Lessons Learned from Migrating Complex Stateful Applications onto Serverless Platforms

Zewen Jin[1], Yiming Zhu[1], Jiaan Zhu[1], Dongbo Yu[1]
Cheng Li[1], Ruichuan Chen[2], Istemi Ekin Akkus[2], Yinlong Xu[1]
{zevin, zym651, zja_pb17151780, ydb2018}@mail.ustc.edu.cn {chengli7, ylxu}@ustc.edu.cn
{ruichuan.chen, istemi_ekin.akkus}@nokia-bell-labs.com
[1]University of Science and Technology of China [2]Nokia Bell Labs

Abstract

Serverless computing is increasingly seen as a pivot cloud computing paradigm that has great potential to simplify application development while removing the burden of operational tasks from developers. Despite these advantages, the use of serverless computing has been limited to few application scenarios exhibiting stateless and parallel executions. In addition, the significant effort and cost associated with re-architecting existing codebase limits the range of these applications and hinder efforts to enhance serverless computing platforms to better suit the needs of current applications.

In this paper, we report our experience and observations from migrating four complex and stateful microservice applications (involving 8 programming languages, 5 application frameworks, and 40 application logic services) to Apache OpenWhisk, a widely used serverless computing platform. We highlight a number of patterns and guidelines that facilitate this migration with minimal code changes and practical performance considerations, and imply a path towards further automating this process. We hope our guidelines will help increase the applicability of serverless computing and improve serverless platforms to be more application friendly.

ACM Reference Format:
Zewen Jin[1], Yiming Zhu[1], Jiaan Zhu[1], Dongbo Yu[1], Cheng Li[1], Ruichuan Chen[2], Istemi Ekin Akkus[2], Yinlong Xu[1]. 2021. Lessons Learned from Migrating Complex Stateful Applications onto Serverless Platforms. In *12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21), August 24–25, 2021, Hong Kong, China.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3476886.3477510

APSys '21, August 24–25, 2021, Hong Kong, China

ACM ISBN 978-1-4503-8698-2/21/08…$15.00
https://doi.org/10.1145/3476886.3477510

1 Introduction

Serverless platforms, such as AWS Lambda [29] and Apache OpenWhisk [4], allow developers to quickly build and deploy applications at scale while eliminating the need for them to manage infrastructure. With serverless computing, developers can focus on the implementation of application logic, which is packed and uploaded to serverless platforms for execution. The platforms are responsible for managing the infrastructure resources and scaling users' applications when needed. Because of its ease-of-use and operational simplicity, serverless computing has generated tremendous interest in both industry and academia [2, 6, 10, 13, 16, 31].

Despite its popularity, the serverless computing concept has only been applied to a few simple scenarios that mostly exhibit the needs of stateless and embarrassingly parallel executions [10]. There is an increasing demand for applying serverless computing to more complex and stateful applications that already exist [31]. Today, to take advantage of serverless computing, developers of these existing applications have to re-architect their applications to match the requirements of serverless computing platforms. This rebuilding of existing applications is time-consuming and ad-hoc, hindering the developers' adoption of serverless computing. A principled approach to this migration can help developers in their effort. At the same time, it can also provide insights for the design of serverless platforms in the future.

In this paper, we target the migration of complex microservice applications onto serverless platforms. Today, the *de facto* standard of building cloud applications is to use the *microservice* architecture [28]. This architecture allows the application to be decomposed into a set of loosely-coupled components with well-defined interfaces. Although each component can be developed and operated individually, this architecture still requires developers operating infrastructure resources (e.g., containers, virtual machines), monitoring the application and scaling the components as needed. Supporting the migration of such existing applications onto serverless platforms will benefit developers as well as increase the applicability of serverless computing.

There are two challenges in providing this support: First, it is costly to build a new serverless version of a complex application; thus, it is preferred to *minimize the modifications* to the codebase and reuse existing code as much as possible.

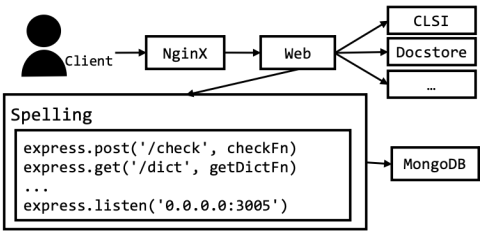


Figure 1. A simplified microservice architecture of Overleaf, with a focus on the flow of a `spelling` task execution.

