The Ifs and Buts of Less is More: A Serverless Computing Reality Check

Jörn Kuhlenkamp  
Information Systems Engineering  
Technische Universität Berlin  
Berlin, Germany  
jk@ise.tu-berlin.de

Sebastian Werner  
Information Systems Engineering  
Technische Universität Berlin  
Berlin, Germany  
sw@ise.tu-berlin.de

Stefan Tai  
Information Systems Engineering  
Technische Universität Berlin  
Berlin, Germany  
tai@tu-berlin.de

Abstract—Serverless computing defines a pay-as-you-go cloud execution model, where the unit of computation is a function that a cloud provider executes and auto-scales on behalf of a cloud consumer. Serverless suggests not (or less) caring about servers but focusing (more) on business logic expressed in functions. Server ‘less’ may be ‘more’ when getting developer expectations and platform propositions right and when engineering solutions that take specific behavior and constraints of (current) Function-as-a-Service platforms into account. To this end, in this invited paper, we present a summary of findings and lessons learned from a series of research experiments conducted over the past two years. We argue that careful attention must be placed on the promises associated with the serverless model, provide a reality-check for five common assumptions, and suggest ways to mitigate unwanted effects. Our findings focus on application workload distribution and computational processing complexity, the specific auto-scaling mechanisms in place, the behavior and strategies implemented with operational tasks, the constraints and limitations existing when composing functions, and the costs of executing functions.

Keywords—serverless computing, function-as-a-service (FaaS), FaaS platforms, FaaS experimentation.

INTRODUCTION

Serverless computing is emerging as an alternative cloud execution model. Instead of leasing infrastructure resources as in traditional cloud systems, developers using serverless computing just submit function code to a cloud provider; thus the term "Function-as-a-Service (FaaS)". The general promise is that the provider runs the code on-demand, scales automatically as needed, and bills the consumer only for the time that the function code is running.

Serverless computing thus suggests to abstract away operational concerns, to offer a simplified programming and deployment model for cloud applications, and to offer a cost advantage. Because of cloud developers "doing less", i.e., not caring about provisioning and operating machines, so the promise, developers can focus and "do more" on the business aspects of their applications.

In this paper, we identify a number of ifs and buts for making this "less is more" promise a reality. We argue to pay careful attention on common assumptions existing with FaaS platforms and to question these assumptions taking the specifics and idiosyncrasies of the (current) FaaS platforms into account. We identify five such common assumptions and corresponding engineering needs, related to application workload distribution and function processing complexity, auto-scaling based on events, the importance of operational tasks, function compositions and dependencies, and execution costs. Without a good understanding of the reality and limitations of (current) FaaS offerings and implementations, developers run danger to face even more responsibilities related to fixing undesirable system behavior, dealing with system failures, or running into unexpected costs. Neither the promises of a simplified execution model nor a cost advantage would then be fulfilled.

Our findings are based on and summarize prior research and experimental work on serverless computing conducted over the last two years. This paper thus presents a cumulative, integrative view of what we have been observing (and are still learning about) serverless computing. Due to the dynamic nature of the FaaS market and concrete FaaS offerings evolving at rapid speed, this paper to some extent is of a situational nature. Still, we believe our findings to be critical for the continued, larger objective of computing conveniently, cost-efficiently, with auto-scaling and minimal infrastructure management efforts, in the cloud.

BACKGROUND

Application developers can choose from a wide range of both commercial fully-managed FaaS platforms [1] and non-commercial open-source FaaS platforms [2]. Examples of fully-managed offers include AWS Lambda1, Google Cloud Functions2, Azure Functions3, and IBM Cloud Functions4. Prominent candidates of open source FaaS platform implementations are OpenWhisk5, Fission6, and Knative7.

Common to all FaaS platforms is that each platform defines a programming model, a function execution model, and a cost model.

At the core of serverless computing is the function abstraction: a function is a specific task that takes some input and

1https://aws.amazon.com/de/lambda
2https://cloud.google.com/functions
3https://azure.microsoft.com/services/functions
4https://www.ibm.com/cloud/functions
5https://openwhisk.apache.org
6https://fission.io/
7https://cloud.google.com/knative
returns some output. From an application programming perspective, functions become the main and dominating programming abstraction. From an application execution perspective, every function execution is isolated and typically ephemeral.

