Migrating from Monolithic to Serverless: A FinTech Case Study

Alireza Goli, Omid Hajihassani, Hamzeh Khazaei*, Omid Ardakanian, Moe Rashidi and Tyler Dauphinee
University of Alberta and York University*
Edmonton and Toronto*, Canada
{goli,hajihass,hkhazaei,oardakan,rashidi,tdauphin}@ualberta.ca

ABSTRACT

Serverless computing is steadily becoming the implementation paradigm of choice for a variety of applications, from data analytics to web applications, as it addresses the main problems with serverfull and monolithic architecture. In particular, it abstracts away resource provisioning and infrastructure management, enabling developers to focus on the logic of the program instead of worrying about resource management which will be handled by cloud providers. In this paper, we consider a document processing system used in FinTech as a case study and describe the migration journey from a monolithic architecture to a serverless architecture. Our evaluation results show that the serverless implementation significantly improves performance while resulting in only a marginal increase in cost.

CCS CONCEPTS

• **Software and its engineering → Software architectures**; • **Computer systems organization → Cloud computing**.

KEYWORDS

Serverless Computing, Monolith Architecture, Performance Evaluation, FinTech

ACM Reference Format:
Alireza Goli, Omid Hajihassani, Hamzeh Khazaei*, Omid Ardakanian, Moe Rashidi and Tyler Dauphinee. 2020. Migrating from Monolithic to Serverless: A FinTech Case Study. In *ACM/SPEC International Conference on Performance Engineering Companion (ICPE '20 Companion), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3375555.3384380

1 INTRODUCTION

Serverless computing is a new paradigm for developing applications and services, and a natural step in the evolution of cloud computing. It has emerged through the development of Function-as-a-Service (FaaS) and Backend-as-a-Service (BaaS) technologies [19]. In this paradigm, the application logic is broken into functions that are stateless in nature and are left to the serverless platform to manage. The granularity of serverless units has shifted from functions to stateless containers in recent years [5, 15]. Stateless containers are similar to stateless functions in the sense that they are triggered by an event, such as a HTTP request, and after the job execution completes, they disappear and do not save or carry any state information over to the next request. However, using stateless containers, developers are free to use the programming language and libraries of their choice to write code, which is not the case with stateless functions [16].

By supporting provisioning and autoscaling of resources while offering fine-grained pay-per-use billing [21], serverless computing has gained the attention of many enterprise and small-scale companies [3]. In recent years, several machine learning and analytics applications have been successfully migrated to serverless computing. The Siren distributed machine learning framework [24] and Graphless serverless graph analysis toolkit [23] are examples of these applications.

Today cloud providers, such as Amazon and Google, provide a comprehensive list of benefits and caveats for serverless application developers (see for example AWS Lambda [4] and Cloud Functions [13]). However, the serverless computing paradigm has a number of disadvantages which have been explored in the literature. One example is the potential performance degradation of applications with data intensive and communication dependent tasks and functions [18, 21, 23]. Thus, whether the transition to a serverless implementation makes sense, in terms of performance and cost, depends on the target application. This paper aims to answer this question for a rudimentary financial technology (FinTech) application.

We present a CPU and data intensive FinTech application as a case study and discuss the migration of this application from its crude monolithic architecture to a distributed, performant serverless architecture. We evaluate the system with serverless architecture in terms of performance and cost, and compare it with the original system before migration. We present different challenges and pitfalls that we faced in our implementation journey and discuss how we addressed them. For example, to overcome the performance issue of communication between services, we used non-blocking I/O, and to meet the resource constraints of the serverless platform, we divided machine learning models. In this work, instead of using stateless functions, we use stateless containers executed in a serverless manner by the Cloud Run platform [15].

The main contribution of our work is as follows:

- We migrate a real-world application with traditional monolithic architecture to serverless architecture.
- We address the gap in the literature by identifying the barriers and challenges in the process of migrating a monolithic document processing system to serverless architecture.

ICPE '20 Companion, April 20–24, 2020, Edmonton, AB, Canada

ACM ISBN 978-1-4503-7109-4/20/04…$15.00
https://doi.org/10.1145/3375555.3384380

- We evaluate the new system with serverless architecture in terms of performance, scalability and cost, and compare it with the original system before migration.

