MTR200236
MITRE TECHNICAL REPORT

MITRE

Edge and Serverless

A Review of Vendors and the Practicalities of Implementation

Author(s): Brinley Macnamara
Michael J. Vincent

May 2020

Sponsor: MITRE MIP
Dept. No.: T864
Project No.: 10MSRF20-PA

Bedford, MA

Approved for Public Release; Distribution Unlimited. Public Release Case Number 20-2050

This page intentionally left blank.

Abstract

Edge and serverless computing have been hailed as emerging technologies with the potential to both reshape internet architectures and redefine the way internet services are developed and deployed. Edge computing is a model for internet architecture that brings compute closer to the network edge, where data is collected or consumed. Serverless computing is a paradigm for software development wherein application code is packaged into modular sub-components and deployed to platforms and infrastructure fully managed by a provider. Together, edge and serverless promise to provide a platform for real-time applications at scale – including self-driving cars, drone delivery services, and virtual and augmented reality experiences. However, the supporting infrastructure for edge computing at scale is far from fully deployed and serverless still requires complex software orchestration to realize applications. In this paper we define terms, review vendors, and discuss use cases. We present a rich set of recommendations for security and deployment backed by hands on experience with the technology. Finally, we provide an overview of the current state of edge and serverless technology and outline our thoughts on these technologies' promising outlook.

This page intentionally left blank.

Executive Summary

Edge and serverless computing have been hailed as emerging technologies with the potential to both reshape internet architectures and redefine the way internet services are developed and deployed. Edge and serverless computing will impact every end user, developer, and organization, including MITRE's sponsors.

Edge computing is a model for internet architectures that brings compute closer to the network edge, where data is collected and consumed, which has the effect of reducing latencies and conserving bandwidth to improve end user experience. Serverless computing is a platform for software deployment that packages application code into modular sub-components – typically referred to as "microservices" or "functions" – that are deployed to infrastructure that is fully managed by a provider. This "server-less" abstraction allows developers to focus on application development instead of infrastructure management.

Because serverless microservices are inherently lightweight, they can be deployed to the edge – including on low compute devices that sit directly within Local Area Networks (LANs). For example, serverless microservices deployed to Internet of Things (IoT) devices can listen for events and process this data in real-time on the host devices themselves, leading to faster generation of insights.

After a thorough review of these technologies, market drivers and vendor offerings, as well as hands-on experience with serverless in a simulated edge setting, it is clear there are tremendous possibilities for future use cases, but the gap between current realities and the imagined future is quite large for all but the most tailored use cases. There are several key reasons for this:

- Writing serverless applications involves a fundamental rethinking of software architectures. In addition to adopting a new coding style, cloud-native serverless applications require a disciplined approach to securing access to cloud resources, optimizing performance, catching failures, and ensuring uptime.
- Even with the billions of IoT devices deployed today, edge computing at a global, national or regional scale will not be a widely available commodity service (i.e. have availability comparable to cloud compute) for least a decade. This is primarily due to the requirements for costly managed physical infrastructure buildouts in widespread geographic locations. Moreover, there is not yet clear consensus on who will lead the buildout: governments, Internet Service Providers (ISPs), Cloud Service Providers (CSPs), software companies, and startups are all contenders. This infrastructure bottleneck will inevitably stall the development of a large swath of applications that will rely on edge compute, as the supportive infrastructure must exist before any software can deliver services over it.

There are some narrowly scoped use cases for deploying serverless functions on low-compute edge devices that are *already* ubiquitous. However, many of these devices do not have the necessary power or compute to support one of the key benefits of serverless: automatic scaling of microservice instances. Additionally, serverless architectures are commonly integrated with other cloud resources (like instances of storage and database services) which present their own challenges for migration to the edge.

The underlying reality in these barriers to serverless computing at the edge is that edge nodes with compute power capable of supporting highly scalable serverless applications must be integrated with legacy infrastructure; a process that requires coordination amongst sometimes competing interests. Amazon Web Services' (AWS) Lambda@Edge[1] service is a good example of how "legacy" infrastructure (in this case AWS' Content Delivery Network, CloudFront) is being repurposed to give customers the ability to push serverless endpoints closer to the edge [1]. However, this is still far (in terms of time and distance) from where the edge needs to be for mission critical, delay intolerant, real-time applications that future internet users will demand.

The three main takeaways from our research are:

1. Edge Computing *at scale* is likely a decade away.
2. Serverless is by no means *effort-less*.
3. Edge computing and serverless can work together to *facilitate real-time data processing closer to the points of data collection and consumption*. Though as of today, use cases are still limited.

[1] https://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html

Table of Contents

List of Tables

1 Introduction

This paper investigates the large market of service offerings for serverless and edge computing, as well as offerings for serverless computing at the edge. We provide definitions of key terms, a review of current and emerging vendors, a review of security considerations, a review of use cases, and recommendations for selecting a provider. After an extensive review of available vendor marketing and technical documentation as well as the underlying realities that emerge from the "fine print" of this documentation, we are skeptical that, in the short term, widespread deployment of serverless applications at the edge is mature enough to warrant adoption by our sponsors. However, we remain hopeful that the construction of supportive infrastructure for widescale edge computing over the coming decade will enable a host of beneficial use cases, including near real-time processing and analyzing of data generated by users and devices at the edge. In today's internet, the "nearest" server to support this use case commonly sits in the cloud. However, in a world with ubiquitous edge compute, data analytics could happen at the point of generation, on-premise, or in a nearby edge data center. In the intervening time, our job is to explore edge computing and serverless platforms that comprise the first wave of technologies that aim to migrate traditionally cloud virtualization technologies to the network edge. By analyzing the state of the art in serverless and edge computing, we also hope to highlight the undeniable chasm between the capabilities of today's technologies and our goals for the internet's future.

1.1 Scope

The intent of this paper is to help our sponsors make informed decisions about whether serverless and edge computing are worth adopting at their current level of maturity. The answer to the question of "to adopt or not to adopt" will depend on the sponsor's use case, as the utility of serverless computing at the edge is currently constrained to a small set of use cases. To help our sponsors determine if their use case can benefit from serverless, edge computing or a combination thereof, we provide a set of high level use cases for serverless computing at the edge, as well as which vendor is best equipped to deliver these services based on customer requirements.

This paper provides an overview of leading edge and serverless computing technologies as well as a discussion about how a customer can run cloud-connected applications at the edge through integration of serverless with on-premises resources. This paper also compares leading edge and serverless computing providers through common evaluation criteria. This paper does not, however, cover any technical deep dives conducted with a specific vendor or open source technology.

This paper highlights some of the security costs and benefits of both edge and serverless computing, as well as some important recommendations for building a secure serverless application. We also detail security considerations for partnering and integrating with public Cloud Service Providers (CSPs). Security guidelines and best practices presented in this paper should not be considered a complete, definitive checklist.

This paper provides a glossary of terms that are typically used in the context of cloud, edge and serverless computing but may not be commonly defined. Common definitions of key terms are essential for understanding the capabilities and uses cases of edge and serverless computing. The fast-paced evolution of this market leads vendors to differentiate their services with a wide range of marketing jargon that at best refer to the same underlying technologies and at worst confuse the capabilities of the services. Our glossary provides a set of succinct definitions to clarify the terms we use throughout the paper.

Every new technology that advances some aspect of current operations comes with tradeoffs to current operating models. Edge and serverless computing are no different. Despite the hype around these technologies, our research was conducted with a skeptical viewpoint so that this paper could provide the most practical guide to the state of the art. This paper does this by describing how edge and serverless computing evolved, how they are currently defined, as well as the inherent tradeoffs that come with using these technologies.

1.2 Background

To understand edge and serverless computing and the benefits they can bring as emerging technologies, we present a brief history of virtualization, internet architectures and cloud computing technology. It is within this context that edge and serverless computing are emerging to create advantages for specialized application deployments and unique service delivery models.

1.2.1 A Brief History of Virtualization

The foundation of serverless computing was laid in the early 2000's when some key additions to the Linux kernel, namely *cgroups* and *namespaces*, became the basis for containerization technology – an alternative to virtual machines (VMs) that was revolutionary for its ability to isolate applications from their host machines through the use of portable and extremely lightweight *containers*.

Cgroups and namespaces eventually gave rise to Linux Containers (LXC) in 2008 and subsequently Docker[2] – an open source container project – in 2013. Docker is now the most popular platform for running containers on Linux systems [2]. A container's essential role is to automate the isolation of a process from its host operating system (OS) and any other containers running on the host [3].

Today, Docker containers (or technologies built on Docker, like Kubernetes[3]) form the basis of many serverless platforms. This is because it is impractical to provide low-cost scalability using traditional VMs. Containers are a much more cost effective means for achieving scalability because they share a kernel with their host, thus making them orders of magnitude smaller in terms of disk space and hardware resource requirements (e.g., CPU, memory) than a typical VM [4].

[2] https://www.docker.com/
[3] https://kubernetes.io/

1.2.2 A Brief History of Internet Architectures

According to the *State of the Edge,* the history of the internet can be divided into three phases [5]:

1. Centralization (internet of the past).
2. Regionalization (internet of today).
3. Localization (internet of the future).

During the *centralization phase* of the internet, traffic intended for remote hosts would traverse several networks before reaching its destination. This was extremely inefficient due to the distances and underdeveloped distributed architectures. Nevertheless, this network architecture was sufficient to support most use cases for early internet communications. Moreover, fewer users during the centralization phase meant bandwidth was enough for services to be delivered over roundabout routes.

Ubiquitous deployment of fast transmission media and cheaper compute spurred the *regionalization phase* of the internet [5]. The development of Content Delivery Networks (CDNs) – data centers whose purpose is to cache and deliver content to nearby users – provided much better performance than delivering services directly from the origin server, which could be very far from the end user or application [6]. The regionalization phase also ushered in 3G and 4G cellular networks, which have succeeded in reducing latencies to the extent that users are now able to stream bandwidth-intensive content, like videos, on their mobile devices [5].

In our current phase of regionalization, CDNs have advanced enough that it is unlikely that a user request for a global service would ever have to leave that user's geographic region. However, CDNs will not be sufficient to support the increased pressure on bandwidth of the many billions of future internet users [5]. Technologies that will be in high demand in the coming decades – autonomous vehicles, virtual reality (VR), augmented reality (AR), and drone delivery services – will require much lower latencies for performance and safety that CDNs are ill-equipped to achieve through the caching strategy. This will ultimately drive the transition to the next phase of the internet: *localization* [5].

Edge computing is the term that has been adopted to describe how *localization* of the internet will take place [5]. Edge computing just means moving compute intensive tasks and data storage closer to the devices and users that will create and consume the data. The hope is that edge computing will encourage the use of more intelligent and customized edge delivery services. This will include placing more compute and storage within Local Area Networks (LANs) to achieve dramatically reduced latencies. This supplemental compute could be as tiny as a microprocessor or large as a 50,000 square foot data center. There are even shipping container sized data centers (typically referred to as micro data centers) that are being developed for placement near cell towers and LANs with high needs for supplemental compute [7].

Despite the massive potential benefits of edge computing, large-scale rollout of modernized edge data centers and custom edge servers will take years. There are significant barriers to moving compute and storage to the network edge. Real estate holders, energy suppliers, ISPs, big-tech companies (including CSPs), software engineers and consumers all have a stake in this emerging technology. The necessary investment for building new edge compute infrastructure will not be

made until there is consensus on use cases and a clear path to profitability. This "rearchitecting of the internet" will require a massive amount of financial investment, creative engineering, as well as the coordination amongst these stakeholders who are understandably fearful of investing in edge technologies too soon when neither the consumer demand nor the necessary supportive infrastructure is in place to buttress a lucrative edge deployment model [5] [7] – sometimes referred to as the "first mover *dis*-advantage" [8].

1.2.3 A Brief History of Cloud Services

The explosion of cloud services can be traced back to the early 2000s, when engineers at Amazon became frustrated with having to repeat the same tasks to provision infrastructure every time they wanted to build a new service [9]. They solved this internal problem by developing "a set of well-documented APIs" for provisioning and managing infrastructure [10]. The real innovation, however, was not in solving this problem for themselves, but in offering their APIs *as a service* at low rates to customers who were also facing the same engineering challenges. This idea has come to be known as Infrastructure as a Service (IaaS). Incredibly, what started as a need to solve an internal problem is now a multi-billion-dollar industry that has ushered in a new revolutionary model for software engineering: cloud computing [11].

Amazon Web Services' (AWS) first offering (launched in 2006) was the *Elastic Compute Cloud (EC2)*. This IaaS offering was the innovation that laid the foundation for all subsequent cloud services [10]. IaaS providers take a similar approach to internet services as do real estate developers to housing: they pay the costs of building and maintaining the infrastructure and profit by renting out that infrastructure at a lower price than a "build-it-on-your-own" approach.

AWS introduced the idea of serverless computing in 2014 with the launch of AWS Lambda, giving them a two-year head start on their competitors. Serverless is quite a profound term in that it captures the degree to which software has advanced in recent decades to make virtualization of compute so efficient that software engineers no longer have to concern themselves with physical compute or even virtualization. Six years after AWS launched Lambda, the serverless sector is now estimated to be worth billions of dollars [12]. Today, AWS is worth $500 billion, and nearly every large tech company has followed their lead in either offering or consuming cloud compute [13].

The growth of serverless is going to have a significant impact on everything from the developer experience to cyber security to the requirements for training a new generation of software engineers, who will be increasingly unconcerned with physical hardware, operating systems and virtualization technology. Nevertheless, most software engineering problems will not be solved with serverless. Currently, serverless has a limited set of use cases and there will always be tradeoffs when a team chooses to switch to serverless. The biggest danger of serverless is the complacency (especially with respect to cyber security) that can take root when application developers wrongly assume their CSP has a robust enough set of security measures in place to make up for lax coding and account management habits. Moreover, migrating an application to a serverless platform is complex task. CSPs must be vetted, documentation must be read, security must be built in to the design of a serverless application from the start, and thorough research must be conducted to ensure transitioning to serverless will solve existing engineering problems, rather than add its own set of new challenges.

2 Overview of Edge and Serverless Computing

Edge and serverless computing are emerging technologies driven by the centralized power of cloud compute and the vast scale it provides. Edge and serverless computing each address different aspects of the centralized cloud paradigm – with serverless increasing scale and edge increasing reach and performance. They also provide more flexibility for application developers to innovate and distribute their services. The following section presents definitions for both edge and serverless computing, as well as related technologies that align with the scope of research in this paper.

2.1 Edge Computing

Edge computing refers to a computing architecture that brings compute that would traditionally be done in the cloud closer to the end user. Operating on data at the location of collection and/or consumption helps improve user experience by decreasing latencies, lowering bandwidth requirements, and reducing pressure on backhaul networks as well as core severs. Some examples of edge computing include:

- Running analytics or performing machine learning inference on locally sensed data on edge devices when cloud connectivity is not present.
- Processing customer data on a server deployed at the edge of an Internet Service Provider's (ISP) network (such as in a network switching facility or near a cellular base station) rather than sending the data thousands of miles away to a hyperscale data center in the cloud for processing to occur there.
- Scaling a web service that would typically be hosted in a single region of a CSP's network (e.g., United States east coast) across the all the regions in the CSP's network to ensure the service is redundant and can respond faster to user requests.

Edge computing is still an emerging technology and there is not a single, universally agreed-upon definition of "the edge." Different interpretations of the edge describe the degree to which infrastructure is shifted from the network core to the network edge. The following sections define some of the more popular interpretations of the edge.

2.1.1 Cloud Edge

The cloud edge is a concept defined differently depending on the stakeholder. This paper will use the most literal definition of the cloud edge, which is the edge of a CSP's network, whereon sits edge data centers owned and operated by the CSP's themselves.

As of today, the data centers that CSPs have branded their "edge data centers" are actually just points of presence that make up the CSPs' CDNs [1] [14] [15]. Nevertheless, CSPs like AWS give customers the ability to deploy applications to these edge data centers to increase redundancy and proximity to their user bases. Our definition of the cloud edge emphasizes a *physical* extension of the cloud by placing bare metal servers closer to end users in globally distributed edge data centers.

There is also a concept of a *virtual* extension of the cloud using edge-to-cloud integration platforms that can be deployed directly to Local Area Networks. AWS's IoT Greengrass[4] is an example of this type of software. Using this definition of a *virtual* cloud edge, one could argue the cloud can extend all the way into a customer's LAN. While today this seems like a far-reaching idea, this is what a future internet could look like in less than a decade – a distributed cloud, rather than a centralized or regionalized one.

2.1.2 Fog Computing

Fog computing is a paradigm for internet architectures that integrates services at network edges with the cloud which acts as the "single source of truth" [16]. Fog nodes are a special type of edge node that have "awareness" of a cloud (usually through vendor-specific software installations) and can thus facilitate secure communication between the edge and the cloud. Fog nodes can also act more like "traditional" edge nodes by running processing and analytics on data sent from other edge nodes.

2.1.3 Cloudlet

A cloudlet is a small-scale cloud data center located closer to end users than a hyperscale cloud data center [17]. A cloudlet can be a single server, or several server racks housed in a shipping container-like enclosure. It can also be a data center on the order of 10,000 square feet – still small enough to be strategically placed near a major population center.

Cloudlets are still very much in development and few have been deployed. It is unclear whether ISPs or CSPs will take the lead on deploying and maintaining these miniaturized data centers. ISPs are in the best position to deploy this technology given they have direct access to the infrastructure on which cloudlets will be deployed. A more likely alternative is that deployment of cloudlets will occur through a partnership between ISPs and CSPs [18] [19]. Through these partnerships, ISPs will deploy the infrastructure near internet access points at their network edges and CSPs will rent this infrastructure to bring their services closer to customers [5] [20].

Open source tools like OpenStack[5] and Kubernetes that automate the provisioning and management of data centers and/or compute clusters will make utilization of edge compute easier for smaller players to introduce competitive services. These tools might also open the door for ISPs to acquire a significant amount of the edge computing market share. ISPs are already geographically present at the edge, and with interoperable open-source solutions for edge services, the build out of edge compute may very well be slanted in favor of ISPs becoming major players in the application delivery arena.

2.1.4 Multi-Access Edge Computing (MEC)

Multi-Access Edge Computing (MEC) is a European Telecommunications Standards Institute (ETSI) defined network architecture for how edge computing will be supported in cellular networks. The ETSI has defined standards for deployment of MEC in both 4G and 5G cellular

[4] https://aws.amazon.com/greengrass/
[5] https://www.openstack.org/

networks, though these specifications have yet to be finalized [21]. Moreover, there are very few examples of MEC deployments in the 4G cellular networks that exist today and the rollout of 5G cellular technologies, including MEC for 5G, is only just beginning.

The original model for an edge-oriented cellular network architecture was called Mobile Edge Computing (also MEC), to emphasize this technology was designed specifically for cellular networks and would thus enable computing at the *mobile edge*.

Mobile Edge Computing was eventually rebranded as Multi-Access Edge Computing to highlight that edge computing in cellular networks will be through shared transmission media and therefore reachable through *multiple access* points [22] [23]. MEC technology is intended to be deployed at cellular base stations, using local breakout to redirect some user traffic away from the core to an application deployed to a MEC host [24].

2.2 Serverless Computing

Serverless computing aims to leverage modern virtualization technology to increase efficiency of application development by abstracting away the entire compute stack (i.e. virtualization software, the host operating system, and hardware) that sits underneath a typical web application. The key benefit of serverless is in the automation of the highly repetitive tasks of managing production infrastructure so developers and organizations can focus on the activities that ultimately drive revenue.

