281/zenodo.1252820.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati,

A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store, in: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles, SOSP '07, ACM, New York, NY, USA, 2007, pp. 205–220. doi:10.1145/1294261.1294281.
URL http://doi.acm.org/10.1145/1294261.1294281

[17] T. Erl, R. Puttini, Z. Mahmood, Cloud Computing: Concepts, Technology & Architecture, 1st Edition, Prentice Hall Press, Upper Saddle River, NJ, USA, 2013.

[18] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: Migration to a cloud-native architecture, IEEE Softw. 33 (3) (2016) 42–52. doi:10.1109/MS.2016.64.
URL https://doi.org/10.1109/MS.2016.64

[19] A. L. Lemos, F. Daniel, B. Benatallah, Web service composition: A survey of techniques and tools, ACM Computing Surveys 48 (3) (2015) 33:1–33:41. doi:10.1145/2831270.
URL http://doi.acm.org/10.1145/2831270

[20] M. Plauth, L. Feinbube, A. Polze, A performance survey of lightweight virtualization techniques, in: F. De Paoli, S. Schulte, E. Broch Johnsen (Eds.), Service-Oriented and Cloud Computing, Springer International Publishing, 2017, pp. 34–48.

[21] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: Distributed computing for the 99%, in: Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, ACM, New York, NY, USA, 2017, pp. 445–451. doi:10.1145/3127479.3128601.
URL http://doi.acm.org/10.1145/3127479.3128601

[22] G. Adzic, R. Chatley, Serverless computing: Economic and architectural impact, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, ACM, New York, NY, USA, 2017, pp. 884–889. doi:10.1145/3106237.3117767.
URL http://doi.acm.org/10.1145/3106237.3117767

[23] W. Hummer, C. Inzinger, P. Leitner, B. Satzger, S. Dustdar, Deriving a unified fault taxonomy for event-based systems, in: 6th ACM International Conference on Distributed Event-Based Systems (DEBS), 2012, pp. 167–178.

[24] A. Serebrenik, T. Mens, Challenges in software ecosystems research, in: Proceedings of the 2015 European Conference on Software Architecture Workshops, ECSAW '15, ACM, New York, NY, USA, 2015, pp. 40:1–40:6. doi:10.1145/2797433.2797475.
URL http://doi.acm.org/10.1145/2797433.2797475

[25] V. Ishakian, V. Muthusamy, A. Slominski, Serving deep learning models in a serverless platform, ArXiv e-printsarXiv:1710.08460.

[26] J. Spillner, C. Mateos, D. A. Monge, FaaSter, Better, Cheaper: The Prospect of Serverless Scientific Computing and HPC, in: 4th Latin American Conference on High Performance Computing (CARLA), Vol. 796 of CCIS, Colonia del Sacramento, Uruguay, 2017, pp. 154–168.

[27] G. C. Fox, V. Ishakian, V. Muthusamy, A. Slominski, Status of serverless computing and function-as-a-service(faas) in industry and research, CoRR abs/1708.08028. arXiv:1708.08028.
URL http://arxiv.org/abs/1708.08028

[28] J. Spillner, Serverless literature dataset (Feb. 2018). doi:10.5281/zenodo.1175424.
URL https://doi.org/10.5281/zenodo.1175424

[29] G. Buchgeher, M. Winterer, R. Weinreich, J. Luger, R. Wingelhofer, M. Aistleitner, Microservices in a small development organization - an industrial experience report, in: Software Architecture - 11th European Conference, ECSA 2017, Canterbury, UK, September 11-15, 2017, Proceedings, 2017, pp. 208–215. doi:10.1007/978-3-319-65831-5_15.
URL https://doi.org/10.1007/978-3-319-65831-5_15

[30] C. Pahl, P. Jamshidi, Microservices: A systematic mapping study, in: Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2, CLOSER 2016, SCITEPRESS - Science and Technology Publications, Lda, Portugal, 2016, pp. 137–146. doi:10.5220/0005785501370146.
URL https://doi.org/10.5220/0005785501370146