Second, the migration should *achieve similar performance* compared with the existing microservice applications; otherwise, using serverless computing may not satisfy application requirements. Addressing these two challenges, combined with the inherent features of serverless computing such as auto-scaling, will make serverless platforms deliver their full benefits for even existing complex applications, and reduce their operational and management costs.

As a first step to address these challenges, we have migrated four complex open-source microservice applications onto Apache OpenWhisk serverless platform [4]. These applications include a collaborative cloud-based LaTeX editing application (Overleaf, formerly ShareLaTeX [25]), an online social network application (Social Network [12]), and two example e-commerce applications (LAB Insurance Sales Portal [18] and Robot Shop [32]). Together, they involve 8 programming languages, 5 application frameworks, and 40 application logic services.

During our effort, we made a number of observations on typical application design patterns and identified several associated migration guidelines, covering aspects about communication, request routing and handling, state management, exception handling and operational tasks. Preliminary experiments show that applying our migration guidelines results in fully (or mostly) serverless versions of these applications that can achieve comparable latency under compute-intensive workloads and provide better scalability under varying workloads.

2 Background and Motivations

2.1 Microservice Applications

A typical microservice application consists of a number of long-running microservices (up to hundreds or even thousands [5, 26]), covering computation, communication and storage functionalities [27, 28]. These microservices are loosely-coupled and communicate with each other via well-defined APIs, so that each microservice can be developed and operated individually. Figure 1 shows the microservice architecture of the Overleaf application [25]. For simplicity, we show only four microservices whereby `NginX` acts as a load balancer, `web` acts as a dispatcher for downstream services, `spelling` is a sample service that listens on port 3005 and routes the received requests to internal handlers for execution, and `MongoDB` is responsible for maintaining and synchronizing global state among different microservices.

Each microservice typically runs as a server. This type of deployment precludes the opportunities for executing services only when needed, and may cause application developers to overprovision resources to meet workload requirements. This overprovisioning leads to low resource utilization for infrastructure operators and higher monetary cost for application developers. In addition, although the microservice architecture simplifies how each microservice component is developed and operated, it requires developers to carefully monitor and manage resources (e.g., containers, virtual machines) to scale in/out each microservice properly, making it challenging to serve often varying workloads.

2.2 Serverless Platforms

The serverless computing paradigm aims to remove resource management from developers, scale fast to varying workloads, and provide cost efficiency via `pay-as-you-go` pricing model. The most popular realization of serverless computing is Function-as-a-Service (FaaS), where the unit of computation is a function. There are a number of commercial FaaS Platforms (e.g., AWS Lambda [29], Azure Functions [7]) and open-source, community-maintained platforms (e.g., Apache OpenWhisk [4], OpenFaaS [23], KNIX MicroFunctions [2, 17]). In this paper, we use Apache OpenWhisk which is widely used in the community. However, it is worth noting that the observations and guidelines drawn in this paper can be generally applied to other serverless platforms.

In a FaaS platform, developers write their application logic as a set of stateless functions. Each request is dispatched to one function or a sequence of functions, which causes a function instance to be scheduled to run inside an action container, process the request and generate an output. Upon the completion of a function execution, its state is destroyed or externalized via a storage or messaging system; therefore, freeing the resources for other uses. Together, these mechanisms keep serverless function executions usually short-lived, and enable flexible and fine-grained scheduling and scaling of infrastructure resources to meet varying workloads when needed. This flexibility in turn leads to the resource efficiency for infrastructure operators and the cost efficiency for developers.

The stateless nature of the serverless functions, however, creates a fundamental tension between the complex stateful applications and the FaaS platforms. To take full advantage of serverless computing, developers of existing complex applications have to re-architect their applications, which is costly and somewhat ad-hoc, making it preferable to reuse existing codebase to the maximum extent. Therefore, it is of great interest to have a principled approach to migrating existing applications onto serverless platforms, with minimal code changes and comparable performance (e.g., latency).

Table 1. The statistics of the four complex, stateful microservice applications.