The migration to serverless brings with it a new set of challenges that include, but are not limited to:

- Adjusting to a new application development paradigm.
- Adjusting to new application development and deployment tools.
- Automation and orchestration technologies.
- Understanding a customer's responsibility for securing their serverless application.
- Determining when serverless should be used and when using it would be counterproductive.

This paper discusses the responsibilities of serverless application developers at length in later sections.

Serverless platforms are good for hosting *microservices* [25]. Microservices are components of applications that can be encapsulated into stateless chunks of logic (sometimes referred to as "functions") that run for anywhere between several hundred milliseconds to 10 minutes and require cost-efficient scaling during periods of high traffic.

Commercial serverless platforms have robust support for public-facing endpoints to handle user requests as well as requests initiated by service instances in the cloud or even edge devices. In the latter case, the function usually reads from an event queue that is populated by the connected service [26].

2.2.1 Function as a Service

Cloud Service Providers often have broad interpretations of what serverless means. For example, every one of the leading CSPs – AWS, Microsoft Azure and Google Cloud – describe their flagship database platforms (DynamoDB[6], Cosmos DB[7] and Firebase[8], respectively) as being "serverless"[9]. While this is *technically* true—the compute underlying these database services is fully provisioned and managed by the vendor and increasing of database capacity via instantiation of new "instances" occurs automatically, these platforms do not support development and execution of application logic. In other words, these "serverless" databases scale *data* rather than *custom application logic* [27] [28]. We define serverless as:

> *A platform for building and hosting cloud-native applications wherein the **unit of scalability is a self-contained module or "function"** that is comprised of the customer's own application logic and sits on top of a fully managed and scalable infrastructure.*

Serverless functions are supported by a serverless stack. Though vendors choose different ways of implementing their serverless stacks, it typically means the vendor manages all but the application layer of this stack [29] [30]:

Table 2-1. Conceptual Serverless Stack

	Customer managed application / service	
Provider Managed	The container that encapsulates a function (thereby isolating it from the host machine)	Network infrastructure
	Container orchestration (namely auto-scaling of the function)	
	A virtual machine (referred to as an "instance")	
	A virtual machine monitor (also known as a hypervisor)	
	The host operating system	
	The computer hardware	

Note that in addition to this stack, the vendor manages all networking infrastructure within the cloud to support the path that an event will take from a its source (the trigger) to a function that the trigger has invoked [31].

---

[6] https://aws.amazon.com/dynamodb/
[7] https://azure.microsoft.com/en-us/services/cosmos-db/
[8] https://firebase.google.com/
[9] An important caveat to note here is that Google doesn't explicitly describe Firebase as "serverless," but does emphasize the fact Firebase allows developers to "build apps fast, without managing infrastructure [107]."

Serverless functions are dormant (and cost-free) until they are invoked, at which point billing begins. Once they terminate, they return to a state of zero-cost dormancy. This *pay-as-you-go* pricing model is a key feature which makes serverless a cost-effective approach for microservices. Running microservices in a serverless environment allows developers to focus on coding while the serverless provider takes care of the cost-intensive, non-value-add, but necessary tasks related to hosting like physical security, server provisioning and management, security patching, networking, runtime management, load balancing, concurrency, and scaling [32].

This cloud computing model of supporting deployment of self-contained code onto pre-provisioned, highly available, and scalable infrastructure is known as Function as a Service (FaaS).

2.2.2 Serverless Architectures

A serverless architecture describes the physical infrastructure, networking and software required to create a serverless platform on which to run serverless applications. Contrary to the term "serverless," physical assets are required to run serverless applications and the way in which those assets are deployed, integrated and coordinated effects the delivery and usability of serverless.

A serverless architecture has a different meaning to a CSP than to a customer. While a CSP is responsible for architecting a serverless *platform*, the cloud customer is responsible for building a *serverless application* that is deployed to their serverless platform of choice. An important thing to note about serverless is that even a well-designed serverless platform does not guarantee that any serverless application will perform well on it, and vice versa. Thus, when migrating to serverless, customers must be careful about choosing a service provider with a good track record as well as avoiding poor coding practices when building their applications.

2.2.2.1 Serverless Platforms

In building a platform that supports serverless computing, the CSP has three jobs:

1. **Isolating customer code** on machines that multiple customers may share (*multitenancy*).
2. Guaranteeing **availability**, this includes dynamic **scaling** of customer resources.
3. **Securing the infrastructure** beneath the application layer.

Isolation of customer code is achieved through containerization. While not the only approach, every serverless platform that has seen success is based on containerization technology – or the idea behind containers – which provides a lighter weight alternative to VMs for isolating an application from other processes on the host system. The use of containers comes with tradeoffs, the most important of which is processes running within a container share a kernel with the host and are thus inherently less "isolated" than processes running inside a VM, making it is easier for processes to break out of their containers [33]. Most open source serverless platforms are based on either Docker Swarm[10] (Docker's container orchestration solution) or Kubernetes (Google's

[10] https://docs.docker.com/engine/swarm/

open source container orchestration platform which uses Docker as the underlying container technology) [4] [31] [34].

AWS Lambda[11] isolates customer code by placing each function in a MicroVM[12] which runs on top of a lightweight open source hypervisor called Firecracker [29]. Azure Functions'[13] hosting solution for isolating and scaling serverless code, the Service Fabric Mesh, is closed source, although their documentation says the platform "supports any programming language or framework that can run in a container" [35]. Google does not explicitly say which technology their closed source FaaS platform, Cloud Functions[14], is built on; however, the developer guide implies their execution environments are containers [36]. Moreover, Google's open source framework for serverless computing, Knative[15], is based on Kubernetes, so it is reasonable to believe Google leverages a similar technology to isolate the code deployed to their commercial FaaS platform [34].

Whether a customer's serverless code is isolated within a MicroVM or a container, the CSP must provide a mechanism for **automatic scaling of function instances in order to uphold their guarantees of high availability**. Container orchestrators like Docker Swarm and Kubernetes have many functions, though their primary role is to manage and scale instances of a container in response to fluctuations in demand. Serverless platforms employ container orchestration (or similar technology) to provide *scalability*, and thus, *high availability*. AWS, for example, uses an open source virtual machine monitor to scale execution environments [29]. Azure and Google don't explicitly say how function instances are scaled on their respective commercial FaaS platforms. Finally, most open source serverless platforms run functions in containers and rely on Docker Swarm and/or Kubernetes for scalability.

It's important to note that **securing a "serverless application" is not fully the responsibility of a serverless vendor.** Vendors, even in the context of serverless, adopt a *"shared responsibility model"* when it comes to securing customer code [37]. One of the vendor's primary responsibilities is to isolate their customers' code that is running in multi-tenanted environments. Other tasks that fall to the CSPs include physical security, network security, redundancy within a region, provisioning and maintenance of compute hardware and operating systems [38]. The vendor also exposes access control and security mechanisms to the customer, but it is up to the customer to properly implement access control for their environment and applications.

2.2.2.2 Serverless Applications

It's easy to be fooled by how easy it is to deploy a serverless function from a platform's web console, though building serverless applications requires skills that go beyond just knowing how to code.

Writing function code will, in most cases, be the easiest task in building a serverless app. Integrating a function into a complex system, choosing a runtime, ensuring the proper

---

[11] https://aws.amazon.com/lambda/
[12] https://firecracker-microvm.github.io/
[13] https://azure.microsoft.com/en-us/services/functions/
[14] https://cloud.google.com/functions
[15] https://cloud.google.com/knative/

performance optimizations are in place so it can adequately cope with demand, setting up a system to monitor the app without having any direct access to log files, debugging errors in an environment that cannot be replicated locally, and most importantly, ensuring the application is secure are all challenging architectural problems that a serverless developer will have to address.

2.3 Bringing Serverless to the Edge

Because serverless is a cloud-native infrastructure solution, it is not immediately intuitive why running serverless applications at the edge could be beneficial. In fact, to some, bringing serverless to the edge may seem counter-productive where edge computing simply cannot provide the scalability on par with the cloud. This is a problem that ISPs, CSPs, and application developers are still grappling with.

As of today, the infrastructure necessary to facilitate deployment of "edge-native" services does not exist at cloud-scale. CDNs provide some benefit but are constrained in how close they can get to end users and are still somewhat limited in *processing and analyzing* data, as opposed to just caching content. Nevertheless, pondering future uses cases has identified some unique opportunities for serverless functions at the edge; a main driver being "**event-driven processing"** [39].

Serverless is uniquely helpful for event-driven processing at the edge because it is inherently event-driven, and thus, can be very resource efficient. Moreover, the potential for processing data at the location of collection increases with the exponential growth in IoT and the data IoT devices capture.

There are many parties interested in capitalizing on the vast amounts of data being generated at the edge. ISPs and CSPs are exploring three avenues for deploying serverless functions at the edge:

1. Deployment on the *cloud edge*. AWS offers this with Lambda@Edge. Note this is not quite the distributed local edge; rather, the edge of their existing CDNs [1].
2. Deployment at ISP edges (i.e. ISP access points). This is an option that is still very much in initial development, although as of just last year, ISPs and CSPs have started to form partnerships to explore this approach [19] [18]. One likely outcome of these partnerships is that ISPs will begin deploying supplemental compute within or near these access points, forming "micro data centers," which CSPs could then rent to deploy their cloud-native services much closer to end users than they could ever get with their CDNs [40].
3. Deployment at the *user edge*. As compute continues to get cheaper, user devices get ever more powerful. An increasingly vast ecosystem of tools has been developed to allow data processing and analytics (e.g., lightweight machine learning models) to run on low-compute edge devices in near real time. While promising for innovation, this prospect comes with security risks, like shifting the responsibilities of provisioning and maintaining the edge compute and data security requirements from CSPs to users. Additionally, the question of what amount and kinds of data should be sent to the cloud and what should be kept at the edge remains unresolved.

The ETSI also recognizes the relationship between edge and serverless, saying “MEC will enable serverless computing for a key 5G use case […] massive IoT devices, by hosting Function as a Service (FaaS) in the edge and supporting integration with the cloud service provider” [41].

This page intentionally left blank.

3 Vendor Analysis

This section compares various platforms for edge and serverless computing (both commercial and open source) based on a detailed set of criteria laid out in the common evaluation criteria below.

The purpose of this section is not so much to rank platforms as it is to unpack the fine print in vendor documentation to provide practical guides on how these platforms can be used. Additionally, instances where vendor claims in overview and marketing materials do not match up with specifications in the documents are brought to light.

3.1 Evaluation Criteria

In order to keep comparisons as uniform as possible, common evaluation criteria is used for each of the major vendor offerings we evaluate. The evaluation criteria are shown below in Table 3-1 along with definitions for each of the evaluation criteria.

Table 3-1. Evaluation Criteria

The vendor, provider.	**The product name.**
URL or reference document.	
Category	Edge, serverless, or hybrid.
Maturity	Discussion of how long the platform (or service) has existed as well as the levels of adoption and use within the market.
Cost	A short description of the costs or cost model.
Security	
Confidentiality *Data must only be accessible to those with the proper permissions to do so.*	List of vendor and customer responsibilities and methods for protecting confidentiality. **Tasks related to confidentiality protection:** • Authentication and access authorization • Identity management • Encryption of data at rest • Physical security • Network security • Hardware, OS, and software updates • Writing secure code (i.e. defensive coding) • Logging and monitoring

The vendor, provider.	The product name.
Integrity *Data must remain authentic and consistent, both in transit and in storage.*	List of vendor and customer responsibilities and methods for maintaining integrity. **Tasks related to integrity protection:** • Encryption of data in transit • Access controls • Isolation of customer code on shared servers (i.e. responsible management of multitenancy) • Statelessness of code (whenever possible) • Proper tear down and clean-up of execution environments • Network security (both within and outside of the cloud) • Logging and monitoring
Availability *Data must be reliably and readily accessible. Data must remain retrievable in the event of an outage as well as through periods of bursts in traffic.*	List of vendor and customer responsibilities and methods for ensuring availability. **Tasks related to availability protection:** • Redundancy • Scalability • Load balancing • Fault tolerance (via redundancy, exception handling, and mechanisms for recovery) • Logging and monitoring • Test-driven development • Performance optimization • When applicable, idempotency of code
Performance	
Supported runtimes	List of supported language runtimes. For example, Python, Node.js, Java, Go, etc.
Concurrency / Scalability	The ability to scale services during periods of increased traffic by replicating execution environments and running them in parallel. List of a platform's available mechanisms, limitations, and performance optimizations for scaling a service.
Notable limits	List of vendor-specific outliers in categories of service limitations. Limits that are common across vendors are not noted here.

The vendor, provider.	The product name.
Maintainability	
Essential developer tools	List of vendor-provided tools necessary for application development and deployment of code developed locally to this platform, such as: • Command Line Interfaces (CLIs) • Continuous Integration / Continuous Deployment (CI/CD) tools • Integrated Development Environments (IDEs) • Testing and debugging tools
Documentation	Review of the state of the documentation of the platform, including level of thoroughness, clarity, and navigability of developer documentation, as well as the state of supplementary documentation such as whitepapers, tech talks, and "quick start" guides.
Logging and monitoring	List of platform-native tools for viewing and analyzing logs as well as any third-party plugins for logging and monitoring.
Prerequisite skills	List of prerequisite skills that developers using this platform will benefit from having (for example, experience levels in software development, cloud computing, networking, cyber security, etc.).
Extensibility	
Intra-cloud integrations	If the platform is cloud native, list of other vendor resources that can be linked to applications deployed to the platform. For example, list of vendor resources that can act as triggers for a serverless function.
Edge integrations	Deployment options for the edge.

3.2 Edge Computing

Edge computing is infrastructure intensive and thus cost intensive to deploy with a geographical footprint diverse enough to entice customers with service offerings. As a result, investment in infrastructure-backed edge computing services has been low by most major CSPs. Instead, they are re-branding the edges of their existing networks, including their CDNs, as the "cloud edge". Deploying services in this "cloud edge" allows customers to benefit from lower latencies while keeping their applications very much within the cloud – still somewhat distant from end users and devices).

Nevertheless, even as edge infrastructure lags, there is a growing market for services that integrate customer owned edge compute with the cloud. The following sections examine the major players and some emerging alternatives in this space.

3.2.1 AWS IoT Greengrass

IoT Greengrass is a suite of software-based tools for extending AWS to the edge, namely LANs. IoT Greengrass consists of a Greengrass Core and Greengrass devices, which, when connected, create a Greengrass Group. The Greengrass Core is a software package that can be deployed on most devices running Linux and acts as a gateway between connected devices at the edge and AWS. It can also route messages between connected devices and provide security features such as device authentication, in-transit encryption of communications both between edge devices and to the cloud, and the syncing of local device logs with CloudWatch. Greengrass devices connect to the core via the AWS IoT Device SDK or, if they are running FreeRTOS, they can connect via the AWS IoT Greengrass discovery library.

The Greengrass Core is fully equipped to host Lambda functions, making it ideal for event-driven data processing and even running analytics like machine learning models at the edge. While AWS emphasizes Greengrass' ability to run AWS Lambdas and perform machine learning inference at the edge, it's clear from the documentation that Greengrass' primary use case is in securing an IoT infrastructure.

The proliferation of malicious IoT botnets over the past 5 years has made us aware of the lack of security controls in many IoT devices [42]. IoT Greengrass attempts to mitigate this persistent threat by giving customers a way to centralize control over their IoT infrastructure. Nevertheless, this tool is by no means a silver bullet for solving the IoT security problem. One important limitation to note is that Greengrass deployments at the edge will rarely be able to "operate offline" [43].

Table 3-2. AWS IoT Greengrass: Overview

AWS	IoT Greengrass
https://aws.amazon.com/greengrass/	
Category	Edge *and* serverless
Maturity	Launch date: June 2017 Updates occurred on a quarterly basis for first year after initial launch. Now they occur on a monthly basis. Based on documentation and developer reviews of Greengrass, AWS has maintained a non-committal attitude towards edge computing thus far. There is not enough demand for robust cloud-edge integrations to incentivize big CSPs to heavily invest in their own infrastructure for edge computing. In the short term (~1 year), AWS will be focused on cloud computing. Nevertheless, AWS' growing suite of services for edge computing, including Greengrass, is a sign that AWS is taking seriously this new paradigm delivery of services.
Cost	**Model** Pay-as-you-go (sort of):

AWS	IoT Greengrass
	In practice this is more like the rent-a-server model used by EC2. Rather than pay for uptime of code (as seen in AWS Lambdas' pay-as-you-go model), customers pay for the uptime of a Greengrass Core device on a per-month basis. A core device is considered "up" or "active" as soon as it makes a request over the internet to an AWS API, at which point the customer is charged for one month of uptime. The customer will not be charged for the next 30 days, as they have already paid for the "month." After that period is over, the customer won't be charged again until the core makes another connection to the cloud. Lump sum: Commit to twelve months of uptime for 22% per month discount. With this option, the customer will pay for the entire twelve months of uptime (even if all twelve months are not used). **Charge per device** $0.16 per month (using pay-as-you-go model) $1.4976 per year (using annual commitment/lump sum model) **Networking (data transfer)** Same as EC2 pricing. Any data transferred between AWS and the Greengrass Core is considered to be passing "over the internet," meaning the transfers are NOT billed at the discounted within-AWS price. Rather, they are billed at the standard over-the-internet price. See Lambda evaluation for more details on pricing of data transfers. **Caveats** As with Lambda, if a Greengrass system communicates with another AWS service (like PUT-ting a document into a Simple Storage Service (S3) bucket, for example), the customer will be billed for the use of that service as well. Customers must also consider the probability that Total Cost of Ownership will increase due to the fact that they must stand up and maintain their own edge computing infrastructure in order to run Greengrass. While AWS does provide an extensive catalogue of Greengrass-compatible devices, Greengrass is only software. The customer is therefore responsible for the costs of purchasing, provisioning, and maintaining the underlying hardware.

