Optimizing and Extending Serverless Platforms: A Survey

Maziyar Nazari, Sepideh Goodarzy, Shivakant Mishra, Eric Rozner
Department of Computer Science
University of Colorado Boulder
maziyar.nazari@colorado.edu, sepideh.goodarzy@colorado.edu,
mishras@colorado.edu, eric.rozner@colorado.edu
Boulder, Colorado 80309
Phone: (303) 492–7514
Fax: (303) 492–2844

Eric Keller
Electrical, Computer, and
Energy Engineering Department
University of Colorado Boulder
eric.keller@colorado.edu
Boulder, Colorado 80309
Phone: (303) 492–0125
Fax: (303) 492–2758

***Abstract*—Serverless Computing is a new cloud computing paradigm wherein people in academia and industry are actively proposing either interesting improvements or building excellent applications on top of it. AWS, Google Cloud, Microsoft Azure, and IBM are popular samples of public clouds that offer Function-as-a-Service on top of their Serverless Computing platforms. Although this paradigm has had numerous advantages for software developers and programmers, it has introduced new challenges to cloud providers. Factors like fine-grained pricing and pay-as-you-go manner, eliminating the responsibility of resource management on the developer side, promises of elasticity and highly-available service, fault tolerance, auto-scaling, and being able to run embarrassingly parallel jobs make it a suitable platform for developers. On the other hand, efficient resource management, offering low-latency service, and providing proper security/isolation have been partly the main challenges introduced on the cloud provider side.**

This paper presents a literature review on today's Serverless platform optimizations and extensions that people have proposed and implemented to further capitalize the Serverless infrastructure. In the end, we will provide the current Serverless paradigm's limitations and a few future directions and research opportunities regarding Serverless Computing.

***Index Terms*—Serverless Computing, Function-as-a-Service, Cloud Computing, Optimization, Extension**

I. Introduction

In traditional cloud computing paradigm, VMs formed the original basis for Infrastructure as a Service that enabled developers to skip the steps of buying hardware, and simplified the deployment of applications. Containers took that one step further into a more efficient deployment model. But, in each case, developers still need to develop a lot of software to orchestrate their applications and deal with the boundaries of a machine as a deployment unit.

Serverless was introduced more recently to solve these problems and allow application developers to just define their application functionality. In effect, as the name suggests, it relieves the developers from dealing with the complexities of setting up and maintaining servers. It has become an interesting topic for researchers and the cloud industry that are actively trying to either improve it or build traditional applications atop it. Cloud providers have traditionally offered services like IaaS, PaaS, SaaS for years, and now after the emergence of Serverless, have started offering Function-as-a-Service (FaaS).

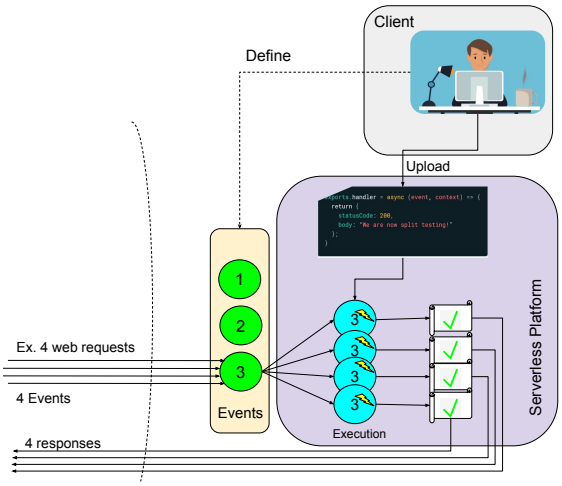

Fig. 1. Serverless Computing Workflow

FaaS has many benefits that have attracted software developers and industry, although it has some limitations as well. Unlike the typical use cases of cloud computing wherein users have to rent VMs or bare metal servers and manage their clusters themselves for a specific amount of time, they only have to upload their function code on a Serverless platform and specify an event to trigger the function execution. Users do not need to worry about any resource management and pay for only the actual function execution time, which can be in the order of seconds, as opposed to renting a VM or bare metal server for an hour. For example, the minimum time we can rent an AWS EC2 instance is an hour [1]. Even if we want to run a highly-parallel function once in an hour for ten seconds, we will have to rent the instance for the entire hour. Contrast that to using the serverless platform AWS lambda [2] instead and only pay for that ten seconds.

