The Pennsylvania State University
The Graduate School

MULTI-CLOUD SERVERLESS DEPLOYMENT

A Thesis in
Computer Science and Engineering
by
Ataollah Fatahi Baarzi

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

May 2021

The thesis of Ataollah Fatahi Baarzi was reviewed and approved by the following:

George Kesidis
Professor of Electrical Engineering and Computer Science
Thesis Advisor

Abutalib Aghayev
Assistant Professor of Computer Science and Engineering

Chita R. Das
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

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 thesis, we propose 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 better services. We discuss the merits of a VSP and show that serverless systems are well-suited to cross-provider aggregation, compared to virtual machines. Finally through experiments we demonstrate how a VSP can prevent from performance degradation due to concurrency limits by a single cloud provider.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Serverless computing is one of the fastest growing paradigms in the cloud industry. It aims to decouple infrastructure management from application development. Under the serverless paradigm, the application developer does not worry about provisioning and specifying the number and type of virtual machines (VMs) that it requires, which is particularly tricky in the shadow of varying demand. Instead, the provider conducts automatic and scalable provisioning.

At the core of existing serverless computing offerings lies the Function-as-a-Service (FaaS) model where functions form the computational building blocks. Functions are often stateless and the application logic is performed in an event-driven style. Deployment of an application then becomes as easy as coding these functional building blocks and connecting them to each other, external triggering events, and/or storage services.

Today, all major cloud providers offer FaaS: AWS Lambda, Azure Functions, Google Cloud Functions, and IBM Cloud Functions. However, FaaS 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. The resulting uncertainty and fast-moving changes to serverless offerings allow developers to exploit competition between providers in order to achieve better performance. In addition, when a developer chooses to use a specific cloud provider to deploy her serverless application, she has to also stick to the other services within that provider in order to implement and build her business logic (consisting of the serverless functions and other services such as database and metric services). This situation is referred to as **vendor lock-in**, where the developer is limited to the scope of a specific cloud provider.

In this thesis, we present and argue for the concept of a Virtual Serverless Provider (VSP), where a third-party entity aggregates the serverless offerings of FaaS providers. In doing so, it 1) prevents vendor lock-in problems and 2) can exploit competition between

providers to achieve a confluence of cost, performance, and reliability goals. Specifically, this thesis presents the merits of having VSPs to take advantage of different pricing schemes as well as variable performance and to achieve superior scalability. The aim is to enable developers to achieve better performance and lower cost as well as circumventing vendor lock-in problems.

We show that certain characteristics of serverless systems and technology trends make the VSP model viable: 1) unlike conventional cloud federations, serverless functions are much faster to deploy compared to VMs; 2) open-source provider-agnostic frameworks and tools such as the Serverless Framework [34] and Gloo [16] already exist; and 3) recent works on container isolation and novel serverless management techniques can improve the predictability of serverless systems. Fast function deployment and interoperability between providers allow VSPs to rapidly deploy functions at different providers, while predictable performance at each provider allows them to better optimize the choice of which provide to utilize.

The rest of this thesis is organized as follows. In Chapter 2, we list and argue for some of the merits of VSPs. Chapter 3 is dedicated to the viability of our proposal. We discuss the main components of our proposed system in Chapter 4. Our case study experiment and evaluation is presented in Chapter 5 The related work is presented in Chapter 6. In Chapter 7 we conduct a detailed discussion of our proposal, its pitfalls, and future work. Finally, we conclude this thesis in Chapter 8.

Chapter 2
Merits

We first give evidence of the merits of virtual serverless providers. By allowing easy access to multiple providers, VSPs allow developers to exploit the different pricing schemes and performance characteristics offered by various providers, both of which may vary over time. VSPs also permit higher scalability limits, helping serverless computing realize its promise of easy scalability according to developer needs.