Application	#Services	#App. Logic Services (LoC)	#Off-the-shelf Services	Language	Application Framework	Communication
Overleaf	15	12 (61,616)	3	Node.js	Express	RESTful API, Redis Pub/Sub
Social Network	26	11 (7,123)	15	C++, Lua	Thrift RPC	RPC
LAB	18	10 (8,295)	8	Java, Kotlin	Micronaut	RESTful API, Kafka
Robot Shop	12	7 (1,679)	5	Node.js, Java, Python, Go, PHP	Express, Flask, Spark	RESTful API, RabbitMQ

3 Migration of Microservice Applications

To gain first-hand experiences, we have migrated four publicly-available, complex microservice applications onto Apache OpenWhisk. These applications are designed using a few application frameworks with diverse choices in programming languages and communication interfaces (see Table 1). They consist of 12-26 microservice components which include application logic components, as well as off-the-shelf software components that are usually used for communications (e.g., Kafka and RabbitMQ), keeping application state (e.g., Redis and MongoDB) or monitoring application status (e.g., Zipkin[24] and Jaeger[14]).

To migrate, we first determine whether a component is an off-the-shelf component. Normally, we do not migrate such components in order to minimize changes to the original application architecture; alternatively, these components could be replaced with their counterparts offered by serverless platforms (e.g., AWS S3 for keeping application state). Furthermore, since serverless functions are short-lived and not directly network-addressable, we keep microservice components unchanged if they interact with other entities in a bidirectional streaming fashion, e.g., via websocket or streaming RPC. Finally, we determine whether and how to adapt the remaining application logic microservice components to be serverless functions, with special care to these microservices that deal with synchronous communication, state management, and a few others (guidelines in Section 4).

3.1 Overleaf

Overleaf, formerly known as ShareLaTeX, is a real-time collaborative LaTeX editor [25]. Besides off-the-shelf components (e.g., NginX, MongoDB and Redis), it has 12 application logic microservices, all of which are written in JavaScript and run using Node.js. These microservices communicate with each other using RESTful APIs and Redis Pub/Sub channels, and MongoDB is used for data storage. Overleaf leverages the Express framework, which allows internal service modules in a microservice to be exposed to other microservices.

Migration. As shown in Table 2, we introduce 1.2% lines of code changes to Overleaf. First, 8 out of 12 application logic components are stateless and communicate via RESTful APIs, making their migration straightforward: we augmented each component with a wrapper function that acts as the entry point required by OpenWhisk. This wrapper is responsible for forwarding parameters to the original Express handler upon invocation (see Section 4.2 for details).

Table 2. The statistics of the four application migrations.

Application	#Services Migrated	#Services Remaining but Changed	ΔLoC (%)
Overleaf	10	1	746 (1.2%)
Social Network	11	1	910 (12.8%)
LAB	10	0	850 (10.2%)
Robot Shop	7	0	590 (35.1%)

In addition, there are 4 stateful microservice components: 1) The `web` microservice acts as a dispatcher for sending requests to downstream microservices, as a request authenticator that reads locally stored credentials to verify requests, and also as a proxy server for file uploading/downloading. 2) The `clsi` microservice compiles LaTeX files and stores the output locally. This local storage acts as a file server for future retrieval. 3) The `filestore` microservice is responsible for serving Overleaf files. 4) The `real-time` microservice exposes addressable network interfaces to clients, accepts incoming connections and interacts with the clients via the connections.

Among these 4 components, we applied changes to the `web` and `clsi` components. In a nutshell, for `web`, we separated its authentication module, persisted the authentication-related credentials in `Redis`, and kept the file proxy server module as in the original microservice (see Section 4.3). For `clsi`, we made changes in 38 lines of code to store generated files in `filestore`. As a result, we could migrate `web` and `clsi` into serverless, while keeping `filestore` and `real-time` still as microservices in order to minimize code changes and comply with the original architecture.

3.2 Social Network

Social Network [12] is an online social networking application based on DeathStarBench which has been used as a benchmark suite for cloud microservices [11]. This application uses Apache Thrift [30] to enable RPC communications among its microservices that are written in C++ and Lua.

Migration. All application logic microservices in Social Network are stateless, and we could migrate all of them. The

key challenge we faced was that Apache OpenWhisk functions are invoked via HTTP requests with JSON-encoded data as input, whereas the Social Network's RPC code uses low-level socket APIs. As a result, we had to convert its original RPC communications to HTTP communications, and implement the extra JSON serialization/deserialization functionality. To assist in this conversion, we have developed a semi-automated script in roughly 700 lines of Python code. With regard to the request routing logic, we bypassed the original RPC listener (see Section 4.2).