The single execution of a function is triggered by events raised by event sources (function triggers), for example, HTTP requests or a database update. A function handler implementation must be provided by the application developer. In general, FaaS platforms support different programming languages and models, e.g., Python, Go, or NodeJS. The handler implementation then contains all the code and libraries necessary to run the function, including, for example, memory settings for the programming runtime and operating system of choice, calls to external services like database or queuing services, or the definition of an explicit return statement.

The execution model determines how a FaaS platform schedules the processing of events and provisions and deprovisions runtimes. Different execution orders and execution guarantees, such as At-Least Once, can be defined. After execution, the runtime is either kept active for a certain period of time or instantly destroyed. A FaaS platform can reuse a function runtime for new event arriving.

Cost models of commercial FaaS platforms only include price components based on actual function executions. Examples of such components are number of executions, execution duration, or used memory.

**RELATED WORK**

This paper synthesizes and discusses insights gained in own prior work related to serverless computing [3]–[8]. Figure 1 illustrates the six main publications, ordered in a timeline, that serve as a basis to this paper. We refer to these publications with the shorthand notation: P1-P6.

Figure 1. Timeline of select own previous work in the area of serverless computing.

Hendrickson et al. [9], Baldini et al. [10], Hellerstein et al. [11] and Jonas et al. [12] relate to our work in that they report on serverless computing experience, identify fallacies and pitfalls of current platforms, and propose an agenda for future research. Our work is in line with these reports, however, our work is not primarily about proposing (yet another) research agenda, but to share lessons learned when addressing a specific serverless computing research problem (for example, related to benchmarking or cost tracing). In P1-P6, we report on extensive experiments conducted and propose concrete methods and solutions to address these challenges.

**COMMON ASSUMPTIONS AND CURRENT REALITY**

Typical applications for which serverless computing has been suggested include: data analytics [13]–[15], scientific workflows [16], [17], video processing [18], [19], chatbots [20], and publish/subscribe systems [21], [22].

Using FaaS when engineering such applications raises a set of questions:

- How well does the function abstraction and execution model fit diverse application workloads and computational processing complexities?
- Are the auto-scaling features of (current) FaaS platforms satisfactory?
- Given the (new) importance of cloud provider-side operational tasks in FaaS architectures, is the set and actual behavior of available pre-defined tasks appropriate?
- Can functions easily be composed without the compositions experiencing undesirable behavior or limitations?
- Does FaaS indeed offer a significant cost advantage over traditional cloud computing models?

In the following, we take a closer look at these questions. We describe five main **common assumptions** and for each common assumption, state the current, observed reality. Further, we discuss for each assumption the corresponding (engineering) challenge, ways to detect or mitigate the challenge, and further opportunities for future research.

**WORKLOAD AND PROCESSING COMPLEXITY**

**Common Assumption 1. The single function abstraction provided by any FaaS platform suits all function needs independent of the complexity of application workloads.**

Different types of applications use different ways to distribute workload in specific units of work that are processed by a single machine. Examples of units of work are: a web request that loads a thumbnail for a product in a webshop, an image that is resized and changed to a different format, or a batch of data records that are aggregated by adding a specific field.

Traditional cloud computing models require developers to select a virtual machine type and application framework that are dependent of the complexity of application workloads. Developers expect FaaS platforms to process work units from different existing application domains in the function abstraction. However, units of work can significantly differ regarding inputs to and outputs from function handlers and handler executions with different computational needs.

**Reality 1. Workload and processing complexity must match a FaaS platform’s function handler settings.**

In current reality, FaaS offerings do not distinguish different function 'types' but only provide a single function abstraction. For the single function abstraction provided, the execution models limit the amount and type of compute resources that
are available to the runtime of a function handler. Furthermore, FaaS platforms can expose a strict upper bound for the execution time of a function handler.

**Challenge:** Many FaaS application scenarios will fit and execute well within the given 'standard' boundaries of a FaaS platform. For example, we found image resizing and processing single measurements in an IoT context to match current offerings well [3]. However, computationally intensive applications can easily exceed the time limit of common function handlers, making the function abstraction infeasible for specific application requirements. An example of such a case is matrix multiplication [4] within the context of big data processing, matrix multiplication being a fundamental type of computation useful in diverse application domains that have high computational needs compared to small input sizes (see Figure 2). In such cases, standard function handlers will fail execution and can quickly degrade the reliability of the application.

If a function abstracts a data-intensive functionality, loading inputs from and writing outputs to external storage can require a significant share of the maximum execution time of a function handler. Thus, only a small amount of time remains for processing inputs. This behavior, for instance, frequently occurs in data analytics scenarios.