2.0.1 Exploiting Different Pricing Schemes

The VSP can exploit even the smallest differences between the pricing schemes of FaaS providers to minimize developers' overall costs while maintaining acceptable performance. To elaborate on this point, we compare the cost of using AWS Lambdas to Azure Functions, which share many similarities. Table 2.1 lists the details of FaaS pricing for these two providers. As seen, the final cost of a function (i.e., excluding storage or data transfer) 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. They charge the same $0.2 per extra million requests. However AWS bills slightly more (~4.2%) for additional GB-seconds.

So far, the pricing models do not appear very different. However, while AWS Lambda always rounds up to the closest 100ms and charges developers based on the rounded value [4], Azure Functions rounds up to the nearest 1ms after billing for a minimum of 100ms [6]. This means that a function duration of 102ms is charged for 200ms and 102ms from AWS and Azure, respectively. Figure 2.1 combines all pricing details to show how much more expensive Lambda can be compared to Azure Functions in a variety of scenarios. As shown, the relative cost depends on the function memory size, the number

	Cost					
	Request Cost		Function Execution Cost			
Provider	Free	Add. Cost	Free (GB-s)	Add. Cost (per GB-s)	Min. Duration	Round-up Resolution
AWS Lambda	1 M	$0.2 per M	400,000	**$0.00001667**	100 ms	**100 ms**
Azure Functions	1 M	$0.2 per M	400,000	**$0.000016**	100 ms	**1 ms**

Table 2.1. The pricing schemes of AWS Lambda and Azure Functions as of 9/28/2020. (M: Million, GB-s: GB-seconds)

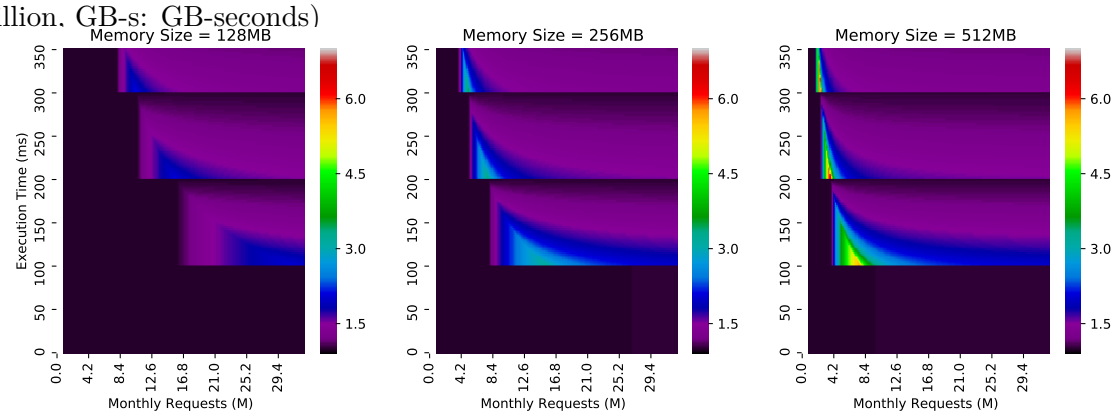

Figure 2.1. The ratio of monthly cost of using AWS Lambda over Azure Functions'. The relative costliness of AWS Lambda over Azure Functions depends on function execution time, number of requests per month, and function memory size.

of invocations, and the duration of function executions.

The above analysis suggests that developers would always incur lower costs by using Azure Functions instead of AWS Lambda. However, AWS Lambda may have other advantages to make it favorable for a serverless workload. For instance, it allows memory allocations up to 3,008MB for functions [4] while Azure's is limited to 1,536MB [6], or it supports a maximum duration of 15 minutes, while Azure's default Consumption plan has a 10 minutes limit. Thus, applications with high memory requirements or long executions may wish to utilize AWS Lambda, but might want to switch to Azure after shrinking. The VSP can facilitate these variable configurations by allowing developers access both FaaS providers, and can conduct careful pricing analyses like the one presented here to optimize the deployment configurations for individual applications.

