A Survey on Serverless Computing and its Implications for JointCloud Computing

Mingyu Wu, Zeyu Mi, and Yubin Xia
Institude of Parallel and Distributed Systems (IPADS)
Shanghai Jiao Tong University
Shanghai, China

***Abstract*—Serverless computing is known as an appealing alternative cloud computing paradigm with its auto-scaling nature and pay-as-you-go charging model. Mainstream cloud vendors have proposed their own serverless platforms, while various kinds of applications have been refactored in a serverless manner for execution. However, the serverless computing model still entails refinement as it introduces performance and security issues. In this paper, we conduct a comprehensive survey on the serverless computing, mainly in three aspects: the type of applications suitable for serverless, the performance issues, and the security issues. We specifically elaborate previous efforts on resolving issues in serverless and shed light on the unresolved issues. We also discuss the opportunities and challenges in integrating serverless computing in the jointcloud infrastructure.**

***Index Terms*—serverless computing, jointcloud computing, survey**

I. Introduction

Serverless is an emerging computing paradigm in the cloud environment. Serverless computing requires breaking the workload into short-running and stateless *functions* to achieve ease-of-maintenance and auto-scaling. Its pay-as-you-go model only charges when functions are actually running, which is also welcomed by cloud customers. To this end, many cloud vendors have released their own serverless platforms, such as Amazon Lambda [11], Google Cloud Functions [25], IBM Cloud Functions [29], and Microsoft Azure Functions [45]. With those serverless platforms, customers only need to upload their functions for scalable and distributed execution. There are also open-sourced implementations available for serverless computing like OpenLambda [28] and OpenWhisk [9].

Despite the virtues, serverless computing also has limitations and problems. First, the short-running nature of serverless functions limits the scope of application. Second, breaking applications into functions introduces performance overhead. Third, in terms of security properties, traditional execution abstractions like VM and container are not the optimal choice to hold these small functions from different customers. Prior work has discussed those problems and provide various solutions, but many still remain unresolved and open. It is also unclear how serverless will evolve when encountering the new trends in cloud, such as jointcloud computing.

This work conducts a comprehensive survey on prior work for serverless computing to shed light upon the challenges, unresolved problems, and opportunities. The survey is able to answer four questions below.

What applications are suitable for serverless? We summarize the applications currently deployed in serverless platforms and categorize them into three: massively-parallel, passive (event-driven), and interconnected.

What performance problems does serverless computing introduce? The rise of serverless computing also introduces performance issues. The most severe ones are the *slow start-up* problem and the *inter-function communication* problem. We will discuss both of them and how prior trials on resolving them.

What security issues does serverless computing raise? Serverless platforms should secure the functions from cloud users from known attacks in the cloud. Therefore, we mainly discuss the design choices of mainstream cloud vendors (Google and AWS) on building secure abstractions for serverless functions.

What happens when jointcloud computing meets serverless? As a new cloud computing model, jointcloud computing allows cloud users to run their workload atop various clouds simultaneously, with user-transparent migration. Serverless seems to be an ideal fit for jointcloud computing as its lightweight computing unit (i.e., function) is very convenient for migration. However, we also notice that the storage dependencies and the opacity of inner-cloud infrastructure make it challenging to combine them together.

The rest of the paper is as follows. Section II informally defines serverless and concludes its basic ideas. Section III describes which kind of applications are suitable to be deployed on the serverless platforms. Section IV summarizes the performance issues while Section V discusses security issues. Section VI elaborates the opportunities and challenges of the combination of serverless and the burgeoning jointcloud computing, which is not covered in prior surveys on serverless [63], [27], [33], [54], [14], [1]. Section VII finally concludes the paper.

II. Definition of Serverless Computing

The Serverless model is also known as function-as-a-service (FaaS), where customers are free to develop, run, and manage the functionalities of their applications without the complexity of building and maintaining the infrastructure typically when

978-1-7281-6981-1/20/$31.00 ©2020 IEEE
DOI 10.1109/JCC49151.2020.00023

developing and launching an app [3]. According to [54], a serverless service should meet the following requirements:

Function-level management. The basic unit in serverless is *function*. All functions are independent from each other, even though they are actually derived from the same application. The unit of deployment, scheduling, and isolation in the serverless platforms are all functions.

Short-running. Since functions are usually small units compared to the entire application, they are expected to complete in a short time period. Commercial serverless platforms like AWS lambda also impose the maximum execution time for each function. However, as the serverless programming model becomes more popular, workload atop serverless now varies, and functions from big-data analytics are not necessarily finished very soon. Therefore, the restriction on the maximum execution time might be loosened in the future[1].