**Detection/Mitigation:** We argue that continuous testing and monitoring of performance, reliability, and cost-efficiency are necessary approaches for detecting application defects that result from the above boundaries currently imposed by FaaS function handlers.

We discuss four mitigation approaches: First, application developers can design and implement an application-specific function decomposition based on a workload distribution model. Second, different FaaS platforms have different limitations for handler settings. Thus, an experiment-driven, comparative assessment of different FaaS platforms can expose these differences and help to select the FaaS platform that fits the application requirements best. Third, developers can mitigate the execution time boundary through the use of function chaining [23]. Function chaining re-executes the same function handler in response to the same event. While function chaining might improve reliability, it can impair cost-efficiency and performance. Lastly, augmenting the runtime life-cycle model of a FaaS platform with application-specific information could enable caching and/or batching of redundant work for multiple events. All mitigation approaches, however, imply significant additional responsibilities for application developers.

**Opportunity:** In our previous work, we found that the requirement of implementing a new workload distribution model for the case of matrix multiplication essentially provided a simple and effective application-level tuning-knob for the cost-performance trade-off. We archived speedup by further decreasing the workload per invocation under the processible limit. Precisely, we increased the number of total splits (see Figure 2), which reduced the number of multiplications for single invocations.

In order to support data-intensive applications features for loading and writing data beyond the current boundaries of function handlers are needed. Specifically, function handler interfaces to respond to runtime life-cycle changes in the execution environment are beneficial. This would allow, for example, a developer to combine the write of the result until the execution environment is about to be paused instead of writing after every invocation.

**Auto-scaling based on events**

**Common Assumption 2.** Functions scale automatically and transparently with a volatile number of events.

Supporting applications that can deal with volatile workloads has always been one of the main drivers for the adoption of cloud computing. In the traditional cloud model, an application developer manages both load balancing and resource provisioning herself. One of the promises of the FaaS model is that developers can now hand these labor-intensive responsibilities off to the cloud provider, allowing the developer to concentrate more on implementing the business logic.

**Reality 2.** Standard provisioning and de-provisioning time of any FaaS platform may not suit an application’s scaling needs.

Each FaaS provider implements a generic elasticity controller to respond to incoming events and to manage function environments on behalf of the user. None of the current FaaS offerings, however, allows a user to control this elasticity controller, which can result in a misalignment with the application needs and the platform.

**Challenge:** One challenge that can occur due to this misalignment is temporal in-elastic behavior. That is, a FaaS platform may temporarily not scale fast enough based on
incoming events resulting in degraded performance and reliability under increasing workloads. In our previous work [8], we detected that, for instance, Google Cloud Functions have a delayed scaling-response to a volatile increase of events, as depicted in Figure 3.

While the challenge of temporal in-elastic results from too slow provisioning of execution environments, a second challenge arises from too fast, i.e., greedy, de-provisioning of execution environments. Greedy de-provisioning hinders the re-use of cachable results from previous executions, for example, downloading a large machine learning model before processing incoming images. Thus, greedy de-provisioning by the FaaS provider might degrade performance and incur undesirable costs under volatile workloads.

**Detection/Mitigation:** Detecting if an application is affected by these challenges can be labor-intensive and require a broad understanding of the application needs as well as of the FaaS platform. In [8] we present an approach to detect and quantify these problems. We also give a general overview of the current elasticity behavior of four major FaaS platforms.

Besides detecting if an application is affected, mitigation strategies exist that help to reduce the impact of these problems. The temporal in-elastic behavior of a FaaS platform might not be avoidable. However, developers can use external high-performance services to offload expensive pre- and post-condition time. An application that needs to acquire additional data might benefit if this data is cached in an ephemeral storage service [12], [25] and expensive authentication tasks could be externalized by relying on provider services instead.

**Opportunity:** New business and research opportunities exist to address the challenges of caching and offloading expensive pre- and post-conditions. Further, the extension of FaaS platforms with a more tune-able elasticity controller is a viable path for a cloud provider to increase her value proposition.

**Operational Tasks**

**Common Assumption 3.** Few fully managed operational tasks that execute transparently solve the diversity of management needs.

The management of applications requires the execution of different operational tasks through its lifetime. Examples include updating deployment packages, changing routes, rolling back updates, or resizing resources. Furthermore, even the same operational task can have varying quality requirements. For instance, some periods of unavailability of a function may be acceptable for the roll-out of a security fix. In contrast, the roll-out of a new version should not impact availability.