2.0.2 Utilizing Variable Performance

The invocation pattern of functions directly affects their latency behavior, as serverless providers cannot keep function containers/VMs live (resident in memory) forever [36, 44]. As different providers deploy various keep-alive policies [36], 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's invocation pattern might fits well in a specific provider's keep-alive policy, for example in Azure Function Hybrid Histogram scheduling policy which currently delivers superior cold start performance [36]. Second, by using real-time performance monitoring of FaaS providers, the VSP can 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 performance degradations are uncorrelated [44]. Thus, it is likely that different providers will show better performance at different times.

Another aspect of variable performance of public cloud providers is the concurrency limit that they set for function invocations. That is, if the number of concurrent invocations of a function exceeds a certain threshold (the concurrency limit set by the provider), the function invocation will either drop or will be queued which can potentially lead to SLO violations. A VSP therefore would keep track of the number of concurrent invocations and schedule the function invocations such that they don't reach the limits set by each cloud provider.

2.0.3 Scaling-Up the Scalable

Scalability is one of the central promises of serverless computing and stems from the fact that developers are not responsible for resource provisioning [8]. It is achieved by FaaS providers using auto-scalers, relying on the fact that FaaS containers/VMs are much more lightweight than IaaS VMs, and taking advantage of less application code. However, there are practical limits to scalability.

McGrath et al. tested the serverless scalability limits by conducting concurrency tests on four FaaS providers [26]. They observed remarkably different scalability numbers from those providers. They also found that the scalability pattern can differ significantly; for instance, while Google Cloud Functions had sub-linear scalability with increasing

concurrency, Azure Functions showed a variable pattern. In a more recent study, Lloyd et al. observed that going beyond fifty concurrent requests significantly increased the cold-run execution time of functions served by Azure Functions [22].

A VSP can increase the scalability limit by deploying multiple serverless domains in parallel and combining the scalability gains at each of them. 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 [17].

2.0.4 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 [46]. 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 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.

Chapter 3
Viability

Having established the potential merits of VSPs, we next consider the *viability* of building a VSP that aggregates services from multiple FaaS providers. 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.

3.0.1 Fast Dispatch

Conventional federated cloud systems are built on top of slow-booting VMs, which 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 [17]. 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 [1]. 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.

3.0.2 Multi-Cloud Event Bridging

In order to have a promising and true multi-cloud serverless application, functions have to be able to access services from other cloud environments or consume events from

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

them. By default, a function can only consume the events within the services from the cloud provider that they belong to. Therefore, in a multi-cloud setting, it is important to enable functions to consume the events from different event sources on multiple clouds. Event bridging enables a function that is running on cloud A, to consume the events from services on cloud B.

New systems have been introduced to support cross-cloud event bridging. Unlike AWS's EventBridge [3], TriggerMesh's EveryBridge [13] 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 [20] can also be used. In particular, Knative Eventing [21] can be used to implement cloud native (and hence compatible with all the cloud providers) event sources and event consumers.

3.0.3 Provider-Agnosticism, a Growing Trend

Vendor lock-in hinders the viability of VSPs: FaaS offerings by cloud providers are designed such that the serverless functions within a cloud provider be suitable with the other services provided by the same 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. Therefore, because of this service-level vendor lock-in problem, a tenant owning a serverless application will not be able to benefit from other services from other cloud providers.

Though the vendor lock-in problem could exist in API-level, there are a number of tools and frameworks which prevent API lock-in on a single provider [40]. Examples include the Serverless Framework [34], Gloo [16], PyWren [18], Fn Project [14], and Nimbella [29]. For instance, Gloo is an API gateway and controller specifically designed to support hybrid applications and clouds.

To overcome the service-level vendor lock-in problem, more tools and platforms need to be designed and developed. These tools support for 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.