There have been several exciting applications leveraging Serverless platforms, especially for running highly parallel jobs. Works like ExCamera [24] and gg [23] are exciting tools to run everyday jobs and common applications such as video encoding atop AWS Lambda. PyWren [26] and IBM-PyWren [31] ease building big data analysis applications using FaaS. Cirrus [21] showed that we could have a complete ML pipeline working atop a Serverless platform and a ton of other exciting applications. Although these papers show groundbreaking results and propose exciting new directions, there are still some major gaps between the current state-of-the-art and the potential benefits that serverless computing can enable, and indeed there is large room for exciting new research in this area.

On the other hand, cloud providers have been trying to optimize Serverless infrastructure with several different goals. These goals include reducing latency [16], [29], [35], increasing resource efficiency [22], reducing the number of Cold Start (described in section III-A) [22], [32], and customizing Serverless platform to perform well for specific use cases [18], [36].

In this paper, we present a survey discussing optimizations that have been done to address different shortages in the Serverless platform, and extensions that people have proposed to improve the Serverless cloud further. The picture depicted in Figure 2, roughly shows the focus of each of the key papers in the area of Serverless cloud environment. We conclude the paper with some future research directions that we believe will be attractive to researchers interested in this field.

II. Overview

Serverless Computing is a recent paradigm that emerged in cloud computing that has a general workflow, as shown in Figure 1. Developers write their application code (function), upload it to the Serverless platform, and define one or more events that can trigger that function on the cloud platform. An event can be an HTTP web request, a read/write request to a database residing in the cloud, another Serverless function, and so on. Developers can also start their function execution themselves regardless of any specific event. For each instance of an event trigger, the Serverless platform creates and executes an instance of the corresponding function with proper inputs according to the event content and returns the result after the execution completes. For example, in Figure 1, a developer has defined three events that each can trigger his/her function, already uploaded to the platform. Event 3 has been triggered four times concurrently that has lead to the spawning of four instances of the function and accordingly four results.

In the current state-of-the-art Serverless platforms that provide Function-as-a-Service, **Containers** play an key role. They are considered the unit of execution in this approach. As we mentioned above, a developer only uploads his/her function code to the cloud platform and defines events that can trigger the execution of their function. They do not need to worry about the underlying VM, runtime environment, hardware, etc. Cloud providers typically provide the runtime environment suitable for function execution by using containers. When an event occurs on the cloud provider side, a container is created, and the function code is executed in that container with the corresponding input.

Serverless computing comes with many popular inherent features. The pricing model is pay-as-you-go, due to which the developer only pays for the resources used for the function execution, and there is no need to rent a VM or cluster to run a regular short job in the cloud. FaaS providers propose auto-scaling, as a result of which the developers do not need to do any workload monitoring or worry about resource management. The cloud provider will upscale and downscale based on demands. Developers can run thousands of function instances simultaneously; thus, they can run embarrassingly parallel jobs (although there are some limitations that we will address in the rest of this paper) atop this platform. Function instances are time-limited, and basically, they are short-lived jobs (for example, AWS lambda currently offers a 15-minute timeout). Each function instance has limited memory and CPU, and they are stateless. Usually, people use external storage services like S3 [3] to maintain state or for inter-function communication. We will go into more detail in the following sections, and at the end of this paper, the reader will have a better insight into Serverless platform designs and implementations.

As it is depicted in Figure 2, Serverless platform generally consists of a management layer that does the function placement, scheduling, orchestration, etc and several server nodes that each has multiple layers to provide the infrastructure to run function instances. Since people have been using a distributed message bus for inter-function communication or persisting program state, we also included this component in the picture. Moreover, we mentioned each project's reference number according to where the corresponding paper has contributed. For example, [32], [34], [36] have some significant contributions related to the management layer to achieve their goals. Likewise, in [18], [22], [23], [26], [31], [34], [36], we indicate contributions in both application code layer and providing appropriate runtime, libraries, etc for that. [16], [22], [29], [35] dug into lower levels and tried to address some drawbacks in the containers, networking, etc, and their optimizations will benefit the entire Serverless platform. Distributed message bus played a vital role in [16], [34] as indicated in the picture, and [35] studied the networking between function instances, thus we placed that on the dashed line between container layers.