**Reality 3.** FaaS platforms implement and apply operational tasks with fixed strategies.
Each FaaS platform implements a small set of operational tasks, e.g., for deployment package or function configuration changes, based on generic platform requirements. Each FaaS platform, however, stops short in offering specialized strategies for executing operational tasks in support of specific management needs, e.g., rollbacks, or security fixes.

**Challenge:** One challenge that the one-solution-fits-all approach creates is that critical management tasks like rolling out security patches are treated the same way by the platform as a common feature update. Our previous experimentation [7] show that some FaaS platforms delay the roll-out of deployment package changes, see the update delay in Google Cloud (ii) in Figure 4. Thus, older code versions remain exposed for a variable amount of time. Such behavior can be undesirable if the older version has a costly defect. Especially, Microsoft’s and Google’s platforms show this behavior [7].

A second challenge can arise during a release that requires the atomic change of multiple functions. For example, a function A depends on a function B. Updating B to B’ makes an update of A to A’ mandatory. Thus, using A and B’ or A’ and B results in runtime errors. Traditionally, release engineers can successfully address this challenge by using one of the two approaches (i) a release unit or (ii) a coordinated release. A single release unit would bundle A’ and B’ in a single artifact for release. A coordinated release would use additional means for coordination that enforce the correct pairing of functions. The fine-granular programming model of FaaS can make release units infeasible. Furthermore, FaaS platforms restrict coordinated releases due to the fully-managed nature of associated operational tasks. These operational tasks are often undocumented, have client-side visible side-effects, and can significantly differ for FaaS Platforms (see Figure 4).

**Detection/Mitigation:** We argue that experimentation and testing of the effects of operational tasks are needed to understand these risks. In prior work [7] we presented a method to evaluate and understand these risks during the development life-cycle of a FaaS application.

We further recommend that developers should evaluate and select FaaS platforms that better fit their risk scenarios as the implementation of operational tasks differs significantly across the major cloud providers. To further mitigate these issues, we recommend implementing feature toggles [26] that can enable more complex roll-out behavior. For example, feature toggles can prevent calls to incompatible downstream services. Lastly, we recommend adding version references to events to allow functions to handle out of date events programmatically.

**Opportunity:** Applications that leverage the pre-defined operational tasks of FaaS providers can introduce significant benefits. For instance, functions can resize resources during the runtime of a job to mitigate errors that would previously result in failures. Furthermore, developers can build more reliable data processing pipelines that do not suffer from out of memory problems for skewed input distributions [7].

Another possibility for applications that can leverage the operational tasks effectively is the ad-hoc customization of deployment packages based on event inspection. Our previous work [7] indicates that providers like AWS apply deployment package changes quickly, allowing, for instance, to deploy different sorting strategies based on the content of events.

**FUNCTION COMPOSITION**

**Common Assumption 4.** *Functions can easily be composed to implement any business logic.*

FaaS platforms lend themselves well to diverse types of applications. Each application has different function composition needs. Ideally, function composition comes with as few constraints and restrictions as possible.

**Reality 4.** *Function compositions experience limitations in choices of runtime environments and resource capabilities outside of the control of the application developer.*

Current FaaS platforms (naturally) limit the control of the deployed execution environment. Moreover, persistent storage in execution environments is often restricted [4]. Furthermore, these environments only have limited connectivity and can only be invoked through a finite set of triggers.

The implementation of even simple business logic may quickly require composing additional services. A FaaS-based...
analytics process, for example, will need external storage services to store intermediate and final results, as experienced during the FaaS-ification of a distributed matrix multiplication scenario [4]. Figure 2 illustrates this application, where both function composition and workload distribution (see Reality 1) were critical.

**Challenge:** There are FaaS-specific service composition needs due to environmental restrictions that result in several challenges for developers.

Debugging complex service compositions can be more complicated than in traditional cloud models. Especially, detecting the root-cause of errors requires to analyze executions in heterogeneous, highly decoupled systems, requiring developers to understand multiple different services and different debugging and logging tools [27].

A further challenge relates to co-locating computation and data. Especially for data-intensive applications, it is beneficial to move computation towards the data. However, FaaS platforms require to move the data to the computation.

It can be further challenging to migrate complex composed applications to another cloud provider. Both the interfaces and qualities of specialized provider services can be unique. Thus, any such migration is very difficult and labor-intensive. Selecting an alternative service on a different cloud provider without significant re-engineering of the application is not trivial.