[31] D. M. Barbosa, R. Gadelha, P. H. M. Maia, L. S. Rocha, N. C. Mendonça, Beethoven: An event-driven lightweight platform for microservice orchestration, in: Software Architecture - 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24-28, 2018, Proceedings, 2018, pp. 191–199. doi:10.1007/978-3-030-00761-4_13.
URL https://doi.org/10.1007/978-3-030-00761-4_13

[32] 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 2018, Orlando, FL, USA, April 17-20, 2018, 2018, pp. 159–169. doi:10.1109/IC2E.2018.00039.
URL https://doi.org/10.1109/IC2E.2018.00039

[33] K. S. Chang, S. J. Fink, Visualizing serverless cloud application logs for program understanding, in: 2017 IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2017, Raleigh, NC, USA, October 11-14, 2017, 2017, pp. 261–265. doi:10.1109/VLHCC.2017.8103476.
URL https://doi.org/10.1109/VLHCC.2017.8103476

[34] M. Yan, P. C. Castro, P. Cheng, V. Ishakian, Building a chatbot with serverless computing, in: Proceedings of the 1st International Workshop on Mashups of Things and APIs, MOTA@Middleware 2016, Trento, Italy, December 12-13, 2016, 2016, pp. 5:1–5:4. doi:10.1145/3007203.3007217.
URL http://doi.acm.org/10.1145/3007203.3007217

[35] J. Spillner, Practical tooling for serverless computing, in: Proceedings of the 10th International Conference on Utility and Cloud Computing, UCC 2017, Austin, TX, USA, December 5-8, 2017, 2017, pp. 185–186. doi:10.1145/3147213.3149452.
URL http://doi.acm.org/10.1145/3147213.3149452

[36] J. Sampé, M. S. Artigas, P. G. López, G. París, Data-driven serverless functions for object storage, in: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, Las Vegas, NV, USA, December 11 - 15, 2017, 2017, pp. 121–133. doi:10.1145/3135974.3135980.
URL http://doi.acm.org/10.1145/3135974.3135980

[37] M. Crane, J. Lin, An exploration of serverless architectures for information retrieval, in: Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval, ICTIR 2017, Amsterdam, The Netherlands, October 1-4, 2017, 2017, pp. 241–244. doi:10.1145/3121050.3121086.
URL http://doi.acm.org/10.1145/3121050.3121086

[38] C. Voskoglou, Developer Economics: State of the Developer Nation 14th Edition, Developer Economics website: https://www.developereconomics.com/reports/developer-economics-state-of-the-developer-nation-14th-edition.

[39] S. Conway, Cloud Native Technologies Are Scaling Production Applications, CNCF website: https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/ (December 2017).

[40] M. Asay, AWS won serverless – now all your software are kinda belong to them, The Register website: https://www.theregister.co.uk/2018/05/11/lambda_means_game_over_for_serverless/ (May 2018).

Appendix

The appendix contains essential data about all study methods according to our mixed-method approach. The complete curated data set is available online [15].

Interview Guidelines

Here we briefly list the questions that guided our interviews. Adopting a semi-structured method, we have not asked every participant each (sub-)question, but followed the flow of conversation. We also collected basic demographic information (summarized in Table 1).

Introduction

- Have you used serverless in the past? How often? Which services / technologies?
- How important was serverless in this process? Was it just a small side-aspect, or a core technology in the project?
- Can you explain the rationale for choosing serverless? What would have been alternatives, and why were they not chosen?
- What services have you been using?

Architecting with Serverless

- How do you choose what goes into a function? How do you split business logic between functions?
- Which features are particularly suitable to be implemented in functions? Which are not?
- Is there a difference between architecting serverless apps to “regular” (micro-)services, or is it just a different way of building a service?
- How large are those functions in your applications, typically? How many do you have?
- Are your services typically event-driven, or are they accessed over e.g., a REST interface?
- Are they standalone components, or do you often end up with functions that are tightly coupled to other parts of the application (e.g., other functions)?

Advantages and Disadvantages

- What are the major advantages that you see in serverless? Are there any besides not having to manage servers?
- Specifically, what about: Reusability; Scalability / Elasticity; Costs; Others?
- What are major disadvantages that you currently see?
- Specifically, what about: Testing functions; Keeping track of control flow in the application; Dealing with state across functions; Tool support; Integration into standard development processes and frameworks (e.g., CI / CD pipelines); Others?