3.2.1.1 Security

Table 3-3. AWS IoT Greengrass: Security

AWS	IoT Greengrass
Confidentiality	**AWS Responsibilities** Securing "the cloud": • Physical security • Network security (cloud network) • Hardware and OS updates (cloud servers) • Encryption of data at rest in the cloud Securing "the edge": • Encryption of "secrets" stored on local devices
	Customer Responsibilities Securing "the edge": • Physical security • Network security (LAN) • Hardware and OS updates (edge devices) • Protection of data at rest on the Greengrass Core and connected edge devices—via Unix file permissions and full disk encryption) Authentication/authorization: • Role-based Access Control (RBAC) ("following the principle of least privilege") • Ensuring mutual authentication (devices to the cloud, devices to the Greengrass Core and the Greengrass Core to cloud) is done according to AWS' recommended best practices • Securing authentication and encryption keys that are stored on local devices Additional responsibilities, including, but not limited to: • Opting for hardware based private key storage, overriding default encryption keys (provided by AWS) with custom keys (and if using own keys, rotating the keys on a regular basis) • Managing local secrets • Writing secure code (like not hardcoding credentials) • Logging and monitoring (ensuring that no sensitive data is logged)

AWS	IoT Greengrass
	• Management of Message Queuing Telemetry Transport (MQTT) (using “Targeted Subscriptions” model)

Integrity	**AWS’s Responsibilities**
	• Encryption of data in transit:
	  ○ Over the internet between the Greengrass Core and cloud
	  ○ Over the Local Area Network between connected devices and the Greengrass Core—via Transport Layer Security (TLS) with mutual authentication
	• Providing the MQTT server and client certificates for mutual authentication in order to establish encrypted connections over the local network
	• Over-the-air (OTA) updates of the Greengrass software

	Customer’s Responsibilities • Management of subscription table for MQTT messaging • Ensuring accurate time keeping on connected devices (so that expiry of certificates will be detected and timestamps in local device logs are accurate) • Securing authentication and encryption keys that are stored on local devices. For select core devices, customers can opt to implement hardware-based encryption of private keys for better security • Creating, validating (for every connection), and updating local device certificates
Availability	**AWS Responsibilities** • In practice, AWS has no responsibility in ensuring availability of IoT Greengrass systems ○ However, AWS is still responsible for ensuring the availability of the IoT Greengrass components that reside in the cloud, such as the AWS IoT Core, which

AWS	IoT Greengrass
	contains endpoints on which Greengrass devices rely for discovery of the Greengrass Core and devices as well as rotation of certificates. In this vein, AWS' main responsibilities are in ensuring IoT Core endpoints are redundant and scalable enough to handle spikes in connections to IoT Core instances from devices in their customers' LANs

	Customer Responsibilities
	• Proper exception handling and code-based performance optimizations in any AWS Lambda functions deployed to the Greengrass Core
	• Logging and monitoring. Options include:
	  ○ Store Greengrass Core device logs locally in the Greengrass Core device's file systems
	  ○ Send logs to the cloud (via AWS CloudWatch)
	  ○ Both
	• Load testing the Greengrass Core. This is especially important given the Greengrass Core device could very well be limited in terms of compute power
	• Fault tolerance for periods of no or intermittent cloud connectivity. For example, handing certificate expiry when the cloud is not available to issue a new certificate and maintaining logging capabilities without access to CloudWatch for log syncing

3.2.1.2 Performance

Table 3-4. AWS IoT Greengrass: Performance

AWS	IoT Greengrass
Supported runtimes	Python, Node.js, Java 8, and C/C++
Concurrency / Scalability	In order to support deployment of Lambda functions to a Greengrass Core device, AWS provides a "containerized Lambda runtime environment" on which user-defined Lambda functions can be deployed. That said, this serverless platform that runs on the Greengrass Core does not provide out-of-the-box support for scaling of user-defined Lambda functions to concurrent instances on the Greengrass Core itself. In fact, there is even an option to run a Lambda function on a Greengrass Core device with no containerization

AWS	IoT Greengrass
	whatsoever, though we discourage this practice. Thus, it is hard to argue that Lambda functions running on Greengrass Core devices are truly “serverless” functions, given they can’t scale.
Notable limits	Some key functions of Greengrass cannot occur without internet connectivity (i.e. connection to the cloud): • Local devices connecting to the Greengrass Core device for the first time (in order to do this, local devices need to make themselves known to the cloud) • Syncing logs to the cloud via CloudWatch • Re-issuing of local device certificates for mutual authentication after certificate expiries

3.2.1.3 Maintainability

Table 3-5. AWS IoT Greengrass: Maintainability

AWS	IoT Greengrass
Essential developer tools	**CLI** AWS CLI[16] For accessing IoT devices, an ssh client like PuTTY may be required. **CI/CD** AWS CloudFormation[17] **IDE** VSCode[18], though terminal-based text editors like vim will suffice **Testing and debugging tools** AWS IoT Device Tester (IDT)[19] for end-to-end testing (i.e. testing Greengrass to cloud integration)
Documentation	The Greengrass documentation is jargon-heavy for most readers unfamiliar with AWS offerings. Documentation is also quite scattered because Greengrass relies on a complex software ecosystem that must all be simultaneously deployed and connected for Greengrass to function. AWS does provide a “getting started” tutorial with seven

[16] https://aws.amazon.com/cli/
[17] https://aws.amazon.com/cloudformation/
[18] https://code.visualstudio.com/
[19] https://aws.amazon.com/greengrass/device-tester/

AWS	IoT Greengrass
	clearly defined steps that is a good place to start[20]. AWS also provides an extensive set of recommendations for "security best practices,"[21] although it is critical that the entire developer guide for Greengrass security be read in order for customers to grasp the full scope of their responsibilities in securing their Greengrass systems.
Logging and monitoring	**CloudWatch** The Greengrass Core can send event logs to CloudWatch[22] (which are then stored in the cloud can be viewed using the CloudWatch service). The Greengrass Core can also log to its local file system. This is the default logging configuration; logging to CloudWatch must be manually enabled. **CloudTrail** CloudTrail[23] gives AWS customers the ability to log and monitor all requests made to the AWS IoT Greengrass API as well as actions of AWS users and roles with access to a Greengrass system. By default, only recent Greengrass API calls/user actions (i.e. events) can be viewed in the CloudTrail web console. A "trail" must be manually created and configured to route all Greengrass "events" to an S3 bucket.
Prerequisite skills	Knowledge of networking, including network security, is essential for IoT Greengrass developers. Software experience is less important given Greengrass is designed to only support deployment of lightweight applications, like Lambda functions. Prior experience with AWS is preferred, as the Greengrass system relies on the cloud for connecting new devices to the Greengrass Core as well as integration of a Greengrass deployment with other AWS resources. Additionally, an IAM administrator schooled in AWS best security practices is required.

[20] https://docs.aws.amazon.com/greengrass/latest/developerguide/gg-gs.html
[21] https://docs.aws.amazon.com/greengrass/latest/developerguide/security-best-practices.html
[22] https://aws.amazon.com/cloudwatch/
[23] https://aws.amazon.com/cloudtrail/

3.2.1.4 Extensibility

Table 3-6. AWS IoT Greengrass: Extensibility

AWS	IoT Greengrass
Intra-cloud integrations	**IoT Core** The Greengrass service running in the IoT Core[24] (a cloud-native service) is required to orchestrate encrypted connections within the Greengrass system as well as to and from the cloud. **AWS Lambda** The Lambda runtime is provided out-of-the-box as part of the Greengrass Core software package, thus allowing the deployment of Lambda functions on the Greengrass Core device; though Lambda functions must be created using the AWS Lambda service in the cloud before they can be re-deployed to a Greengrass Core. **CloudWatch** See description under "Logging and monitoring" (above) **CloudTrail** See description under "Logging and monitoring" (above) Several other AWS services can be linked to a Greengrass system using "connectors".[25]
Edge integrations	On-premises resources can also be linked to a Greengrass system using "connectors." For example, a connector to Twilio[26] can be used to send notifications to end-user devices in response to an event sensed by the Greengrass system.

3.2.2 Emerging Vendors

Established providers do not have many service offerings in edge computing as they do in serverless. This is mainly due to differences in how these technologies are delivered: edge requires physical infrastructure deployed at particular geographic locations, whereas serverless is software that can be deployed anywhere compute is present. The ability to create, package and offer serverless products is orders of magnitude less costly in terms of upfront investment, time-to-market and operating costs given CSPs existing infrastructures and business models.

Nevertheless, there are industry consortiums that have been formed to propel ideas and standards in edge computing. The Telecom Infra Project[27] was formed in 2016 as an engineering-focused

[24] https://aws.amazon.com/iot-core/
[25] https://docs.aws.amazon.com/greengrass/latest/developerguide/connectors.html
[26] https://www.twilio.com/
[27] https://telecominfraproject.com/

collaboration for designing global telecom network infrastructure. Members include network operators, suppliers, developers, integrators, and startups. They've created sub-working groups and one of which, the Edge Application Developer Project[28], focuses on lab and field testing for services/applications for MEC using open source architecture and software stacks [44].

The Open Edge Computing Initiative[29] is another industry group with the goal of driving business opportunities in edge computing. The Open Edge Computing Initiative focuses on the development and scaling of "edge computing platforms and services", providing live edge demonstrations in an edge computing test center and partnering with academia for additional research and development [45].

While the list of members of these and other industry groups is far too numerous to provide full details, there are some standout vendors providing edge computing services with some success.

3.2.2.1 Vapor IO

Vapor IO[30] announced a highly compact "hyper-collapsed" data center offering in 2015 [46]. Datacenter in-a-box services had been around for some time (e.g., VCE Vblock[31], FlexPod[32]), but Vapor IO has differentiated itself by providing a "Data Center Runtime Environment," as well as an interface to their platform to make deploying applications to their infrastructure much easier than deployment to more traditional "drop-in" compute and storage [46]. In early 2018, Vapor IO announced their Kinetic Edge[33] colocation services – micro data centers providing IaaS with full, half and quarter-rack sizing options, including power and bandwidth [47]. Vaper IO calls the their Kinetic Edge service "the unique infrastructure foundation for the third act of the internet" [48]; where "third act" is localization as defined in the *State of the Edge 2020* report [5].

In early 2020, Vapor IO accounced it will be partnering with Cloudflare to deliver Cloudflare cloud services on Vapor IO's Kinetic Edge platform in 36 U.S. cities, "enabling a new class of low-latency edge-to-core applications" [49].

3.2.2.2 MobiledgeX

In 2016, Deutsche Telekom funded an internal 18-month study on the value of edge computing to operators and developers [50]. At the conclusion, MobiledgeX[34] emerged as an independent company. MobiledgeX has large investment and asset backing and is working to create a market of edge infrastructure and services, all exposed in a cloud-like, easy to use interface. They are attempting to solve "first mover *dis*-advantage" by appealing to application developers, device makers, CSPs, CDNs and mobile operators – those on both the supply and demand side of an emerging edge market [8].

---

[28] https://telecominfraproject.com/ead/
[29] https://www.openedgecomputing.org/
[30] https://www.vapor.io/
[31] https://en.wikipedia.org/wiki/VCE_(company)#Vblock
[32] https://flexpod.com/
[33] https://www.vapor.io/kinetic-edge-colocation/
[34] https://mobiledgex.com/

MobiledgeX has packaged their edge offerings into "cloudlets" – container-based infrastructure that works with mobile operators' Network Function Virtualization (NFV) infrastructure [51]. In mid-2019, MobiledgeX partnered with Telus for a trial of multi-access edge computing in Canada [52]. The partnership is deploying "virtualized cloudlets in key locations near the edge of Telus' wireless and wired access networks" [52]. This supplemental compute is designed to address the distribution scale that is required for IoT, augmented reality and other services when 5G goes live [53] [52].

3.3 Serverless

It was easy for CSPs to transition from more traditional cloud computing models, like IaaS, to serverless given their existing infrastructures and business models already heavily relied on container-based technologies. The "big three" CSPs – AWS, Microsoft Azure and Google Cloud Platform – all have serverless offerings that integrate with their respective suites of cloud service offerings. While other CSPs (e.g., IBM Cloud, Rackspace) have similar offerings, they are not nearly as widely used, documented and tested as the big three's serverless platforms.

The following sections compare serverless offerings from the big three as well as provide some insight into open source options for building and deploying serverless applications.

3.3.1 AWS Lambda

Launched in 2014, AWS Lambda is AWS' platform for building and deploying highly available event driven serverless functions (Lambdas). AWS Lambda stands out for its detailed documentation, ultra-competitive pricing scheme, and extensive integration with other AWS services like S3, DynamoDB, CloudWatch, and IoT Greengrass. The AWS CLI is powerful but does not support local testing of functions, so using the Serverless Application Model (SAM) CLI[35] to develop Lambda functions is strongly encouraged.

Because of its popularity, a strong ecosystem of plugins and third-party extensions have emerged to enhance developer experience while using the Lambda service. Among them are:

- **The Serverless Framework**[36]: A framework for developing serverless applications on a variety of cloud platforms. It provides a CLI that is much simpler and easier to use than the even AWS' own CLI.
- **Claudia.js**[37]: An open source framework for writing Node.js functions for deployment on AWS Lambda.
- **SCAR Framework**[38]: An open source framework for running Docker containers as AWS Lambda functions, essentially Docker-as-a-Function.

[35] https://github.com/awslabs/aws-sam-cli
[36] https://serverless.com/
[37] https://claudiajs.com/
[38] https://scar.readthedocs.io/en/latest/intro.html

- **node-lambda**[39]: Another open source framework for deploying Node.js applications as Lambda functions.
- **Architect**[40]: A very simple open source framework for building applications with AWS Lambda. Architect is basically a wrapper for CloudFormation (AWS' CI/CD tool) and the AWS CLI. Architect provides nearly identical functionality to the Serverless Framework with a simple API and ample documentation.
- **Chalice**[41]: An open source framework for writing Lambda functions in Python.
- **Dapr**[42]: An open source framework for building serverless functions for deployment on any platform – including local deployment with support for Kubernetes integration.
- **Flogo**[43]: An open source framework for development of serverless applications written in Go for a variety of CSPs.
- **Sparta**[44]: An open source framework for writing serverless functions in Go. Sparta seems much simpler and more straightforward than Flogo but only supports deployment to AWS Lambda.

Table 3-7. AWS Lambda: Overview

AWS	Lambda
https://aws.amazon.com/lambda/	
Category	Serverless
Maturity	Launch date: November 2014 First to the market; beat the next-fastest competitor, Azure, by two years. Updates to the platform occur (roughly) on a monthly basis. A limited amount of research has shown that Lambda is the most popular FaaS platform among developers [54].
Cost	**Model** Pay-as-you-go **Free tier** • First 1M free requests each month • First 400,000 GB-seconds consumed each month

[39] https://github.com/motdotla/node-lambda
[40] https://arc.codes/
[41] https://chalice.readthedocs.io/en/latest/
[42] https://github.com/dapr/dapr
[43] https://tibcosoftware.github.io/flogo/introduction/
[44] https://gosparta.io/

AWS	Lambda

	• First 1GB of outgoing internet traffic each month
	Requests (flat rate)
	• $0.20 per 1M requests
	Resource consumption
	• $0.000016667 per GB-second (note this figure is for *1024 MB* of memory; this rate varies based on amount of memory allocated)
	Sub-second metering
	• Yes. But a Lambda function's runtime will always be *rounded up* to the nearest 100ms. This means a function that runs for 101ms will be billed for 200ms of runtime.
	Networking (data transfer)
	• Inbound data: Always free
	• Outbound data:
	  ○ Free for first 1 GB going out from Lambda to internet each month.
	  ○ After first GB out, always free for transferring data from Lambda instances to other AWS resources *within the same region*.
	  ○ After first GB out, customers are charged on per-GB basis at a rate that depends on total data being transferred (rates are always lower when data is travelling *within AWS*).
	Caveats
	• Some free tier offerings DO NOT apply to AWS GovCloud.
	• AWS GovCloud[45] follows the typical AWS pricing models, though GovCloud customers have the option to reserve instances with a lump-sum payment that will remain in service for the period that was paid for.
	• Data transfer in and out of Lambda functions in GovCloud are typically more costly.
	• If a Lambda function is linked to other AWS resources (like S3), then the customer will be charged for use of those resources as well.
	• Use of a Virtual Private Cloud (VPC)[46] and VPC peering incurs additional charges.

[45] https://aws.amazon.com/govcloud-us/
[46] https://aws.amazon.com/vpc/

3.3.1.1 Security

Table 3-8. AWS Lambda: Security

AWS	Lambda
Confidentiality	**AWS Responsibilities** • Physical security • Securing the AWS network • Hardware, OS, and some software updates • Encryption of data at rest • Virtualization and multitenancy
	Customer Responsibilities • Role-based Access Control (RBAC) via AWS' Identity Access Management (IAM) service • Authentication/authorization (via IAM) • Securing of authentication, encryption, and API keys (via AWS' Key Management Service (KMS)[47]) • Logging and monitoring • Writing secure code • Service-to-service authentication (via IAM) • Optional: Overriding default memory wiping and encryption settings
Integrity	**AWS Responsibilities** • Securing the AWS network • Encryption of data in transit (via TLS) • Ensuring statelessness of Lambda function execution environments • Isolation of customer code on shared servers • Teardown and clean-up of execution environments

[47] https://aws.amazon.com/kms/

AWS	Lambda
	Customer Responsibilities • Securing their LANs and VPCs • Encrypting data written to long-term storage (like S3) • Optional: Using stronger transport-layer encryption
Availability	**AWS Responsibilities** • Redundancy • Scaling • Load balancing • Fault tolerance • Upholding guarantees in AWS Lambda's Service Level Agreement[48]
	Customer Responsibilities • Logging/monitoring (both code and billing) • Exception handling in code • Test-driven development • Runtime choice • Optional: Performance optimizations (such as Provisioned Concurrency).

3.3.1.2 Performance

Table 3-9. AWS Lambda: Performance

AWS	Lambda
Supported runtimes	Node.js*, Python*, Ruby*, Java, Go, .NET * Available in web console
Concurrency / Scalability	**Default concurrency limit** • 1000 Lambda function instances (account wide, can be overridden upon request to the Support Center)

[48] https://aws.amazon.com/lambda/sla/

AWS	Lambda
	Burst concurrency limit • 500-3000 Lambda function instances (depends on the region) **Optional performance optimizations** • Reserved concurrency (increase concurrency limit on per-function basis up to the account wide concurrency limit - 100) • Provisioned concurrency (pre-instantiation of execution environments) • Autoscaling with provisioned concurrency (more cost effective as this means that provisioned concurrency will only be triggered by traffic spikes) **Automatic scaling?** Yes. Via instantiation of replicated execution environments.
Notable limits	Lambda functions timeout after 15 minutes. Web console has max deployment package size of 3 MB. Max 10 test events per function when testing from within the web console.

3.3.1.3 Maintainability

Table 3-10. AWS Lambda: Maintainability

AWS	Lambda
Essential developer tools	**CLI** AWS CLI (discouraged) AWS Serverless Application Model (SAM) CLI (recommended) The Serverless Framework CLI (third party; partially open source) **CI/CD** CodeDeploy[49] (comes with the SAM CLI) **IDE** Cloud9[50] (for hybrid development) VSCode with AWS Toolkit (for local development)

[49] https://aws.amazon.com/codedeploy/
[50] https://aws.amazon.com/cloud9/

AWS	Lambda
	Testing and debugging tools AWS Serverless Application Model (SAM) CLI Cloud9 There are a variety of AWS-native tools for logging and monitoring (listed below) as well as support for inserting correlation ids (this is the central function of X-Ray[51]) into client payloads so developers can track a request through the entire invoke path. Using the SAM CLI, developers can now test Lambda functions locally.
Documentation	Documentation for this service is robust and reasonably well-organized, though the overview of Lambda on the service's landing page is vague. The "getting started"[52] page is better, providing links to both an easy-to-follow "hello-world" Lambda tutorial and the developer guide. The "getting started" page also has a list of AWS-endorsed "serverless developer tools"[53]. There is also a "resources"[54] page that links to whitepapers, case studies, and tech talks, all of which offer valuable insights into use cases for Lambda, as well as best practices. The whitepaper on Lambda security[55] is a must-read for any Lambda developer.
Logging and monitoring	**CloudWatch** View performance metrics; embedded in Lambda web console **CloudTrail** View event logs for all AWS services used by an account **X-Ray** Trace invoke path (across AWS network) from the event trigger to the Lambda function **Config**[56] Monitor config changes to any AWS service, including Lambda
Prerequisite skills	A Lambda developer will benefit from some experience in software development (preferably with some web development experience). Building serverless applications is as much of an architecture challenge as it is a coding challenge, so proficiency in data structures, web APIs, web security, and RBAC are preferred. At least one team

[51] https://aws.amazon.com/xray/
[52] https://aws.amazon.com/lambda/getting-started/
[53] https://aws.amazon.com/serverless/developer-tools/
[54] https://aws.amazon.com/lambda/resources/
[55] https://d1.awsstatic.com/whitepapers/Overview-AWS-Lambda-Security.pdf
[56] https://aws.amazon.com/config/

AWS	Lambda
	member must have a background in networking. Also, it is critical that at least one team member (preferably an IAM administrator) has deep knowledge of AWS best security practices and strongly communicates and strictly enforces those practices within the team.