The rest of this paper is organized as follows. Section 2 provides background information about the terminologies that are used throughout the paper. Section 3 introduces the case study system which is used in this study. Section 4 explains the migration journey, motivation, challenges, and the proposed serverless architecture. Section 5 introduces the data set used in this work and evaluates the proposed serverless solution. Section 6 reviews related work on serverless computing, and Section 7 concludes the paper.

2 BACKGROUND

In this section, we provide background information and define the terminology used throughout the paper.

2.1 Serverless Computing

In serverless computing, despite what its name suggests, we still have servers that work in the background, but they are not visible to end users [19]. The cloud provider takes care of infrastructure management, allowing the developers to focus on the business logic of their applications [18]. Hence, as a result the developers do not need to worry about allocating resources and preventing under/over provisioning of resources. As the number of requests for a service increases the serverless platform scales the number of running instances to accommodate the increased demand. Similarly, when there is no workload it scales them down to zero. The cost model is also different from traditional computation resources in the cloud. In serverless platforms, users only pay for the time that they use a specific resource.

The serverless paradigm also brings several benefits to the cloud providers. Specifically, it benefits cloud providers by increasing the utilization of their servers that might not be appealing to users or platforms such as ARM and RISC-V to handle computations [19]. Moreover, multitenancy reduces the amount of idle infrastructure, thereby reducing the degree of under-utilization. These advantages make serverless appeal to cloud providers to the extent that today all main cloud providers offer serverless solutions, including Amazon AWS Lambda [4], Google Cloud Functions [13], and open source projects such as Apache OpenWhisk [8].

2.2 Monolithic Architecture

Monolithic architecture is a simple way of designing and implementing software systems. In monolithic architecture, all of the component are combined into a single tightly coupled piece of software. For brand new projects, this architecture may work well at the beginning, but as requirements evolve this architecture makes it more difficult for developers to adapt. This is one of the main disadvantages of monolithic architecture.

3 A FINTECH CASE STUDY

We focus on a document processing system which is commonly used by financial institutions as a case study. The document processing pipeline is a replacement for the laborious tasks of reading, classifying, and extracting information from financial documents such as bank statements, balance sheets, letters, etc. Historically, these tasks were done manually in banks and financial institutions but have been automated in the past couple of years.

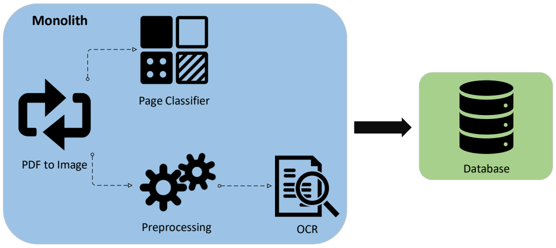

Figure 1: Original document processing pipeline before migration

The first version of this system is implemented in Python programming language and has a monolithic architecture. As shown in Figure 1, this system has five main components:

- PDF to Image Converter: Converts each page of a PDF document to image so that it can be processed by other components. This component is implemented using the PDF2Image library [10].
- Page Classifier: Gets an image as input and classifies it into one of the predefined document classes. This component is implemented using the Keras library [12] with Tensorflow backend [1].
- Preprocessing: Applies image processing and enhancement techniques to the input image, making it ready for the OCR component. This component is implemented using the OpenCV library [11].
- Optical Character Recognition (OCR): Extracts the text content from the input image. This component is implemented using the Tesseract OCR Engine [22].
- Database: Indexes and stores the output of the document processing pipeline for each input document.

The output of this system is fed to a Key-Value extractor system which is capable of extracting key-value pairs that the financial institutions are interested in.

4 MIGRATION JOURNEY

In this section, we walk through the migration journey and describe the challenges faced in each step. We start with the motivation behind the transition from a monolithic architecture to a serverless architecture. At last, we present the new serverless architecture.