3.3 LAB Insurance Sales Portal

LAB Insurance Sales Portal (abbreviated as LAB) [18] is an e-commerce application that uses the Micronaut microservice framework [21]. Its microservices are written in Java and Kotlin, with dependencies on the Micronaut libraries. The inter-service communication happens via either RESTful APIs or the Apache Kafka message broker [3].

Migration. There are 6 stateless application logic microservice components. These components originally communicate with RESTful APIs and Kafka, so we made necessary changes to adapt their communications to OpenWhisk's required patterns and removed Kafka entirely (see Section 4.1). For the remaining 4 stateful components, we noticed that each of them runs a local H2 in-memory database [9] for state management. We unified these H2 databases into a new single stateful component, and this change enabled us to migrate these 4 components also into serverless.

3.4 Robot Shop

Robot Shop [32] is another e-commerce application. It uses several popular programming languages including Node.js, Java, PHP, Python and Golang, along with web frameworks such as Express, Flask and Spark. Its microservices communicate with each other via RESTful APIs and RabbitMQ.

Migration. All its application logic microservices are stateless, and their migration is similar to that of the Overleaf application described before. We kept the original listeners to forward requests with additional request encoding/decoding steps. Most of our efforts have been focused on making this conversion in different languages and frameworks used in this application. Furthermore, we removed the RabbitMQ component which was originally used to asynchronously trigger some subscriber service (i.e., `dispatch`), because its functionality has already been supported by OpenWhisk. Altogether, though the codebase of Robot Shop is the smallest, the ratio of code changes (35.1%) is the highest among all four application migrations.

4 Observations and Migration Guidelines

Here, we present our observations in the aforementioned migration efforts and summarize a set of general guidelines for migrating complex applications onto serverless platforms.

4.1 Synchronous vs. Asynchronous Communication

Microservice components communicate with each other via well-defined interfaces. Depending on the application needs, this communication can take place synchronously or asynchronously. In synchronous communication, the sender of a message blocks until it receives a response from the receiver of the message. Examples include RESTful APIs and synchronous RPCs. On the other hand, in asynchronous communication, the sender of a message does not wait for the response from the receiver. Examples include communication via message brokers and asynchronous RPCs.

We could apply the synchronous communication in the context of serverless computing via blocking serverless function calls, which however might cause inefficiencies and double billing [8]. For example, in Overleaf, a synchronous `spelling` request goes through the `web` microservice, meaning the migrated serverless `web` would have to wait until the serverless `spelling` finishes processing the request. This problem gets amplified when there are multiple functions interacting with each other. As a result, microservices communicating over synchronous channels require more changes to benefit from serverless computing, due to the conversion of communication patterns. We leave the exploration of optimizations for future work.

Communication among serverless functions usually happens in asynchronous channels. Migrating microservice components that exchange messages via asynchronous channels (such as message brokers) can be more straightforward, and the modification can be performed in various ways.

Invoke with publishing. Since a serverless function is triggered only when there is a request, one cannot simply take a microservice component that is the message receiver (working in a subscriber mode) and use it in a serverless function that executes when there is a request (see Figure 2a). Thus, the sender (i.e., publisher) has to invoke the serverless function asynchronously, so that the message can be received and processed (see Figure 2b).

Invoke without publishing. The sender can also invoke the receiver function with the request message (see Figure 2c). This approach has the advantage of eliminating the overhead caused by publishing and receiving the message from the message broker. Furthermore, it makes the message broker redundant, so that it can be removed to simplify the application architecture. On the other hand, features provided by the message broker, such as message ordering or multi-subscriber support for parallelism, would be lost and have to be otherwise supplied. For example, the sender may need to invoke multiple subscribing functions for parallelism.