3.0.4 Increasing Performance Predictability

Performance predictability enables the VSP to plan and optimize for scheduling policies, i.e., deciding which applications/functions should be invoked on which provider. Without performance predictability, it is difficult for the VSP to adaptively exploit performance variations as discussed in earlier, which limits the potential benefits of a VSP. While current serverless computing systems are not predictable [17], there are some active research areas that can improve the predictability of serverless systems.

Function cold start is a major source of latency variation in serverless environments. A new study by Microsoft researchers showed how adaptive lifetime management of function images can reduce cold starts significantly [36]. Others have explored ways to reduce cold start latency [1, 27, 30, 43]. Lack of isolation between functions is another source of performance degradations [44]. New research systems, such as X-Containers [38] and FAASM [39], and production systems, such as Firecracker [1], have tackled the isolation issues. With the current trend, we anticipate cold start overhead to continue shrinking in the next few years and isolation mechanisms to further improve, leading to more performance predictability.

Resource disaggregation is another relevant research area that has gained significant interest in recent years. It is well-suited to the serverless model since functions are typically stateless [28]. In fact, researchers have already proposed language runtime [24], OS [2], and architecture [11] solutions for such serverless resource disaggregation. Traditional data centers use various server types with different hardware configurations, causing performance variations [17, 37]. Disaggregation enables allocating the right amount of resources to the serverless functions regardless of server type and capacity. This way, performance becomes less server-specific and the intra-provider performance variations are reduced. The VSP can take advantage of increased predictability.

Chapter 4
Proposed Architecture

4.0.1 Architecture

While Chapter 3 establishes the viability of a VSP, building and deploying a VSP requires careful design of an architecture that can ensure seamless transitions between FaaS providers and exploit variations in pricing and performance. Figure 4.1 shows a high-level overview of our proposed architecture. In order to invoke application functions at different FaaS providers, it uses five main modules to ensure high performance and low cost: a utility-driven scheduler, controller, event bridge, performance monitor, pre-loader, and local cache. We describe the roles of each of these modules below.

Utility-driven Scheduler: This module uses recent performance 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 (e.g., Amazon's US-East and US-West regions), 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 an adaptive 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

