Serverless Cloud Computing: State of the Art and Challenges

Vincent Lannurien, Laurent D'orazio, Olivier Barais, Jalil Boukhobza

Serverless Cloud Computing: State of the Art and Challenges

Vincent Lannurien, Laurent D'Orazio, Olivier Barais and Jalil Boukhobza

Abstract The serverless model represents a paradigm shift in the cloud: as opposed to traditional cloud computing service models, serverless customers do not reserve hardware resources. The execution of their code is event-driven (HTTP requests, cron jobs, etc.) and billing is based on actual resource usage. In return, the responsibility of resource allocation and task placement lies on the provider. While serverless in the wild is mainly advertised as a public cloud offering, solutions are actively developed and backed by solid actors in the industry to allow the development of private cloud serverless platforms. The first generation of serverless offerings, "Function as a Service" (FaaS), has severe shortcomings that can offset the potential benefits for both customers and providers – in terms of spendings and reliability on the customer side, and in terms of resources multiplexing on the provider side. Circumventing these flaws would allow considerable savings in money and energy for both providers and tenants. This chapter aims at establishing a comprehensive tour of these limitations, and presenting state-of-the-art studies to mitigate weaknesses that are currently holding serverless back from becoming the *de facto* cloud computing model. The main challenges related to the deployment of such a cloud platform are discussed and some perspectives for future directions in research are given.

Vincent Lannurien
b<>com Institute of Research and Technology, ENSTA Bretagne, Lab-STICC, CNRS, UMR 6285, Brest, e-mail: vincent.lannurien@ensta-bretagne.org

Laurent D'Orazio
Univ. Rennes, Inria, CNRS, IRISA, b<>com Institute of Research and Technology, e-mail: laurent.dorazio@irisa.fr

Olivier Barais
Univ. Rennes, Inria, CNRS, IRISA, b<>com Institute of Research and Technology, e-mail: olivier.barais@irisa.fr

Jalil Boukhobza
ENSTA Bretagne, Lab-STICC, CNRS, UMR 6285, Brest, b<>com Institute of Research and Technology, e-mail: jalil.boukhobza@ensta-bretagne.fr

1 Introduction

In 1961, John McCarthy imagined that the time-sharing of computers could make it possible to sell their execution power as a service, just like water or electricity [50]. Due to the democratization of high-speed Internet access in the mid-2000s, McCarthy's idea was implemented in what is known as *cloud computing*: companies and individuals can now drastically reduce the costs associated with the purchase and maintenance of the hardware needed to run their applications by delegating responsibility for the infrastructure to service providers. This model is called "Infrastructure as a Service" (IaaS) [78].

Over the years, new trends appeared with the aim of reducing the customer's responsibilities. For example, in the "Platform as a Service" (PaaS) model, customers do not have direct access to the machines that support their applications and perform most of the management tasks via specialized interfaces. In these models, the customer pays for resources that are sometimes dormant. This is because reserved resources must often be over-provisioned in order to be able to absorb the surge in activity and handle hardware failures [52].

The serverless model represents a paradigm shift in the cloud: as opposed to traditional models, serverless customers do not reserve hardware resources. The execution of their code is event-driven (HTTP requests, cron jobs, etc.) and billing is based on actual resource usage. In return, the responsibility of resource allocation and task placement lies with the provider [106].

While serverless in the wild is mainly advertised as a public cloud offering (table 3), solutions are actively developed and backed by solid actors in the industry to allow the development of private cloud serverless platforms (table 4).

The first generation of serverless offerings, "Function as a Service" (FaaS), has severe shortcomings that can offset the potential benefits for both customers and providers – in terms of spendings and reliability on the customer side, and in terms of resources multiplexing on the provider side. Circumventing these flaws would allow considerable savings in money and energy for both providers and customers. This chapter aims at establishing a comprehensive tour of these limitations and presenting state-of-the-art studies to mitigate weaknesses that are currently holding serverless back from becoming the *de facto* cloud computing model.

This chapter is organized as follows: after an introduction, some background and motivations related to serverless computing are given. Those include an introduction to cloud computing and virtualization technologies. Then, the serverless paradigm is introduced: we discuss its characteristics, benefits and offerings for both public and private cloud. A state of the art related to the challenges on resources management for serverless computing is then drawn. It is built around six issues: cold start, cost of inter-function communication, local state persistence, hardware heterogeneity, isolation and security, and programming model and associated risks of vendor lock-in. Finally, we introduce perspectives and future directions for research in the field of serverless computing.