3.3.1.4 Extensibility

Table 3-11. AWS Lambda: Extensibility

AWS	Lambda
Intra-cloud integrations	**Full list of integrated services** Almost any other AWS service can be integrated with Lambda[57]. **Indirect vs Direct triggers** • AWS's DynamoDB is an example of an indirect trigger of Lambda, meaning it can write data to a queue from which Lambda functions can read. • S3, CloudFront, API Gateway, and AWS IoT are all direct triggers, meaning they can generate their own events that will be passed to associated Lambda functions upon invocation. **Integrations for security** Additionally, CloudWatch, CloudTrail, X-Ray, and Config are all AWS-native services that can be used to monitor performance and security of a particular Lambda function or AWS account with minimal initial set up required.
Edge integrations	**IoT Greengrass** Provides a Lambda runtime for deployment of non-scalable Lambda functions on low-compute IoT devices. **Lambda@Edge (CloudFront)** Platform to deploy scalable Lambda functions to AWS' Content Delivery Network (CDN).

[57] https://docs.aws.amazon.com/lambda/latest/dg/lambda-services.html

3.3.2 Microsoft Azure Functions

Microsoft Azure Functions is its platform to build and deploy event-driven serverless functions as "function apps" that can leverage other Azure-native services. Azure Functions stands out for its global reach – having the most PoPs of the "big three."

Azure Functions has a complicated pricing scheme tied to other Azure services. It appears to offer more flexibility than its peers with both Windows and Linux-based deployment options [55]. However, non-premium users are arbitrarily locked out of key developer tools like the portal editor and most Linux deployment methods [55]. This economy/premium tier paradigm for limiting developer experience has no equivalent in AWS.

Usability is somewhat degraded by lack of simplicity and "feature overload" – whereby a large set of options, some of which Azure itself advises against using, increases confusion and development time [55].

Table 3-12. Microsoft Azure Functions: Overview

Microsoft	Azure Functions
https://azure.microsoft.com/en-us/services/functions/	
Category	Serverless
Maturity	Launch date: 2016 Second major vendor to the market; 2 years after Lambda
Cost	**Model** • **Consumption** uses the "traditional" FaaS pay-as-you-go model. • **Premium** (basically Consumption with perks) is also pay-as-you-go, but allows users to unlock optimizations, like keeping instances "warm" for better performance. • **App Service** allows users to deploy long-running functions with no limit on maximum duration. **Free tier** • First 1M requests (i.e. executions) • First 400,000 GB-seconds consumed each month • First 5 GB of outgoing internet traffic each month **Requests (flat rate)**

Microsoft	Azure Functions

	• $0.20 per 1M executions (i.e. requests)[58]
	Resource consumption
	• $0.000016 for every GB-second
	Sub-second metering
	• No. Rounding occurs on per-second basis. This means a function that runs for 0.01 seconds will be billed as 1 second of execution time.
	Networking (data transfer)
	• Inbound data: Always free
	• Outbound data:
	  ○ Free for first the 5 GB every month.
	  ○ After the first 5 GB out, transfers are billed on a pay-per-use basis, with price per GB going down as total size of monthly transfers goes up.
	  ○ Even after the first 5 GB out, outbound data transfers to an instance *within* an Azure Availability Zone are always free.
	Caveats
	• Storing any persistent data will cost you. Azure uses a pay-as-you-go model for pricing storage, though the specifics of the model are complex as costs vary based on type of storage plan (standard vs. premium) as well as the level of guaranteed redundancy.
	• The integrated logging and monitoring tool, Application Insights[59], has a daily limit on the number of logs it will ingest for free. Logs generated after this limit has been reached will not appear in query results unless the customer opts to pay for increased logging.

[58] "Requests" and "executions" have slightly different meanings in the context of serverless but can be conflated for the purposes of describing the FaaS cost model as in generic serverless architectures. It is reasonable to assume there is a one-to-one mapping of requests to function executions.
[59] https://docs.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

3.3.2.1 Security

Table 3-13. Microsoft Azure Functions: Security

Microsoft	Azure Functions
Confidentiality	**Azure's Responsibilities** • Physical security • Securing the Azure network • Hardware updates, OS, and some software updates • Virtualization and multitenancy
	Customer's Responsibilities • Identity management (RBAC) via Azure's IAM service[60] • Authentication/authorization • Securing encryption and authentication keys (via Key Vault[61]) • Encryption of data at rest • Logging and monitoring • Writing secure code • Service-to-service authentication
Integrity	**Azure's Responsibilities** • Securing the Azure network • Isolation of execution environments • Proper teardown of execution environments
	Customer's Responsibilities • Writing stateless serverless functions • Securing their LANs and Azure Virtual Networks (VNets)[62] • Opting to use stronger transport layer encryption
Availability	**Azure's Responsibilities** • Redundancy • Scaling • Load balancing

[60] https://azure.microsoft.com/en-us/product-categories/identity/
[61] https://azure.microsoft.com/en-us/services/key-vault/
[62] https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-overview/

Microsoft	Azure Functions
	• Fault tolerance • Maintaining commitments in the Azure Functions Service Level Agreement[63]
	Customer's Responsibilities • Logging and monitoring • Exception handling • Testing edge cases • Runtime choice • Optional performance optimizations

3.3.2.2 Performance

Table 3-14. Microsoft Azure Functions: Performance

Microsoft	Azure Functions
Supported runtimes	C#, Node.js, F#, Java, PowerShell, Python, and Typescript Offers "Language Extensibility" feature that gives the customer the ability to allow a single event to trigger multiple "worker" nodes, each with different runtimes that nevertheless run the same function logic implemented in the corresponding runtime language[64]. *Mastering Azure Serverless Computing*[65] has a better overview of the Language Extensibility feature, as the Azure-provided documentation for this feature is unnecessarily complicated.
Concurrency / Scalability	**Consumption plan concurrency limit** • 200 host instances **Premium plan concurrency limit** • 100 host instances **App service plan concurrency limit**

[63] https://azure.microsoft.com/en-us/support/legal/sla/functions/v1_1/
[64] https://github.com/Azure/azure-functions-host/wiki/Language-Extensibility
[65] https://www.oreilly.com/library/view/mastering-azure-serverless/9781789951226/

Microsoft	Azure Functions
	• 10-20 host instances[66] **Scaling** • Consumption plan scaling: automatic. • Premium plan scaling: automatic with optional performance optimizations, such as "pre-warming" instances[67]. • App service plan scaling: manual. Although the customer can turn on the "autoscale" feature, which requires them to author "rules" dictating how many instances an app can scale up to. It's worth noting, however, that this level of customer intervention that is needed to "autoscale" in the app service plan seems to defeat the purpose of auto scaling and is not serverless in nature.
Notable limits	Logging limits in free tier of Insights (as noted above). The Service Level Agreement for Azure Functions is very similar to AWS Lambda's SLA, except for one critical component: Azure Functions customers pay for *75% of uptime costs* when monthly uptime percentage is anything lower than 99%. This is not the case for AWS, which guarantees *full reimbursement* when monthly uptime percentage is less than 95% due to a failure that AWS has deemed to be its own fault. AWS' policy is good for ensuring customers are protected in the event that an extenuating circumstance damages AWS' infrastructure enough to induce Denial of Service (DoS). Azure does not provide this level of protection.

3.3.2.3 Maintainability

Table 3-15. Microsoft Azure Functions: Maintainability

Microsoft	Azure Functions
Essential developer tools	**CLI** The Azure CLI[68] (required for using Azure Functions Core Tools)

[66] Each host instance can run 10 worker processes (the default is 1) and each worker process runs a single instance of a function, so the function-wise concurrency limits can be higher and more fluid than those noted above. For example, the Consumption plan allows for up to 200 concurrent host instances, though if host instances are configured to run 10 function instances, then the function-wise "concurrency limit" for the Consumption plan is closer to 2000 in practice.
[67] https://docs.microsoft.com/en-us/azure/azure-functions/functions-premium-plan#pre-warmed-instances
[68] https://docs.microsoft.com/en-us/cli/azure/

Microsoft	Azure Functions
	Azure Functions Core Tools[69] (comes with its own CLI) for which the Azure CLI is a dependency The Serverless Framework CLI **CI/CD** "Source control integration" (using git/Kudu)[70] Azure DevOps/Azure Pipelines[71] **IDE** VSCode with Azure extension Visual Studio[72] **Testing/debugging** Azure Functions Core Tools (comes with its own CLI) Azure Functions supports local development, testing, and debugging of functions with Azure Core Tools—a feature that greatly enhances developer experience. The ability to test locally (rather than having to re-deploy to the cloud after a small code change) is a huge time saver. The ability the develop and test locally also means developers can stick to their preferred IDE and debugging tools, rather than having to be constrained to cloud-based tools. Typically, experienced software developers have strong opinions about their debugging and code editing tools, so eliminating the need to debug with unfamiliar cloud-based tools that suffer from unavoidable latencies can be a game changer for many teams. Remote debugging of Azure Functions is supported by plugins for Visual Studio and VSCode. Jest[73] is the e2e testing framework used by the VSCode plugin and is useful when it works but nearly impossible to customize. It is recommended that developers stick to local testing using the Azure Core Tools. Azure Functions diagnostics[74] offers developers a way to "troubleshoot" errors from the Azure web portal with the help of a bot named Genie. Unfortunately, it was designed with too low of a common denominator in mind and thus is too sluggish to be of any use. A simple static webpage listing common issues that developers can search against would be more useful and accessible.

[69] https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local
[70] https://docs.microsoft.com/en-us/azure/azure-functions/functions-continuous-deployment
[71] https://docs.microsoft.com/en-us/azure/devops/pipelines/targets/azure-functions
[72] https://azure.microsoft.com/en-us/products/visual-studio/
[73] https://jestjs.io/
[74] https://docs.microsoft.com/en-us/azure/azure-functions/functions-diagnostics

Microsoft	Azure Functions
	Lastly, Azure Functions supports the use of correlation IDs for tracking the invoke path of a request from end to end, similar to AWS' X-Ray service.
Documentation	The Azure Functions developer guide is disorganized and difficult to grasp for developers who are inexperienced with Azure. It is recommended that developers skip the "Concepts" section in the documentation entirely and begin learning the platform through a "Quick Start"[75] or basic tutorial, like "Create a serverless web app"[76]. Developers should be required to read Azure's whitepaper on serverless security[77] before writing any code that is intended for production. Working through the tutorial on deploying a function app inside a VNet[78] will also be critical, as VNets offer essential security controls for an Azure customer's public endpoints.
Logging and monitoring	**Azure Application Insights** Collects and stores "log, performance, and error data" in a SQL-like database in the cloud. Using Application Insights, raw logs specific to a function app can be viewed in the from within the Azure Functions web interface and SQL-like (Kusto) queries can be executed against this log data. Results are returned in tables and charts. **Azure Monitor Logs**[79] Another logging and monitoring tool that allows customers to run queries over *all* their function app logs in one place.
Prerequisite skills	All developers will need to have a strong background in core concepts in software engineering, like data structures. Some web programming experience is helpful. At least one team member should have experience with securing cloud infrastructures and hands-on experience with Azure's IAM service is preferred. This person should be the designated IAM admin of the team's Azure account. It is preferred that all team members have networking experience, as this is critical for securing any cloud-based service.

[75] https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-first-azure-function-azure-cli
[76] https://docs.microsoft.com/en-us/learn/modules/automatic-update-of-a-webapp-using-azure-functions-and-signalr/
[77] https://azure.microsoft.com/mediahandler/files/resourcefiles/azure-functions-serverless-platform-security/Microsoft Serverless Platform.pdf
[78] https://docs.microsoft.com/en-us/azure/azure-functions/functions-create-vnet
[79] https://docs.microsoft.com/en-us/azure/azure-functions/functions-monitor-log-analytics

3.3.2.4 Extensibility

Table 3-16. Microsoft Azure Functions: Extensibility

Microsoft	Azure Functions
Intra-cloud integrations	The Azure Functions service has the overlapping and thus confusing concepts of "triggers" and "bindings." It seems that "triggers" are Azure services wherein events can *trigger* an Azure function, whereas bindings are Azure services that can be programmatically (via an API) linked to an Azure function, thus allowing the function to read data from and write data to instances of that service. Many Azure services (including Blob storage, Cosmos DB, and the Event Grid) are capable of being both triggers and bindings[80].
Edge integrations	**Azure IoT Edge[81]** Deploy functions at the edge. **Azure Stack Edge[82]** Functions can be indirectly with Stack Edge via IoT Edge when an Azure function is contained within an "edge module" deployed to a Stack Edge instance. **Azure IoT Hub[83]** Events emitted by edge devices connected to the Hub can act as triggers for Azure functions

3.3.3 Google Cloud Functions

Google Cloud Functions is Google Cloud's event-driven serverless compute platform providing automatic load-based scaling of user-defined Functions. Cloud Functions is less mature and feature dense than both AWS Lambda and Azure Functions. This can help limit the number of design decisions a developer is required make, thus allowing for a clearer path to getting started and less room for security errors. Documentation for Google Cloud Functions is the most readable and navigable of all the big three FaaS platforms. Google Cloud Functions falls short in the number of allowable regions for deployment of serverless applications.

[80] https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings
[81] https://azure.microsoft.com/en-us/services/iot-edge/
[82] https://azure.microsoft.com/en-us/products/azure-stack/edge/
[83] https://azure.microsoft.com/en-us/services/iot-hub/

Table 3-17. Google Cloud Functions: Overview

Google	Cloud Functions
https://cloud.google.com/functions/	
Category	Serverless
Maturity	Launch date: 2017; last to market of the big three Updates to the platform occur on a monthly or quarterly basis, depending on the time of the year.

Cost	**Model**
	Pay-as-you-go
	Free tier
	• First 2 million requests each month
	• First 400,000 GB-seconds consumed each month
	• First 200,000 GHz-seconds consumed each month
	• First 5GB of outbound internet traffic each month
	Requests (flat rate)
	• $0.40 per 1M requests
	Resource consumption
	• Fee computed based on memory *and* CPU consumption:
	  ○ $0.0000025 per GB-second (memory)
	  ○ $0.0000100 per GHz-second (CPU)
	Sub-second metering
	• Yes. Runtimes are rounded up to the nearest 100ms.
	Networking (data transfer)
	• Inbound data: Always free
	• Outbound data:
	  ○ $0.12 per GB of data traveling outside of the region
	  ○ Free for any traffic traveling *within* the same region
	Caveats
	• No “specific fee” for writing to disk but there is a cost for memory consumed during transferal of data to mount point /tmp.

3.3.3.1 Security

Table 3-18. Google Cloud Functions: Security

Google	Cloud Functions
Confidentiality	**Google Cloud Responsibilities** • Physical security • Securing the Google Cloud network • Hardware updates, OS, and software updates • Encryption of data at rest • Virtualization and multitenancy • Authentication of function invokers (default setting as of Jan 15, 2020[84])
	Customer Responsibilities • Identity management (RBAC via Google Cloud Platform's IAM service[85]) • Authentication and authorization via IAM (for developers, functions, *and* function invokers—unauthenticated invocation of a function must be explicitly turned on) • Service-to-service authentication (via IAM) • End user authentication (via Google Sign In or Firebase authentication) • Proper use of API keys, including rotation of the keys and use of keys in conjunction with authentication tokens • Logging and monitoring • Writing secure code
Integrity	**Google Cloud Responsibilities** • Securing the Google Cloud network • Isolation of serverless execution environments • Proper teardown of execution environments
	Customer Responsibilities • Ensure Cross-Origin Resource Sharing (CORS) is safely implemented (preferably using a proxy)

[84] https://cloud.google.com/functions/docs/securing/managing-access-iam
[85] https://cloud.google.com/iam

Google	Cloud Functions
	• Writing secure code (optimizing for statelessness and idempotency) • Securing their LAN and VPC • Restricting allowed function invokers (for example, restricting function invokers to resource instances within the function's same VPC) • Securing outgoing traffic (for example, opting to use NAT in conjunction with a VPC to avoid assigning a function a public IP address for outgoing traffic)
Availability	**Google Cloud Responsibilities** • Automatic scaling • Load balancing • Redundancy • Upholding commitments in Service Level Agreement • Site Reliability Engineering
	Customer Responsibilities • Logging and monitoring • Deleting any temporary data written to /tmp, as this is an in-memory storage resource that will begin to hog memory (reducing performance and driving up costs) as long as a function is alive. • Opting to implement scaling/concurrency optimizations or limit scaling to abide by the restrictions of integrated resources (such as a database a function may need to interact with). • Exception handling (including abiding by Google's strict exception handling guidelines for Functions). • Writing idempotent/stateless functions • Load testing

3.3.3.2 Performance

Table 3-19. Google Cloud Functions: Performance

Google	Cloud Functions
Supported runtimes	Node.js, Python3, and Go
Concurrency / Scalability	**Default limit** 1000 instances of a single function. This limit *cannot* be overridden. **Automatic scaling?** Yes. Function instances are scaled *within* a region automatically.
Notable limits	Cloud Functions offers only 8 regions for deployment and offers no equivalent to AWS Lambda or Azure it terms of allowing global distribution of a function across its CDN.

3.3.3.3 Maintainability

Table 3-20. Google Cloud Functions: Maintainability

Google	Cloud Functions
Essential developer tools	**CLI** gcloud (Google Cloud native CLI)[86] Serverless Framework CLI (third party) **CI/CD** CloudBuild[87] **IDE** VSCode with Cloud Code extension **Testing and debugging tools** Errors (including uncaught/unhandled exceptions) are viewable in the Cloud Console. As with Google's logging utilities, more fine-grained error reporting can be done via the Stackdriver Error Reporting API[88], for which Google

[86] https://cloud.google.com/sdk/gcloud
[87] https://cloud.google.com/cloud-build
[88] https://cloud.google.com/error-reporting/docs

Google	Cloud Functions
	provides client languages for three languages that Functions supports (Node.js, Python, and Go). Local development and testing of Cloud functions is supported with the use of the open source Functions Framework[89].
Documentation	The documentation is the most approachable of all the "big three." In fact, the documentation across the entire suite of Google Cloud services abides by the same frontend styling guidelines, which further improves readability. Google Cloud does a good job of giving developers a peek "under the hood" of Google's layer of the cloud. While these details are not necessary for getting started, some engineers will appreciate this level this supplementary technical information.
Logging and monitoring	**Simple Logging** When using default, "simple logging" feature, autogenerated Functions logs can be viewed in the Cloud Console or locally using the gcloud command line tool. **Stackdriver** The Stackdriver Logging Client Library is available for "more fine-grained error reporting[90]." This tool "provides an idiomatic interface to the Logging API" that makes it easier to get custom logging logic up and running in a function[91]. **Cloud Monitoring[92]** Cloud Monitoring can be used to set up custom alerting for notable metrics. **Audit Logs[93]** Audit Logs can be leveraged for monitoring account-related activities (such as user log in events). Audit Logs are scoped by project rather than by account. Other entities, such as folders, organizations, and billing accounts have their own instances of Audit Logs as well. Moreover, only project-specific logs can be viewed in the Cloud Console. For reading account-wide logs, the CLI (gcloud) or an API must be used.
Prerequisite skills	Google Cloud seems to have the lowest barrier entry of the big three, mainly because their services put more constraints on what developers can do. Their documentation also tends to be more intuitive. They routinely include code snippets in the docs as well as concrete steps for

[89] https://cloud.google.com/functions/docs/functions-framework
[90] https://cloud.google.com/functions/docs/monitoring/error-reporting
[91] https://cloud.google.com/functions/docs/monitoring/logging
[92] https://cloud.google.com/monitoring
[93] https://cloud.google.com/logging/docs/audit

Google	Cloud Functions
	setting up their serverless infrastructures—for everything from "Hello World" functions to more advanced security features like VPC configuration. Some of experience in web programming (that presumably comes with knowledge of web security) is always helpful when working with any serverless platform. Networking experience is useful but not required for Google Cloud Functions. Like with AWS and Azure, an IAM administrator on the team is required and this person's role should be dedicated to securing the infrastructure. If developers have experience with AWS and Azure, they will find the transition to Google Cloud quite easy, though previous experience with cloud services is not required.