Due to the promise of Serverless computing, there has been an increasing amount of research in the space that we discuss partly in this paper. Generally, the research has centered on three main topics:

- **What can we do on Serverless today?**
  Serverless has many great benefits, but it's an architecture with a specific way to design applications for it. The first area of research revolves around understanding what applications are possible to run on current platforms. The research has shown that embarrassingly parallel

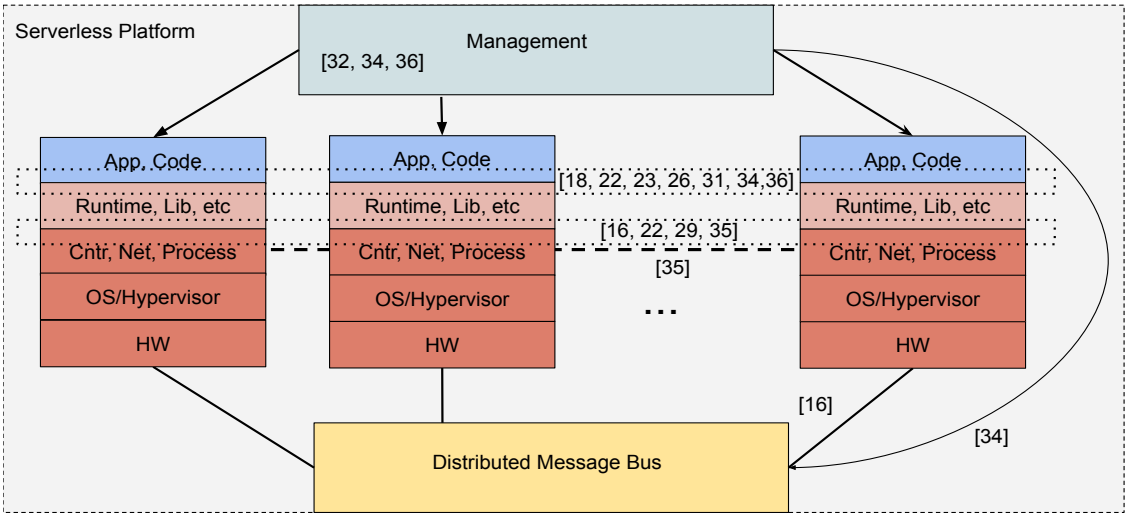

Fig. 2. Overview of the projects discussed in this survey

applications, like machine learning, are a good fit right now.

For instance, PyWren [26] and IBM-PyWren [31] proposed a new way of running distributed and big data analytics applications on top of Serverless infrastructure instead of using popular platforms like PySpark [4] and Hadoop [5]. Furthermore, to offload processing and computation to the cloud, gg [23] opened a new window on offloading everyday jobs like program compilation with 1000-way parallelism using the Serverless cloud. ExCamera [24] and Sprocket [17] offered video encoding/decoding using Serverless platform. People have also studied the use case of Serverless Computing in special areas. For example, SNF [34] has provided a Network-Function-as-a-Service infrastructure in which it has addressed major challenges in offering VNF service atop Serverless platforms.

- **How can we improve execution?**
  There have been lines of research on reducing the function start-up latency (including solving the Cold Start problem) and proposing more efficient inter-function communication solutions to make the Serverless platform a viable option for a diverse range of applications.
- **How can we extend the applicability of Serverless?**
  This research line proposes new programming language primitives, fault tolerance mechanisms, concurrency patterns, etc to enrich the Serverless computing platforms further and make them suitable for more general-purpose applications and easy to use.

The rest of the paper is outlined as follows: Section III summarized the projects that optimized the Serverless platforms for either cold starts and startup latency or inter-function communication. Section IV will list a few projects that tried to extend the Serverless platform to make it suitable for more general-purpose applications. In the end we propose some gaps and future directions in Section V and we conclude the paper in Section VI.