4.2 Listening and Routing

Regardless of the choice of synchronicity in the communication, the interacting microservice components have to listen for new requests and route them to their associated internal

```
# publisher
  publisher.publish(newMessage)
# subscriber
  subscriber.register(fetchAndProcessNewMessage)
  def fetchAndProcessNewMessage():
    ...
```

(a) Original publication and subscription.

```
# publisher
  publisher.publish(newMessage)
  publisher.invokeAsync(subscriber, 'notifyNewMessage')
# subscriber
  def serverless_handler(msg):
    if msg == 'notifyNewMessage':
      subscriber.register(fetchAndProcessNewMessage)
  def fetchAndProcessNewMessage():
    ...
```

(b) Serverless: Invoke with publishing.

```
# publisher
  publisher.invokeAsync(subscriber, newMessage)
# subscriber
  def serverless_handler(newMessage):
      processNewMessage(newMessage)
```

(c) Serverless: Invoke without publishing.

Figure 2. Asynchronous communication migration.

handlers. With RESTful APIs and RPCs, the microservice usually exposes a network port and accepts requests over the port (see Figure 1). With message brokers (e.g., Kafka, RabbitMQ, Redis Pub/Sub), the microservice subscribes to a topic or channel, and waits for incoming requests.

In the serverless computing context, such scaffolding code for receiving messages becomes unnecessary, because the exchanges of message between functions are handled by the serverless platform. These exchanges usually invoke the serverless functions with the input parameters as a JSON-encoded object inside the HTTP request. In a migration effort with the goal of minimal code changes, this input needs to be converted properly to dispatch the request to its appropriate handler. There are two approaches one can take.

Keep the listener. This approach will keep the original listener of a microservice component inside the serverless function. In order to process the request, some glue code is required so that the incoming HTTP request on the serverless platform can be converted to the request format accepted by the microservice component. For example, if the original microservice component has an RPC server, the glue code will have to transform the HTTP request into RPC inside the serverless function. This approach can reduce the migration effort drastically. For example, in Overleaf, we only need to write 66 lines of JavaScript glue code to wrap existing listeners, which are then applied to 10 of its microservice components. One potential disadvantage is the additional latency introduced by the glue code. Furthermore, care must be taken when the original components expose network ports: if original microservices use the same port numbers, migrated versions (i.e., functions) in this approach may execute on the same host and create conflicts.

Bypass the listener. This approach will remove or bypass the original listener. It is more efficient, but it requires extra efforts to find and call the proper inner handler for each type of requests. In RPC-based microservices, the decoupling of the RPC stub and the handler code makes the migration fairly straightforward. In comparison, in microservices using REST APIs, the code of handlers and listeners is more tightly coupled, for instance, due to the underlying frameworks (e.g., Express for JavaScript, and Flask for Python). This migration requires more efforts and changes, making it more tedious and error-prone. For example, the Overleaf's `spelling` microservice written using the Express framework requires the modification of about 150 lines of code, even though the microservice itself consists of only 800 lines of code. For a larger microservice like `web`, which consists of about 40K lines of code, this modification will require more efforts [20].

4.3 State Management

Serverless functions are typically stateless and short-lived, with an intent to make them easy to manage and scale. Multiple requests to the same serverless function may be directed to different function instances for execution, and therefore, the state produced by one execution may be inaccessible to the next execution. This approach, however, complicates the state management that is required by many microservice applications.

To share the state between multiple instances of the same function or across multiple functions, one common approach is to externalize the state to some intermediary (e.g., a storage system such as MongoDB, Redis, or AWS S3) for state sharing. Extra care needs to be taken to deal with state during migration. As mentioned in Section 3, if the application uses off-the-shelf storage components (e.g., MongoDB), we can either simply keep these components to minimize code changes, or modify the application to use storage services offered by the serverless platform (e.g., AWS S3).

In contrast, if the state is stored locally inside custom microservice components, we have to manually identify the application state before externalizing it. For example, there are two types of local state in Overleaf. One is the `csrf` tokens used for authentication in the `web` microservice. The other is the compiled files within the `clsi` service, which are stored in `filestore` after migration. These data are stored in Redis after our migration. Another example of such locally-kept state is the local database in LAB Insurance Sales Portal. Several microservices of this application have their own H2 in-memory database running locally. When these microservices are migrated, the corresponding H2 database engine

could be deployed as a standalone server if one wants to minimize the code changes.

4.4 Miscellaneous

Below, we further report two more migration considerations.