4.1 Motivation

The original document processing system works well for a small number of input documents; however, it cannot handle a large number of documents in a reasonable amount of time within the non-functional requirements of large financial institutions. The two main issues with the original system are therefore speed and scalability.

We can imagine different ways to improve the performance of this system and make it process documents faster. For example, we can use multi-threading and parallel processing [25], but multi-threading may not be the most cost-effective solution because of the extra effort needed to tailor an existing software solution to a multi/many core execution environment. Furthermore, to achieve the best multi/many core performance there is a need for certain infrastructure and hardware scale-ups which may not be affordable. For these reasons the multi-threaded solutions could be less appealing compared to the serverless alternative.

For scalability, due to monolithic nature of the system, the only option for scaling without changing the implementation is vertical scaling by adding more resources to the machine that runs the application. This approach provides limited scalability and at some point cannot catch up with the growing demand [2]. Thus, the main motivation for migrating to the serverless architecture is to improve the performance and scalability of the system.

4.2 Breaking the Monolith

We started the migration journey by decoupling the system and breaking it down to a number of well-defined and loosely coupled services. As depicted in Figure 1, it is possible to identify four well-defined services in the original monolithic architecture. These four services are: PDF to image conversion, page classification, preprocessing, and OCR.

After further consideration and taking into account the communication overhead and also the relationship between services, we decided to merge the preprocessing and OCR services into one service. This resulted in three primary services that together comprise the document processing pipeline.

Afterward, we started to package each service along with dependencies into a Docker container that follows the stateless principle of serverless computing. Each service resides in a container and listens for HTTP requests and input arguments. We trigger the stateless services via HTTP request and pass them the necessary arguments. The page classification service takes the image of a document as input and returns the prediction result as output. The input for the OCR service is also the image of the document; it returns the extracted text from the image.

However, after deploying this first architecture we faced several challenges that led to modifying this architecture. In the next section, we review these challenges and changes that we implemented.

4.3 Migration Challenges

After implementing the first version of the serverless document processing pipeline based on the architecture in the previous section, we faced two main challenges. They can be attributed to the constraints that exist in the implementation and serverless solution provided by the cloud provider, which in our case was Google Cloud Platform (GCP).

The first challenge is the constrained resource of the serverless platform. This challenge is mainly due to the limitations and constraints that GCP imposes on instances of Cloud Run service. In particular, the maximum amount of memory that can be allocated to a Cloud Run service is 2 GB, but the page classification service needs more memory. To mitigate this challenge, we split the classifier component into two parts by breaking the trained neural network model, as shown in Figure 2. The first part performs a part of the prediction task and sends the output of the last hidden layer to the second part, which subsequently provides us with the final prediction, i.e., the document class. Therefore, the page classification service is replaced by two separate services Page Classifier 1 and Page Classifier 2.

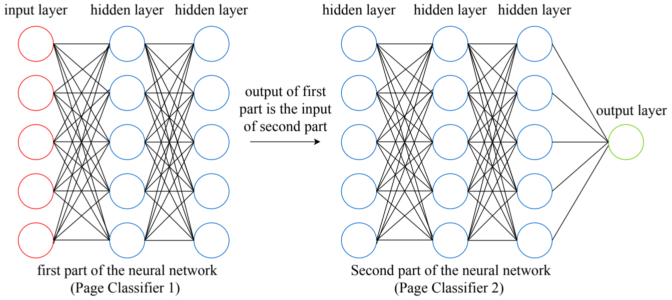
input layer
hidden layer
hidden layer
output of first part is the input of second part
hidden layer
hidden layer
hidden layer
output layer
first part of the neural network (Page Classifier 1)
Second part of the neural network (Page Classifier 2)

Figure 2: Structure of page classifier neural network after splitting.