---

[2] `https://www.redhat.com/en/topics/cloud-computing/iaas-vs-paas-vs-saas`

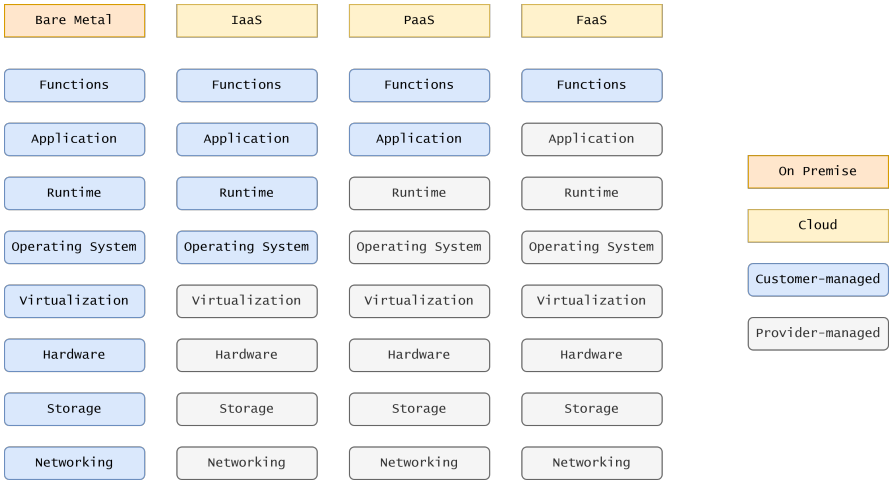

Fig. 1 Comparison of the different cloud service models in terms of customer responsibility (inspired from Red Hat's documentation [2])

2 Background and motivations

In this first section, we introduce characteristics of cloud computing; its service models and associated technologies. We also take a glance at recent transformations in the ways developers program for the cloud, and the consequences in terms of applications deployment.

2.1 The promises of cloud computing

The NIST definition of cloud computing [82] gives five essential characteristics for the cloud computing model, as opposed to on-premises and/or bare metal deployments:

- **On-demand self-service** – Customers can book hardware resources through e.g. a web interface rather than by interacting with operators. In return, they usually have limited control over the geographic location for these resources;
- **Broad network access** – These resources are immediately made available over public broadband connections by the provider;
- **Resource pooling** – Compute time, storage capacity, network bandwidth are all shared between customers. Virtualization techniques are used to ensure isolation between workloads;
- **Rapid elasticity** – Applications can benefit from increased or decreased computing power through scaling, as resources are dynamically provisioned and released to absorb variations in demand;

- **Measured service** – Cloud providers instrument their infrastructures so as to precisely monitor resource usage. Customers can then be charged in a fine way according to their needs.

These characteristics are found in three cloud service models, as shown in Figure 1:

- **Software as a Service** (SaaS) – Targets the end user by offering access to fully managed software (the application);
- **Platform as a Service** (PaaS) – Targets developers and DevOps teams who want to deploy their applications quickly without managing servers, generally through the use of containers. Customers deploy their applications on top of a runtime environment that is managed by the provider;
- **Infrastructure as a Service** (IaaS) – Targets architects and system administrators who want fine-grained control over their infrastructures. IaaS customers are in charge of their own servers, usually virtual.

It is debatable how much traditional IaaS and PaaS cloud offerings hold up to their promise regarding rapid elasticity and measured service. SaaS is out of the scope of this study, as it targets end users rather than application developers.

Elasticity, in the sense of "automatic scaling", intrinsically cannot be offered by IaaS or PaaS deployments. Developers are responsible for planning ahead and specifying their needs, that is, booking an adequate quantity of resources [4]. These are usually called "instances" by cloud providers. Cloud instances are typically distinguished according to their specifications in resource type and capacity: for example, there can be instances with many CPU cores, whereas others provide access to a GPU or some other hardware accelerator.

The choice of instance type(s) and quantity for an application depends on a) the nature of the computations it runs, and b) the acceptable latency and the desired throughput [134]. However, it is the customer's responsibility to not over-provision beyond their actual need.

This offering design has further consequences. First, it means that billing is done coarse-grained: per instances booked rather than per resources actually used. Besides, idle resources are always paid for, and scaling to zero cannot be achieved in this setting, because an application will always require at least one instance running to handle an incoming request.

Cloud computing platforms have to accommodate for an important number of customer jobs, leading to massive multitenancy which requires adequate isolation and virtualization techniques. Those are introduced in the next section.

2.2 Virtualization technologies

Multitenancy is a defining characteristic of cloud computing. It is the ability for a cloud provider to share resources between multiple customers to secure cost savings

[130]. As resources are pooled and different applications make use of them, it creates direct channels between processes in the user space: multitenancy comes with the responsibility for the provider to guarantee privacy and security across the different workloads of the customers [124].