Transparency. Users of serverless are agnostic about the execution environment. All they need is to upload their functions (usually wrapped in a container image) for processing. Transparency is vital for serverless as it liberates users from the burden of code deployment and management.

Stateless. Functions are stateless and only describe the application logic for task processing (or how the states are transformed). All required states are imported from external storage. The stateless nature of functions makes serverless more robust to crashes, as recovering states within crashed functions are not required.

Pay-as-you-go. The billing model of serverless is pay-as-you-go, i.e., the cloud provider charges only when the uploaded functions are actually executed. In contrast, cloud vendors now mainly charge users by the resource number rented for running instances (virtual machines), no matter if they are active, under-utilized, or idle. Therefore, serverless is quite attractive for cloud users as it suggests less cost.

The virtues of serverless computing are appealing for both cloud vendors and customers, so mainstream cloud vendors have developed their own serverless frameworks, such as AWS Lambda, Google Cloud Function, Microsoft Azure Functions, IBM Cloud Functions, and Alibaba Cloud Function Compute.

III. Application

Current serverless platforms have supported more and more functionalities [15], [31], [50], [5]. Hence, many applications can be migrated to a serverless platform. But due to the current limitations of the serverless platform (e.g., large startup latency and slow communication performance between functions), only a small number of applications have been successfully ported to a serverless version. The currently ported serverless applications can be classified into three categories:

Massive and independent parallelism: The first typical use case of serverless computing is to build an application that consists of thousand-way mostly independent functions. In this use case, each function contains all the necessary code and data, which means it does not need to communicate with other functions. Therefore, such application can be easily boosted by creating thousands of workers to fully enjoy the automatic scalability support of the platform. Most scientific computing or big data processing tasks belong to this type. PyWren [32] uses AWS Lambda functions for linear algebra and machine learning hyperparameter optimization. Werner et al. propose to use AWS Lambda to implement distributed matrix multiplication [65]. Serverless version Mapreduce [6] and Spark [24] are two other examples of this approach.

[1]Actually, the per-function maximum execution time has been increased from 5 minutes to 15 in Lambda to support various kinds of applications [10].

Event-driven handlers: This type of applications has passive handlers which wait for a specific kind of events. Once an event is triggered, the handler wakes up to process corresponding business logic. For example, a user uploads a picture to a website. Then the picture uploading event triggers a serverless function to generate a thumb picture based on this uploaded one. Web and mobile app API serving are two of the most common use cases of serverless computing. It is common that API servers do not need long-running servers keeping running and waiting for user requests. If the number of API requests is small, most server computing time is wasted on looping or sleeping for waiting. By using serverless, these API services do not need to be held in a long-running server which causes unnecessary financial and management costs.

General task-based applications: It is natural to adapt applications in the previous two categories into serverless architecture. However, the rapid development of serverless has encouraged researchers to explore more complicated applications. Those applications include software compilation, unit tests, and video encoding. Although some of them have been supported by some serverless platforms [8], [23], those platforms are application-specific and thus not general enough to support various task-based applications. To this end, gg [22] has proposed an intermediate representation (IR) to connect general task-based applications with serverless platforms. gg IR requires developers to divide the applications into *thunks* so that each of them will be translated into functions and executed in different containers. The experiment results show that the execution time on serverless platforms with gg IR is comparable with traditional multi-core execution, which suggests the great potential of the serverless architecture for efficiently supporting various kinds of applications.

For the third type of applications, although they can be divided into fine-grained modules, the modules may intensively cooperate and communicate with each other during runtime. The relationships among the connected workers are either static or dynamic. A static relationship means the connections among function workers are determined before installation and will not be changed during runtime. For a dynamic relationship, the connections between workers dynamically change according to the runtime behaviors of functions when handling user requests.

According to the relationship among functions, prior work [21], [35] has classified the applications into three forms, as shown in Figure 1.

The first form is a chained pipeline where functions are ex-

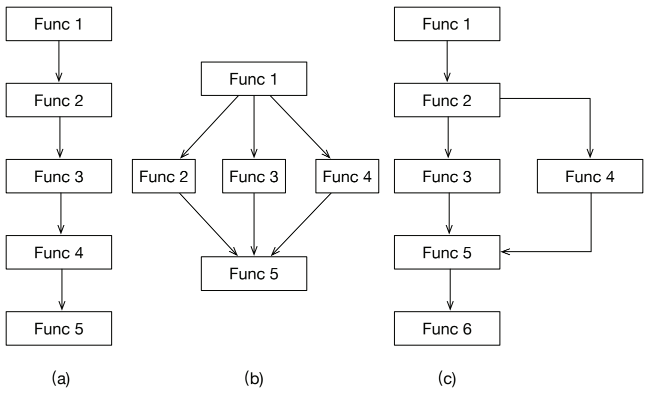

Figure 1: The connection forms of a serverless application. Figure (a) shows a pipeline form, whose steps are connected in sequential order. In this form, a function should wait until all previous functions have finished. Figure (b) is a parallel form, where some functions are running in parallel. Figure (c) is a branch form, where some functions choose one of the next functions according to runtime behavior.

ecuted sequentially. For example, a facial recognition pipeline consists of five sequential stages (decode, scene change, facial recognition, draw, and encode). The second form represents a parallel workload where many workers receive inputs from the upstream worker and output their data to a final merging worker. Those two forms are common in the serverless scenario and have been intensively studied. Since the execution topology is static, the serverless platform is able to conduct optimizations during deployment. For example, SAND [4] accelerates the communication between chained functions by co-locating them into the same container.

The third form has a condition checking worker to choose which function to invoke based on the current condition value. Since the execution topology is dynamic, fewer optimizations can be adopted and most prior work has not covered this form of applications.

IV. Performance Issues

Serverless has become a new computing paradigm where an application is split into multiple short-running, on-demand functions for separated processing. Serverless breaks applications into smaller units for better portability and maintainability, but it also introduces performance issues. First, to launch a function, serverless frameworks need to prepare the runtime environment, whose time cost is sometimes more than the execution time of the function itself. For applications in high-level languages, the warmup time for language runtime further exacerbates the problem. Second, since functions are now separated from each other, normal function calls now become inter-process communications and remote network invocations, which induces severe communication overhead. Prior work also uncovers other performance issues in serverless computing, such as scheduling [34], [56] and performance predictability [27], [47]. However, we will focus on the two problems above as plenty of work has been proposed to solve them.

A. Fast boot

To achieve fault isolation and portability, mainstream serverless frameworks mainly deploy functions into containers. However, launching a container is quite time-consuming: it takes more than 100ms to initialize a container from the very beginning. In contrast, the execution time for a function usually only costs some milliseconds [20]. Therefore, the boot overhead of containers becomes a major bottleneck in the serverless scenario.

Container cache. A straightforward solution to the boot problem is *container cache*. When a function is finished, the serverless framework can retain its runtime environment so that the same function from the same user can reuse it. The mainstream commercial serverless platforms have adopted this mechanism so that it is less likely to boot a new container for new-coming functions [63].

Pre-warming. Container cache is mainly designed for reusing containers when a function is repeatedly invoked. However, for the same function from different users, the container cannot be reused due to security issues, and users may still suffer from the slow boot. To this end, another work introduces *pre-warmed containers*, which prepare a runtime environment for functions which share similar characteristics. For example, OpenWhisk [59] can pre-launch Node.js containers if it has observed that the workload mainly consists of Node.js-based functions. Those containers are generally spawned without information from users, so users can directly exploit them without worrying about their privacy. SOCK [49] adopts a similar strategy but for python-based functions. It identifies popular python packages and pre-installs them on their pre-warmed containers to cut down the boot time of the python runtime.

Since pre-warmed containers are initialized with the same workflow, their in-memory states are quite similar. Therefore, prior work [62], [20] proposes a fork-based scheme to reduce the memory footprint of pre-warmed containers. When a new container is required, it will be forked from a pre-warmed one. The memory pages of containers are also shared and only become private when write operations occur. This solution works well especially for functions running atop language runtime, as the initialization of the runtime is nearly the same among different runs. GraalVM Native Image [66] proposes an alternative solution to launch new functions from a prepared JVM image to save startup time. It also adopts the sharing and copy-on-write mechanism to reduce the memory footprint. The major difference is that GraalVM Native Image leverages more programming-language techniques, such as ahead-of-time (AOT) compilation and point-to analysis.

Container optimization. Since the cost of launching a container is prohibitive, a line of work tries to optimize the launch time considering the characteristics of serverless functions. SOCK [48], [49] breaks down the launching process of containers and locates bottlenecks in file systems, namespaces

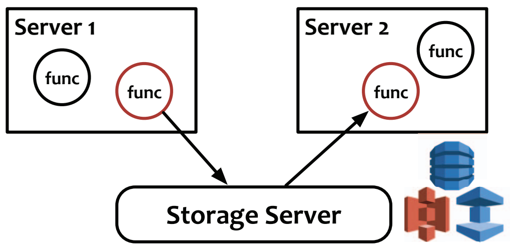

Figure 2: The communication model in serverless platforms

and cgroups. It further provides *lean containers* with much faster boot time than vanilla ones. Mohan et al. [46] focus on the scalability problem of network creation for a vast number of containers and exploits network pre-creation to solve the problem.

Looking for other abstractions. Due to the slow launch time in containers, serverless frameworks also search for alternative abstractions to isolate functions. Virtual machines (VM) can provide stronger isolation compared to containers, but they usually have longer startup time. LightVM [43] suggests that VMs can achieve fast boot time (much faster than vanilla containers) by trimming down unnecessary components, which makes VM a competitor for serverless scenarios. SAND [4] indicates that containers provide too strong isolation for functions uploaded by the same user. It allows functions from the same user to co-reside in the same container and leverages different processes to isolate them from each other. Besides, there are many alternative abstractions for serverless, such as Google gVisor [26], AWS FireCracker [16], Unikernels [41], [42], light-weight contexts [39], etc. We anticipate more abstractions as the workload in serverless becomes more various and critical.

B. Communications

Since serverless frameworks have broken applications into separated functions, normal function calls are now turned into remote invocations through network, which introduce considerable overhead. The communications between functions usually happen at the end of the preceding one. As Figure 2 shows, the serverless platforms are oblivious about the relationship among functions, so functions may reside in different servers. Before the preceding function exits, it will store its output data into the storage server (e.g., S3 in AWS) so that the subsequent one can retrieve it as the input. As the two functions and the storage server are in three different machines, the function call is significantly slowed down as it involves at least two network round-trips. To optimize the communications between functions, prior work (1) optimizes the performance of the storage server or (2) optimizes the whole communication path.

Optimizing the storage server. Mainstream serverless frameworks directly reuse the cloud storage services to store intermediate data among functions, such as S3 and DynamoDB [57]. Unfortunately, those storage services are not designed for short-running functions and thus become a performance bottleneck. The problem is exacerbated in *serverless analytics*, which entails transmitting large intermediate data among functions, such as MapReduce [6]. Klimovic et al. [37] study the performance of S3 on various serverless analytic applications and compare it against a DRAM-based key-value store (ElastiCache [7]). They conclude that the storage service should be highly elastic to meet the high IOPS and throughput requirement of serverless analytics. Their follow-up work, Pocket [38], achieves this goal by introducing multi-tier storage including DRAM, SSD and HDD. Pocket also proposes a controller to rightsize the resource allocation according to the characteristics of serverless functions and current resource utilization. Locus [52] also combines different kinds of storage devices to achieve both performance and cost-efficiency for serverless analytics. Crail [58] is also built for storing and exchanging large-scale temporary data, but it targets for high bandwidth and low latency. Carreira et al. [17] argue that storage systems above are not suitable for serverless machine learning workflows and thus propose their own serverless storage [18] to support sparse data structure and rich ML-related APIs. Shredder [67] further proposes *storage functions* to move the serverless computation onto the storage servers to reduce the data exchanging overhead. There is also growing interest to combine serverless frameworks with the ”serverless” architecture, namely disaggregated hardware [55], [51].

Optimizing the communication path. Serverless frameworks are usually oblivious or unconcerned about communication among functions. Therefore, even though two functions happen to be scheduled onto the same server, prior frameworks like OpenWhisk still leverage the storage server for their data exchanging [4]. To this end, prior efforts have been made to optimize the communication path when the relationship between functions is known in advance. Fortunately, mainstream platforms have allowed applications to forge their functions together as a chain of execution (such as IBM Action Sequences [30] and AWS Step Functions [12]). SAND [4] will specially handle those chains and co-locate them on the same machine. Besides, it also proposes hierarchical message queuing, where local messages inside a function chain in the same machine will be directly delivered without storage servers. Its experiments on an image-processing pipeline [13] show a great reduction in the communication overhead. Vodrahalli et al. [60] also co-locate related functions and propose a software-defined cache to avoid accessing the storage server. The co-location mechanism may also be helpful for serverless analytics, where applications can be represented as direct acyclic graphs (DAGs). Furthermore, communication among functions ao-located at the same machine can also be optimized by using traditional IPC techniques in microkernels [44], [36].

Another line of work tries to kick the storage server out of the communication path with network mechanisms. ExCamera [23] leverages a rendezvous server to facilitate inter-function communications. When a function requires sending a message to others, it will send it to the rendezvous server, which forwards the message to the destination. However, the rendezvous server will become a bottleneck when the number of active functions grows larger. ExCamera mentions that this problem can be resolved by introducing "a hole-punching NAT-traversal strategy" [23], but it does not implement this strategy in the paper. gg [22] also exploits this strategy to exchange small objects among functions, but it does not go into detail in the paper. Although this solution seems appealing to mitigate communication overhead, it complicates the design by introducing many channels among function executors. The problem still remains open as none of prior systems provide a full-fledged solution to support direct and rapid communication among functions.

V. Security Issues: Abstraction for Function

In serverless computing, the most common abstraction to hold functions is Linux container [63], [27], [33], [54], [14], [1]. Linux Container is based on namespace and control group (cgroup) mechanisms. The namespace mechanism provides a security isolation scheme, where each namespace can specify its own system resources, including process, IPC, network and file. The resources outside a namespace are inaccessible to it. The cgroup mechanism limits the number of resources used by each process such as CPU, memory, network and disk. It can prevent a process from comsuming too much system resources and affecting the normal use of resources by other processes.

A Linux container instance is a set of processes running on an operating system shared with other containers. The security of a container depends on the shared operating system. However, the LOC of the operating system is rapidly expanding. Specifically, the LOC of Linux kernel has risen from 150,000 of version 1.0 to more than 20 million of version 4.15. Hence, it is inevitable that the operating system contains a large number of hidden vulnerabilities. A failed or breached container instance can easily trigger a vulnerability in the operating system, then it may have access to or tamper with other container instances above the operating system. For example, the published shocker[2] attack exploits a bug in system calls related to file handles, allowing container instances to access arbitrary files on the operating system, including files from other container instances.

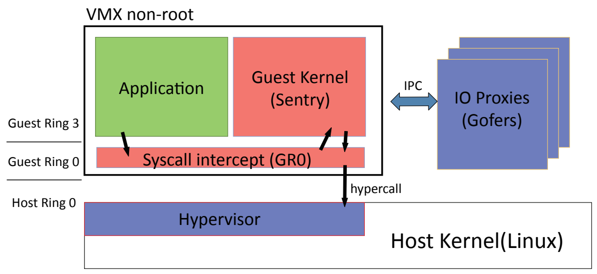

Figure 3: The architecture of Google gVisor

To enhance the security of containers, Google introduced gVisor[26] for securing serverless functions. Figure 3 shows the overall architecture of gVisor. The serverless function is running in non-root mode Ring 3. gVisor provides a kernel called Sentry, which intercepts and processes all system calls from the function. The Sentry is written in the Go language and can completely isolate the function from the operating system. Sentry carefully implements a large number of system calls including memory management and thread scheduling, so that most of the function's system calls can be processed in Sentry. Only a small part of the system calls need to rely on the kernel. For example, when performing I/O operations, gVisor will forward the request to a module called Gofer in root mode Ring 3, which further requests the host Linux services.

The weak isolation of Linux containers is caused by the shared and vulnerable kernel. The design of gVisor avoids by providing a separate Sentry for each serverless function. Furthermore, since Go language is type-safe and guarantees memory safety, Sentry has less vulnerability than Linux. Last but not least, gVisor leverages hardware virtualization as the last line of defense. Therefore, the design of gVisor achieves at least the same isolation level as a virtual machine. At the same time, its startup performance is much faster than a VM because the specialized kernel Sentry is highly optimized and small. However, since the Gofer architecture requires multiple context switches, a serverless function that frequently uses I/O does not perform well on gVisor.

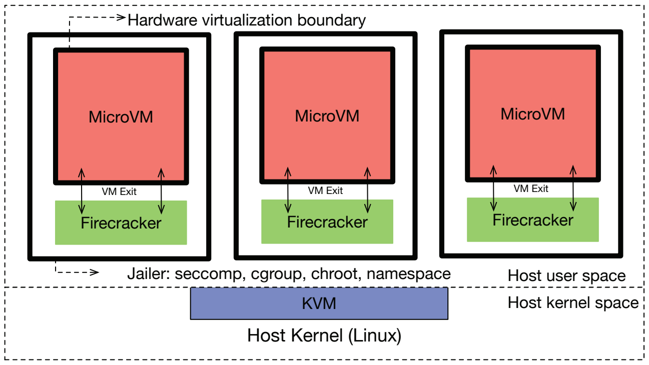

Figure 4: The architecture of AWS FireCracker

Another approach to protecting a container is to put it within a virtual machine. However, a serverless function requires a small number of resources, which means the VM is a heavier abstraction for this scenario. A normal VM is equipped with full-fledged OS kernels like Linux, which may severely affect the startup and runtime performance. To this end, Amazon presents a new hypervisor called FireCracker [16] written in Rust language, which uses a microVM to hold a serverless function. Figure 4 shows the architecture of FireCracker. FireCracker uses two levels of isolation boundary to harden the security of a microVM. The first isolation boundary implemented is by the hardware virtualization. The second one is a jailer box based on traditional isolation techniques in Linux, such as seccomp and namespace. The OS kernel used a microVM is a tailored Linux that removes all unnecessary functionalities like unused drivers. Therefore, the startup and

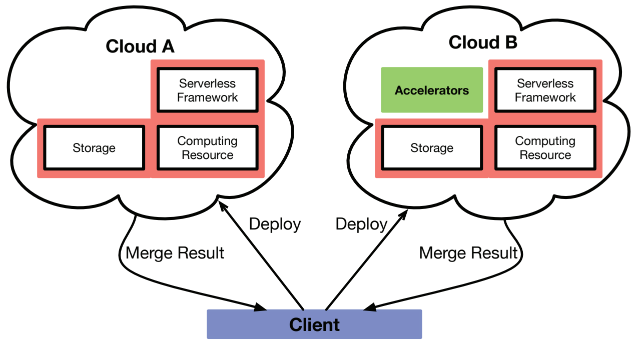

Figure 5: An example to show the combination of serverless computing and jointcloud computing.

memory usage of a microVM is comparable to a gVisor container.

Even though the container and VM abstraction is the main-stream practice used in today's commercial serverless platform, other abstractions are also proposed to reduce the resource consumption and improve the security of serverless functions. For example, Shredder [67] relies on JavaScript V8 instance called "Isolate" to provide an isolated execution environment for a function, which can achieve inter-tenant isolation. The unit of isolation is V8 "context" that groups both code and data. They only contain necessary code and data, making it possible for a process to hold thousands of contexts and seamlessly switch among them. Furthermore, the cost of context switches is much smaller than that of a normal process. However, Shredder now can only support hash-table-based storage, and it is still unclear whether the context abstraction can be implemented in other languages and deployed in more practical scenarios.

VI. Opportunity and Challenges for JointCloud Computing

The jointcloud computing is a new model that enjoys the benefits of multiple clouds at the same time. By leveraging the jointcloud model, a cloud user could be served by collaborative clouds at low cost, high availability and guaranteed QoS [61].

However, the traditional VM abstraction used in today's clouds may not be the ideal way to implement the jointcloud computing. The most important reason is that migrating a heavy VM across clouds would involve the transferring of a large number of useless states, most of which are not used by the application's core logic. Therefore, the VM abstraction hinders the wide adoption of the jointcloud computing.

A. Opportunities

The serverless computing presents opportunities to implement the jointcloud computing model efficiently. First, the serverless model forces a cloud user to organize an application in the unit of functions by using high-level programming languages, which means that the migration unit is not the VM any more but the much smaller serverless function. A smaller function can be easily migrated among clouds. If the overhead to migrating a function is still considerable, the stateless property of serverless functions allows us to restart a new instance on a new cloud and abort the old one. Second, today's serverless platform supports short-running functions and most functions have to finish within minutes. By using serverless computing to implement the jointcloud model, a user can deploy functions at different clouds to enjoy the lowest prices and best performance simultaneously. Suppose there is one cloud that provides cheap serverless service without fast accelerators, while the other cloud owns accelerators but has much more expensive serverless service. Under such circumstance, we can deploy most computing functions at the first cloud and upload accelerator-related functions to the second cloud, as shown in Figure 5.

B. Challenges

However, it is still not straightforward to combine the serverless model and the jointcloud computing. First, the definitions and APIs used in today's serverless clouds are different from each other. For example, different clouds provide various storage services: Amazon has an object storage service called S3 while other clouds may not provide similar services. Even if other clouds have similar ones, these services have distinct APIs, which hinders the switch or migration of a function among clouds. Second, it is challenging to decide which clouds or platforms are most suitable for a user. The choice of clouds depends on a variety of factors, such as the geographical distance between cloud and users, dynamic pricing and service qualities of each cloud. The decision should also take communications among clouds into consideration. An unwise choice could result in an increase in communications among functions located at different clouds. Although distributed data storages [19], [53], [64] can be exploited to implement efficient inter-cloud data exchange, the communication overhead is still considerable. Third, it is still difficult to handle the dynamic variance of clouds service quality. Migrating an existing function instance across clouds may not a better choice than restarting a new one at the target cloud. Although the restarting strategy fits well with the serverless computing, which is usually stateless, restarting may still involve persistent state migration among databases. Therefore, a performance model is demanded to estimate the overhead of migration and restarting so as to make the most cost-effective choice.

HCloud [40] is a preliminary serverless-based system which tries to address some challenges. HCloud builds unified APIs among different cloud vendors to resolve the API problem. It also provides per-cloud managers to monitor resource utilization and price to achieve cost-efficiency. Its scheduler relies on users to explicitly define the inter-function relationship to reduce the inter-cloud communications. However, it does not cover the overhead of migration and provides limited language support (python only).

VII. Conclusions

The emerging serverless computing model is striking the traditional cloud infrastructure by splitting applications into short-running, dynamically scalable, and stateless functions. However, the rise of serverless also introduces new problems in both performance and security areas. This paper conducts a thorough survey on serverless computing in three aspects: applications suitable for serverless computing, performance problems, and security issues. It also describes the challenges and opportunities when combining serverless with the joint-cloud computing model. Our future work will focus on the unresolved problems in serverless computing and extend the serverless model to the jointcloud infrastructure.

VIII. Acknowledgments

We sincerely thank the anonymous reviewers for their insightful comments. This work was supported by the National Key Research & Development Program (No.2016YFB1000104). Yubin Xia (xiayubin@sjtu.edu.cn) is the corresponding author.

References

[1] 2018 serverless community survey: Huge growth in serverless usage. https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/.

[2] shocker: docker poc vmm-container breakout. http://stealth.openwall.net/xSports/shocker.c.

[3] Serverless architecture. https://martinfowler.com/articles/serverless.html#unpacking-faas, 2018.

[4] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 923–935, Berkeley, CA, USA, 2018. USENIX Association.

[5] K. Alpernas, C. Flanagan, S. Fouladi, L. Ryzhyk, M. Sagiv, T. Schmitz, and K. Winstein. Secure serverless computing using dynamic information flow control. *Proc. ACM Program. Lang.*, 2(OOPSLA):118:1–118:26, Oct. 2018.

[6] Amazon. Ad hoc big data processing made simple with serverless mapreduce. https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simplewith-serverless-mapreduce/, 2016.

[7] Amazon. Amazon elasticache. https://aws.amazon.com/elasticache/, 2019.

[8] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 263–274, New York, NY, USA, 2018. ACM.

[9] Apache OpenWhisk. Apache openwhisk - open source serverless cloud platform. https://openwhisk.apache.org/, 2019.

[10] AWS. Aws lambda enables functions that can run up to 15 minutes. https://aws.amazon.com/about-aws/whats-new/2018/10/aws-lambda-supports-functions-that-can-run-up-to-15-minutes/, 2018.

[11] AWS. Aws lambda. https://aws.amazon.com/lambda/, 2019.

[12] AWS. Aws step functions. https://aws.amazon.com/cn/step-functions/, 2019.

[13] AWS. Github - aws-samples/lambda-refarch-imagerecognition. https://github.com/aws-samples/lambda-refarch-imagerecognition, 2019.

[14] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, et al. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer, 2017.

[15] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, and O. Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, pages 89–103, New York, NY, USA, 2017. ACM.

[16] J. Barr. Firecracker lightweight virtualization for serverless computing., 2018.

[17] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, volume 2018, 2018.

[18] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz. Cirrus: a serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24. ACM, 2019.

[19] P. F. Corbett and D. G. Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems (TOCS)*, 14(3):225–264, 1996.

[20] D. Du, T. Yu, Y. Xia, G. Yan, C. Qin, Q. Wu, and H. Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2020.

[21] T. Elgamal. Costless: Optimizing cost of serverless computing through function fusion and placement. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 300–312. IEEE, 2018.

[22] S. Fouladi, F. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, pages 475–488, Berkeley, CA, USA, 2019. USENIX Association.

[23] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, pages 363–376, Berkeley, CA, USA, 2017. USENIX Association.

[24] Github. Apache spark on aws lambda. https://github.com/qubole/spark-on-lambda/.

[25] Google. Cloud functions - google cloud. https://cloud.google.com/functions/, 2019.

[26] Google. gvisor: A container sandbox runtime focused on security, efficiency, and ease of use., 2019.

[27] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[28] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA, 2016. USENIX Association.

[29] IBM. Ibm cloud functions. https://www.ibm.com/cloud/functions, 2019.

[30] IBM Cloud. Ibm cloud functions - concepts. https://cloud.ibm.com/functions/learn/concepts, 2019.

[31] A. Jangda, D. Pinckney, Y. Brun, and A. Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA):149:1–149:26, Oct. 2019.

[32] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 445–451, New York, NY, USA, 2017. ACM.

[33] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Menezes Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019.

[34] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, pages 158–164, New York, NY, USA, 2019. ACM.

[35] R. S. Kannan, L. Subramanian, A. Raju, J. Ahn, J. Mars, and L. Tang. Grandslam: Guaranteeing slas for jobs in microservices execution frameworks. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, and et al. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP 09, page 207220, New York, NY, USA, 2009. Association for Computing Machinery.

[37] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. Understanding ephemeral storage for serverless analytics. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 789–794, Berkeley, CA, USA, 2018. USENIX Association.

[38] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 427–444, Berkeley, CA, USA, 2018. USENIX Association.

[39] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight contexts: An {OS} abstraction for safety and performance. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 49–64, 2016.

[40] J. Liu, Z. Mi, Z. Huang, Z. Hua, and Y. Xia. Hcloud: A serverless platform for jointcloud computing. In *IEEE International conference on jointcloud computing*, 2020.

[41] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. *Acm Sigplan Notices*, 48(4):461–472, 2013.

[42] A. Madhavapeddy, D. J. Scott, J. Lango, M. Cavage, P. Helland, and D. Owens. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, 57(1):61–69, 2014.

[43] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233. ACM, 2017.

[44] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen. Skybridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys 19, New York, NY, USA, 2019. Association for Computing Machinery.

[45] Microsoft. Microsoft azure functions. https://azure.microsoft.com/services/functions/, 2019.

[46] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[47] H. D. Nguyen, C. Zhang, Z. Xiao, and A. A. Chien. Real-time serverless: Enabling application performance guarantees. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, pages 1–6, New York, NY, USA, 2019. ACM.

[48] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400. IEEE, 2017.

[49] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 57–69, Berkeley, CA, USA, 2018. USENIX Association.

[50] M. Obetz, S. Patterson, and A. Milanova. Static call graph construction in aws lambda serverless applications. In *Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'19, pages 20–20, Berkeley, CA, USA, 2019. USENIX Association.

[51] N. Pemberton and J. Schleier-Smith. The serverless data center: Hardware disaggregation meets serverless computing.

[52] Q. Pu, S. Venkataraman, and I. 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, 2019.

[53] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[54] H. Shafiei and A. Khonsari. Serverless computing: Opportunities and challenges. *arXiv preprint arXiv:1911.01296*, 2019.

[55] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang. Legoos: A disseminated, distributed {OS} for hardware resource disaggregation. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 69–87, 2018.

[56] A. Singhvi, K. Houck, A. Balasubramanian, M. D. Shaikh, S. Venkataraman, and A. Akella. Archipelago: A scalable low-latency serverless platform. *arXiv preprint arXiv:1911.09849*, 2019.

[57] S. Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730. ACM, 2012.

[58] P. Stuedi, A. Trivedi, J. Pfefferle, A. Klimovic, A. Schuepbach, and B. Metzler. Unification of temporary storage in the nodekernel architecture. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 767–782, 2019.

[59] M. Thömmes. Squeezing the milliseconds: How to make serverless platforms blazing fast, 2017.

[60] K. Vodrahalli and E. Zhou. Using software-defined caching to enable efficient communication in a serverless environment.

[61] H. Wang, P. Shi, and Y. Zhang. Jointcloud: A cross-cloud cooperation architecture for integrated internet service customization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1846–1855, June 2017.

[62] K.-T. A. Wang, R. Ho, and P. Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, page 39. ACM, 2019.

[63] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift. Peeking behind the curtains of serverless platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 133–145, Berkeley, CA, USA, 2018. USENIX Association.

[64] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320, 2006.

[65] S. Werner, J. Kuhlenkamp, M. Klems, J. Mller, and S. Tai. Serverless big data processing using matrix multiplication as example. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 358–365, Dec 2018.

[66] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):184, 2019.

[67] T. Zhang, D. Xie, F. Li, and R. Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12. ACM, 2019.