Exception handling. In microservice applications, there are many exceptions thrown at runtime to handle communication failures. To comply with the original application logic, we should keep the identical exception handling behaviors in the migrated version. For RESTful APIs, supported by OpenWhisk natively, we can simply reuse the original exception handling mechanism. However, this cannot be applied to RPC-based communications, since the migration replaces RPCs with HTTP requests. To address this issue, we have to identify the distinct types of RPC exceptions, and explicitly throw exceptions when the corresponding failures occur, in such a way that these exceptions can be handled by the original exception handling code.

Redundant components. Serverless platforms usually provide utilities for monitoring, load balancing, and auto-scaling. When migrating microservice applications onto serverless platforms, some existing components can be removed or replaced with the built-in options provided by serverless platforms. For instance, `Consul`, the registry and discovery service in LAB, can be removed as serverless platforms already route messages to associated functions. Similarly, `circuit breakers` for limiting request rates may also become redundant, because applications can use the monitoring, load balancing and scaling utilities from the serverless platforms.

5 Preliminary Evaluation

We report our preliminary evaluation on three applications: Overleaf, Social Network and Robot Shop[1], and compare the original microservice architecture with our migrated version. We first break down the request handling of one task in each application (i.e., 'Compile' in Overleaf, 'StorePost' in Social Network, and 'SubmitOrder' in Robot Shop). Afterwards, we increase the load on the 'Compile' task in Overleaf and the 'SubmitOrder' task in Robot Shop to evaluate the scalability offered by OpenWhisk, and report throughput and latency values. We use Locust [19] for load generation.

Our experiments were conducted in a virtual machine (VM) with 40 2.2GHz vCPUs, 40GB RAM and 250GB SSD. This VM hosts the OpenWhisk installation with one controller, one invoker and an unlimited container pool size, as well as the microservice components that are not migrated. These components run in separate containers with no resource limits. All serverless functions are deployed with 256MB RAM in OpenWhisk. To make a fair comparison with the always-deployed microservice components, we ensure that our OpenWhisk functions are running in warm containers. This setting is reasonable, and we reason our decision as follows: First, under steady workloads, function containers will be warm to better serve requests. Second, decreasing cold-start and invocation latencies is an active research topic [2, 15, 22] and is orthogonal to this work.

[1]We observed similar trends for the LAB Insurance Portal application and omit its results due to space limit.

Table 3. Latency breakdown in milliseconds for Overleaf's 'Compile' task, Social Network's 'StorePost' task, and Robot Shop's 'SubmitOrder' task. Here, 'SL' stands for serverless and 'MS' stands for microservice.

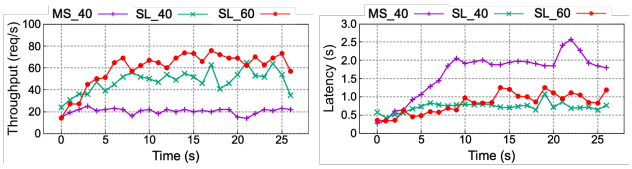
Task	Version	Invocation	Init	Compute	Store	E2E
Compile	SL	28.1	-	283.3	19.0	330.4
	MS	3.8	-	275.0	-	278.8
Store-Post	SL	33.9	25.7	0.7	-	60.2
	MS	0.2	-	0.6	-	0.8
Submit-Order	SL	26.9	-	158.7	-	185.6
	MS	6.4	-	141.9	-	148.3

5.1 Latency Comparison and Breakdown

Table 3 presents the latency breakdown of the requests for the three tasks in our applications. As one can see, the 'compute' step is roughly equal for both serverless (SL) and microservice (MS) deployments. The reason is that, for computation, there are no resource limits for both the OpenWhisk functions and the original microservice containers. In the serverless functions, there is a common 'invocation' overhead. This overhead is due to OpenWhisk's handling of HTTP requests to trigger a function execution, which has to pass through multiple platform components before reaching the function (i.e., NginX, controller, Kafka and invoker). On the other hand, for the original microservice deployment, the request happens directly via Thrift RPC in Social Network, and RESTful API in Overleaf and Robot Shop.