Wrap-Up

- Do you see serverless as a fad, or do you think it will be here to stay?
- Is there something that conceptually needs to change before serverless will be ready for prime-time?

Questionnaire Form

Here we provide the outline to the seven groups of questions asked in the online survey.

Demographics

- How much experience in IT in general do you have?
- How much experience in programming do you have?
- How much experience do you have with using cloud services, such as AWS EC2, Lambda, or Heroku?
- Which of the following cloud services have you, or members of your team you work with closely, been using?*
- What's your role in your team?

Terminology

- Select the most fitting definition: To me, the term “serverless” describes...

Application Architecture

- Have you used a FaaS service in the past?
- How many months have you used a FaaS service?
- How many different projects have you used a FaaS service for?
- Which programming languages do / did you use in a FaaS context? (which languages are the actual functions implemented in)
- Which kinds of applications do / did you use FaaS mostly for?
- What do you use FaaS for in end-user facing applications?
- If you use FaaS to implement a REST endpoint or HTTP service, how fine-grained are functions?
- What do you use FaaS for in the backend?
- Which other kinds of cloud services are you using in conjunction with FaaS?
- When I build a system with FaaS, typically ...
- I use serverless functions to wrap library code, without extending it significantly.
- I reuse individual cloud functions (deployed functions used across many services, applications, teams, departments).
- I compose cloud functions (e.g., use one function that itself calls other functions).
- What number of functions do the FaaS systems / applications that you build typically consist of?

FaaS Development Practices and Patterns

- I build a routing function that acts as the central entry point and dispatches requests to other functions.
- I externalize state between FaaS calls in a key/-value datastore, such as Redis.
- I chain function calls to increase or work around timeouts.
- I ping functions to keep containers warm.
- I select more memory for my functions than required because I want to get a stronger CPU.
- I build FaaS systems or applications from scratch.
- I build FaaS systems or applications by gradually migrating an existing system or application.
- Are there any other, similar recurring patterns of FaaS development that were not listed previously? Please describe them briefly below.
- How do you typically test FaaS functions?
- Which of the following third-party libraries have you used for deploying and interacting with FaaS services?

Mental Model

- Building FaaS applications requires a different mindset or mental model than building an application with EC2 or Docker.
- The mental model behind FaaS is difficult to grasp for developers.
- Novice developers may actually have an easier time getting started with FaaS as they do not need to "unlearn" so much previous knowledge about cloud application development.
- Knowledge or experience with which of the following techniques or practices is helpful to understand the mental model behind FaaS better?
- Feel free to comment on the mental model when using FaaS.

Advantages and Challenges

- Select what the most significant advantage of using FaaS is for you.
- Select which of the following you consider significant challenges for using current FaaS services.
- Do you think that using FaaS at the moment is cheap in terms of cloud hosting costs?

The Future

- Do you plan to use or continue to use a FaaS service in the future?
- Why not?

ID	Author	Article Title and Link	Published
A1	Mike Roberts	Serverless Architectures	04 August 2016
A2	Matt Wood	Serverless Map/Reduce	03 November 2016
A3	Paul Kinlan	Serverless Data Sync in Web Apps with Bit Torrent	June 14 2016
A4	Alex Ellis	Your Serverless Raspberry Pi cluster with Docker	20 August 2017
A5	Pete Johnson	30K Page Views for $0.21: A Serverless Story	16 August 2016
A6	Kevin Vandenborne	Serverless: A lesson learned. The hard way.	n/a
A7	Bryan Liston	Going Serverless: Migrating an Express Application to Amazon API Gateway and AWS Lambda	04 October 2016
A8	Guy Podjarny	Serverless Security implications—from infra to OWASP	19 April 2017
A9	Dmitri Zimine	Serverless is cheaper, not simpler	28 August 2017
A10	Marc Cuva	Writing a cron job microservice with Serverless and AWS Lambda	30 January 2017
A11	Joe Stech	Going Serverless: AWS and Compelling Science Fiction	11 October 2016
A12	David Wells	How To Schedule Posts for Static Site Generators (Jekyll, Hugo, Phenomic etc.)	07 March 2017
A13	Obie Fernandez	What if we didn't need an app server anymore?	30 November 2015
A14	Kevin Deisz	Serverless Slackbots Powered by AWS	08 March 2016
A15	Ron Miller	AWS Lambda Makes Serverless Applications A Reality	24 November 2015
A16	Charity Majors	WTF IS OPERATIONS? #SERVERLESS	31 May 2016
A17	Rafal Gancarz	Serverless Takes DevOps to the Next Level	28 April 2017
A18	Ryan Kelly	Going Serverless with AWS Lambda and API Gateway	07 August 2016
A19	Andrew Walker	Google not Amazon. Make fantastic savings in a serverless world	19 July 2017
A20	Todd Hoff	Is Serverless The New Visual Basic?	15 May 2017