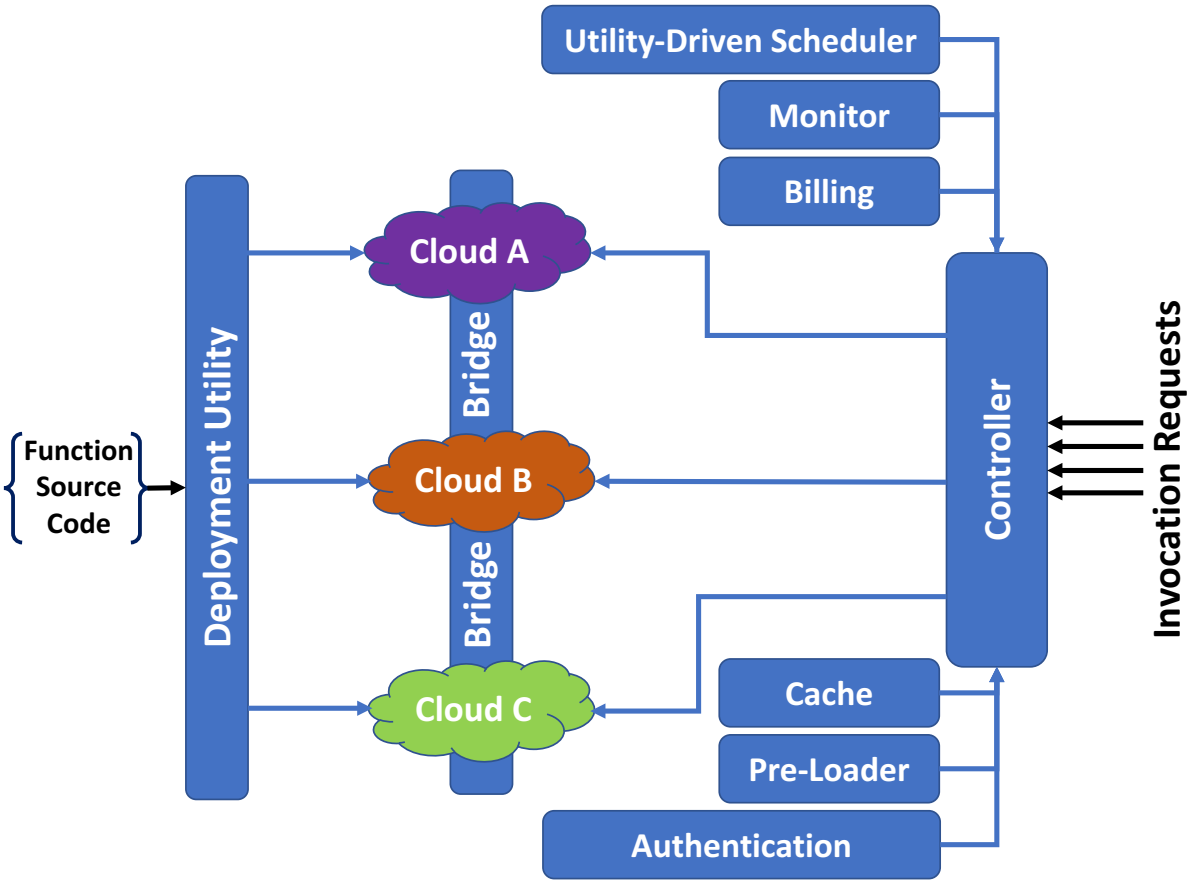

Figure 4.1. The high-level overview of the proposed VSP.

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 consuming.

Performance Monitor: Since function performance can vary over time, 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. It then notifies the controller module to update the set of available regions, and the controller can begin to utilize the new provider. Providing this "warm start" will reduce perceived latency from switching between providers.

Cache: Maintaining a local cache at the VSP allows it to save time and money when encountering similar requests.

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 (memory-seconds), 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.

4.0.2 Deployment Workflow

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 her application can be deployed on. A deployment utility is used to deploy the application on a VSP backed by multiple cloud environment options. Note that the modules of VSP described above themselves are deployed on one (or more than one) cloud environment.

Chapter 5
Case Study Evaluation

In this Chapter we evaluate the effect of function invocation concurrency limit on the quality of service for serverless functions.

As mentioned in 2.0.2 public cloud providers set a concurrency limit on the number of concurrent invocations of the serverless functions. Once the number of concurrent invocations exceed the concurrency limit, the invocation will be dropped and returns a error code to the users.

Major cloud providers' limitations are as follow:

- **Amazon AWS** has a capacity for each tenant which is divided among all the serverless functions for that tenant [5]. A tenant can configure the concurrency for each individual function. By default AWS allows up to 1000 concurrent invocations for each function in a region.

- **Google Cloud Platform** has a limit on individual functions instead of the tenant. In practice, we observed that GCP allows up to 2500 concurrent invocations for each serverless function.

- **Microsoft Azure** has the notion of an "app" which consists one or more than one function. Each app can have up to 200 concurrent instances running where each instance can handle multiple function invocations. Therefore, the number of concurrent function invocations in Azure are limited to 200 times each instances' concurrency level which is configurable.

The above concurrency limit examples by major cloud providers can affect the end to end performance of serverless applications hosted on a single cloud provider. Example scenario include a flash crowd or a surge in traffic can result in degrading the performance as well the user experiences as the invocations get dropped or fail.

A VSP in the other hand, can prevent from performance degradation and poor quality of service experience by scheduling invocations among multiple cloud platforms under the hood. The VSP's scheduler monitors the number of active invocation on each cloud platform and avoids scheduling more invocations on a cloud platform that is near to exceed its limits.