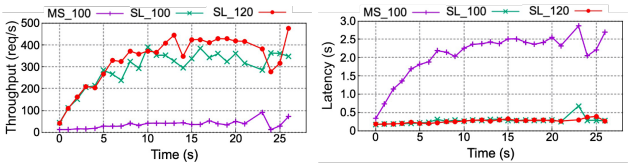
There are two more steps that incur overhead for the serverless deployment. The 'Store' step for the 'Compile' task in Overleaf is due to our migration-related changes: instead of keeping the generated files inside the microservice component, we send these files to the `filestore` component (see Section 3). This externalization of state to a storage service is common in stateless serverless functions and can only be alleviated by providing fast access to such storage.

The 'Init' step in Social Network's 'StorePost' task is more involved. This step refers to the loading of serverless function executables and the setup of active database connections to MongoDB and Memcached in the original microservice component. First, there exists language mismatches between OpenWhisk's action runner and the function, i.e., OpenWhisk's action runner uses Python to listen for new requests, but the 'StorePost' function is written in C/C++. Thus, the Python runner loads the executable binary dynamically at every request. Second, we had expected that these connection requests could be moved outside the function handler



(a) Throughput **(b)** Latency

Figure 3. Scaling of the 'Compile' task in Overleaf. Here, 'SL' stands for serverless, 'MS' stands for microservice, and '_num' indicates the number of clients (e.g., 40).



(a) Throughput **(b)** Latency

Figure 4. Scaling of the 'SubmitOrder' task in Robot Shop. Here, 'SL' stands for serverless, 'MS' stands for microservice, and '_num' indicates the number of clients (e.g., 100).

to eliminate the incurred latency, so that they are established only once at the action container start and stay alive for subsequent requests. Nevertheless, because the database connections are set up inside the binary, this step cannot be easily eliminated. There are two approaches: If the action container's runner was written in the same language, this issue would not exist. More generally, these connections are required for storing and retrieving state, thus could be removed if the serverless functions have access to a common, efficient backend storage service.

5.2 Auto-scaling Benefits

To show the auto-scaling benefits of serverless computing, we conduct experiments with the Overleaf's 'Compile' task and Robot Shop's 'SubmitOrder' task, in which we increase the workload by adding 5 and 10 more clients per second, respectively, until the system gets saturated. Note that, the purpose of our preliminary evaluation is not to showcase that our migration to serverless can perform better than microservices, but should rather be seen as a 'sanity check' on our changes. Figure 3a shows that the serverless version of the 'Compile' task supports up to 80 requests per second with 60 clients, whereas the original microservice component can only handle 40 clients and up to 19.3 requests per second. Recall that the original microservice component is running without resource limits; however, the `clsi` component responsible for compilation is bottlenecked by the single process it runs. On the other hand, though the OpenWhisk actions are limited to 256MB RAM, they can still scale out with concurrent requests.

Figure 3b shows the experienced latency with these workloads. With the increasing load, the original microservice experiences an increasing delay and reaches up to about 2 seconds under the peak load. In comparison, the latency produced by the serverless function is more stable and is about 900ms under the peak load. As shows in Figure 4a and 4b, we also observe similar performance trends for the 'SubmitOrder' task in Robot Shop, validating the auto-scaling ability of serverless platforms.

6 Discussion and Future Directions

We think that serverless computing can benefit a wide range of complex, stateful applications; however, there are still some open challenges that need to be addressed. First, our preliminary evaluation as well as other studies [2, 22] reveal that the visible invocation delays may impede the adoption of serverless computing, especially for latency-sensitive applications. The extra delay would be amplified when executing a sequence of functions, required by complex application logic. Therefore, the migration methodology should incorporate the trade-off between latency and serverless benefits (e.g., auto-scaling) to avoid 'over-migration'. Fortunately, there is a plethora of work to eliminate such overheads from both industry [1] and academia [2, 15]. Second, a tool that helps developers migrate their legacy microservice applications to serverless or that automates this migration will be tremendously helpful for enabling serverless benefits for such applications. Such a tool would also accelerate the adoption of serverless computing for a wider range of applications. Our manual migration already makes a first step towards this goal. However, in order to automate this process, more work is needed to address the challenges introduced by different communication patterns as well as tight coupling of state with computations. Third, the state management plays a key role in migrating stateful microservices onto serverless platforms. To reduce migration complexity, state management systems should have well-defined interfaces covering various types of states (e.g., files, memory, key-value pairs). In addition, fast state access will likely improve the performance of serverless functions after migration.