3.3.3.4 Extensibility

Table 3-21. Google Cloud Functions: Extensibility

Google	Cloud Functions
Intra-cloud integrations	Google's logging solution, Stackdriver, can act as a trigger for a Cloud Function. This is the recommended way of responding to notable log events, though it might not be optimal given serverless functions are not well suited to handle streaming data due to prevalence of cold starts. In addition to Stackdriver, other Google resources can generate event triggers, including Cloud Storage, Cloud Pub/Sub, Cloud Firestone, and Firebase[94].
Edge integrations	While Google does not explicitly support deployment of Cloud Functions to the "edge" (neither on its CDN nor to low-compute edge devices), Google might be very well on its way to making deployment of serverless applications on-premises much simpler with Google Anthos[95], a suite of software, including the Google Kubernetes Engine and Cloud Run (based on Google's open source serverless platform, Knative). Anthos is a big step in the direction of making serverless applications more portable and thus edge-deployable.

[94] https://cloud.google.com/functions/docs/concepts/events-triggers
[95] https://cloud.google.com/anthos

3.3.4 Open Source Options

Running serverless applications locally using an open source serverless framework pushes the burden of infrastructure management back on the customer. Hosting serverless applications on customer-owned devices from within customer LANs can be counter-productive for two reasons:

1. Most open source serverless platforms (including those detailed below) are already offered as services by CSPs. For example, Google's open source serverless platform, Knative[96] can be deployed on customer-managed infrastructure, or the customer can opt to use a Google-operated deployment of Knative[97] in the cloud.
2. Local infrastructure management is antithetical to the core principles of serverless, and in many cases, it is more costly than most cloud-based hosting options.

Nevertheless, there are still some cases where a locally hosted infrastructure must be maintained and moving from traditional infrastructure management practices towards infrastructure as code can help automate and streamline repetitive deployment tasks. Deploying open source serverless platforms on-premises requires the adoption of technologies like containers and container orchestration that are the bedrock of serverless platforms. Some of these open source serverless platforms even allow developers to deploy serverless applications using a hybrid approach—where code is deployed locally and to commercial CSPs. This abstraction provides increased flexibility in hosting and cloud migration options.

The following Table 3-22 lists open source options throughout the serverless stack – from running basic containers to container orchestration to serverless application development platforms – that can be used in local deployments.

Table 3-22. Open Source Serverless Options

Name	Category	Developer / Maintainer	License	Supported Languages
Docker[98]	Container	Solomon Hykes Docker, Inc.	Apache 2.0	All[99]
	Platform for no-hassle containerization of applications. Handles development and deployment and makes migration of highly portable container images trivial.			
Docker Swarm[100]	Cluster (container orchestration)	Docker, Inc.	Apache 2.0	All

[96] https://knative.dev/
[97] https://cloud.google.com/knative/
[98] https://www.docker.com/
[99] Although Docker is language agnostic (any language can be used in a Docker container), developers can download one of their pre-built language stacks (Docker images preloaded with language-specific runtime environments) via Docker Hub. Currently, Docker Hub offers a language stack for 20 languages [108].
[100] https://docs.docker.com/engine/swarm/

Name	Category	Developer / Maintainer	License	Supported Languages
	Docker plugin for management of a multi-(Docker) container environment (cluster).			
Kubernetes[101]	Cluster (container orchestration)	Google Cloud Native Computing Foundation	Apache 2.0	All
	Tool for management of multi-container environment (cluster). Runs on top of Docker, provides more features.			
Knative[102]	Function	Google	Apache 2.0	C#, Go, Java, Kotlin, Node.js, PHP, Python, Ruby, Scala, Shell
	Serverless framework that adds a thin layer of functionality over Kubernetes.			
Kubeless[103]	Function	Bitnami	Apache 2.0	Python, Node.js, Ruby, PHP, Go, .NET, Ballerina[104]
	Kubernetes-native FaaS framework. Claims to be the "most Kubernetes native of all."			
OpenFaaS[105]	Function	Alex Ellis and community	MIT	Node.js, C#, Go, Java, PHP, Python, Ruby[106]
	Kubernetes and Docker-native FaaS framework			
OpenWhisk[107]	Function	IBM Apache Software Foundation	Apache 2.0	.NET, Node.js, Go, Java, PHP, Python, Ruby, Swift, Ballerina (experimental), Rust (experimental)
	FaaS framework that runs on top of a variety of container platforms.			
Fn[108]	Function	Oracle	Apache 2.0	Go, Java, Node.js, Ruby, Python

[101] https://kubernetes.io/
[102] https://knative.dev/
[103] https://kubeless.io/
[104] Claims to also support "custom runtimes" (most likely means Kubeless can run in custom Docker images).
[105] https://www.openfaas.com/
[106] OpenFaas currently offers "templates" for these languages (i.e. Docker images with built-in support for these languages). According the docs, the templates can be customized, but only if a developer is willing to fork the repository and make their own changes.
[107] https://openwhisk.apache.org/
[108] https://fnproject.io/

Name	Category	Developer / Maintainer	License	Supported Languages
	Docker-native FaaS platform.			
Fission[109]	Function	Platform9 and community	Apache 2.0	Go, Java, Node.js, Python
	Kubernetes-native FaaS framework.			
The Serverless Framework[110]	Framework	Serverless.com and community	MIT	Node.js, Python, Java, Go, C#, Ruby, Swift, Kotlin, PHP, Scala, F#
	The Serverless Framework is a free and open source serverless application framework written mainly in Node.js and used to develop serverless applications for AWS Lambda, Azure Functions, Google Cloud Functions, IBM Cloud Functions and others.			

[109] https://fission.io/
[110] https://www.serverless.com/

4 Security Considerations

Moving event-driven data processing from central control to distributed edge servers running microservices presents a host of security related issues. While edge computing will be integral in allowing new applications to meet their performance requirements, security must not be an afterthought.

The recent news about proliferation of malicious IoT botnets highlights the need for security centric development practices when it comes to writing distributed applications for cloud, edge, or hybrid deployment. The following sections summarize best practices along with guidance for confidentiality, integrity and availability in edge and serverless computing.

4.1 Best Practice Recommendations

4.1.1 Development Process

Read relevant parts of the vendor documentation in their entirety. Hands-on learning is always a great way to get started, though developers will always make assumptions during hands-on training that only a developer guide can correct.

For example: If a developer is getting started with AWS Lambda, and wants to write a Lambda function that is integrated with DynamoDB, it will be important for the developer to read the following sets of documentation:

1. AWS Lambda's documentation on working with DynamoDB[111].
2. The DynamoDB section of AWS' whitepaper on its security processes[112]. This provides an overview of default DynamoDB security configurations as well as opt-in settings that are critical for any customer to be aware of, like how to ensure database instances are automatically backed up to guarantee data is recoverable in the event that an unforeseen disaster takes a database server offline.
3. Security and Compliance in Amazon DynamoDB[113]

Do not rush the development process. While time to market is always an important factor in the success of a service, a single security incident can cost an organization millions of dollars and even its reputation.

Embrace test-driven development. Test-driven development means developers write tests (which requires them to think about edge cases) *before* they start writing any code. Thinking about edge cases from the start means developers will be more apt to recognize how their code can be broken, and in turn, identify hot spots for security vulnerabilities. This method for software development is also good for debugging, as the short feedback loop between a small change to code and a test means bugs can be identified much more quickly.

---

[111] https://docs.aws.amazon.com/lambda/latest/dg/with-ddb.html
[112] https://d0.awsstatic.com/whitepapers/Security/AWS_Security_Whitepaper.pdf
[113] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/security.html

Prefer automation and scriptable toolsets over web consoles for serverless development. Developing serverless functions within a CSP's web console is a good way for new developers to learn the platform, as these web consoles typically support rapid prototyping and testing of functions; however, most vendors limit which languages and capabilities can be used inside the web interface. Moreover, developers will inevitably experience some degree of latency and discomfort when developing in a web-based text editor rather than their local environment that has been configured with personalized settings.

When choosing a language or runtime for a serverless function, be flexible. You must fully consider your use case, developer proficiency, and performance requirements. Compiled languages like Go may have better performance. Node.js and Python may have a lower barrier to entry as well as lightweight runtimes, which mean cold starts of applications written in these languages will be tolerable if all other best practices are followed. Languages with relatively heavy runtime environments, like Java, should be avoided.

When developing on a serverless platform, do not take a hands-off approach to scalability even though this is provided out of the box. Failure to optimize concurrency and/or limit concurrency can be costly both in terms of performance and billing [56] [57]. Consider options to customize how scaling of functions occurs on a serverless platform. For example, if consistently high traffic is anticipated, **provisioned concurrency** will reduce the number of cold starts in response to high request volumes. On the other hand, **limiting concurrency** prevents a function from scaling too much for other resources to keep up, which would degrade availability [57].

4.1.2 Security is a Shared Responsibility

Do not assume a label of "serverless" means that the serverless provider is solely responsible for securing a customer's serverless application. Remember the **shared responsibility model** applies to serverless too. While the CSP will take on more responsibility in securing serverless applications (for example, the CSP will take care of OS updates) than in traditional IaaS environments, there is no shortage of security tasks that the customer will remain responsible for. The three main ones include:

1. Securing their application's code and data.
2. RBAC (via IAM).
3. Securing integrated services.

Do not assume vendor-provided lists of "best security practices" are a comprehensive checklist of everything that needs to be done to secure a serverless application. Vendor-provided lists of security best practices should be viewed as a list of measures that a customer can take to prevent *some* (but not all) common mistakes and/or address *some* edge cases that are routinely overlooked, but the vendor cannot anticipate every possible mistake that could lead to a security vulnerability.

Do not ignore history; learn from other's mistakes. If a security breach occurs on a major vendor platform, reading about it and understanding the details will often highlight security holes and better practices to ensure the breach is not repeated. The recent Capitol One hack was an enlightening demonstration of the dangers that result from sloppy coding practices combined

with a lack of understanding of AWS' default configuration settings that are applied to *every* EC2 instance [58].

Do not assume that the CSP is fully responsible for securing their networks. While most CSPs leverage TLS by default to encrypt data moving within and outside of the cloud, TLS is not a silver bullet for network security. It is for this reason that CSPs offer many tools to provide additional security. One ubiquitous and highly effective tool is a Virtual Private Cloud (also referred to as a VPC for short or VNet by Azure). Out of the box, VPCs include features like IP address whitelisting that can help to mitigate any fallout from misconfigurations of compute and storage within the cloud infrastructure (like an endpoint that is accidentally left open) that are extremely common [59]. Additionally, VPCs usually come with a platform for centralized management of network security settings for a customer's cloud infrastructure.

4.1.3 Choosing a Cloud Service Provider

Prioritize reliability over flexibility when choosing a vendor. Be wary of feature overload. This is when vendors provide too many options to achieve the same thing. Over-burdening developers with too many choices can lead to mistakes with security consequences [60].

Prioritize maturity over "newness" when choosing a cloud service provider. Mature vendors have had time to stress-test their systems and have robust processes for finding and fixing bugs. This means their customer applications are more likely to be highly available and secure [60].

Prioritize vendor reputation over price when choosing IoT devices. Follow vendor recommendations for deploying devices at the edge. AWS, for example, provides a vast catalogue of devices that have been approved for running AWS IoT Greengrass.

Choose a Cloud Service Provider and stick with them. A multi-cloud deployment strategy is far too complicated to provide any real benefits. Using a single public cloud alone requires great effort to vet services, APIs, and security protocols. Opting to deploy across multiple public clouds will double or triple this workload as well as introduce new interoperability challenges. The big three CSPs: AWS, Azure, and Google Cloud, are all good options for building serverless applications, as well as integrating cloud deployments with infrastructure at the edge. If applicable, one should also consider FedRamp[114] compliance of the cloud service provider.

Avoid Cloud Service Provider solutions for operating an edge deployment "offline" [43]. CSPs benefit from fog computing architectures that are designed to promote integration of edge devices with their own infrastructure. This is antithetical to hyper resilient computing at the edge – a model where decreased need for connectivity from the edge to the cloud means less bandwidth backhaul requirements. If there is a use case for fog computing, consider leveraging "Hardware-as-a-Service" by deploying a CSP's own hardware at the edge. AWS Outposts and Azure Stack Edge are viable options for this type of service and will be easier to integrate and maintain.

---

[114] https://www.fedramp.gov/cloud-service-providers/
AWS: https://aws.amazon.com/compliance/fedramp/
Azure: https://docs.microsoft.com/en-us/azure/azure-government/compliance/azure-services-in-fedramp-auditscope
Google: https://cloud.google.com/security/compliance/fedramp

4.1.4 Integrating Third-Party Services

Do not include untrusted third-party dependencies (like libraries and/or modules) in serverless applications. Limit the number of trusted third-party dependencies as well. Serverless functions are prone to performance issues when having to undergo cold starts. Thus, be cognizant of how your chosen dependencies effect system image size and cold start times.

Avoid closed source third party services for extended or enhanced functionality. If there is a desire for non-native functionality, use an open source third party tool. Open source allows inspection of the code and verification of the functionality to ensure there are no security vulnerabilities. The Serverless Framework is an example one such open source option; however, its commercial features must be purchased. This framework is well-liked among developers and has support for AWS, Azure, and Google Cloud, as well as various other serverless platforms [61].

4.1.5 Edge Computing

Do not overlook physical security when deploying edge computing. Most of the above recommendations focus on best practices for securing software; however, deploying edge computing devices includes physical assets at remote locations. Physical security becomes an even greater concern when controlling access to geographically dispersed assets, sometimes in collocations which your organization does not control.

4.2 Confidentiality

Do not use your cloud account's root user on a regular basis. AWS recommends that the root user only be used to create the first IAM user. After that, secure the root credentials and perform all other tasks with IAM users whose privileges are strictly limited [62].

Do not over-assign privileges (i.e., avoid wildcards (*) in IAM policies). Assignment of privileges to users, roles, and groups should be done using the least-privileges model, meaning identities should have the minimum number of privileges required to perform their tasks.

Do not allow everyone to invoke a serverless function. If a serverless function must be publicly accessible, ensure only authenticated users can invoke it by default. Google Cloud Functions has this as a default setting on all Cloud functions [63]. Only re-consider allowing unauthenticated invocation of a function or service if legitimate use cases arise.

Do not assume that developers will secure their cloud accounts on their own accord. Even software engineers who are aware of the inherent dangers of lax security practices are prone to cutting corners, so it is important to have fail safes in place for when that happens. Compel developers to secure their accounts using multi-factor authentication and strong passphrases. Additionally, developers should be compelled to encrypt any access credentials at rest, preferably in a system keychain or password manager. This is not always possible to do with API keys, as they have to remain unencrypted so than can be used by developer CLIs, though developers should practice good security hygiene by never hard-coding API keys and opting to place them in restricted configuration files or environment variables. Passphrases and API keys should never be checked in to source control.

4.3 Integrity

Do not write serverless functions with "side effects" [64] [65]. Write idempotent functions**.** This is a good way to ensure a serverless app maintains consistency, meaning it will always respond to client requests with the most accurate and up-to-date information *or* an error message on failure [65] [64]. It is especially important for a function to have no "side effects" when handling any persistent data. Google has a good example of how consistency in customer data can be adversely affected if functions are not idempotent[115].

Writing idempotent functions requires thinking about edge cases and making small changes to the code to ensure that a function will not produce unexpected results no matter how many times it is executed in succession [64]. Due to their inherent characteristic of having no side-effects, idempotent functions are more fault tolerant as they will be unlikely to precipitate the failure of a subsequent re-try call to themselves.

Do not manually handle Cross Origin Resource Sharing (CORS) in source code. CORS is an important security feature that most modern browser have by default. Safely enabling CORS is especially important in serverless architectures, as the inherent modularity of serverless applications means that underlying functions can be hosted on different domains, thus cross-function communication will oftentimes be restricted by a CSP's default restrictions on CORS.

It is recommended that CORS be enabled by using a vendor-provided proxy. For example, Google Cloud recommends the Cloud Endpoints proxy for cross function communication [66]. In addition to enabling the *safe* use of CORS, proxies can provide a host of other useful security features, such as request validation – a critical first line of defense [67].

4.4 Availability

When working with a Function-as-a-Service (FaaS) platform, **avoid writing stateful functions** [65]. **Write stateless functions**. A stateless function is one that does not require knowing anything about its own state during runtime. This means the function is simply passed an input by the caller with which it executes a task – usually manipulating the input data in some way -- but does not need any extra data to complete the task. A stateless function also does not cache or store any data about its state that could be accessed by another process. Statelessness is important in serverless applications for several reasons:

- **Performance**. Increasing the RAM allocation of a serverless function to store state data that can be leveraged by a subsequent invocation of that function is often more costly than its worth. The timing cost of retrieving that information written to an external database or datastore is even worse, especially given the serverless customer can rarely anticipate how far away that persistent datastore will be from their function.
- **Cost efficiency.** Following on from performance, a slower function runs longer and since billing is tied to runtime and resource consumption, shorter running and RAM-efficient functions accumulate less runtime costs.

[115] https://cloud.google.com/blog/products/serverless/cloud-functions-pro-tips-building-idempotent-functions

- **Fault tolerance.** Serverless customers have very little control over the environments in which their code is executed, so assuming that data will persist across separate invocations of the same function is will lead to failures when the unavoidable wiping of memory occurs when a function is torn down [60].

Do not use the same cloud account for production and testing [60]. The concurrency limits set by most cloud platforms apply to the *entire account*, so it is easy to Denial of Service yourself by exhausting concurrency limits through automated testing [60].

Do not forgo testing serverless code in the cloud because the provider offers tools for local testing [60]. While local testing has benefits and can be leveraged for the bulk of testing during development, serverless code *must* be tested in the cloud, as it is impossible for a developer to replicate a CSP's execution environment on their local machine. A function that succeeds locally could very well fail in the cloud.

Use local testing for prototyping and debugging during development [60]. When debugging serverless code, a good rule of thumb is to test bug fixes in the local environment until the bug is resolved and then verify the solution works in the cloud as well. For example, AWS provides the SAM CLI for local testing and Cloud9 (a cloud-based IDE) for hybrid testing and debugging [68] [69].