5.0.1 Experimental Setup

Function. To evaluate the effect of concurrency limit on the performance of serverless function invocations we deploy a simple function which sleeps for 10 seconds and returns. **Baseline.** Our baseline is deploying the function in a single region (us-east-1) on AWS. **Workload.** The workload that we use for our evaluation consists of N concurrent invocations per second for 5 consecutive seconds. That is in total we have $5 * N$ invocations.

5.0.2 VSP deployment

As part of VSP deployment, we deploy the function on using the VSP client on three cloud provider platforms: AWS, GCP, and Azure. Figure 5.1 shows part of the function deployment configuration for VSP. Note that in addition to the required configuration, under VSP deployment the developer provides the target public cloud platforms as well.

5.0.3 Results

We offer different load levels ranging from 10 requests per second to 1000 requests per second. Each load level runs for 5 seconds therefore, in total we have 50 to 5000 invocation requests. Figure 5.2 shows the percentage of completed requests at each offered load level. In the baseline scenario where there is a 1000 concurrency limit set on the number of concurrent function invocations the percentage of completed requests starts to decrease at 300 request per second load level. This because at the 300 requests per second offered load, the total number of concurrent invocations exceed the limit and hence some of the invocation requests fail. As can be seen in this figure, as the load increases the percentage of completed requests drops from 100% at 200 requests per second and lower to about 28% at 1000 requests per second.

In the other hand, the VSP scheduler schedules the function invocation among the three configured cloud platforms and for all the offered load levels the percentage of

```
name: SleepyFunction
handler: sleepy_handle
runtime: Golang
...
providers:
  - name: GCP
    secrets: {{secrets.GCP}}
    limits: {{limits.GCP}}

  - name: AWS
    secrets: {{secrets.AWS}}
    limits: {{limits.AWS}}

  - name: Azure
    secrets: {{secrets.Azure}}
    limits: {{limits.Azure}}
...
```

Figure 5.1. Part of he function deployment configuration. The configuration consists of the underline cloud platforms: AWS, GCP, and Azure

completed requests is near 100%. As can be seen in this figure, at 1000 requests per second offered load, the VSP improves the percentage of completed requests by more than 70%.