III. Serverless Platform Optimizations

In this section we focus on the optimizations that have been done so far to improve Serverless infrastructure categorized in 2 topics:

A. Cold Start & Startup Time Optimizations

A famous problem in the Serverless FaaS that people are actively trying to optimize for is the containers' cold start. As mentioned earlier, typically, each function instance runs in a container in the FaaS platform. Cold start refers to when a container boots up (downloads the code and required libraries, initializes its runtime, and bootstraps) from scratch for the first time. When the container has already done these steps, it is ready to run and execute the function code and return the result. In the latter case, we call the container warm. There is a trade-off between wasted memory and the number of cold starts. Thus, if the cloud providers keep the containers hosting function instances always warm, they would have colossal memory wastage since each container's code, libraries, runtime, etc should permanently reside in memory. On the other hand, if they throw the container and its related content out of the memory immediately after the function execution, we would encounter cold start latency at each function invocation. Consequently, there is a sweet spot on which we can have reasonable cold start numbers and resource waste on balance.

In a more general context, researchers have been trying to reduce the start up time of the function instance to make the Serverless platform more reactive and resource efficient. We summarize some of the related works below.

To begin with, firstly, we explain a few works wherein reducing the number of cold starts and start up time have been one of their major goals categorized by their methodologies.

1) Smart Algorithms: [32] is a joint work of Microsoft Research and Microsoft Azure, in which they analyzed Azure Functions [6] workload statistically in detail, and they presented their solution implemented on their Serverless platform, taking the analysis into account. Accordingly, they monitored the Azure Functions workload for two weeks [7] and analyzed that to know how functions are accessed, what resources they use, and how long they run. They provide a great deal of workload characterization in their paper, although in this survey, we focus on the Serverless platform optimization part of their project. They found out that the Serverless workload types are so variant (constant, bursty, Poisson distribution, etc) and that there is no one-size-fits-all solution to determine the memory residency interval for the functions or predict the function start times.

They claimed current Serverless platforms incorporate the fixed keep-alive policy (keep alive: the time during which the container is kept warm between 2 invocations). They added, AWS Lambda keep-alive interval is around 10 minutes (obtained by reverse engineering), and Azure Functions is around 20 minutes. However, it does not perform well on a very simplistic case in which the function is called, for instance, every 11 minutes on AWS Lambda, and the function faces a cold start every time.

They proposed and implemented the Hybrid Histogram Policy for predicting the container's keep-alive time to have a sort of dynamic policy adapting to each container workload. They used a combination of Histogram-based approach, ARIMA [20] time series forecast when there are too many out of bounds, and being conservative (specifying maximum keep alive time) in their model instead of just using a fixed keep-alive policy. After they observed good results applying their solution to the workload, they implemented their system on OpenWhisk [8], and showed that it only has less than 1ms added latency to the critical path while reducing memory wastage, container cold starts, and function execution time.

2) Relaxing Function Isolation: Looking deeper into the cold start problem, many researchers have figured that current isolation between the Serverless function instances is not necessary. Each container has its own runtime, libraries, etc, built on top of lower layers that provide Namespace isolation and Control Groups concept, making the typical container a heavyweight and slow option for Serverless functions. Starting a Serverless function instance in a separate isolated container may cause inefficiencies in terms of resource and latency in special cases that several projects have addressed. For example, function instances of a single application coming from a tenant may trust each other and share some prerequisites; thus, they can be run in a less isolated environment than containers which are more lightweight, quicker, and more efficient.

More Appropriate Containers - SOCK's [29] main focus is to reduce the startup time of the cloud function instances. For example, they claimed, considering running a Python function that uses "scipy" library as a Serverless function, it is expected to encounter a couple of seconds for runtime and library initialization before the actual execution, which can be by orders of magnitude more than the execution time of the function itself.

They proposed a new serverless-optimized container as a part of the greater project called OpenLambda [9]. Three significant contributions that they have made in this project include: Changing the predefined Linux isolation mechanisms (namespaces) for containers to more lightweight approaches like bind mount, "namespaces + chroot," instead of using flexible, expensive union file system used in Docker [10]. Applying the idea of Zygote process in Android OS on the Serverless platforms, they could put common libraries in the Zygote process. Thus, by forking new instances from the Zygote process and adding libraries if needed, they can reduce the function startup time and leverage kernel features like copy-on-write mechanisms, and reduce the containers' overhead. Finally, they implemented a 3-tier caching mechanism to reuse initialized runtime within a lambda, reuse initialized Zygotes between lambdas, and reuse installed packages between lambdas using handler cache, import cache, and install cache, respectively.

They evaluated their system for the first contribution using initiating no-op handlers with different parallelism levels using SOCK against Docker containers. As for the second contribution, they created and destroyed handlers' runtime as quickly as possible and measured the operations per second against Docker. Furthermore, to evaluate the last contribution, they sent two random requests per second to an OpenLambda [9] worker machine hosting 100 distinct lambdas, all importing django [11] library, and measured the end-to-end latency. They could achieve 18x better performance using SOCK instead of Docker containers as a result of their first contribution, 3x startup latency improvement using the idea of Zygote processes and forking new instances from that, and 3 - 16x speed up using a 3-tier caching mechanism to reuse the runtime and libraries that are already deployed.

Application-level Sandboxing - SAND [16] starts with explaining an image processing pipeline running on the Serverless platform consisting of 5 functions as their motivating example. By running this pipeline on multiple Serverless platforms (AWS Lambda, IBM Cloud Functions, and Apache OpenWhisk), they noticed that this application's total execution time is around twice as much of the actual computation time. They mentioned some of the best practices in the existing Serverless platforms. As described earlier, if the FaaS provider starts a new container for every function trigger, it will encounter so many cold starts. On the other hand, if they keep the containers warm to handle the new incoming requests, it has too much resource inefficiency. Since they have a fixed timeout, they still suffer from a non-optimal number of cold starts which will add delays.

Their first key idea is proposing different levels of isolation for applications and functions. They isolate applications in containers, and they run application functions in the same container as processes. Whenever a new event happens, they fork from the corresponding process in the application container. This way, they can leverage OS features like copy-on-write and resource de-allocation when the function execution

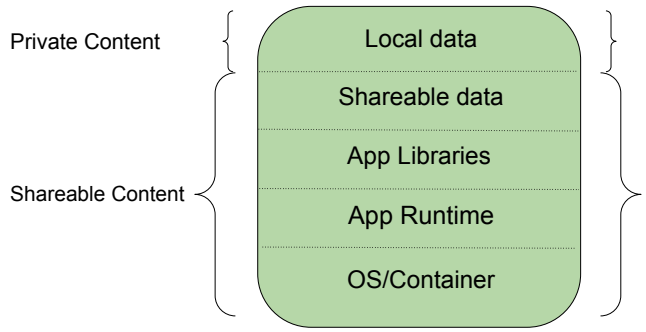

Fig. 3. Function Execution Stack. This graph is especially for concurrent functions of an application that can share data according to [22]

is finished. Consequently, by forking, their system reduces overhead compared to creating a container for each function instance. We discuss their second key idea in the next section to improve inter-function communication.

Their application-level sandboxing mechanism provided faster startup, automatic de-allocation, and low-execution footprint of Serverless applications.

Share the Shareables - In Photons [22], they started by describing a motivating example that leads to their work. Let's say we are running a Tensorflow for image classification on top of the Serverless platform. Currently, each instance of the function will have a separate execution stack, including the container, runtime, libraries, and shared data, which is the ImageNet model [12] and the local request data. They argued that this level of isolation between the functions (function execution stack) of the same application is not necessary. They added that 70-90 percent of these functions' memory footprint is shareable (The first four layers in Figure 3), and only the rest of it is private and specific to the function instance.

They used language level isolation to isolate different invocations of the same application, and they ran the function instances on the same JVM runtime. As for the shareable data, they included a shared object store accessible by the function instances that has GET/SET API and locking mechanism so that functions can use for intermediate communication. Since language level isolation is not as strong as VM/container isolation, they only co-locate the same function instances on the same runtime with shareable data.

Using Photons, they reduced the memory consumption of running 100 concurrent functions up to 5x. Their project works best if the number of concurrent functions increases in the Serverless platform. According to the analysis that people have done on Microsoft Azure Functions [32], 99 percent of the invocations are coming from less than 10 percent of the functions, which means Serverless functions typically run concurrently. Benchmarking their system using Microsoft Azure workload, they saved 30 percent memory space using Photons, and they claimed if they used this memory space to keep more containers warm, they could reduce the number of cold starts by 52%.

Network Startup Optimization- Another issue regarding the existing Serverless platforms is the communication between cloud functions. People used to use S3 for intermediate communication, which is slow for a short-lived job. ElastiCache [13] and Pocket [27] are faster, but sometimes they are expensive, and since they are all storage-based mechanisms, they are not a viable option for bursty parallel short-lived jobs.