Do not adopt a manual deployment and traffic shifting strategy [70] [60]. Manually shifting traffic to a function as soon as it has been deployed is a bad idea, especially when CSPs offer CI/CD solutions that integrate automatic fail safes into the deployment process [60] [70]. For example, the AWS SAM CLI is integrated with AWS' CodeDeploy, which offers features such as running sanity checks on new versions of functions before redirecting traffic, as well as configuration options to gradually shift traffic and automatic roll back and alerts if a function fails during the deployment process [70] [71].

Think about edge cases that will break the application code. Write serverless applications with the assumption that user input will be wrong and/or malicious. Exceptions thrown by serverless applications are much harder to debug than traditional applications. Additionally, exceptions can affect availability of a service. Take a whitelisting approach to processing user input – meaning only accept user input if it abides by certain strict criteria, and above all, **sanitize user input**.

4.5 Summary

Our last and perhaps most important recommendation is to be wary of cloud security checklists, including the list of best practices presented in this paper. While checklists can offer a helpful starting point, cyber defense cannot be condensed to a list of tasks. Security is a process that needs to be incorporated into the architecture design and application development lifecycles. This includes ongoing reassessment of the effectiveness of security controls during the operational phase of a system.

A critical component of cyber security is the humans who design, build, configure, maintain, and use the security systems. Humans are highly fallible; thus, human error is far more likely to be the source of a breach in the cloud than an obscure zero day [72]. Thus, an honest and nuanced risk assessment—at the organization, project, and application levels—that results in

recommendations about which defenses are worth the implementation and maintenance costs will lead to far better results than following a checklist of “best practices” that may often leave out critical pieces of context, such as constraints on budgets and the abilities of staff.

This page intentionally left blank.

5 Example Use Cases

Serverless computing is best done in the cloud wherein CSPs have nearly full control over the infrastructure. However, the technology is ready now to have a potentially beneficial impact outside the cloud thanks to a large ecosystem of open source tools for serverless application development. These tools make it easier for developers to stand up resilient cloud-like infrastructures on their own commodity hardware for all sorts of applications.

By contrast, edge computing relies on a highly distributed infrastructure that does not yet exist and will be controlled by many competing parties among whom coordination will be complex. The timeline for edge computing's beneficial impact is expected to be significantly longer – on the order of years. That said, the use cases for edge computing are more compelling than the current use cases for serverless alone.

5.1 Large-scale Distributed Application Architectures

Any large-scale application that requires a distributed architecture (i.e., deployment across many edge nodes) will benefit from serverless computing at the edge. The benefits increase with Function-as-a-Service (FaaS) due to the extremely high costs incurred when deploying a distributed application:

- **Overhead cost.** Standing up and maintaining a distributed system is prohibitively expensive for most organizations from an infrastructure standpoint. Organizations who want to distribute a system globally look to CSPs for their existing scale and points of presence for delivery. Even if an organization considers hosting a smaller-scale distributed system, infrastructure and engineering costs are still high.

  Serverless can lower this cost through more efficient use of infrastructure leveraging lightweight containerization technology. This allows for easier scaling of a services both *within* and *across* edge nodes deployed at various edge locations – such as edge nodes in a CDN or even within a customer LAN.

- **Opportunity cost.** Building distributed systems is technically difficult. Even with all the open source languages, libraries, and frameworks that are available to help, organizations still need to hire highly skilled engineers and pay them high salaries to design, build, and maintain these complex systems.

  AWS, Azure, and Google Cloud have a proven track record of competency in managing distributed systems in the public cloud. Choosing CSPs for serverless significantly reduces the opportunity cost of infrastructure maintenance, as the provider takes care of all the infrastructure to support distributed computing.

Nevertheless, there are many tradeoffs in deploying to the cloud versus deploying locally, as well as in choosing serverless over more traditional IaaS platforms for application development. For many use cases, customers will find deploying monolithic code to a single EC2 instance or just hosting an application locally will suffice. However, for cloud native applications of tomorrow, serverless is a compelling approach for building applications with redundancy, scalability, and geographic distribution.

5.2 Event-driven Data Processing at the Edge

Another use case for serverless in conjunction with edge computing is event-driven data processing at the edge. The following are three applications that will be of interest to many sponsors.

5.2.1 Real time data processing at the edge

Edge computing and serverless technologies can enable processing of data at or near the location of data collection/generation, which has the potential to be a game changer for mission critical applications that have any degree of latency intolerance. This includes health care, surveillance, and disaster mitigation, among others. Lowering latencies is a key driver of edge computing adoption and given the fact that most edge devices are inherently lightweight and oftentimes sit within bandwidth constrained LANs, leveraging serverless technologies at the edge will also be important in ensuring compute and power on edge devices as well as local bandwidth is consumed conservatively.

5.2.2 Analytics on Edge Devices

Machine Learning (ML) models are notorious for being compute intensive. Nevertheless, technologies are emerging to enable the testing of ML models on low-compute edge devices. Today, a device as tiny as a Nvidia Jetson or Raspberry Pi can host an ML model that has been optimized for the edge.

IoT devices are ubiquitous and are generating massive amounts of data. One thousand automobiles, for example, will generate "the total daily data volume processed by Facebook" [39]. A modern commercial plane engine has enough sensors to generate "up to 10 GB of data per second, resulting in 844 TB of data from a 12-hour flight for a twin engine aircraft" [39]. With that amount of data, it will become necessary to deploy ML models on edge devices closer in proximity to the data-collecting sensors for the following reasons:

1. **Bandwidth constraints.** Sending 10 GB of data per second from an aircraft to the cloud is impossible due to the aircraft's inherent bandwidth constraints. Moreover, reliable internet connectivity is still never a guarantee for some users, especially those in Denied, Disrupted, Intermittent, and Limited (DDIL) bandwidth environments.
2. **Latency.** Even if there were enough bandwidth to go around, there are still physical laws that limit the speed at which the cloud can ingest, analyze, and respond to massive amounts of sensor data. When it comes to planes and self-driving cars, latency becomes a safety issue if data is being used to prevent a crash or respond to environmental changes.
3. **Time to action.** The ability for an application to autonomously react to environmental changes in real-time is especially important for mission-critical applications, like monitoring the health of medical devices.

Google and others (e.g., NVIDIA[116], Intel[117]) are already providing lightweight ML technology. Two of Google's most notable innovations include:

1. TensorFlow Lite[118]: A lightweight ML framework based on the popular TensorFlow technology that is designed specifically for deployment to low-compute edge devices. TensorFlow Lite also has a Python API, which means it's relatively easy for developers to get a TensorFlow Lite model up and running.
2. The Edge TPU[119]: A "Tensor" Processing Unit designed by Google to speed up ML computations done within their TensorFlow framework. Google has launched an entire suite of products with embedded Edge TPUs as well as accessory hardware like cameras and sensors that can be easily attached to compute with an embedded TPU [73].

5.2.3 Securing IoT

There is already lots of computing being done at the edge in the form of IoT. However, a wide array of vendors, standards and haphazard implementation decreases IoT's potential benefits. In some cases, the data collected is either kept at the edge or lost when overwritten by new data. Moreover, a majority of IoT devices have a less than reputable track record when it comes to cyber security. We must begin to confront the reality that "cheap IoT threatens the internet" and will probably never cease to be a threat [74].

To help mitigate this issue, CSPs have begun developing fog computing technologies to provide stronger linkages between IoT devices and the cloud. AWS and Azure have both developed specialized routing software, IoT Greengrass and IoT Edge, that can orchestrate connections to IoT management services in the cloud, enforce stricter security requirements for these connections, and increase reliability within customer LANs by supporting more secure intra-IoT messaging without the need for cloud connectivity. IoT Greengrass and IoT Edge also come bundled with serverless runtimes that allow for the hosting of Lambda and Azure functions, respectively, to listen for notable events with customer LANs, process those events, and send insights to the cloud.

[116] https://developer.nvidia.com/tensorrt
[117] https://software.intel.com/en-us/neural-compute-stick
[118] https://www.tensorflow.org/lite
[119] https://cloud.google.com/edge-tpu

This page intentionally left blank.

6 Summary

The three main takeaways of our research are:

1. Edge Computing *at scale* is likely a decade away.
2. Serverless is by no means *effort-less*.
3. Edge computing and serverless can work together to *facilitate real-time data processing closer to the points of data collection and consumption*. Though as of today, use cases are still limited.

6.1 Summary of Serverless Benefits and Challenges

6.1.1 Benefits of Serverless

Serverless is good for microservices. While containerization and container orchestration – layers of virtualization that underly serverless technology – have wider applicability than fully managed serverless platforms, Function-as-a-Service (FaaS) is particularly well-suited to cloud-native application architectures. There are several key reasons why this is the case, including:

- **Scalability** [32]. Monolithic applications do not scale easily. By contrast, developers are encouraged to keep their serverless functions small to ensure scalability. As a result, serverless applications are generally stateless and microservice-based, which helps to guarantee that orchestration of starts and stops is done efficiently. Serverless functions can scale horizontally on commodity hardware and are typically lightweight enough to share this hardware with other applications. Monolithic applications generally need single instances of more capable (i.e., costly) hardware.
- **Resiliency** [32]. Resiliency is significantly increased by leveraging serverless in the public cloud. Cloud serverless providers design their platforms to encourage microservice application architectures that make isolation of services trivial. Thus, when an inevitable unhandled exception occurs, only a self-contained piece of the application will be affected.
- **Inherent modularity of microservices** makes integration of new features easier [32]. Automating build and deploy processes is much easier when applications are broken down into small, microservice-sized chunks rather than large, monolithic code bases. In monolithic application architectures, the smallest update to back end code can require a full re-build, test and re-deployment [60]. In a microservice-based architecture, each service exposed by an application is self-contained and therefore separately instantiated from the rest of the microservices that comprise the application. Thus, updating an existing microservice or adding a new one can be done with offline integration testing and the feature can be deployed in a limited production push until confidence in the change is established.
- **Cost transparency.** Every instantiation of a serverless function is logged for billing, giving customers a granular view into which components of their serverless applications

are most costly. This insight is valuable to managers when making assessments on where to invest to increase cost efficiency. The cost transparency of serverless can help CSP customers use resources more methodically as well as indicate how dollars translate into capability.

Serverless is good at **reducing the amount of time developers will spend on non-value add tasks** like infrastructure maintenance and upgrades. AWS was born from Amazon's own need to abstract away compute provisioning and management tasks to ease the burden on Amazon engineers and cloud computing is delivering these benefits at scale [10] [9].

Serverless is good for **reducing total cost of ownership** if cloud deployments. IaaS eliminated the need to pay for a local data center. PaaS eliminated the need to maintain operating systems. FaaS abstracts all underlying infrastructure and eliminates the requirement to pay for uptime of a particular instance inside the cloud when all that is needed is a few seconds of compute for a microservice to run.

6.1.2 Challenges of Serverless

Most CSPs emphasize the *increased productivity* developers will experience when building serverless applications on their platforms. While it is true that developers no longer manage infrastructure when using serverless platforms, **serverless application development introduces ancillary tasks that can quickly eat away into a developer's productivity.** These tasks are not unique to cloud-based serverless platforms and are unavoidable when using serverless as the framework for application development:

- **Identity Access Management (IAM)**. Role Based Access Control (RBAC) is almost always the customer's responsibility in securing cloud environments. RBAC can be quite complex and require detailed knowledge of the vendor's IAM service. In the context of serverless, IAM is important for ensuring that developers have the least privileges allocated to perform their tasks as well as ensuring that serverless functions are securely assigned roles that will allow them to safely access and operate on the data they will process. Teams who are building serverless applications in the cloud can benefit from at least one dedicated IAM administrator to ensure RBAC is handling properly in their deployment.
- **Managing architectural complexity.** While monolithic applications suffer from scalability problems, serverless applications composed of microservices can suffer from complexity problems. Breaking up applications into modular components offers benefits including scalability, resiliency, and increased portability of components; however, it complicates application logic and require robust networking and integration to ensure continuous operations. Moreover, the level of complexity in these architectures can increase when developers are working with closed-source platforms in the public cloud. This requires CSPs' documentation be clear, well organized, and thorough.
- **Testing and debugging are difficult on serverless platforms.** Cloud-hosted execution environments in which serverless functions run cannot be replicated locally; therefore, reproducing errors for debugging is very difficult. Moreover, accessing error logs in the cloud is also difficult. And even when logs can be accessed and reviewed, some error

messages are too cryptic to provide any useful information out of context – the crucial context being the non-local, cloud environment in which the functions are being executed.

When building a serverless application in the cloud using a Function as a Service (FaaS) platform, the eventual deployment can be **inherently disorganized** [60]. This is because each endpoint will invoke its own microservice. Moreover, the customer has little control over where these services run. When client request handlers are distributed in such a way, so too are the potential vulnerabilities in the request handling routines. This equates to a **larger attack surface for a serverless application** [60].

When working with serverless platforms, there are many details to consider:

- **Documentation does not always align with vendor claims.**

  > *With Lambda, you can run code for **virtually any type of application** or backend service - all with **zero administration*** [75].

  Serverless implies a lack of infrastructure beneath the application code, though assuming that it "just works" securely, is a mistake. With AWS Lambda, for example, a developer can launch their first function by writing some source code in the web console, setting a few configurations (mainly providing their function with an IAM role), and clicking "Create Function" [76]. Testing the Lambda from within the web console is equally easy. Nevertheless, despite the low barrier to entry, the created function can open several types of security vulnerabilities if the roles are misconfigured, a bug is present, credentials are mishandled, or data is not secured.

- **Make sure the solution fits the problem.** Serverless is best suited for microservices architectures. Thus, re-architecting a small-scale application or a long-running program to become "serverless" would be counter-productive. While supporting technologies like containers and container orchestration may prove useful in development, testing, and deployment, migration to a microservice-based architecture to make use of fully managed serverless may not be the best approach for many use cases.

- **Understand the trade space.** Serverless rids developers of the burden of non-value add infrastructure management tasks. Nevertheless, serverless introduces more complex microservice interconnection and orchestration tasks, as well as complicated access control administration tasks. Moreover, when leveraging serverless in the public cloud, local environments are traded for remote environments wherein familiar toolsets may no longer be supported.

- **Complicated IAM protocols that are "easy to get wrong"** [60]. All the "big three's" IAM services have steep learning curves and require a detailed review of the IAM documentation and best practices. When building a large-scale serverless application, at least one team member should devote themselves to securing the infrastructure and developing expertise in IAM.

- **Hidden fees** (see specific details in the evaluation criteria); the biggest drivers of which are:
    - Networking (data transfer) – Additional fees that a customer must pay to transfer data across the cloud and to hosts outside the cloud. Additionally, deploying a Virtual Private Cloud (VPC) will incur an extra cost.
    - Costs associated with the use of other cloud resources. For example, putting a record in a database instance will incur an extra charge on top of a serverless function's invocation and resource consumption charges.

6.2 Summary of Edge Compute Benefits and Challenges

6.2.1 Benefits of Edge Computing

The key benefit of Edge Computing is decreased latencies for the delivery of internet services. This is critically important for new use cases that cannot tolerate latency on the order of 50 to 100 milliseconds typical in today's regionalized internet. New applications promise to dramatically enhance user experience through the delivery of "multidimensional experiences" and facilitate increased automation of everything from cars to drone delivery services [5]. For these use cases, latencies of microseconds to 10 milliseconds are expected, without which user experience is degraded, or worse, serious safety hazards arise.

The exponential growth of IoT devices means more data than ever is being collected about everything from our physical world to our habits and personal preferences. This puts a premium on bandwidth in the core for centralized aggregation and processing. Thus, it is critical that more compute be placed closer to end users to achieve latencies that are low enough for use cases like Augmented Reality and machine-to-machine communication that cannot tolerate bottlenecks in backhaul networks [5].

Other benefits of edge computing include:

- **Preservation of backhaul bandwidth to core networks.** This is especially important as more data at the edge is collected. Edge computing helps to offload processing of this data to lightweight IoT devices to generate intelligence that can be immediately acted upon. Edge computing can also accomplish this with deployment of high-power compute devices at the edge that will enable bandwidth preservation and real-time analytic generation—this model for data analytics has been labeled the "Intelligent Edge" [39] [77].
- **Reduced pressure on core (origin) servers.** While the hyperscale data centers in the cloud are gigantic, cloud servers are also subject to resource limitations. Additionally, renting compute in the cloud can become quite expensive when autoscaling occurs to meet higher than forecasted demand. If a customer needs to analyze data collected at the edge, it can be more cost-effective to run these analytics on-premises.
- **Increased resiliency to network outages.** Denial of service can happen in a variety of ways; anything from an accidental oversight to bad weather or malicious actors can be the culprit of a DoS. Edge computing increases resiliency of internet services by

migrating them from the cloud to the edge and by distributing services across many edge nodes, thereby insulating services from centralized failure and segmenting failure domains to smaller subsets of users.

6.2.2 Challenges of Edge Computing

Edge compute nodes cannot yet be hyper-scaled; a key advantage of cloud services. The key feature of an edge compute node is that it is more lightweight than a typical hyperscale data center sitting in the core and thus more able to be placed near locations of data collection and consumption. This shifting of data analytic tasks from the cloud to the edge trades the power of cloud for more agile response times for critical services. The question of "how much more lightweight" drives this compromise. While there are some startups calling their relatively large data centers (~50,000 square feet) "edge data centers," an edge node can be anything from a Raspberry Pi to a shipping container-sized micro data center that sits near a cellular base station [78] [79].

The primary challenge with edge computing is the **barriers to implementation**. In the case of serverless, CSPs have full control over their infrastructure and can deploy services and developer tools to this infrastructure in an agile fashion. Edge computing requires wide-scale physical infrastructure build-outs before any type of virtualized workloads can be delivered with enough distribution to achieve any meaningful reduction in latencies. This regionalized internet of today is a resource controlled by many competing parties and depended upon by nearly every person on earth, so moving from our current phase of regionalization to more localized internet services will be a slow process [5]. Current realities can help to shape a picture of what the future internet may or may not look like, and so far, it seems like the scaling of edge computing technologies could go one of two ways:

1. **ISPs will take the lead.** This is plausible because ISPs control much of the internet's infrastructure and can supply compute to key access points. Multi-Access Edge Computing (MEC) defines one such deployment model for edge computing in the context of 5G cellular networks. While 5G and edge computing are both a long way from having widespread availability, ISPs are already laying the groundwork. AWS Wavelength and similar partnerships between ISPs and CSPs are pushing forward with the goal of embedding cloud services into ISP access points [19] [18]. While CSPs are taking a cautious approach to edge computing, ISPs are making it clear they are no longer satisfied with simply being transit providers. With 5G and edge computing, we will likely see ISPs pushing to develop their own edge computing service offerings.
2. **ISPs will not lead the way**, resulting in competition between ISPs, CSPs, and startups to acquire market share in a potential edge computing service delivery sector. This could lead to a variety of competing partnerships, proprietary interfaces and slow adoption until the inevitable convergence as larger players consume or decimate rival startups. We discussed successes of some edge computing startups in section 3.2.2. One such startup is deploying their hardware directly on an ISP's network. The other is more focused on extending the public cloud to ISP access points.

Whether or not the ISPs lead on buildout of edge-computing, the internet architecture that we will see in the coming decades will be quite different than what we see today. Our current

internet architecture is ill-equipped to meet the demands of modern users, who will be increasingly reliant on edge compute to satisfy their latency-intolerant needs. The transition from hyper-centralized to hyper-*de*centralized internet services will be slow, as the reliability of the new edge services will have to be proven before mission critical applications can be migrated.