The second challenge was related to the way that serverless services communicate with each other. In the first implementation, synchronous HTTP requests were used for communication between services. So when we send an HTTP request in a synchronous manner, it blocks the execution of the program and waits for the response to arrive. In scenarios where we can execute other tasks at the same time, this paradigm diminishes the performance of the system and leads to wasting a considerable amount of computation time without doing useful computation, considering the cost model of serverless platform. To deal with this problem, we replaced synchronous HTTP requests with asynchronous HTTP requests for communication between services. Asynchronous HTTP requests follow the non-blocking programming paradigm in which after sending a request, the program can continue running other statements and do some useful task until the response arrives. Hence, in the document processing pipeline, when we send an asynchronous HTTP request the program can continue sending other I/O requests, aggregate, and store the results of requests and complete the job in a shorter time in comparison with synchronous mode. We observe in the evaluation section that using asynchronous HTTP requests instead of synchronous HTTP requests improves the performance of the system tremendously. In the next section, we present the final serverless architecture after migration for the document processing pipeline.

4.4 Serverless Architecture

Figure 3 provides a high-level overview of the serverless architecture of the document processing pipeline after migration. In the new architecture we have the following serverless services:

- Dispatcher: Receives a Pub/Sub message that contains the bucket name and the list of document names. It then assigns each document to an instance of the coordinator service.
- Coordinator: Plays the role of an orchestrator in the pipeline. This service is responsible for executing the pipeline logic on a single document, aggregating the results, and storing the results in BigQuery.

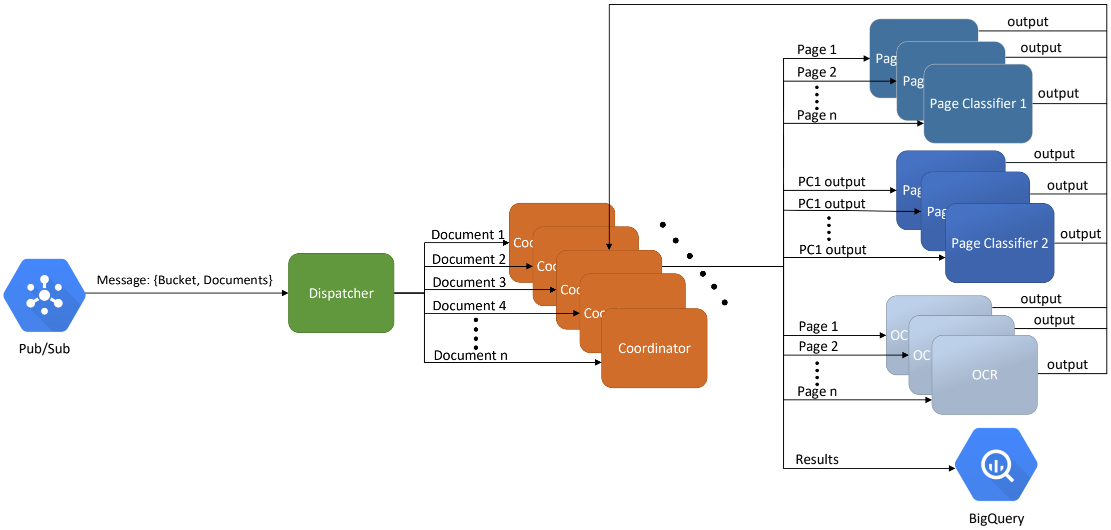

Figure 3: Serverless Architecture of Document processing pipeline.

- Page Classifier 1: Gets an image as input and performs the first part of the classification task and sends the last hidden layer output to the coordinator service.
- Page Classifier 2: The input for this service is the output of Page Classifier 1. It executes the second part of the classification task and returns the final prediction to the coordinator service.
- OCR: Gets an image as input, extracts the text content from the image after preprocessing, and sends it to the coordinator service.

In this architecture, we leverage the inherent parallelism in the serverless platform which is the ability to launch a new instance per request to speedup the document processing pipeline. As Figure 3 shows, in this new architecture the coordinator service gets a document as input and splits the document into pages. Next it feeds each page to the Page Classifier 1 and OCR services at the same time using asynchronous HTTP requests. This means that each page is sent to a separate instance of OCR and Page Classifier 1 services and processed in parallel with other pages; this leads to higher throughput and increased performance.

Page Classifier 1 carries out the first part of the whole classification job and sends the output of the last hidden layer to the Page Classifier 2 through the coordinator service. Page Classifier 2 carries out the rest of the classification job and returns the final prediction to the coordinator service. In the OCR service, each page is preprocessed first and the text content is extracted and returned to the coordinator service. The coordinator service receives the results from all services, aggregates and stores them in BigQuery.

To simply show the interaction between different serverless services in the new architecture, we pass a single document with one page to the pipeline as depicted in Figure 4.

5 EVALUATION

In this section, we evaluate the serverless document processing pipeline in terms of performance, scalability, and cost, and compare it with the old monolithic version.

5.1 Dataset Description

Due to privacy issues regarding financial documents, we cannot publicly share the dataset of financial documents that we used in our system; instead we take advantage of the publicly available RVL-CDIP dataset for classifier training and performance/cost evaluation of the new system[1] [17]. This dataset contains 400,000 document images in 16 different classes. These 16 classes are letter, form, email, handwritten, advertisement, scientific report, scientific publication, specification, file folder, news article, budget, invoice, presentation, questionnaire, resume, and memo. We stitch random samples from RVL-CDIP together to form a synthetic dataset of PDFs between 7-15 pages long.

5.2 Experimental Setup

We deploy each service in the proposed serverless architecture on GCP as a Cloud Run service. 2 GB of memory is allocated to each instance of the service. We also run the monolithic version on two separate virtual machines with different specifications on GCP to observe the effect of adding more resources on the scalability and performance of the monolithic version. The first virtual machine has 4 vCPU and 15 GB of memory, while the second one has 96 vCPU and 360 GB of memory.

We conduct two experiments. In the first experiment, we compare the serverless architecture with the monolithic one in terms of performance and cost by benchmarking both systems on a set of 100 documents that have between 7 and 15 pages. In the second experiment, we evaluate the performance and scalability of the

[1] https://www.cs.cmu.edu/ aharley/rvl-cdip/

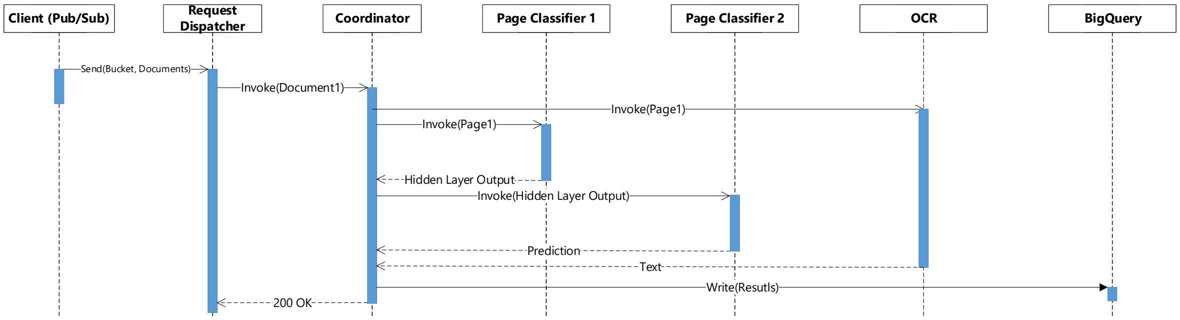

Figure 4: Sequence diagram of serverless document processing pipeline for one document with one page.

Table 1: Comparing the monolithic architecture with serverless architecture in terms of performance and cost

Architecture	Spec	Turnaround Time	Cost
Monolithic (Single VM)	4 vCPU, 15 GB (n1-standard-4)	~ 6.27 h	~ $1.23
Monolithic (Single VM)	96 vCPU, 360 GB (n1-standard-96)	~ 5.07 h	~ $23.12
Serverless (Cloud Run)	1 vCPU, 2 GB	~ 4.05 min	~ $2.43

new architecture as the number of input documents increases. We repeat each experiment three times and ignore the result of the first run to eliminate the cold start effect. Hence, we report the average of the results of the second and third runs.

5.3 Results

Table 1 shows the performance and cost results obtained for the old system and the new system on 100 documents in the first experiment. It can be readily seen that the serverless version is about 93x faster than the monolithic version; it processes the documents in about four minutes, whereas it takes about six hours for the monolithic version with 4 vCPU and 15 GB of memory to finish the same job. Turning our attention to cost, the monolithic version with 4 vCPU and 15 GB of memory turns out to be about 49.38% cheaper than serverless version. Note that we have not taken into account the idle time for the virtual machine in our cost calculations. The cost for the virtual machine is only for the duration that it processes the document, so if we consider the idle time in the cost calculation (which is the case in the real-world setting), it would become more expensive than the serverless solution.

We also repeated the first experiment for the monolithic version on a more powerful virtual machine with 96 vCPU and 360 GB of memory to understand to what extent we can improve the performance and reduce the turnaround time by adding more resources. As expected, adding more resources did not greatly improve the performance despite increasing the cost dramatically. We attribute this to the sequential execution of the monolithic version.

In the second experiment, we change the number of input document to the serverless pipeline from 1 to 500 documents to measure the performance of the new system on different loads. Figure 5 illustrates the turnaround time as we increase the number of input documents. The x-axis of this figure shows the total number of documents submitted to the system where each document consists of multiple pages (11 pages on average).

Due to the limit set by GCP, each Cloud Run service can currently scale to a maximum of 1000 instances simultaneously. Thus, the number of instances varies between 0 to 1000 based on the input workload. As we invoke some services such as Page Classifier 1 for each page, at some point, these services reach this instance limit, causing the next requests to experience a waiting time before getting service. Therefore, we observe an almost linear increase in the turnaround time as the number of documents increases.

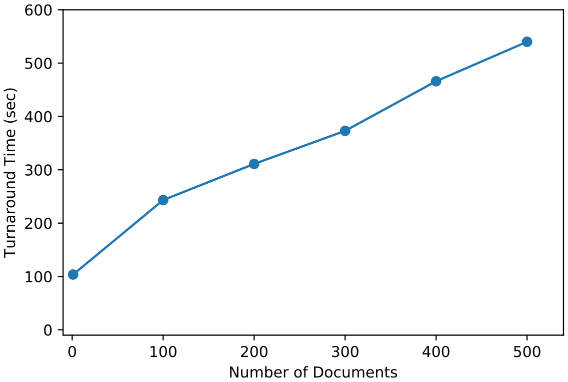

Figure 5: The turnaround time of jobs in the serverless document processing pipeline.

6 RELATED WORK

The body of research on serverless computing can be categorized into two groups: studies on serverless computing evolution and open problems [9, 20, 21], and studies on migrating applications to the serverless computing paradigm which is the focus of this paper [3, 7, 23, 24].

In recent years, due to the appealing characteristics of FaaS serverless solutions such as fine grained billing benefits and implicit autoscaling, many companies and developers have migrated their solutions to the serverless architecture to benefit from these advantages and provide better services to their costumers. Such companies include Reuters, iRobot, Autodesk and many more that reportedly use AWS Lambda to better serve their customers and internal processes [6]. In Toader et al. [23], the authors proposed a detailed implementation of a serverless graph processing framework implemented with AWS Lambda which benefits from automated resource autoscaling and provisioning. Graphless abstracts away resource management and configuration from the users. This will benefit end users who are not familiar with HPC concepts. However, the authors have shown that because of the variability of network characteristics under certain communication intensive workloads, the Graphless efficiency degrades.

In [24], the authors proposed a serverless framework for machine learning tasks, called Siren, which is fully asynchronous and achieves different levels of parallelism and elasticity. Siren, through stateless serverless functions, much like Graphless eliminates the burden of resource management and scaling machine learning algorithms imposed on the end users by the current well established serverfull frameworks. The Siren framework is deployed on the AWS Lambda serverless platform. The authors in [3] discussed economic and architectural benefits of serverless computing and study two real world services, namely MindMup and Yubl [3], that have been migrated to and adopted serverless computing. They discussed how MindMup and Yubl could benefit from serverless deployment; these benefits include reduced time to feature delivery and time to market for developers and faster request processing for customers. However, it is mentioned that serverless platforms are not well suited for mission critical and time sensitive tasks [3].