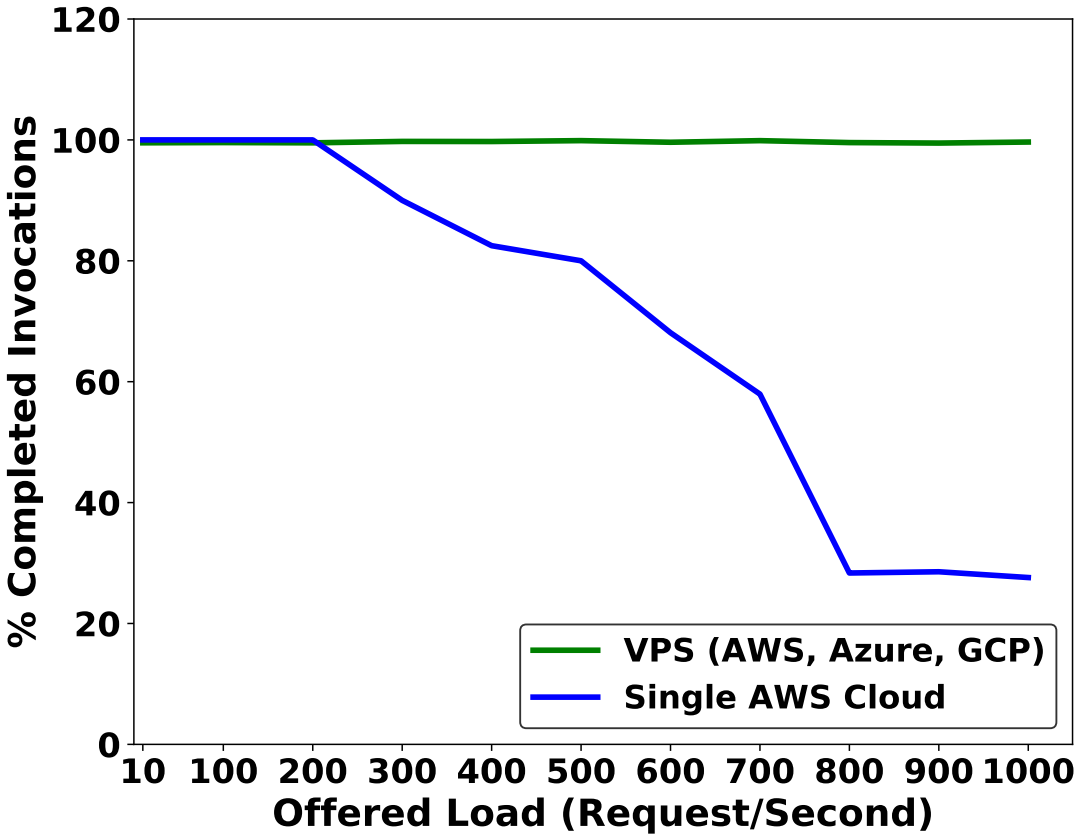

Figure 5.2. Percentage of completed invocation at each load level.

Chapter 6
Related Work

Researchers have studied various aspects of serverless computing systems in the past few years. Those span over scheduling and resource management [30, 36], isolation and virtualiztion [1, 39], novel applications and services [15, 33, 42], performance analysis of public offerings [23, 26, 36, 44] or open source serverless platforms [9, 35], and economics of serverless architecture within a single provider [12, 18]. To the best of our knowledge, our work is the first to explore the merits and viability of virtual serverless providers that aggregate multiple FaaS providers.

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 [19]. Such federations allow cloud providers to pool their resources so that users can build and access services that operate across multiple providers [32]. 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 [25]. Other work has explored building systems for integrating cloud storage services [10, 45], which, much like our proposed VSP, allows users to take advantage of differentiated pricing at different storage services. 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, [41] 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 [7, 31, 47]. 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 [48] has examined the viability of a virtual provider that aggregates user jobs at multiple cloud providers in order to save cost. 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.

Chapter 7
Discussion and Future Work

Controversial points: As we discuss in earlier, 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, 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.

Open issues: 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. 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.

Points of failure: Maintaining consistent data storage across multiple will likely incur additional costs, which may cancel out any savings from utilizing multiple providers. Individual cloud providers might also attempt to block virtual providers from accessing serverless functions in order to preserve the competitive advantages of vendor lock-in.

Chapter 8
Conclusion

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 thesis, we proposed 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 better services. We discussed the merits of a VSP and show that serverless systems are well-suited to cross-provider aggregation, compared to virtual machines. Finally through experiments we demonstrated how a VSP can prevent from performance degradation due to concurrency limits by a single cloud provider.

Bibliography

[1] Alexandru Agache et al. Firecracker: Lightweight virtualization for serverless applications. USENIX NSDI, 2020.

[2] Zaid Al-Ali, Sepideh Goodarzy, Ethan Hunter, Sangtae Ha, Richard Han, Eric Keller, and Eric Rozner. Making serverless computing more serverless. IEEE CLOUD, 2018.

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

[4] AWS Lambda pricing. `https://aws.amazon.com/lambda/pricing/`. Last accessed on 9/28/2020.

[5] Amazon AWS lambda concurrency limit. `https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html`. Last accessed on 02/02/2021.

[6] Azure Functions pricing. `https://azure.microsoft.com/en-us/pricing/details/functions/`. Last accessed on 9/28/2020.

[7] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. ACM SoCC, 2019.

[8] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017.

[9] Daniel Barcelona-Pons, Pedro García-López, et al. FaaS orchestration of parallel workloads. WoSC, 2019.