In Particle [35], they dug into overlay networks to make them suitable for bursty Serverless functions. According to the paper, overlay network startup time is responsible for a huge portion of the entire container startup latency. They proposed Particle, that interconnects related Serverless functions with low overhead in overlay network startup.

To give the reader an overview of overlay networks and how they are created, there are six steps to have a fully functional overlay network: 1- Setup network namespace 2- Setup new host devices (create guest and local Virtual Ethernet Devices) 3- Move the guest VETH into the container 4- Move local VETH into VTEP namespace (controller ns residing on the host) 5- Attach local to VXLAN VNI bridge and 6- Attach IP and MAC to guest VETH. Using eBPF, they profiled each step's latency, and they came up with the result that 92% of the total network startup time is for steps 3 and 4, wherein moving to namespaces happens. Mainly this overhead is because of calling a kernel function named dev_change_net_namespace, which is called in both startup and cleanup. This function holds a lock during this move. Thus, each container's overlay network creation makes the kernel go through locking twice (in both steps 3 and 4). They argued that this level of isolation is unnecessary for hundreds of serverless threads for the same tenant or user job (like a program compilation using gg [23]).

In Particle, they do steps 1 to 4 only once at the beginning of starting a Serverless job and do steps 5 and 6 for every new instance of that user job based on their assumption that these instances do not need to be isolated strongly. Thus, they put all the instances in the same network namespace and skip creation and moving of network namespaces and devices steps by using Particle. Their system has a limitation that if we change a network rule, it will affect all the containers created for that specific user job.

They can reduce the network startup time by up to 5x compared to Weave, Docker Swarm, and Linux Overlay. They also ran Sprocket [17], a Serverless video processing pipeline, and they showed that by using Particle, they reduced the startup time to 18% of the total runtime while 71% of the total runtime is for the actual processing. This number for Docker Linux Overlay is 62% and 35% for startup and processing, respectively.

B. Inter-function Communication

One of the exciting research topics in Serverless computing is to improve the communication between the function instances. Since the Serverless instances are not network-addressable (although they can initiate connections with external services), people should use storage services or distributed message bus for function interaction, but they are not suitable, especially for bursty Serverless workload. For example, on

AWS, using S3 adds much overhead to the function execution time. Using in-memory k/v stores and cache may help have less latency, but it is more expensive than external storage services or less efficient than direct network communication. In this section, we discuss a few projects related to this objective, and we will discuss some further works in the next section since the authors provided communication primitives included in their proposed extensions for Serverless platforms.

In SAND [16] that we discussed in the previous section, besides their first key idea of having application-level sandboxing and leveraging application functions co-location, they proposed a second contribution to optimize the inter-function communication. Since function instances are not able to usually talk to each other through a network, if functions of the same application want to interact, they have to do it via the distributed message bus (or distributed storage like S3 in AWS) even if they reside on the same host which adds extra latency to the application total runtime.

Their second key idea is implementing a hierarchical message bus to address high communication overhead between function instances. They implemented a local message bus on each host, so the functions interacting on the same host do not go all the way through the globally distributed message bus to interact with each other. Along with their application-level sandboxing mechanism, this hierarchical message bus offers a notable improvement in execution time and startup time since now the co-located functions can interact via local message bus. They also implemented a coordination mechanism between the global message bus and the local message bus to keep them synchronized firstly and secondly, the global message bust acts as a back up for the local ones, which provides fault tolerance.

They showed their local message bus is also 3-5X faster than the global one. Function interaction is 8.3x faster than Apache OpenWisk and 6.3x faster than AWS Greengrass.

In Photons [22], we also mentioned that function instances co-located on the same Java Virtual Machine could share states with each other via shared object store by the API they provided to write photon-friendly functions, and in this way, they have less communication overhead than using distributed storage service like S3.

IV. Serverless Platform Extension

Now that we covered a few novel, exciting works in the area related to Serverless infrastructure and runtime optimizations, we will continue to describe some other interesting works that extended the Serverless platform by adding new primitives and features.

As we mentioned earlier, in the introduction and overview sections, people have built applications atop the Serverless platforms and for specific purposes like ExCamera [24] and Sprocket [17] (Video Processing), PyWren [26] and Locus [30] (Data Analytics), gg [23] (Parallel Compilation), Cirrus [21] (Machine Learning), and NumPyWren [33] (Numerical Analysis).