**Detection/Mitigation:** One way to mitigate the challenges of debugging FaaS-based applications is to introduce better monitoring and testing that covers all services used by an application. Especially, the usage of distributed tracing and distributed logging services with automated analysis can help to reduce this overhead.

In order to address the data-locality challenges, we recommend the usage of specialized storage services for sharing ephemeral data that can deal with the workload of FaaS applications [25]. We also recommend evaluating the different services offerings to get data to a FaaS application to find the best fit for a given application. Lastly, the extension of the FaaS programming and execution model with data management operations might enable better data processing performance.

**Opportunity:** FaaS allows implementing ad-hoc large-scale computations in big data processing systems with low entry barriers [6]. FaaS can enable large-scale computation within seconds, as many experiments have shown [13], [19].

**Execution Costs**

**Common Assumption 5.** FaaS offers a significant cost advantage compared to traditional cloud computing models.

The cost model calculation of traditional cloud deployment models typically is based on the size and duration of the provisioned machines. Thus, provisioned machines with low utilization are not cost-efficient. It is the responsibility of application developers to ensure that appropriate mechanisms for combined load-balancing and provisioning are in place that ensure the high utilization of provisioned machines over time.

In contrast, in serverless computing using FaaS, the cloud provider bills the consumer only for the time that a function is actually running. Thus, application developers need to invest less time and work in load-balancing and provisioning while avoiding paying for idle machines. As consequence, cost-savings are the main expectation of application developers for using FaaS [23].

**Reality 5.** The marginal costs of running code are higher, and compositions can become additional cost drivers.

The cost model of FaaS can realize significant cost benefits for volatile cost models by avoiding charges for idle machines.
However, case studies show that for specific workloads, the marginal costs of executing code are lower [28] or higher [29] compared to traditional cloud computing models. Thus, configuring the “right” amount of computing resources for executing a function, i.e., sizing, remains a critical management task for developers. In practice, sizing is often non-trivial because of the implicit cost-performance trade-off that has to be solved based on the requirements of a specific application context.

Moreover, functions do not execute in isolation but compositions. The composition logic can become an additional driver of the cost. We provide three examples of composition-related cost-drivers in [3]. They include:

- Re-execution: Failing downstream services can result in multiple executions of upstream functions.
- Synchronous Calls: Functions that make synchronous calls to backend services count as executing and are billed during the wait-time.
- Repeated Calls: A fine-granular workload distribution (see Reality 1), isolation, and ephemeral storage (see Reality 4) can increase the total number of calls to backend services, e.g., databases, for the same amount of work [30].

**Detection/Mitigation**: Testing, experimentation, and monitoring are appropriate means for detecting the need for sizing-related cost-debugging. Sizing is a re-occurring task for new code commits. Thus, sizing tasks should be automated. First approaches in support of automating sizing decisions for isolated AWS Lambda functions exist. While such approaches are a promising start, sizing decisions should not occur for isolated functions but taking composition logic into account. Debugging composition-driven cost defects can prove difficult.

Leitner et al. [31] propose an approach for detecting costs per service and application. However, in many cases, detecting the root-cause of cost-defects requires cost-tracing infrastructure. Unfortunately, due to the high level of abstraction, tracing can be challenging in a serverless environment. Thus, we propose a corresponding cost-tracing approach [3]. We argue that for some applications, static code analysis can be appropriate to detect composition-driven cost defects. However, to the best of our knowledge, respective approaches are still missing.

**Opportunity**: The fine-granular metering model of the serverless abstraction enables a plethora of new opportunities. For example, application developers could offer corresponding fine-granular cost models for application-level APIs. For large computational tasks, e.g., big data analytics, the serverless abstraction could enable job-level billing, budgets, and runtime budgets.

**Conclusion**

Serverless computing and Function-as-a-Service platforms are gaining traction for good reasons. The serverless programming model, however, introducing a specific function abstraction, and the serverless execution model, introducing dependencies to event and runtime handler implementations and configurations, are inherently different from conventional cloud computing practice. In this paper, we describe a set of five common serverless assumptions and provide a state-of-the-art assessment for each. This reality check, based on a series of extensive experimentation over the last two years, reveals important challenges that, if not carefully addressed, can easily counter the serverless promises. We expect that some observed realities will change as FaaS platforms and implementations change. FaaS platform propositions and developer expectations must, however, be continuously be questioned and verified. To that end, the need for continuous experimentation and benchmarking along with the definition of appropriate metrics and measurement methods remains to be an important future work.

**References**