6.3 Deployment Strategy

When choosing a cloud deployment strategy, vendor "lock-in" is a pervasive concern in both the private and public sectors. However, vendor "lock-in" should *not* be a deciding factor in how an organization chooses to architect their cloud infrastructure [80]. Taking steps to mitigate the risk of vendor "lock-in," like a multi-cloud deployment approach, introduce a new set of challenges including incompatible APIs, data migration issues and complex cost accounting. The main risk in using a single provider to host an entire cloud deployment lies in "data lock-in," as Yan Cui puts it [80]. That said, the probability that one of the big three would do something to warrant the migration of a customer's entire cloud infrastructure to another provider is extremely low, while the **complexity of migrating** from one CSP to another is very high – and potentially costly [80].

We strongly recommend that organizations pick a single vendor after proper vetting and stick with that vendor. Additionally, we also recommend that organizations choose one of the big three (AWS, Microsoft Azure, or Google Cloud Platform) as their provider unless there is an extremely niche use case which cannot be satisfied or is better tailored to a custom solution. The risk of an unpleasant surprise is decidedly less when using one of the big three CSPs, while working with a small-scale vendor introduces challenges outside the technical realm (like reliability, reputability, and longevity). Moreover, small-scale vendors are not able to provide the level of extensibility that the big three can guarantee. Thus, risk of failure due to vendor lock-in is much higher with small scale vendors.

AWS arguably invented the cloud with the launch of the first successful IaaS platform, EC2, in 2006 [9]. AWS is also the most mature of the big three; however, the increasing complexity of their services' APIs and configuration options is troubling [81]. AWS's flagship serverless offering, Lambda, has a well-documented open source CLI that greatly enhances developer experience. Moreover, AWS also supports the deployment of Lambda functions to AWS points of presence via the Lambda@Edge service [82] [1].

In addition to Lambda@Edge, AWS has early stage initiatives to bring their own compute services to the edge, including AWS Local Zones, which intends to deploy small-scale AWS edge data centers near major metro areas (starting with Los Angeles) and AWS Wavelength, a partnership between AWS and telecoms to "embed AWS compute and storage services within the telecommunications providers' datacenters at the edge of the 5G networks" [83] [82]. For now, AWS IoT Greengrass offers a virtual extension of a small subset of AWS services, including Lambda.

Despite its history of revolutionizing cloud computing—first by inventing the cloud with IaaS and its subsequent invention of Function-as-a-Service with Lambda—it remains to be seen

whether AWS will be first-to-the market on an edge-native service that will propel it to front runner status in the edge computing sector. All that considered, AWS is a best fit for:

- Highly extensible serverless application development, specifically microservices (i.e., FaaS) using AWS Lambda. This recommendation is made based on the maturity of the Lambda service, as well as its popularity, evidenced by the robust ecosystem of open source tools for enhancing software development with Lambda.
- Deploying serverless to the "cloud edge" (i.e., AWS' CDN) via Lambda@Edge. For example, DataDome has had success in leveraging this early-stage edge computing platform to dramatically decrease latencies for their customers [84].

Azure has been trailing AWS in the cloud computing market for over a decade. However, Azure is ahead of its competitors in real estate acquisition – which could signal a pivot towards increasing their edge computing capabilities – starting with an aggressive expansion of their CDN [85].

On the serverless side, Azure Functions' documentation is difficult to grasp without prior knowledge of the Azure ecosystem. As with AWS, the increasing complexity of Azure's APIs and configuration options can easily overwhelm new adopters. For example, customers can deploy their functions through *nine* different methods. Moreover, when these methods are broken down based on hosting platform (Windows or Linux) and hosting plan ("Consumption," "Premium," or "Dedicated") there are a total of 42 possible deployment scenarios for an Azure function [55].

Azure is best suited to organizations that are heavily invested in Microsoft technologies and committed to Microsoft as a strategic partner. Azure is mature enough not to warrant any re-deployment of a cloud infrastructure onto a different platform; nevertheless, given a choice, AWS Lambda and Google Cloud Functions are better options for running serverless applications.

While also facing pressure to catch up with AWS, Google Cloud Platform has taken a different approach to building their cloud computing services. Rather than optimize for options and customizability, Google Cloud Platform has optimized for core features and clear documentation. Google Cloud Platform stands out for the consistency, readability, navigability, and clarity of their documentation, and Cloud Functions is no exception. While it's true that the developer documentation on any cloud platform as massive as AWS, Azure, and Google Cloud Platform will be somewhat scattered, Google mitigates this by providing easy-to-follow steps for how services can be integrated, including code-snippets. Their best practices recommendations are thorough and even include code examples to make the implementation of these best practices more straightforward.

Of all the big three, Google seems to be the least focused on real estate acquisition. Instead, Google has been a leader in the development of virtualization software (starting as far back as the early 2000s) that has laid the foundation for serverless computing [2]. In 2014, Google launched Kubernetes, a container orchestration platform based on Docker. Since then, Kubernetes has become the favorite container orchestration platform among developers and is ubiquitous in CSP service offerings [86] [87] [88] [89]. More recently, Google developed another open source platform for building serverless applications, called Knative. Knative

combines the best of many worlds: Container-as-a-Service, container orchestration, and FaaS. As ISPs look to build their own edge computing infrastructure, deployment of platforms like Kubernetes and Knative may be prioritized for the fact that they are open source and thus have widescale support in the developer community. Google seems to be anticipating this prospect with the development of Google Anthos, a suite of open source software, including the Google Kubernetes Engine as well as a Knative-based platform for hosting serverless applications called Cloud Run, that is designed for deployment "in both cloud and on-premises environments." [90] Even more recently, Google has started looking into deploying Cloud Run on key AT&T access sites [91]. Thus, given its leadership in developing open source virtualization technology, Google Cloud is recommended for:

- Running Kubernetes-based applications in the cloud.
- Running applications that require advanced machine learning capabilities at the edge, as Google has also been a cultivator of open source machine learning technology that is lightweight enough for deployment to low compute edge devices.

6.4 Conclusion

Moving from monolithic code bases to microservice-based architecture is necessary to make applications more suitable for cloud deployment. Containerization of services is an important prerequisite for both scalability and security. A continuation of this trend will prepare applications and services for "edge-native" deployments, as modularized code bases are more easily distributed. However, it is important to note that a robust "Infrastructure Edge" does not yet exist, and even as it starts to materialize, demand for cloud computing services will persist [5].

The big three are here to stay and their cloud computing businesses will continue to thrive due to unceasing demand, large swaths of market share, and high profitability. In other words, a transition to "edge-native" software deployment models will not undermine the popularity of "cloud-native" services. These technologies complement each other, and will often be used in conjunction with each other, depending on the use case. Serverless applications are ideal for edge deployment, as the event-driven nature of Function-as-a-Service (FaaS) fits in nicely with the event-driven nature of data generation and consumption at the edge. While *serverless at the edge* technologies are still relatively primitive, their development is a clear sign that edge-native deployment models might very well be as ubiquitous as cloud-native deployment models in a decade's time, perhaps sooner.

7 References

[1] AWS, "Lambda@Edge," AWS, [Online]. Available: https://aws.amazon.com/lambda/edge/.

[2] EllMarquez, "The History of Container Technology," 2018. [Online]. Available: https://linuxacademy.com/blog/containers/history-of-container-technology/.

[3] Docker, "Docker security," [Online]. Available: https://docs.docker.com/engine/security/security/.

[4] OpenFaaS, "Introduction," OpenFaas Project, 2019. [Online]. Available: https://docs.openfaas.com/.

[5] State of the Edge, "State of the Edge 2020: A Market and Ecosystem Report for Edge Computing," State of the Edge, 2020.

[6] KeyCDN, "What is a CDN?," 2020. [Online]. Available: https://www.keycdn.com/what-is-a-cdn.

[7] S. Gossett, "What is Edge Computing?," 2020. [Online]. Available: https://builtin.com/cloud-computing/what-is-edge-computing.

[8] S. P. Choudary, "First-mover disadvantage: The challenges of hitting the market before it's ready," The Next Web, 13 July 2013. [Online]. Available: https://thenextweb.com/entrepreneur/2013/07/13/first-mover-disadvantage-the-challenges-of-hitting-the-market-before-its-time/. [Accessed 2020].

[9] B. Black, "EC2 Origins," 2009. [Online]. Available: http://blog.b3k.us/2009/01/25/ec2-origins.html.

[10] R. Miller, "How AWS came to be," [Online]. Available: https://techcrunch.com/2016/07/02/andy-jassys-brief-history-of-the-genesis-of-aws/.

[11] Statista, "Total size of the public cloud computing market from 2008 to 2020," [Online]. Available: https://www.statista.com/statistics/510350/worldwide-public-cloud-computing/.

[12] MarketsAndMarkets, "Serverless Architecture Market by Service type, Deployment Model, Organization Size, Verticals And Region - Global Forecast to 2023," [Online]. Available: https://www.reportlinker.com/p05486760/Serverless-Architecture-Market-by-Service-type-Deployment-Model-Organization-Size-Verticals-And-Region-Global-Forecast-to.html.

[13] E. J. Savitz, "Amazon Web Services Is Worth Half a Trillion Dollars, Analyst Estimates," 2019. [Online]. Available: https://www.barrons.com/articles/amazon-stock-web-services-worth-half-a-trillion-dollars-51559060451.

[14] Azure, "Edge Nodes," [Online]. Available: https://docs.microsoft.com/en-us/rest/api/cdn/edgenodes.

[15] Google Cloud, "Google Cloud networking in depth: Cloud CDN," [Online]. Available: https://cloud.google.com/blog/products/networking/google-cloud-networking-in-depth-cloud-cdn.

[16] B. Butler, "What is fog computing? Connecting the cloud to things," IDG Communications, Inc, 17 January 2018. [Online]. Available: https://www.networkworld.com/article/3243111/what-is-fog-computing-connecting-the-cloud-to-things.html.

[17] "Cloudlet," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Cloudlet.

[18] Google, "AT&T and Google Cloud Team Up to Enable Network Edge 5G Computing Solutions for Enterprises," Google, 2020. [Online]. Available: https://cloud.google.com/press-releases/2020/0305/google-cloud-att-collaboration.

[19] AWS, "AWS Wavelength," AWS, 2020. [Online]. Available: https://aws.amazon.com/wavelength/.

[20] Y. Sverdlik, "AWS Reveals Two Different Edge Data Center Plays at re:Invent," Data Center Knowledge, 2019. [Online]. Available: https://www.datacenterknowledge.com/edge-computing/aws-reveals-two-different-edge-data-center-plays-reinvent.

[21] F. Giust, G. Verin, K. Antevski, J. Chou, Y. Fang, W. Featherstone, F. Fontes, D. Frydman, A. Li, A. Manzalini, D. Purkayastha, D. abella, C. Wehner, K.-W. Wen and Z. Zhou, "MEC Deployments in 4G and Evolution Towards 5G," ETSI, 2018.

[22] L. Peterson and B. Davie, "Multi-Access Networks," in *Compute Networks: A Systems Approach*, 2012.

[23] Award Solutions, "Multi-Access Edge Computing (MEC)," in *5G Networks and Services*, Award Solutions, 2018.

[24] Athonet, "SGW-LBOsolution for MEC: Taking services to the Edge," Athonet, 2016.

[25] AWS, "AWS Lambda Limits," AWS, [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-limits.html.

[26] Google Cloud, "Background Functions," [Online]. Available: https://cloud.google.com/functions/docs/writing/background.

[27] AWS, "Partitions and Data Distribution," AWS, 2020. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/.

[28] Azure, "Azure Cosmos DB," Microsoft, 2020. [Online]. Available: https://azure.microsoft.com/en-us/services/cosmos-db/.

[29] AWS, "Secure and fast microVMs for serverless computing," AWS, 2020. [Online]. Available: https://firecracker-microvm.github.io/.

[30] Azure, "What is Service Fabric Mesh?," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/service-fabric-mesh/service-fabric-mesh-overview.

[31] OpenWhisk, "Documentation," Apache Software Foundation, 2016. [Online]. Available: https://openwhisk.apache.org/documentation.html.

[32] Cloudflare, "What Is a Serverless Microservice? | Serverless Microservices Explained," [Online]. Available: https://www.cloudflare.com/learning/serverless/glossary/serverless-microservice/.

[33] LiveOverflow, "Google Paid Me to Talk About a Security Issue!," [Online]. Available: https://www.youtube.com/watch?v=E-P9USG6kLs.

[34] Google, "Knative," Google, [Online]. Available: https://cloud.google.com/knative.

[35] Azure, "What is Service Fabric Mesh?," Microsoft, 2018. [Online]. Available: https://docs.microsoft.com/en-us/azure/service-fabric-mesh/service-fabric-mesh-overview.

[36] Google Cloud, "Cloud Functions Execution Environment," Google, [Online]. Available: https://cloud.google.com/functions/docs/concepts/exec.

[37] AWS, "Security in AWS Lambda," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-security.html.

[38] "Resilience in AWS Lambda," [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/security-resilience.html.

[39] M. Wolf, "Machine Learning + Distributed IoT = Edge Intelligence," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, 2019.

[40] "Micro data center," [Online]. Available: https://en.wikipedia.org/wiki/Micro_data_center.

[41] S. Kekki, W. Featherstone, Y. Fang, P. Kuure, A. Li, A. Ranjan, D. Purkayastha, F. Jiangping, D. Frydman, G. Verin, K.-W. Wen, K. Kim, R. Arora, A. Odgers, L. ontreras and S. Scarpina, "MEC in 5G networks," ETSI, 2018.

[42] Radware, "A Brief History of IoT Botnets," Radware, 1 March 2018. [Online]. Available: https://blog.radware.com/uncategorized/2018/03/history-of-iot-botnets/.

[43] AWS, "AWS IoT Greengrass," [Online]. Available: https://aws.amazon.com/greengrass/.

[44] Telecom Infra Project, "EAD," [Online]. Available: https://telecominfraproject.com/ead/.

[45] Open Edge Computing Initiative, "Open Edge Computing Initiative," [Online]. Available: https://www.openedgecomputing.org/.

[46] Vapor IO, "Vapor IO Exits Stealth to Provide Industry's First Intelligent, Hyper Collapsed Data Center," 2015. [Online]. Available: https://www.vapor.io/vapor-exits-stealth/.

[47] Vapor IO, "Service Offering - Vapor IO," Vapor IO, 2019. [Online]. Available: https://www.vapor.io/service-offering/. [Accessed 2020].

[48] Vapor IO, "Kinetic Edge - Vapor IO," Vapor IO, 2019. [Online]. Available: https://www.vapor.io/kinetic-edge/. [Accessed 2020].

[49] Vapor IO, "Vapor IO Partners with Cloudflare on Nationwide Deployment," Vapor IO, 22 January 2020. [Online]. Available: https://www.vapor.io/vapor-io-partners-with-cloudflare-on-nationwide-deployment/. [Accessed 2020].

[50] MobiledgeX, "About Us," [Online]. Available: https://mobiledgex.com/about.

[51] MobiledgeX, "Cloudlets and MobiledgeX," 2020. [Online]. Available: https://mobiledgex.com/product/cloudlets.

[52] MobiledgeX , "MobiledgeX and TELUS to trial mobile edge network in Canada," MobiledgeX , May 2019. [Online]. Available: https://mobiledgex.com/press-

releases/2019/05/07/mobiledgex-and-telus-to-trial-mobile-edge-network-in-canada. [Accessed 2020].

[53] TELUS, "TELUS and MobiledgeX first in North America to unleash live edge computing access," 2019. [Online]. Available: https://www.telus.com/en/about/news-and-events/media-releases/telus-and-mobiledgex-first-in-north-america-to-unleash-live-edge-computing-access.

[54] S. Conway, "Cloud Native Technologies Are Scaling Production Applications," 2017. [Online]. Available: https://www.cncf.io/blog/2017/12/06/cloud-native-technologies-scaling-production-applications/.

[55] Azure, "Deployment technologies in Azure Functions," Microsoft, 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-deployment-technologies.

[56] Google Cloud, "Cloud Functions Execution Environment," Google, 2020. [Online]. Available: https://cloud.google.com/functions/docs/concepts/exec.

[57] Google Cloud, "Controlling Scaling Behavior," Google, 2019. [Online]. Available: https://cloud.google.com/functions/docs/max-instances.

[58] R. Walikar, "An SSRF, privileged AWS keys and the Capital One breach," [Online]. Available: https://blog.appsecco.com/an-ssrf-privileged-aws-keys-and-the-capital-one-breach-4c3c2cded3af .

[59] Google Cloud, "VPC Service Controls," [Online]. Available: https://cloud.google.com/vpc-service-controls.

[60] M. Roberts, "Serverless Architectures," 2018. [Online]. Available: https://martinfowler.com/articles/serverless.html.

[61] S. Maarek, "AWS Lambda and the Serverless Framework - Hands On Learning!," Udemy, 2020.

[62] AWS, "Identity and Access Management for AWS IoT Greengrass," [Online]. Available: https://docs.aws.amazon.com/greengrass/latest/developerguide/security-iam.html.

[63] Google Cloud, "Authenticating Developers, Functions, and End-users," [Online]. Available: https://cloud.google.com/functions/docs/securing/authenticating.

[64] S. Walkowski, "Cloud Functions pro tips: Building idempotent functions," [Online]. Available: https://cloud.google.com/blog/products/serverless/cloud-functions-pro-tips-building-idempotent-functions.

[65] Azure, "Optimize the performance and reliability of Azure Functions," 2019. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-best-practices#scalability-best-practices .

[66] Google Cloud, "Authenticating Developers, Functions, and End-users," 2020. [Online]. Available: https://cloud.google.com/functions/docs/securing/authenticating.

[67] Google Cloud, "Architectural overview of Cloud Endpoints," 2020. [Online]. Available: https://cloud.google.com/endpoints/docs/openapi/architecture-overview.

[68] "CLI tool to build, test, debug, and deploy Serverless applications using AWS SAM," [Online]. Available: https://github.com/awslabs/aws-sam-cli.

[69] AWS, "Working with AWS Lambda Functions in the AWS Cloud9 Integrated Development Environment (IDE)," [Online]. Available: https://docs.aws.amazon.com/cloud9/latest/user-guide/lambda-functions.html.

[70] C. Munns, "Implementing safe AWS Lambda deployments with AWS CodeDeploy," Amazon, 2018. [Online]. Available: https://aws.amazon.com/blogs/compute/implementing-safe-aws-lambda-deployments-with-aws-codedeploy/.

[71] AWS, "Deploying Serverless Applications Gradually," AWS, [Online]. Available: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/automating-updates-to-serverless-apps.html.

[72] J. Rundle, "Human Error Often the Culprit in Cloud Data Breaches," Dow Jones & Company, Inc., 2019. [Online]. Available: https://www.wsj.com/articles/human-error-often-the-culprit-in-cloud-data-breaches-11566898203.

[73] T. Klopfenstein and M.-N. L. , "Smart Bird Feeder," 2019. [Online]. Available: https://coral.ai/projects/bird-feeder.

[74] J.-L. Gassée, "Cheap IOT Threatens The Internet," [Online]. Available: https://mondaynote.com/cheap-iot-threatens-the-internet-c7b44ab390f9.

[75] AWS, "AWS Lambda," AWS, [Online]. Available: https://aws.amazon.com/lambda/.