Kappa [36] has focused on providing a framework for more general-purpose applications using Serverless infrastructure and make it more than just for event handling generally. Here are the problems that they focused on in their general-purpose framework:

- Lambda functions are short-lived jobs; however, the computation can be long-running in general.
- existing FaaS offerings lack concurrency primitives and patterns like fork-join and message passing.

To address the first challenge, Kappa, provides a checkpointing mechanism at the language level based on continuation, which helps to resume the lambda functions from where it timed out. As for the second challenge, they provided a concurrency API in which they provided concurrency primitives familiar to developers like ”future” and ”message passing” to handle synchronization between lambda functions. They have implemented fault tolerance mechanisms to ensure they do not have non-determinism and side effects while checkpointing. Kappa needs no modification to the Serverless platforms and can be running on current platforms.

Kappa consists of 3 main components, Compiler, Coordinator, and Library. The compiler is responsible for program transformation and generates code for checkpointing. The coordinator is responsible for launching and resuming Kappa tasks and has the underlying features for implementing the concurrency primitives, and Kappa tasks use Library for checkpointing and synchronization. In the beginning, the application code will be transformed by the compiler to a Kappa-friendly code (which includes the code for checkpointing), and then it will be enpackaged with the Kappa library at runtime and passed to the coordinator. The coordinator will launch the Kappa tasks that are logically considered long-lived threads. Each Kappa task will communicate with the coordinator through RPC for checkpointing and synchronization purposes. They also used AWS S3 to save and retrieve the checkpoint contents and continuation functions. Kappa has other features like coordinator state persistence, checkpoint replication, and call external services like storage services.

As for the evaluation, they provided Kappa's overhead and ran several applications to show the generality of the framework. Checkpointing overhead is not too much (less than 100ms upto 1000KB of checkpoint size) even if we use S3. They showed that their checkpointing mechanism is also scalable by testing it over up to 1000 concurrent lambdas doing checkpointing every 100ms with the size of 0.5 KB. They ran TPC-DS [14], word count in Map-Reduce style, Parallel grep, which counts the occurrences of a string in a single file split into chunks, and streaming application to calculate the average number of hashtags per tweet in a stream of tweets, and a web crawler based on UbiCrawler [19] which downloads web pages starting from seed domains [15].

Crucial [18] also addressed two challenges in current Serverless platforms: 1- The problem of shipping data to code and high bandwidth usage for the fine-grained data access (moving data back and forth) 2- It is hard to synchronize concurrent cloud functions since it lacks concurrency prim-

itives. They proposed having a mutable shared state across cloud functions and synchronization primitives so that people can quickly run cloud functions using their system.

They provided a library in Java with the abstraction of Cloud Threads, and with that, programmers can run their Java thread as a Serverless cloud function. They have implemented a Distributed Shared Object built atop infinispan [28], which plays a role as an in-memory data store to share mutable states across cloud functions. Since they are using object methods and can handle some operations at the storage level, they mitigated high bandwidth usage for shipping data to code for fine-grained accesses. Furthermore, they have provided synchronization primitives like CyclicBarriers and Semaphores to handle concurrency issues in their code. According to their evaluation, they handle fine-grained access, provide low latency (as they are using in-memory space), propose concurrency primitives, support operations on data, have a simplified programming model, and provide fault tolerance using object replication.

They claimed that less than 3% changes needed to native Java thread's code to make it suitable for Serverless platform using Crucial. They showed, Crucial atop Serverless will outperform the Spark cluster running Logistic Regression, and Crucial has a negligible overhead compared to native Java threads atop their Distributed Shared Object store running locally in the cloud.

V. Discussion & Future Directions

There is a number of interesting research directions that we can take the work. Here we discuss two of them:

Disaggregated Serverless Platform - According to [25] Serverless platforms without any of the co-location/placement mechanisms mentioned in this paper deploys Serverless function instances on the nodes non-deterministically, which leads to inefficiency. As mentioned in works like Photons [22], and SAND [16], we can leverage function co-location to share common resources across the functions of an application and also add a more efficient communication channel locally so that an application's functions do not have to communicate through a globally distributed message bus. We are sharing runtime, libraries, etc across concurrent function instances of an application which leads to better resource utilization and reducing cold starts because we have the shareable portion of the function in the target node locally. Now that this function placement and co-location has shown its benefits in the area of Serverless computing, we can take this one step further.