To meet these guarantees, providers must resort to protection measures providing airtightness between processes that are not supposed to be aware of each others. This mechanism of transparently presenting separate execution environment with distinct address space, filesystem and permissions is called isolation [40]. For this purpose, providers may rely on virtualization technologies.

Virtualization is an isolation technique that allows one to run an application within the boundaries of a secure execution environment, called a *sandbox*, by introducing a layer of indirection between the host platform and the application itself [113].

Virtualization of the host resources can be done with virtual machines (VMs) or containers. These sandboxes give the underlying processes the impression of having an entire machine at hand, but while VMs virtualize the physical resources of the host, relying on the CPU's architecture to achieve isolation, containers depend on the host operating system's system call API to isolate workloads [77].

These techniques are beneficial both on the provider and the developer side. The former leverages virtualization to achieve isolation of customers' workloads as well as flexibility to manage scaling of sandboxes given a finite amount of hardware resources. The latter organizes their development pipelines so as to replicate a production-like environment during development stages, and to deliver and deploy their products.

Virtualization became such a cornerstone in cloud computing that Kubernetes [30], an orchestration system that leverages containers[3] to manage applications from deployment to scaling of services, is increasingly referred to as "the operating system for the cloud" [103].

When choosing the isolation model they want to rely on to achieve multitenancy, cloud providers have to trade off between performance and security. Containers are frequent targets of privilege escalation attacks ([137], [16]) but they perform multiple orders of magnitude better than virtual machines: containers startup time is in the hundreds of milliseconds, while VMs boot in seconds [77]. Designing lightweight virtual machines that offer performances comparable to containers is a critical research topic ([1], [10]).

2.2.1 Virtual Machines

Virtual machines virtualize the physical resources of the host: hardware-assisted virtualization allows multiple full-fledged *guest* operating systems to run independently on shared physical resources, regardless of the nature of the *host* operating system [64].

[3] Note that Kubevirt [31] aims at making Kubernetes suitable for VM workloads.

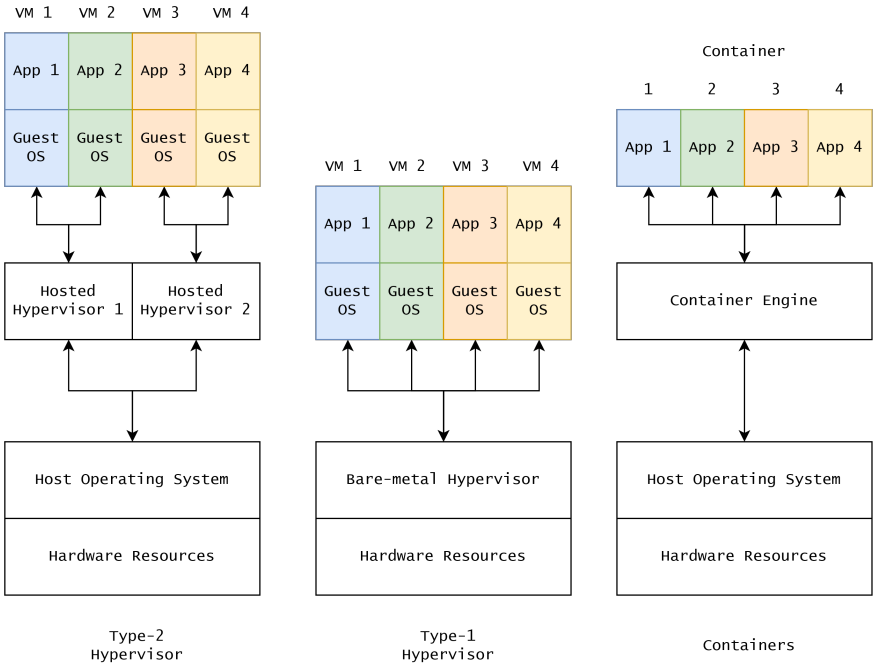

Fig. 2 Overview of different isolation models: virtualization and containerization

From a VM point of view, the execution sandbox is seen as a complete platform, while it actually is a subset of the host computer resources, determined by the hypervisor (or VMM for Virtual Machine Manager), a low-level software that can run on bare metal or as a process of the host operating system.

The hypervisor has the responsibility to manage VM lifecycle: creation, execution, destruction, and sometimes migration of virtual machines are handled by the hypervisor.

Hypervisors exist in two different abstractions, as shown in figure 2:

- Type-1 (bare-metal) hypervisors run directly on the host machine's hardware. They rely on the host's CPU support for virtualization. Given that they do not depend on an underlying operating system, they are considered more secured and efficient than their hosted counterpart. Common examples of type-1 hypervisors include VMware ESXi [127], KVM [71], Xen [70] and Hyper-V [85];
- