On Merits and Viability of Multi-Cloud Serverless

Ataollah Fatahi Baarzi
The Pennsylvania State University
ata@psu.edu

George Kesidis
The Pennsylvania State University
gik2@psu.edu

Carlee Joe-Wong
Carnegie Mellon University
cjoewong@andrew.cmu.edu

Mohammad Shahrad
University of British Columbia
mshahrad@ece.ubc.ca

Abstract

Serverless computing is a rapidly growing paradigm in the cloud industry that envisions functions as the computational building blocks of an application. Instead of forcing the application developer to provision cloud resources for their application, the cloud provider provisions the required resources for each function "under the hood." In this work, we envision virtual serverless providers (VSPs) to aggregate serverless offerings. In doing so, VSPs allow developers (and businesses) to get rid of vendor lock-in problems and exploit pricing and performance variation across providers by adaptively utilizing the best provider at each time, forcing the providers to compete to offer cheaper and superior services. We discuss the merits of a VSP and show that serverless systems are well-suited to cross-provider aggregation, compared to virtual machines. We propose a VSP system architecture and implement an initial version. Using experimental evaluations, our preliminary results show that a VSP can improve maximum sustained throughput by $1.2x$ to $4.2x$, reduces SLO violations by 98.8%, and improves the total invocations' costs by 54%.

CCS Concepts

• **Computer systems organization → Cloud computing**.

Keywords

serverless, multi-cloud, cloud computing

ACM Reference Format:
Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. 2021. On Merits and Viability of Multi-Cloud Serverless. In *ACM Symposium on Cloud Computing (SoCC '21), November 1–4, 2021, Seattle, WA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3472883.3487002

SoCC '21, November 1–4, 2021, Seattle, WA, USA

ACM ISBN 978-1-4503-8638-8/21/11…$15.00
https://doi.org/10.1145/3472883.3487002

1 Introduction

Serverless computing is one of the fastest growing cloud paradigms. It aims to decouple infrastructure management from application development. Under the serverless paradigm, the application developer does not worry about provisioning and up-/down-scaling resources, which can be particularly tricky in the shadow of varying demand. Instead, the provider conducts automatic and scalable provisioning. This, complemented with the pay-as-you-go model and scale-down-to-zero capabilities have made serverless one of the most exciting cloud models for developers. Today, all major cloud providers have serverless offerings, mainly in the form of Function-as-a-Service (FaaS): AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. Serverless is still growing rapidly, where new system features are frequently introduced and various aspects of pricing and quality of service (QoS) are constantly re-visited.

In this paper, we present and argue for the concept of Virtual Serverless Provider (VSP), a third-party entity that aggregates serverless offerings from multiple providers and seamlessly delivers a superior unified serverless to developers. In doing so, it prevents vendor lock-in problems and exploits competition between providers to achieve a confluence of cost, performance, and reliability goals. Specifically, this paper presents the merits of having VSPs take advantage of different pricing schemes as well as variable performance and to achieve enhanced cost, performance, and scalability.

Certain characteristics of serverless systems and ongoing technology trends make the VSP model viable. Unlike conventional cloud federations of VMs, serverless functions are much lighter and faster to deploy, eliminating the pain of slow migrations. Additionally, there are already a number of open-source provider-agnostic serverless frameworks and tools that pave the way towards building efficient and cost-effective multi-cloud serverless aggregation. Finally, a number of recent technical advances and ongoing research areas will improve the performance predictability of today's

	Cost						
	Request Cost		Function Execution Cost				
Provider	Free	Add. Cost	Free (GB-s)	Add. Cost (per GB-s)	Min. Duration	Round-up Resol.	Mem. Metering
AWS Lambda	1 M	\$0.2/M	400,000	$\$1.67 \times 10^{-5}$	1 ms	1 ms	Static
Azure Functions	1 M	\$0.256/M	400,000	$\$2.1 \times 10^{-5}$	100 ms	1 ms	Dynamic
GCP Functions	2 M	\$0.4/M	400,000*	$\$2.5 \times 10^{-5}$*	100 ms	100 ms	Static

Table 1: The pricing schemes of AWS Lambda, Azure Functions and GCP Cloud Functions consumption plan as of May 28, 2021. (M: Million, GB-s: GB-seconds). *: GCP Cloud Functions charges additional $\$10^{-4}$ per GHZ-seconds with 200,000 free GHZ-seconds.

highly variable serverless systems [18]. Increased predictability allows the VSP to better optimize choices.

Designing and implementing VSPs opens new research directions. One such avenue is developing domain-specific VSPs (e.g., for ML [11], etc.) to enhance the optimality of the multi-cloud deployment. This can potentially lead to new cloud market structures.

2 Merits

2.1 Harnessing Cost and Service Variances

Table 1 summarizes the pricing elements for three popular serverless providers as of May 28, 2021. As seen, the final cost of a serverless function[1] depends on the number of invocation requests as well as execution. The execution cost is a combination of memory usage and execution time. AWS Lambda and Azure Functions offer the same limits on their free tiers: the first one million requests and 400 GB-seconds in a month are free. Beyond that, AWS Lambda charges slightly less than Azure Functions for additional requests and GB-s. However, the way the GB-second consumption is calculated varies between these two providers. Azure Functions comes with the overhead of charging for at least 100ms of execution, whereas Lambda is limited by static memory allocation and charging for memory even when the function is not using it. This makes running a sub-hundred-millisecond function with consistent memory usage cheaper on Lambda, and running a two-hundred-millisecond function with bursty memory usage cheaper on Azure Functions. There exist many more subtle differences in serverless pricing, especially when the entire ecosystem including storage and third-party services is considered. However, this simple example should suffice in elaborating that the cost-optimally of each serverless provider depends on the workload. The VSP can exploit this to minimize developers' overall costs while maintaining acceptable performance.

In addition to using provider differences to reduce cost, the VSP can use it for performance optimizations. An example is how different providers deploy various function keep-alive policies, considering that function containers/VMs cannot be kept alive (resident in memory) indefinitely [44, 51]. The VSP can pick the one that suits the application the best. For instance, if invocations are infrequent and too irregular and bound to get cold starts anyway, the VSP would rather map it to the FaaS provider with the cheapest offering, regardless of performance. On the other hand, an application with a predictable invocation pattern can benefit from Azure Functions' adaptive Hybrid Histogram policy [44] to get fewer cold starts and thus better average and tail latency.

Pushing this angle further, the VSP can also use the real-time performance monitoring data to adaptively distribute its function requests away from those providers experiencing a slowdown. This strategy is supported by empirical evidence that the performance variation for different providers is statistically independent. Wang et al. monitored the cold start latency of AWS, Google, and Azure for over a week and showed that their performance degradation is uncorrelated [51]. Performance variation is not limited to different providers and can exist across various regions of the same provider. Ginzburg et al. [18] reported significant performance variations in two AWS Lambda regions: an 11% difference in end-to-end performance as well as a 12-hour lag in daily performance degradation hour. In Section 5 we demonstrate the feasibility and performance benefits of such adaptive scheduling across serverless platforms.

2.2 Fusing the Benefits of Providers

Scalability is a central promise of serverless computing and stems from the fact that developers are not responsible for resource provisioning. It is achieved by state-of-the-art auto-scalers, relying on function images being lightweight, and functions being stateless often. Despite all of these, this scalability is not infinite. Previous characterization studies on production serverless systems have observed different scalability rates and patterns for various providers [31]. Lloyd et al. observed that going beyond fifty concurrent requests significantly increased the cold-run execution time of functions served by Azure Functions [29]. Parallel deployment on multiple serverless providers allows the VSP to increase the scalability limits, both in terms of ramp-up speed as well as sustained throughput. This parallel deployment requires low-latency load balancing at the VSP, which is viable, as the VSP is not performing high latency tasks such as language runtime or application-specific initializations [48]. We demonstrate this later in Section 5.

In addition to scalability benefits, using multiple providers for each application creates a larger virtual monthly free-tier by combining free-tier limits from multiple providers. This potential should not be overlooked, as recent production serverless traces released by Azure [44] revealed that the majority of serverless applications are invoked infrequently (*"81% of the applications are invoked once per minute or less*

[1]This excludes additional storage or data transfer costs.

on average" [44]). This means that while a small percentage of applications that are heavily invoked would probably see no cost-saving from free tier extension, a significant share of medium class applications would enjoy it.

2.3 Data-Aware Deployment

In a multi-cloud setting, it is possible that the business data and compute resources are distributed among different cloud providers due to business decisions or other technical or economic incentives. In such cases, putting computation closer to the data is the desired approach [54]. A business might not be able to migrate parts of data out of a provider or region to comply with data governance and protection laws such as GDPR. A VSP would ease the deployment of such applications by scheduling and deploying functions such that 1) they are as close as possible to the data that they consume, 2) the data transfer between regions is minimized to reduce data transfer (and processing) costs, and 3) data protection laws are complied with.

3 Viability

We next consider the *viability* of building a VSP. To do so, we compare serverless to conventional cloud systems already federated. We also enumerate a number of new systems and technology trends that either have paved or will pave the way for VSPs.

3.1 Fast Dispatch

Conventional federated cloud systems are built on top of slow-booting VMs. This requires caching VM images or keeping some backup VMs alive in order to ensure smooth transitions between providers. Building a federation of serverless offerings, however, is easier to handle as the cold boot of functions is much faster than that of VMs. Initiating a medium-size VM on AWS EC2 with 2 vCPUs, 8 GB of memory, and 8 GB of SDD storage (100/3000 IOPS) took us ~250 seconds [2]. On the other hand, initiating Node.js 8 or Python 3.6 functions on AWS Lambda took around ~1 seconds, which is 250 times faster. These VM initiation measurements were conducted without transferring any VM or container images; if those transfers were included, the gap in initiation times would have been even more dramatic due to the significantly larger size of VM images. Thus, VSP users would likely experience significantly lower delays than in traditional cloud federations, making them a viable option for application developers.

3.2 Provider-Agnostic APIs and Bridges

Vendor lock-in hinders the viability of VSPs: Serverless offerings are often designed to be compatible with other cloud services within the cloud provider. For example, they might have native integration with other services such as event sources, logging and metrics services, queue services, and other specific services on the cloud provider they belong to. Because of this service-level vendor lock-in problem, serverless developers cannot always benefit from services offered by other cloud providers.

Fortunately, there are several tools and frameworks which prevent API lock-in at the API-level [45]. Examples include the Serverless Framework [42], Gloo [19], PyWren [22], Fn Project [15], and Nimbella [36]. For instance, Gloo is an API gateway and controller specifically designed to support hybrid applications and clouds.

In a multi-cloud setting, it is also vital to allow functions to consume events from different event sources on multiple clouds. New systems have been introduced to support cross-cloud event bridging. Unlike AWS's EventBridge [3], TriggerMesh's EveryBridge [14] can consume event data from various sources to trigger functions on any public or private cloud. Open source and cloud-native initiatives such as Knative [24] can also be used. In particular, Knative Eventing [25] can be used to implement cloud-agnostic event sources and event consumers.

Regarding the service-level vendor lock-in problem, there remain opportunities to develop more tools and platforms to support seamless cross-cloud bridging in order to enable serverless functions on one cloud to utilize services on another cloud. Collectively, these tools and platforms circumvent the locking effect from provider-specific APIs and services and pave the way towards production class VSPs.

3.3 Performance Predictability

Increased performance predictability of serverless systems would facilitate building VSPs. Currently, one major source of latency variation in serverless environments is cold start overhead. There have been many advances on this topic in the past few years. These include techniques that reduce cold start latency, including lightweight virtualization [1, 49], sandboxing [2], snapshotting [10, 13], ahead-of-time allocation of network interfaces [32], and language runtime optimizations [12]; and techniques that reduce the number of cold starts through adaptive lifetime management of functions [8, 16, 44]. Given the ongoing research in this angle, we anticipate cold start effects to be mitigated to a great degree. Data caching is another practical solution for performance predictability and cost reduction [35, 40].

4 Proposed Architecture

Building and deploying a VSP requires careful design of an architecture that can ensure seamless transitions between providers and exploit variations in pricing and performance. Figure 1 shows a high-level overview of our proposed architecture for the VSP, consisting of eight modules: a utility-driven scheduler, controller, event bridge, performance monitor, pre-loader, local cache, billing, and authentication. We describe the role of each of these modules below.

[2] This measurement was taken using a Ubuntu Server 18.04 LTS image in the North Virginia region.

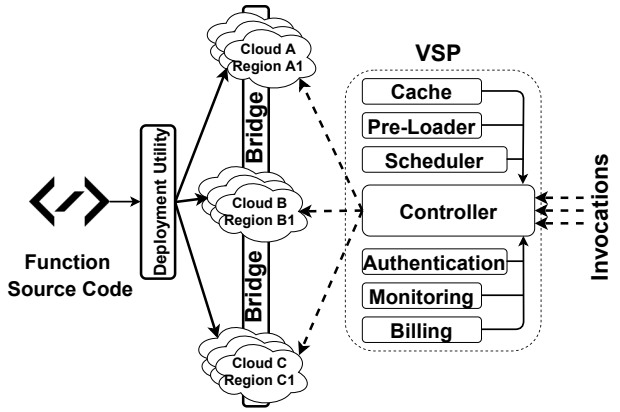

Figure 1: The high-level overview of the proposed VSP.

Utility-driven Scheduler: This module uses performance metrics and cost information to find the right provider(s) to maximize user utility. Utility optimization policies could include cost minimization, performance maximization, or arbitrary combinations of the two objectives. The scheduler also considers hard constraints such as compliance, resource allocation limits, etc. to eliminate unacceptable providers. To account for potential variations in performance and to avoid overloading a single provider (and potentially degrading its performance), the scheduler may identify multiple providers that provide near-optimal performance. Since function performance can vary not only between providers but also between different regions of a single provider, the granularity of cloud environment options that we consider is a single region of a single provider.

Controller: The controller maps requests of each application to its set of providers according to the scheduler's utility optimization results, acting as a lightweight load balancer across the providers identified by the scheduler. For instance, if the performance at one provider suddenly degrades, the controller can quickly shift to another provider.

Event Bridge: Once the scheduler schedules different functions on different cloud environments, it is possible that a function on cloud A, needs to consume events from cloud B. This module as discussed before enables cross-cloud event sourcing and consumption.

Performance Monitor: A function's performance can vary over time due to varying inputs or as a result of change in the QoS of underlying providers. As a result, the set of optimal providers may vary as well. This module logs and tracks performance metrics such as latency and execution times of functions running on different providers. If a major deviation from the performance history is observed, this module can trigger the utility-driven scheduler to update the optimized mapping of functions/applications to providers.

Pre-loader: In case the scheduler identifies new providers (or new regions within the same provider) as part of the set of optimal providers, this module starts initializing and invoking the application functions in the new provider/region to warm them up. It then notifies the controller module to update the set of available regions, and the controller can begin utilizing the new provider. This module reduces perceived latency caused by switching to new providers.

Cache: Maintaining a local cache at the VSP enables a number of optimizations. For pure and memoizable functions, this eliminates sending repetitive invocations to providers. For functions with annotated data intent [47], the VSP can use the cached data to accelerate data movement across providers to keep compute and data co-located. For generic functions without annotations or hints, recent serverless data cache designs such as OFC [35] and Faa$T [40] can be extended to operate at the VSP-level. The main challenges in designing the VSP's cache are delivering consistency guarantees and prioritizing what to cache and for what duration.

The VSP finally needs **billing** and **authentication** modules to facilitate developer payments and authentication between users and FaaS providers. The billing service keeps track of the number of requests from each user, the execution, and the corresponding providers; the VSP can then bill developers directly for both its own aggregation services and the serverless computing resources that they use at each provider. The authentication module should bridge the end-to-end authentication between users and FaaS providers.

In terms of developer experience, deploying a serverless application is similar to deploying to the cloud environment that the developer is currently using. In addition to the required deployment configuration, the developer will provide a list of cloud environments that their application can be deployed on. A deployment utility is used to deploy the application on a VSP backed by multiple cloud environment options.

5 Preliminary Results

To further showcase the merits mentioned above, we implemented the prototype of a VSP. Using our preliminary results, we demonstrate how the VSP allows maintaining the QoS when a provider slows down and how it reduces the costs by dynamically prioritizing cheaper providers.

5.1 Experimental Setup

The VSP controller is implemented using GoLang programming language. We deploy the controller on a *m4.2xlarge* AWS VM instance with 4 vCPU and 32 GB of memory. We use an open-loop load generator as our client and deploy it on a separate *m4.2xlarge* instance. Our evaluation benchmarks cover a range of applications and consist of these five functions[3]:

- *NLP*: A function that scrapes a random article from Wikipedia and builds a bag-of-words model from it.

[3]Available here: https://github.com/PSU-Cloud/vsp-benchmark

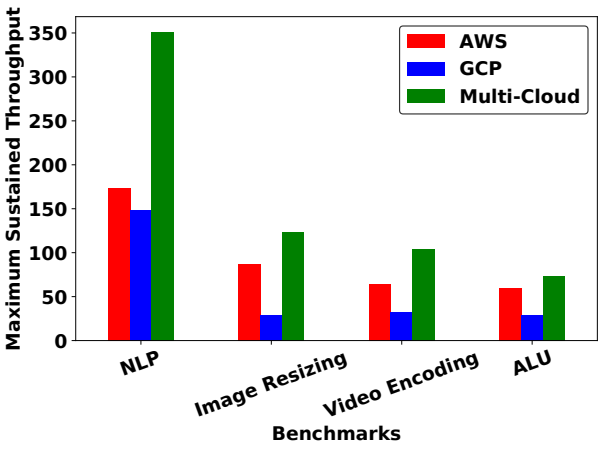

Figure 2: Given the same concurrency limit (1,000), the maximum sustained throughput is different for AWS and GCP. The multi-cloud VSP seamlessly increases this throughput.

- *Image Resizing*: A function that gets an image from an S3 bucket and resizes it to three target sizes and a thumbnail.
- *Video Encoding*: A function that gets a video file from an S3 bucket and converts it to GIF format using the *ffmpeg* utility.
- *ALU*: A CPU-intensive function from the Serverless-Bench [53] that launches a random number of threads to perform arithmetic calculations in a loop with a random number of iterations.
- *Sleepy*: A function that sleeps for one second, then sends a request to a back-end web service and it returns once the response is received. The web service receives a random number from the functions and performs an addition to it and returns.

Unless otherwise mentioned, all the functions are configured to have 512 MB of memory. We use AWS Lambda and Google Cloud Functions as the underlying providers of the VSP.

5.2 Results

Increasing the concurrency limit. In order to provide highly available managed services, cloud providers set limits on individual tenant's usage. In particular, for FaaS offerings, they set limits on the number of concurrent instances of the function serving invocations. Additional invocations beyond the concurrency limit will fail with throttling errors. A VSP can prevent such failures and thus improve the quality of service by avoiding scheduling further invocations to the cloud provider that has reached its concurrency limit.

To demonstrate this, we deployed our benchmarks on both providers to find the maximum throughput at which we achieve 100% success. Lambda has a default concurrency limit of 1,000. The concurrency of Google Cloud Functions is configurable and we set it to 1,000 for a fair comparison. Figure 2 depicts the maximum sustained throughput for each benchmark using AWS Lambda, GCP Functions, and the multi-cloud VSP setting. We excluded the Sleepy function as it is not a representative workload for throughput. For *Image Resizing* and *Video Encoding*, the GCP-only setting has significantly lower rate as the data resides on an AWS S3 bucket. We included this case intentionally as this might happen for a VSP too. The VSP uses *Least Outstanding Invocations* (*LOI*) as its load balancing policy to distribute invocations among the providers. As seen, the VSP achieves throughput close to the sum of the two cloud providers'. This is due to its low added latency – an average of 150μs in our measurements. In particular, compared to the AWS-only setting, the VSP improves the quality of service by 1.2x (for *ALU*) to 2x (for *NLP*). Compared to GCP only setting, the VSP improves the quality of service by 2.5x (for *ALU*) to 4.2x (for *Image Resizing*).

Latency-aware deployment. As discussed in Section 2.1, VSPs can exploit the variable performance of different cloud providers to reduce latency and cost (through execution time reduction). In particular, by monitoring the performance of each provider, a VSP can identify latency SLO violations and avoid scheduling the invocations to the problematic provider to mitigate SLO violations. To demonstrate this, we deploy the *Sleepy* function and the back-end web services on both AWS and GCP cloud and set the latency SLO to 1.2 seconds. We run a fixed load of 30 requests per second for 4 minutes. To identify the SLO violation, the VSP maintains the Exponentially Weighted Moving Average (EWMA) of latency values for each cloud provider. The VSP de-prioritizes scheduling to the cloud provider(s) with EWMA latency violating the SLO. To emulate performance degradation at a provider, we inject a synthetic anomaly [38] at time t = 30s to add an additional *250ms* latency to the GCP function executions. The anomaly lasts for 170 seconds, and after that, the web service continues to perform normally.

Figure 3 shows the end-to-end latency of invocations. We compare the performance of three settings for the VSP: single provider (GCP), two-provider latency-agnostic VSP with round-robin load balancing, and two-provider setup with latency-aware scheduling. As seen, once the anomaly starts, the invocation latency for the single provider and the latency-agnostic settings increases and lasts during the entire 170-second window. Note that as the back-end web service gets overloaded by the constant invocation rate, latency variance also increases. The latency-aware VSP adapts to the situation in roughly two seconds (given the EWMA averaging, it is not instantaneous) and diverts invocations to the healthy provider (AWS Lambda here).

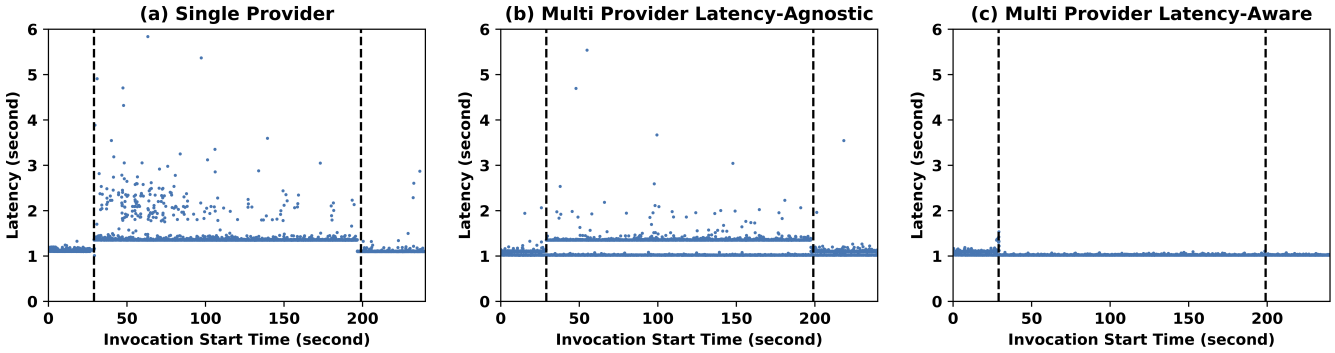

Figure 3: Performance of various VSP scheduling strategies under injected latency anomaly in the 30s-200s window. A latency-aware approach diverts invocations from the slow provider and guarantees meeting the SLO.

Cost improvements. In the pay-as-you-go pricing model of serverless, applications are charged for their true execution. Thus, a higher execution time for non-application-related reasons such as high context switches in overloaded servers [43] would entail a higher cost for developers [5, 17].

In the last experiment, we added the *250ms* extra latency by increasing execution times. In reality, latency variances can stem from non-execution-related sources, such as a network slow-down, an overloaded API gateway, or other sorts of gray failure [21]. However, for brevity, we use the same simple experiment to show the impact of cost-aware scheduling when execution times are monitored by the VSP. Figure 4 shows the distribution of invocation cost and latency values for the experiments shown above in Figure 3. As seen, using a multi-cloud cost-/latency-aware VSP reduces the cost as well as the latency by diverting invocations from the slow provider during the slow-down window. In particular, under the cost-aware VSP, 99% of the requests cost less than $\$1.02 * 10^{-5}$ compared to $\$1.4 * 10^{-5}$ and $\$2.25 * 10^{-5}$under the cost-agnostic VSP and the single provider deployments, respectively. This is equivalent to 27% and 54% cost improvements, respectively.

Note that the cost numbers in this experiment only reflect the execution cost that is often the primary cost factor for serverless systems. However, depending on the application, the total service cost can include other components such as storage cost. Specifically, in situations where data need to be replicated across cloud platforms storage costs can become more significant. The cost calculations and comparison in such scenarios are left for future work.

6 Discussion

Feedback: We are looking to receive feedback from the community on the technical and economic viability of virtual serverless providers, i.e., whether they see any obstacles that we have not already identified to building or deploying a VSP, and potential barriers to the adoption of a VSP. We would also be interested in existing frameworks that might

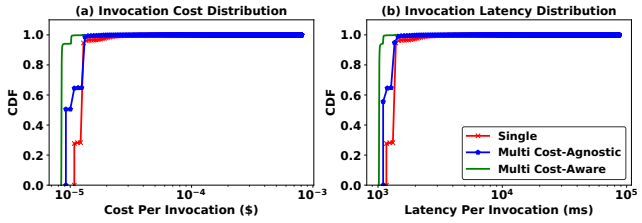

Figure 4: Cost and latency of invocations distribution.

be leveraged to realize this vision, and lessons from prior work on federated cloud computing that might be applicable. While we believe that serverless computing is uniquely suited to federations, as detailed in the body of the paper, some of the challenges that we expect to face (e.g., the need for policy compliance across provider domains) are similar to those in other types of cloud federations.

Controversial points: As we discuss in Section 7, federated architectures and algorithms have been proposed for cloud services for many years. However, these architectures are not widely deployed today, in part due to vendor lock-in effects. Serverless computing is less susceptible to many obstacles to realizing federated clouds (see Section 3), but it remains to be seen whether others would prove fatal to building VSPs. The economic viability of such virtual providers is also a concern; the VSP's scheduler would need to be reconfigured whenever an individual cloud provider significantly changed its serverless pricing or execution logic, which could prove infeasible in practice. Additionally, one may argue that due to economic incentives, cloud providers may try to sabotage the VSPs. In contrast, the VSP can improve the adoption of serverless computing and hence expanding the market for cloud providers. Note that, in our proposed model, the VSP runs third-party code within its own purchased resources and is thus the sole customer dealing with each provider.

Open challenges: Several research challenges remain before production-scale VSPs can be realized. Since a (if not the) major use case of cloud computing is data analytics, many functions invoked by serverless applications might

run on data stored at the FaaS provider. Thus, a viable VSP will require effective mechanisms for maintaining consistent data storage across multiple providers without incurring significant additional costs. Developing algorithms to optimally exploit temporal cost and performance variations at the different providers is another open area of research; it is not clear how the scheduler should determine the optimal set of FaaS providers, or how the presence of a VSP would affect FaaS providers' pricing and performance policies.

VSP operation costs: The reported cost numbers in this paper do not include the operation of the VSP itself. We demonstrate the total savings from aggregating serverless providers. Based on those savings, the VSP, which is a separate entity from the providers, developers, and end-users, decides the profitability of the adoption.

7 Related Work

Researchers have studied various aspects of serverless computing systems in the past few years. Those span over scheduling and resource management, isolation and virtualization, novel applications and services, performance analysis of serverless platforms, and economics of serverless architecture within a single provider. However, to the best of our knowledge, the research community has not explored the multi-cloud serverless angle extensively yet.

Lithops [41] uses Python's multi-processing library to enable running Python programs on multiple serverless providers. It enables the developers to write programs at once and leverage the scalability of multi-cloud deployment. Lowgo [27] is a tracing system for serverless applications deployed on multiple cloud environments and can be used to record the events and dependencies between to ease the correctness and performance debugging. Aske et al. [4] proposed deploying serverless applications on the resource-constrained edge platforms along with the cloud providers. It is limited to scheduling the application on one environment at a time based on latency SLO. In contrast, we propose simultaneous utilization of multiple providers adaptively based on a variety of factors including workload patterns, variable performance, concurrency limits, costs, latency SLOs, data locality, and security. Furthermore, our design aims to improves the total costs by exploiting different providers and multiplexing them. Using simulations, other works [7, 26] have investigated the cost-performance trade-off for executing the serverless functions in environments with computation resource heterogeneity and disparity. They introduce new optimization objectives the results of which can be used by the VSP's utility scheduler for better function placement in a multi-cloud setting.

Cloud computing users have faced challenges like vendor lock-in and differences in pricing even before the advent of serverless computing. To overcome these challenges, several works have proposed *federating* or interconnecting cloud services [23]. Such federations allow cloud providers to pool their resources so that users can build and access services that operate across multiple providers [39]. Since users can freely move between providers, the providers are forced to compete to offer better, less expensive services, thus benefiting users. Indeed, prior work has examined cloud providers' incentives for joining such a federation [30]. A concurrent work recently described the merits and viability of aggregating cloud computing platforms [46]. In this paper, we explore aggregating serverless offerings, its technical challenges, and conduct preliminary experiments with a prototype. Other work has explored building systems for integrating cloud storage services [9, 52], which, much like our proposed VSP, allows users to take advantage of differentiated pricing at different storage services. Similarly, other work has studied the aggregation of content delivery network (CDN) providers [20, 28, 33, 34]. In doing so, distributing an object across multiple CDN providers is somewhat similar to our setup. Building such a virtual provider for serverless computing, however, raises new challenges: in particular, the decision of which provider to use at which time must account for not only storage costs but also execution costs and temporal variations in the performance of the serverless functions. Recently, [50] showed the feasibility of the serverless aggregation idea using a small local cluster. However, we are advocating for large-scale public cloud federation.

Another line of research aims to build algorithms to dynamically utilize multiple cloud services or providers in order to minimize cost. Much of this work has focused on utilizing Amazon EC2's spot and burstable offerings, which offer discounted services at lower availability [6, 37, 55]. These works generally attempt to minimize user costs while participating in multiple markets, with the hope that at least one market will have available resources that the user can utilize at any given time. Thus, their focus is on managing availability by exploiting the dynamics of the spot market. Other work [56] has examined the viability of a virtual provider that aggregates user jobs at multiple cloud providers in order to save costs. However, the temporal variation in serverless computing performance, as well as the relative complexity of its pricing policies, likely require the VSP to develop new scheduling algorithms for mapping functions to providers.

Acknowledgments

We thank David Wentzlaff, Yiying Zhang, and Samuel Ginzburg for their valuable feedback on this work. We also thank the anonymous reviewers and our shepherd, Dongsu Han, for helping us improve the paper. This work was supported in part by NSF CCF grant 2028929, NSF CNS grant 2122155, NSF CNS grant 1751075, NSERC grant RGPIN-2021-03714, and the AWS Cloud Credits for Research program.

References

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

[2] 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, 2018.

[3] Amazon EventBridge. https://aws.amazon.com/eventbridge/. Last accessed on 9/28/2020.

[4] Austin Aske and Xinghui Zhao. Supporting multi-provider serverless computing on the edge. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, pages 1–6, 2018.

[5] AWS lambda pricing. https://aws.amazon.com/lambda/pricing/. Last accessed on 5/28/2021.

[6] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. BurScale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 126–138, 2019.

[7] Matt Baughman, Rohan Kumar, Ian Foster, and Kyle Chard. Expanding cost-aware function execution with multidimensional notions of cost. In *Proceedings of the 1st Workshop on High Performance Serverless Computing*, pages 9–12, 2020.

[8] David Bermbach, Ahmet-Serdar Karakaya, and Simon Buchholz. Using application knowledge to reduce cold starts in FaaS services. SAC '20, page 134–143. ACM, 2020.

[9] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. DepSky: dependable and secure storage in a cloud-of-clouds. *ACM TOS*, 2013.

[10] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. SEUSS: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[11] 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*, pages 13–24, 2019.

[12] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. From warm to hot starts: Leveraging runtimes for the serverless era. HotOS, 2021.

[13] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo 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*, pages 467–481, 2020.

[14] TriggerMesh EveryBridge. https://triggermesh.com/cloud_native_integration_platform/everybridge/. Last accessed on 9/28/2020.

[15] Fn project. https://fnproject.io/. Last accessed on 5/28/2021.

[16] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[17] GCP cloud function pricing. https://cloud.google.com/functions/pricing. Last accessed on 5/28/2021.

[18] Samuel Ginzburg and Michael J Freedman. Serverless isn't server-less: Measuring and exploiting resource variability on cloud FaaS platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, pages 43–48, 2020.

[19] What is gloo edge. https://docs.solo.io/gloo-edge/latest/. Last accessed on 5/28/2021.

[20] Oliver Hohlfeld, Jan Rüth, Konrad Wolsing, and Torsten Zimmermann. Characterizing a meta-CDN. In *International Conference on Passive and Active Network Measurement*, pages 114–128. Springer, 2018.

[21] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The Achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 150–155. ACM, 2017.

[22] 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*, pages 445–451, 2017.

[23] Kiranbir Kaur, DR Sandeep Sharma, and DR Karanjeet Singh Kahlon. Interoperability and portability approaches in inter-connected clouds: A review. *ACM Computing Surveys (CSUR)*, 50(4):1–40, 2017.

[24] Knative. https://knative.dev/. Last accessed on 5/28/2021.

[25] Knative Eventing. https://knative.dev/docs/eventing/. Last accessed on 5/28/2021.

[26] Rohan Kumar, Matt Baughman, Ryan Chard, Zhuozhao Li, Yadu Babuji, Ian Foster, and Kyle Chard. Coding the computing continuum: Fluid function execution in heterogeneous computing environments. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 66–75. IEEE, 2021.

[27] Wei-Tsung Lin, Chandra Krintz, and Rich Wolski. Tracing function dependencies across clouds. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 253–260. IEEE, 2018.

[28] Hongqiang Harry Liu, Ye Wang, Yang Richard Yang, Hao Wang, and Chen Tian. Optimizing cost and performance for content multihoming. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 371–382, 2012.

[29] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless computing: An investigation of factors influencing microservice performance. IEEE IC2E, 2018.

[30] Lena Mashayekhy, Mahyar Movahed Nejad, and Daniel Grosu. Cloud federations in the sky: Formation game and mechanism. *IEEE Transactions on Cloud Computing*, 3(1):14–27, 2014.

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

[32] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[33] Matthew K Mukerjee, Ilker Nadi Bozkurt, Bruce Maggs, Srinivasan Seshan, and Hui Zhang. The impact of brokers on the future of content delivery. In *proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 127–133, 2016.

[34] Matthew K Mukerjee, Ilker Nadi Bozkurt, Devdeep Ray, Bruce M Maggs, Srinivasan Seshan, and Hui Zhang. Redesigning CDN-broker interactions for improved content delivery. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, pages 68–80, 2017.

[35] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. OFC: an opportunistic caching system for FaaS platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 228–244. ACM, 2021.

[36] Nimbella. https://nimbella.com/platform. Last accessed on 5/28/2021.

[37] Hojin Park, Gregory R Ganger, and George Amvrosiadis. More IOPS for less: Exploiting burstable storage in public clouds. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[38] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for slo-oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, 2020.

[39] Benny Rochwerger, David Breitgand, Eliezer Levy, Alex Galis, Kenneth Nagin, Ignacio Martín Llorente, Rubén Montero, Yaron Wolfsthal, Erik Elmroth, Juan Caceres, et al. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4–1, 2009.

[40] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. Faa$T: A transparent auto-scaling cache for serverless applications. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21. ACM, 2021.

[41] Josep Sampé, Pedro Garcia-Lopez, Marc Sánchez-Artigas, Gil Vernik, Pol Roca-Llaberia, and Aitor Arjona. Toward multicloud access transparency in serverless computing. *IEEE Software*, 38(1):68–74, 2020.

[42] The Serverless framework. https://serverless.com. Last accessed on 5/28/2021.

[43] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1063–1075, 2019.

[44] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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, 2020.

[45] Josef Spillner. Practical tooling for serverless computing. In *Proceedings of the 10th International Conference on Utility and Cloud Computing*, pages 185–186, 2017.

[46] Ion Stoica and Scott Shenker. From cloud computing to sky computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 26–32, 2021.

[47] Yang Tang and Junfeng Yang. Lambdata: Optimizing serverless computing by making data intents explicit. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, pages 294–303. IEEE, 2020.

[48] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 311–327, 2020.

[49] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, 2018.

[50] Adbys Vasconcelos, Lucas Vieira, Italo Batista, Rodolfo Silva, and Francisco Brasileiro. DistributedFaaS: Execution of containerized serverless applications in multi-cloud infrastructures, 2019.

[51] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, 2018.

[52] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308, 2013.

[53] Tianyi Yu, Qingyuan Liu, Dong Du, Yubin Xia, Binyu Zang, Ziqian Lu, Pingchao Yang, Chenggang Qin, and Haibo Chen. Characterizing serverless platforms with ServerlessBench. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '20. ACM, 2020.

[54] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–12, 2019.

[55] Yang Zhang, Arnob Ghosh, and Vaneet Aggarwal. Optimized portfolio contracts for bidding the cloud. *IEEE Transactions on Services Computing*, 2018.

[56] Liang Zheng, Carlee Joe-Wong, Christopher G Brinton, Chee Wei Tan, Sangtae Ha, and Mung Chiang. On the viability of a cloud virtual service provider. *ACM SIGMETRICS Performance Evaluation Review*, 44(1):235–248, 2016.