Moreover, a recent innovative step and improvement in the serverless computing research and development is the replacement of functions in the FaaS serverless architecture with containers and images from the developers. These containers are stateless and their autoscaling and all the other deployment details are handled by the cloud providers. Such container-oriented serverless solutions include products such as the Google Cloud Run [15] or the Amazon Fargate [5], both of which are serverless solutions for containers. The upside of serverless containers compared to functions is the level of concurrency introduced by containers which refers to the number of user requests processed by each container. The developers are free to assign concurrency levels higher than 1 to each container, for example, a concurrency level of 4 indicates that 4 user requests are processed by each container. The concurrency level in each function in the FaaS serverless solutions is set to 1, meaning that each function processes 1 request at a time [14].

7 CONCLUSION

With the advent of serverless computing, several monolith applications which were previously developed for a single-core or multi-core execution environment are implemented from scratch following the serverless paradigm to take advantages of auto-scaling and automated resource provisioning. In this article, by observing the gap in the literature and the lack of studies concerning the performance of the serverless implementation of financial services, we present the migration journey of a FinTech application from its monolithic form to a high-performance serverless implementation. Based on our evaluations, the proposed serverless implementation outperforms the previous monolith serverfull implementation by 93 times, while increasing the cost by 50%. Our cost calculation does not take into account the under utilization of the serverfull infrastructure and the investment required to build the serverless system. In conclusion, the serverless implementation provides unparalleled speedup and performance improvement over the serverfull implementation without making drastic changes to the software design.

REFERENCES

[1] Martín Abadi et al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Martin L Abbott and Michael T Fisher. 2009. *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise.* Pearson Education.

[3] Gojko Adzic and Robert Chatley. 2017. Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 884–889.

[4] Amazon. 2019. *Amazon AWS Lambda.* https://aws.amazon.com/lambda/

[5] Amazon. 2019. *Amazon Fargate.* https://aws.amazon.com/fargate/

[6] Amazon. 2019. *AWS Lambda Costumer Case Study.* https://aws.amazon.com/lambda/resources/customer-case-studies/

[7] Lixiang Ao et al. 2018. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*. 263–274.

[8] Apache. 2019. *OpenWhisk.* https://openwhisk.apache.org/

[9] Ioana Baldini et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[10] Belval et al. 2020. A python module that wraps the pdftoppm utility to convert PDF to PIL Image object. https://github.com/Belval/pdf2image.

[11] G. Bradski. 2000. The OpenCV Library. *Dr. Dobb's Journal of Software Tools* (2000).

[12] François Chollet et al. 2015. Keras. https://keras.io.

[13] Google Cloud. 2019. *Cloud Functions.* https://cloud.google.com/functions/

[14] Google. 2019. *Cloud Run Concurrency Concept.* https://cloud.google.com/run/docs/about-concurrency

[15] Google. 2019. *Google Cloud Run.* https://cloud.google.com/run/

[16] Google. 2019. *What is serverless?* https://cloud.google.com/serverless-options

[17] Adam W Harley, Alex Ufkes, and Konstantinos G Derpanis. [n.d.]. Evaluation of Deep Convolutional Nets for Document Image Classification and Retrieval. In *International Conference on Document Analysis and Recognition (ICDAR)*.

[18] Joseph M Hellerstein et al. 2018. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651* (2018).

[19] Eric Jonas et al. 2019. Cloud programming simplified: a berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[20] Nima Mahmoudi, Changyuan Lin, Hamzeh Khazaei, and Marin Litoiu. 2019. Optimizing serverless computing: introducing an adaptive function placement algorithm. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*. 203–213.

[21] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[22] Ray Smith. 2007. An overview of the Tesseract OCR engine. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, Vol. 2. IEEE, 629–633.

[23] Lucian Toader et al. 2019. Graphless: Toward serverless graph processing. In *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 66–73.

[24] Hao Wang, Di Niu, and Baochun Li. 2019. Distributed Machine Learning with a Serverless Architecture. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 1288–1296.

[25] Jianlong Zhong and Bingsheng He. 2013. Medusa: Simplified graph processing on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 25, 6 (2013), 1543–1552.