[76] AWS, "Run a Serverless "Hello, World!"," AWS, [Online]. Available: https://aws.amazon.com/getting-started/hands-on/run-serverless-code/.

[77] Azure, "The future of computing: intelligent cloud and intelligent edge," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/overview/future-of-cloud/.

[78] EdgeConneX, "Boston Data Center," EdgeConneX, 2020. [Online]. Available: https://www.edgeconnex.com/locations/north-america/boston-ma/.

[79] S. I. Fulton, "How hyperscale data centers are reshaping all of IT," ZDNet, 5 April 2019. [Online]. Available: https://www.zdnet.com/article/how-hyperscale-data-centers-are-reshaping-all-of-it/.

[80] Y. Cui, "You are wrong about serverless vendor lock-in," Lumigo, 2019. [Online]. Available: https://lumigo.io/blog/you-are-wrong-about-serverless-vendor-lock-in/.

[81] C. Majors, "Operational Best Practices #serverless," 2016. [Online]. Available: https://charity.wtf/2016/05/31/operational-best-practices-serverless/.

[82] AWS, "Regions and Availability Zones," AWS, [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/regions_az/.

[83] "AWS Local Zones," [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/localzones/.

[84] DataDome and B. Fabre, "AWS re:Invent 2017: Taking Serverless to the Edge (SRV312)," AWS, 2017. [Online]. Available: https://www.youtube.com/watch?v=3iknsVpfYr0&feature=youtu.be&t=2098.

[85] Azure, "Azure global network," Microsoft, [Online]. Available: https://azure.microsoft.com/en-us/global-infrastructure/global-network/.

[86] A. Yigal, "The Rise of Kubernetes in 2017," 2020. [Online]. Available: https://logz.io/blog/rise-kubernetes-2017/.

[87] Azure, "Azure Kubernetes Service (AKS)," [Online]. Available: https://azure.microsoft.com/en-us/services/kubernetes-service/.

[88] AWS, "Amazon Elastic Kubernetes Service," [Online]. Available: https://aws.amazon.com/eks/.

[89] Google Cloud, "Google Kubernetes Engine," [Online]. Available: https://cloud.google.com/kubernetes-engine/.

[90] Google Cloud Platform, "Anthos," [Online]. Available: https://cloud.google.com/anthos.

[91] S. F. III, "Why Google Cloud and AT&T May Merge their Telco Edges," Informa USA, Inc, 16 March 2020. [Online]. Available: https://www.datacenterknowledge.com/edge-computing/why-google-cloud-and-att-may-merge-their-telco-edges.

[92] P. Koutoupis, "Everything You Need to Know about Linux Containers, Part I: Linux Control Groups and Process Isolation," 2018. [Online]. Available: https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-i-linux-control-groups-and-process.

[93] "Linux namespaces," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Linux_namespaces.

[94] S. Kenlon, "Demystifying namespaces and containers in Linux," 2019. [Online]. Available: https://opensource.com/article/19/10/namespaces-and-containers-linux.

[95] M. Kerrisk, "Namespaces in operation, part 1: namespaces overview," 2013. [Online]. Available: https://lwn.net/Articles/531114/.

[96] M. Ridwan, "Separation Anxiety: A Tutorial for Isolating Your System with Linux Namespaces," [Online]. Available: https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces.

[97] Google, "Google aims to deliver its services with high performance, high reliability, and low latency for users, in a manner that respects open internet principles.," [Online]. Available: https://peering.google.com/#/infrastructure.

[98] "Modular Data Centers: Is Anyone Buying?," [Online]. Available: https://www.wired.com/insights/2012/02/modular-data-centers/.

[99] Azure, "Deployment technologies in Azure Functions," [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-deployment-technologies#portal-editing.

[100] Azure, "Deployment technologies in Azure Functions," [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-deployment-technologies#external-package-url.

[101] Azure, "Deployment technologies in Azure Functions," Microsoft, [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/functions-deployment-technologies#local-git.

[102] N. Wingfield, "Amazon's Cloud Business Lifts Its Profit to a Record," New York Times, 28 April 2016. [Online]. Available: https://www.nytimes.com/2016/04/29/technology/amazon-q1-earnings.html.

[103] Wikipedia, "Information Security," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Information_security#Key_concepts.

[104] L. Ablon and A. Bogart, "Zero Days, Thousands of Nights," RAND Corporation, Santa Monica, California, 2017.

[105] "Cloudlet," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Cloudlet.

[106] "OpenStack," 2020. [Online]. Available: https://en.wikipedia.org/wiki/OpenStack.

[107] Google, "Firebase," [Online]. Available: https://firebase.google.com/.

[108] Docker Inc., "Docker Hub," 2020. [Online]. Available: https://hub.docker.com/search?q=&type=image&category=languages&page=1.

Appendix A Glossary of Terms

5G	Fifth generation of cellular network technology.
Application Programming Interface (API)	A collection of methods (functions) that define the interface to an application's back-end logic, which developers can then utilize to integrate that application's functionality into their own service.
Asynchronous	Code that is not executed serially (meaning asynchronous code may be executed out of order depending on what the execution engine decides to do). In the context of the web, Javascript is widely used because of its asynchronous properties. This is optimal for the web because users cannot tolerate waiting for a long task (like GET-ing data from a web server) to finish before receiving feedback from a webpage that their request has at least been acknowledged. Thus, the Javascript engine will not pause the execution of a script while it waits for a server to respond to a GET request. Rather, it forwards the request to the server, moves on to subsequent code, and eventually comes back to the response-handling code via a "callback" once the server has responded.
Augmented Reality (AR)	An interactive experience where objects that reside in the real world are enhanced by computer-generated information.
Autoscaling	Application monitoring that automatically adjusts capacity to maintain steady, predictable performance at low cost.
Backend as a Service (BaaS)	Cloud service model in which the internals of a web or mobile application is outsourced so the developers only need to maintain the front-end/user interface.
Bare metal server/Instance	A server that is not running (or hosting) any virtualized environments. Applications are deployed on bare metal typically when high performance is critical (and thus the overhead of virtualization cannot be tolerated) and cost of compute is less important. **Examples***:* EC2 Bare Metal instance (AWS)
Batch processing	Processing of datasets (batches) that are constantly changing (like logs) in a scheduled fashion that does not require any user intervention.

Binding	In the context of serverless, the linking (i.e. "binding") of a function to a trigger (typically an event-emitting cloud resource). For example, a serverless function can be bound to a cloud-hosted database and triggered by an update to that database. Mature serverless platforms (like AWS Lambda and Azure Functions) have built-in support for bindings to their own resources, thus reducing the need to hard-code event listeners.
Cloud, Cloud computing	On-demand availability of compute resources without direct active management by the user/customer.
Cloud Edge	Cloud services at points of presence closer to the consumer/producer of the data involved.
Cloudlet	See entry for Edge data center.
Cloud-native	Describes any type of software (be it a platform, framework, developer tool, IDE, etc.) that is specifically designed to be deployed in a hyperscale data center. This term is also used to describe vendor-native deployments, like "AWS-native" or "Azure-native."
Cluster	(Container Cluster) A dynamic system that manages containers, grouped together in pods, running on compute nodes, along with all the internetworking.
Cold start	In serverless computing, a serverless function will require a "cold start," meaning its entire runtime, environment store, and application logic must be loaded into memory if it has not been previously run in a given timeframe. Depending on the runtime, cold starts can negatively impact performance, thus it is important for developers to think carefully about the runtime they choose for their serverless functions. Developers can also choose to implement optimizations to minimize the number of cold starts that must take place.
Concurrency	When a computer is executing multiple tasks over the same time span. Depending on if the computer is a multi-core or single-core machine, these concurrent tasks may or not actually be executing at the same time. In a multi-core machine, concurrent tasks are typically executed *in parallel*, meaning these tasks (threads) are handed off to different CPUs, which in turn execute their own thread, thus achieving simultaneous execution of multiple tasks. In a single core machine, tasks *appear* to run concurrently through the OS's graceful use of task switching, though in

	reality each thread must pause periodically in order to efficiently share the single CPU's resources. In the context of serverless, concurrency means that multiple functions are executing at the same time—they may or may not be executing on the same host machine, though this detail is abstracted away by the vendor so the developer will never know. A **concurrency limit** is typically a cap on the number of function instances that can execute simultaneously in response to a "normal" rate of requests. Scaling up to the default concurrency limit usually happens linearly (AWS). A **burst concurrency limit** is a cap on the number of functions that will be instantiated as a result of exponential scaling that is triggered by a large spike in invocations (AWS).
Container	A bundle of application code and all its related configuration files, libraries, dependencies and a minimal runtime environment. A container is typically a writeable layer of software running on a process instantiated from a very lightweight read-only Linux image. Most container implementations are, by default, strictly isolated from the host OS.
Container orchestration	The automatic process of managing and scheduling the work of individual containers and their applications throughout a cluster.
Content Delivery Network (CDN)	A geographically distributed network of servers that provides high availability and performance through caching of data closer to end-users.
Continuous Delivery (CD)	In software development, an automation workflow to deploy an application from a software repository into production, usually by a trigger event.
Continuous Integration (CI)	In software development, an automation workflow to build and test applications whenever new commits are pushed to a software repository.
Control Group (cgroup)	A feature in the Linux kernel that enables the assigning of one or more processes to a controller, creating a Control Group (cgroup) to which the controller allocates a limited amount of compute and memory. This is often viewed as a security feature in the kernel, as it can contain the impact of a rogue process to that process' Control Group, thus

	mitigating any adverse impact on the rest of the OS and its other subsystems.
Docker	An open source platform for developing, managing, deploying, and distributing containerized applications (either locally or in the cloud).
Docker Swarm	A Docker-made tool for the management of a multi-container environment (aka “a cluster”).
Edge, Edge Computing	A distributed computing approach to bring compute and data closer to the consume/producer to improve performance and save bandwidth
Edge device	A compute resource that is close to an end user. The precise definition of an “edge device” varies greatly, but for the purpose of this paper, an edge device is a computer with a direct connection to a Local Area Network (LAN) thus giving it the ability to respond to other hosts on the LAN very quickly.
Event	A collection of data (usually a JSON object) that is transmitted to a serverless function upon a trigger firing, which the function then processes upon its invocation. A serverless function will typically treat this as its input. In the context of serverless, the term “event” is usually used interchangeably with the idea of a “trigger,” but they are two separate things. While a trigger is tied to a specific function and says: *this is the kind of thing that will need happen for me (the trigger) to invoke my function*, an event is more or less decoupled from a function and, whenever a trigger fires, says: *here is some interesting data for a function to process.* For example, in an IoT Greengrass/Lambda system, if a trigger is set to be a rise in room temperature, a possible corresponding event would be a temperature (like 78 degrees) that is sensed by a connected thermostat, encoded in JSON, and sent to a nearby Lambda instance for processing.
Event-driven	A programming model that requires code to only be executed in response to a significant event (which could be anything from a user clicking a button to the uploading of a file to an S3 bucket). There are lots of systems built around this paradigm. Web browsers and web servers, for example, are event driven. If a user clicks a button on a web page, a chunk of code listening for that event will be invoked. Serverless functions are largely event driven—as

	they are only "awoken" upon receiving an event (like an HTTP request) tied to a trigger (like an API Gateway configured to route certain HTTP requests to a particular function).
Event-driven processing	A model for processing data that aims to conserve compute and bandwidth by dictating that data processing modules only run upon the firing of *events* that meet a certain threshold of significance.
Event source	A change, be it in the physical world (an end user submitting a form) or logged in a computer system (an update to a record in a database), that sets off a trigger, which in turn, invokes a serverless function.
Execution environment	An isolated runtime that is spun up on the fly in response to the invocation of a serverless function. It contains all the resources required by the function to perform its task and is typically torn down immediately after the function terminates, so to ensure that the function is executed as efficiently as possible and that no customer data is leftover on the host machine (that could be vulnerable to compromise if the machine is hosting multiple tenants).
Executor	This is a key component of most FaaS infrastructures. It is a process that accepts events from a gateway, and in turn invokes the Serverless functions to which the event's trigger is bound by spinning up an execution environment on an available worker node (server). It is also referred to as an "Invoker."
Fog computing	A model for internet architectures that enables the integration of services and data collected at the edge with the cloud.
Function	A modular block of code (typically deployed to serverless platform) that is invoked in response to a trigger firing. A trigger, when fired, passes an event to the function. The event can be thought of as the argument to the function—a chunk of input data which the function is responsible for doing something with.
Function as a Service (FaaS)	A cloud computing service that allows developers to write modular chunks of code (functions) and deploy them to a platform that is almost entirely managed by a Cloud Service Provider (CSP). Functions deployed to FaaS platforms are typically lightweight and event driven, so scaling them up and down is cost efficient.

Hybrid cloud	A compute infrastructure that is partially located in the cloud and partially located at the "edge" (usually this means "on premises").
Hypervisor	Also known as a Virtual Machine Monitor (VMM). Software for running a Virtual Machine (VM). Its job is to act as the middleman between a host OS and a VM running on the host to ensure the VM is thoroughly isolated from the host (mainly through provisioning of resources).
Image	A file that contains a copy of a computer system.
Image (Docker)	A file generated by Docker that contains all the information (runtime environment, libraries, dependencies, application code, etc.) needed to instantiate a Docker container. Typically, a Docker image is comprised of several layers, most notably the base image (a very lightweight Linux VM image) and the container layer (the sandboxed application).
Infrastructure-as-a-Service (IaaS)	A model for cloud computing in which the cloud service provider provides the bare-bones infrastructure (hardware, networking, and host operating systems) necessary for customers to run their applications in the cloud.
Instance (Docker)	A Docker container that is spun up at runtime via the instantiation of a Docker image. FaaS functions typically run in container instances.
Instance (Virtual Machine (VM))	A computer program simulating an operating system that is different from the one it is running on. We typically use the term Virtual Machine to describe an instantiated image of an operating system running on top of another (host) operating system. A VM can appear as if it is a genuine operating system through the use of a Hypervisor, another program that contains all of the "magic" to efficiently allocate some of the host's compute, memory, and storage resources to the VM. As far as the host is concerned, the VM is just a resource-intensive process, and as far as the VM is concerned, there is nothing else to see on the host aside from the resources to which it has been given access.
Internet of Things (IoT)	The explosion of the number of Internet-connected devices that is the result of the modern trend of embedding cheap processors and Network Interface Cards (NICs) into consumer goods like toys, appliances, cars, watches, thermostats, etc.
Keeping warm	A technique used by some serverless developers to minimize the number of cold starts their functions will

	require upon intermittent invocations. One way to keep a function "warm" is to run a cronjob that invokes a function at steady intervals to ensure it stays loaded in memory and can therefore have the fastest possible response time when it receives a request from an actual end user.
Kubernetes	Initially developed by Google, Kubernetes is a popular open source platform for container orchestration that now serves as the foundation for many open source serverless platforms.
Last mile problem	An obstacle to scaling next-generation networking technologies (like 5G and edge computing) that comes from the fact that most end users do not a fiber connection to their ISPs' edge routers.
Latency	How long it takes a packet to reach its destination from its source.
Microservice	In software development, an arrangement of application code as a coupled set of services instead of a singular monolithic application.
Namespace	A set of contextual information about the kernel that a process running on a Linux system relies on to execute mission critical services like networking, data storage, and inter-process communication. In Linux, a namespace can refer to a namespace type (there are currently seven namespace types in Linux systems) or an instance of a particular type of namespace assigned to a particular process.
Network edge	This is a relative term but usually means the edge of an ISP's network.
On-premise	Defines the installation a compute resource (i.e. service or device) at the location of the end user (usually this means within a Local Area Network).
Pay-as-you-go	A pricing model used by many cloud service providers in which the vendor only charges the customer for compute used. Typically, this means the customer is not responsible for explicitly starting up or shutting down an instance to prevent over-billing when that instance is not in use. Rather, the vendor will auto-calculate charges in response to customer code (like a function or microservice) being invoked. This pricing model (of which pay-per-request is a sub-category) is ideal for serverless.

Pay-per-request	Pricing model used by many Serverless vendors (AWS, Azure, Google Cloud Platform, to name a few) that stipulates the customer must only pay for the time it takes to execute their function that is invoked in response to a user request. Essentially, the clock starts when the request handler is invoked and ends when the handler terminates.
Scale-to-zero	The ability of a serverless function to scale down to zero instances during a period of zero demand, thereby maximizing cost efficiency. This is a key component of serverless, as function instantiation requires memory consumption which is costly and therefore should be limited.
Serverless, serverless computing	A cloud computing service that allows its customers to deploy application code to a Cloud Service Provider's infrastructure that is entirely managed by the provider (from bare metal to containerization). The term "serverless" is misleading because it implies there is no physical compute resource underneath a serverless application. This is never the case. Rather, we can think of serverless as describing a paradigm for application development in which the physical compute resource is completely abstracted away from the application developer.
Stateless	While every process executed by a CPU must consume at least a small amount of virtual memory (and therefore has some "state" during runtime), a stateless application is one that does not write nor read any data to/from storage and/or a cache in memory when it is executed, and hence has no persistent state that can be accessed by another process.
Trigger	A generic user action that kicks off a chain of events to invoke a serverless function to which it is bound.
Virtual Machine (VM)	See entry for Instance (Virtual Machine).
Virtual Machine Monitor (VMM)	See entry for Hypervisor.
Virtual network	A subnet (created by a cloud service provider and configured by the cloud customer) that, when assigned to a serverless function, isolates the microservice from the public internet by giving it a private IP address. If the function requires access to the internet, virtual networks can add an additional layer of security via features like Network Address Translation (NAT) and IP whitelisting. **Synonyms:** Virtual Private Cloud (AWS), VNet (Azure)

Warm start	In serverless computing, if a function has run within a given timeframe and thus may have been cached or have cached data available (i.e. its code is still instantiated in memory), then it can be invoked straight from memory to improve execution time (thus reducing latency for the end user). Straight-from-memory invocation is called a "warm start." Warm starts are typically much faster than "cold starts" as the costly operation of loading the entire function runtime from disk into memory is entirely avoided. Serverless vendors offer various advanced methods to increase the likelihoods of warm starts, including "provisioned concurrency" (AWS) and "pre-warming" (Azure).

This page intentionally left blank.

Appendix B Abbreviations and Acronyms

3G / 4G / 5G	Third / Fourth / Fifth Generation (cellular networks)
AR	Augmented Reality
AWS	Amazon Web Services
CDN	Content Delivery Network
CI/CD	Continuous Integration / Continuous Deployment
CLI	Command Line Interface
CORS	Cross-Origin Resource Sharing
CSP	Cloud Service Provider
DDIL	Denied, Disrupted, Intermittent, and Limited
EC2	Elastic Compute Cloud
ETSI	European Telecommunications Standards Institute
FaaS	Function as a Service
IaaS	Infrastructure as a Service
IAM	Identity Access Management
IDE	Integrated Development Environment
IoT	Internet of Things
ISP	Internet Service Provider
LAN	Local Area Network
MEC	Multi-Access Edge Computing (formally Mobile Edge Computing)
MQTT	Message Queuing Telemetry Transport
NFV	Network Functions Virtualization
OS	Operating System
PID	Process Identifier
PoP	Point of Presence
RAM	Random Access Memory
RBAC	Role-based Access Control
S3	Simple Storage Service
SAM	Serverless Application Model
TLS	Transport Layer Security

UX	User Experience
VM	Virtual Machine
VPC	Virtual Private Cloud
VR	Virtual Reality