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

[11] Syrivelis Dimitris et al. A software-defined architecture and prototype for disaggregated memory rack scale systems. IEEE SAMOS, 2017.

[12] A. Eivy. Be wary of the economics of "serverless" cloud computing. *IEEE Cloud Computing*, 4(2):6–12, March 2017.

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

[14] Fn project. `http://fnproject.io`. Last accessed on 9/28/2020.

[15] Sadjad Fouladi et al. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. USENIX NSDI, 2017.

[16] Gloo. `gloo.solo.io`. Last accessed on 9/28/2020.

[17] Eric Jonas et al. Cloud programming simplified: A Berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[18] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. ACM SoCC, 2017.

[19] Kiranbir Kaur, DR Sharma, and DR Kahlon. Interoperability and portability approaches in inter-connected clouds: A review. *ACM CSUR*, 2017.

[20] Knative. `https://knative.dev/`. Last accessed on 9/28/2020.

[21] Knative Eventing. `https://knative.dev/docs/eventing/`. Last accessed on 9/28/2020.

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

[23] Pedro García López et al. Comparison of faas orchestration systems. In *IEEE/ACM UCC*, 2018.

[24] Martin Maas, Krste Asanović, and John Kubiatowicz. Return of the runtimes: Rethinking the language runtime system for the cloud 3.0 era. ACM HotOS, 2017.

[25] Lena Mashayekhy, Mahyar Movahed Nejad, and Daniel Grosu. Cloud federations in the sky: Formation game and mechanism. *IEEE TOCC*, 2015.

[26] Garrett McGrath and Paul R. Brenner. Serverless computing: Design, implementation, and performance. IEEE ICDCSW, 2017.

[27] Anup Mohan et al. Agile cold starts for scalable serverless. USENIX HotCloud, 2019.

[28] Shripad Nadgowda, Nilton Bila, and Canturk Isci. The less server architecture for cloud functions. ACM WoSC, 2017.

[29] Nimbella. `https://nimbella.com/platform`. Last accessed on 9/28/2020.

[30] Edward Oakes et al. SOCK: Rapid task provisioning with serverless-optimized containers. USENIX ATC, 2018.

[31] Hojin Park, Gregory R. Ganger, and George Amvrosiadis. More IOPS for less: Exploiting burstable storage in public clouds. USENIX HotCloud, 2020.

[32] Benny Rochwerger et al. The reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):4–1, 2009.

[33] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. Middleware, 2018.

[34] The Serverless framework. `https://serverless.com`. Last accessed on 9/28/2020.

[35] Mohammad Shahrad, Jonathan Balkind, and David Wentzlaff. Architectural implications of function-as-a-service computing. IEEE/ACM MICRO, 2019.

[36] Mohammad Shahrad, Rodrigo Fonseca, et al. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. USENIX ATC, 2020.

[37] Vaishaal Shankar et al. numpywren: serverless linear algebra. *arXiv preprint arXiv:1810.09679*, 2018.

[38] Zhiming Shen et al. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. ACM ASPLOS, 2019.

[39] Simon Shillaker and Peter Pietzuch. FAASM: Lightweight isolation for efficient stateful serverless computing. USENIX ATC, 2020.

[40] Josef Spillner. Practical tooling for serverless computing. ACM UCC, 2017.

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

[42] Ao Wang et al. InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. USENIX FAST, 2020.

[43] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. ACM EuroSys, 2019.

[44] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. USENIX ATC, 2018.

[45] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. ACM SOSP, 2013.

[46] Tian Zhang, Dong Xie, Feifei Li, and Ryan Stutsman. Narrowing the gap between serverless and its state with storage functions. ACM SoCC, 2019.

[47] Yang Zhang, Arnob Ghosh, and Vaneet Aggarwal. Optimized portfolio contracts for bidding the cloud. *IEEE TOSC*, 2018.

[48] Liang Zheng et al. On the viability of a cloud virtual service provider. ACM SIGMETRICS, 2016.