Acknowledgments

We sincerely thank all anonymous reviewers for their insightful feedback, and especially thank our shepherd Yongle Zhang for his guidance in our camera-ready revision. This work was supported in part by National Nature Science Foundation of China (61802358, 61832011, 61772486), and USTC Research Funds of the Double First-Class Initiative (YD2150002006). Cheng Li is the corresponding author.

References

[1] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[2] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. SAND: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference*. 923–935.

[3] Apache. 2021. Apache Kafka is an open-source distributed event streaming platform. https://kafka.apache.org/. [last access: May 26, 2021].

[4] Apache. 2021. Apache OpenWhisk is a serverless, open source cloud platform. https://openwhisk.apache.org/. [last access: May 26, 2021].

[5] AppCentrica. 2021. The Rise of Microservices. https://www.appcentrica.com/the-rise-of-microservices/. [last access: May 26, 2021].

[6] AWS. 2021. AWS Lambda Customer Case Studies. https://aws.amazon.com/lambda/resources/customer-case-studies/. [last access: May 26, 2021].

[7] Microsoft Azure. 2021. Azure Functions Serverless Compute. https://azure.microsoft.com/en-us/services/functions/. [last access: May 26, 2021].

[8] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. 2017. The Serverless Trilemma: Function Composition for Serverless Computing *(Onward! 2017)*. Association for Computing Machinery, New York, NY, USA, 89–103. https://doi.org/10.1145/3133850.3133855

[9] H2 Database Engine. 2021. H2 Database Engine. http://www.h2database.com/html/main.html. [last access: May 26, 2021].

[10] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 363–376. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi

[11] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[12] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2021. SocialNetwork - One Project of DeathStarBench. https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork/. [last access: May 26, 2021].

[13] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *arXiv preprint arXiv:1812.03651* (2018).

[14] Jaeger. 2021. Jaeger: open source, end-to-end distributed tracing . https://www.jaegertracing.io/. [last access: May 26, 2021].

[15] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 152–166.

[16] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019). arXiv:1902.03383 http://arxiv.org/abs/1902.03383

[17] KNIX. 2021. KNIX - A High-performance, Open-source Serverless Computing Platform. http://knix.io/. [last access: May 26, 2021].

[18] ASC LAB. 2021. LAB Insurance Sales Portal - A Simplified insurance sales system. https://github.com/asc-lab/micronaut-microservices-poc/. "[accessed-April-2021]".

[19] Locust. 2021. Locust: An Open-Source Load Testing Tool. https://locust.io/. [last access: May 26, 2021].

[20] Maxogden. 2021. Callback Hell: Intuitively Code of Asynchronous JavaScript. http://callbackhell.com/. [last access: May 26, 2021].

[21] Micronaut. 2021. Micronaut Framework. https://micronaut.io/. [last access: May 26, 2021].

[22] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[23] OpenFaaS. 2021. OpenFaaS - Serverless Functions Made Simple. https://www.openfaas.com/. [last access: May 26, 2021].

[24] OpenZipkin. 2021. OpenZipkin: A distributed tracing system. https://zipkin.io/. [last access: May 26, 2021].

[25] Overleaf. 2021. Overleaf - An Open-Source Online Latex Editor. https://www.overleaf.com/. [last access: May 26, 2021].

[26] Tracy Ragan. 2021. Navigating the Microservice DeathStar With DeployHub. https://dzone.com/articles/navigating-the-microservice-deathstar-with-deployh. [last access: May 26, 2021].

[27] Mark Richards. 2021. Chapter 4. Microservices Architecture Pattern. https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch04.html. [last access: May 26, 2021].

[28] Chris Richardson. 2021. Pattern: Microservice Architecture. https://microservices.io/patterns/microservices.html. [last access: May 26, 2021].

[29] Amazon Web Services. 2021. AWS Lambda – Serverless Compute - Amazon Web Services. https://aws.amazon.com/lambda/. [last access: May 26, 2021].

[30] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable Cross-Language Services Implementation. *Facebook white paper* 5, 8 (2007), 127.

[31] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proc. VLDB Endow.* 13, 12 (July 2020), 2438–2452. https://doi.org/10.14778/3407790.3407836

[32] Stan. 2021. Robot Shop - A Sample Microservice Application. https://github.com/instana/robot-shop/. [last access: May 26, 2021].