Table 3: Complete list of analyzed articles (results for search keyword "serverless").

Complete Article List

Tables 3 and 4 provide a full list and hyperlinks to all articles analyzed as part of the multi-vocal literature review. All hyperlinks have been visited on April 27, 2018. Note that article A42 has been deleted by the platform Medium and is not available any longer.

ID	Author	Article Title and Link	Published
A21	"Flynn"	3 Reasons AWS Lambda Is Not Ready for Prime Time	09 February 2016
A22	n/a	Dirt Cheap Recurring Payments with Stripe and AWS Lambda	05 May 2017
A23	Larry Land	The future is now, and it's using AWS Lambda	16 May 2015
A24	Jim Pick	Introducing lambda-comments	05 May 2016
A25	n/a	Ask HN: How was your experience with AWS Lambda in production?	n/a
A26	Nick Malcolm	Using AWS Lambda to call and text you when your servers are down	05 December 2016
A27	Sumit Maingi	Best practices – AWS Lambda function	02 March 2017
A28	Sumit Maingi	.Net Core Web API on AWS Lambda Performance	13 Februar 2017
A29	Vineet Gopal	Powering CRISPR with AWS Lambda	25 September 2015
A30	Matthew Fuller	AWS Lambda: "Occasionally Reliable Caching"	07 December 2015
A31	Chris Anderson	Azure Functions with Serverless, Node.js and FaunaDB	06 September 2017
A32	Troy Hunt	Azure Functions in practice	22 September 2016
A33	Paul Batum	Processing 100,000 Events Per Second on Azure Functions	19 Swptember 2017
A34	Tamizhvendan S	Scale Up Azure Functions in F# Using Suave	n/a
A35	Thomas Ardal	Configure and deploy Azure Functions with Kudu	29 March 2017
A36	Brent Schooley	How to Send Daily SMS Reminders Using C#, Azure Functions and Twilio	24 January 2017
A37	Frederic Lardinois	Microsoft's Azure Functions adds support for Java	04 October 2017
A38	Thomas Ardal	Monitoring Azure Functions with the Portal and elmah.io	06 April 2017
A39	Thomas Ardal	An introduction to Azure Functions and why we migrate	20 March 2017
A40	Thomas Ardal	Migrating a Topshelf consumer to a Function running on Azure	23 March 2017
A41	James Thomas	Playing With OpenWhisk	22 April 2016
A42	n/a	OpenWhisk loves Python 3 and you should too... (*Article removed by Medium during the preparation of this manuscript.*)	20 April 2017
A43	Alex Glikson	Extending OpenWhisk to the IoT Edge with Node-RED, Docker and resin.io	14 February 2017
A44	James Thomas	OpenWhisk and MQTT	15 June 2016
A45	Joab Jackson	IBM Launches Bluemix OpenWhisk, an Event-driven Programming Service	22 February 2016
A46	Markus Thömmes	Uncovering the magic: How serverless platforms really work!	11 October 2016
A47	Carl Osipov	Polyglot serverless computing using Docker and OpenWhisk	08 June 2016
A48	Ryan S. Brown	Interview: Andreas Nauerz Of OpenWhisk/Bluemix	16 May 2016
A49	James Thomas	Serverless APIs With OpenWhisk and API Connect	26 April 2016
A50	Lionel Villard	Deploying Express.js apps to OpenWhisk (Part 1)	03 May 2017

Table 4: Complete list of analyzed articles (results for search keywords "aws lambda", "azure functions", and "openwhisk" in order).