Since all function instances that are potential candidates for co-location might not fit into a node as our unit of underlying hardware resources is the actual server node, we can leverage disaggregated computing to extend hardware resources beyond a single server. Combining these two directions can create some trade-offs. For example, suppose we implement a memory disaggregation mechanism in order to provide the functions with more memory using remote node's memory to share the shareable portions. In that case, we will certainly add overhead to the function execution since we make functions do paging remotely in part. However, on the other hand, we have to initialize all those shareable portions (cold start) if we want to start a function from scratch on some other node. This trade-off can be analyzed in detail and lead to efficient scheduling decisions in the management layer of the disaggregated Serverless platform.

Serverless & Specialized Hardware - Function instances are currently time-limited, and they are inherently short-lived jobs. As authors said in Kappa [36], they are not suitable for long-running jobs, and there may be some faults happening in this situation unless we use mechanisms like checkpointing in Kappa framework, which needs minor code modifications. Besides, in [25], the authors argued that current Serverless platforms do not support specialized hardware.

To persist intermediate state/data for a single function or across functions, developers are using the distributed storage layer like S3 or in-memory cache like Redis (which is a bit more expensive) as they do in Kappa checkpointing.

One interesting idea could be leveraging NVM as the persistence layer and provide fault tolerance and fast resumption/re-execution of serverless functions locally on the node (no need to reach S3) to run long-running jobs and provide Serverless platform's users with specialized hardware support. It should be noted that usually, local NVM is cheaper than memory, while it typically comes with more space for persisting data. Besides, local NVM is faster than remote storage for fine-grained accesses in long-running jobs.

VI. Conclusion

In this paper, we explained the promise of Serverless computing. We presented a literature review on the optimizations that people applied to state-of-the-art to address and improve the existing drawbacks of Serverless infrastructure. We also summarized the extensions that people have added to build either a Serverless platform, especially for a purpose, or make the Serverless platform suitable for general purpose applications. In the end, we summarized a few limitations and proposed several future directions to make this promising Serverless Computing idea even more promising.

Acknowledgment

This research was supported in part by VMware and the NSF as part of SDI-CSCS award number 1700527, and by the NSF as part of CAREER award number 1652698.

References

[1] https://aws.amazon.com/ec2/pricing.
[2] https://aws.amazon.com/lambda.
[3] https://aws.amazon.com/s3/.
[4] https://spark.apache.org/docs/latest/api/python/getting_started/index.html.
[5] https://hadoop.apache.org/.
[6] https://azure.microsoft.com/en-us/services/functions/.
[7] https://github.com/Azure/AzurePublicDataset.
[8] https://openwhisk.apache.org/.
[9] https://github.com/open-lambda/open-lambda.
[10] https://www.docker.com/.
[11] https://www.djangoproject.com/.
[12] http://www.image-net.org/.

[13] https://aws.amazon.com/elasticache/.

[14] https://github.com/ooq/tpcds-pywren-scripts.

[15] http://s3.amazonaws.com/alexa-static/top-1m.csv.zip.

[16] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association.

[17] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[19] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: A scalable fully distributed web crawler. *Softw. Pract. Exper.*, 34(8):711–726, July 2004.

[20] G. E. P. Box and David A. Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 65(332):1509–1526, 1970.

[21] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery.

[23] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[24] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. 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)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[25] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[26] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99 In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[27] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[28] Francesco Marchioni and Manik Surtani. *Infinispan data grid platform*. Packt Publishing Ltd, 2012.

[29] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association.

[30] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, February 2019. USENIX Association.

[31] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, page 1–8, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[33] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 281–295, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Arjun Singhvi, Junaid Khalid, Aditya Akella, and Sujata Banerjee. Snf: Serverless network functions. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 296–310, New York, NY, USA, 2020. Association for Computing Machinery.

[35] Shelby Thomas, Lixiang Ao, Geoffrey M. Voelker, and George Porter. Particle: Ephemeral endpoints for serverless networking. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 16–29, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 328–343, New York, NY, USA, 2020. Association for Computing Machinery.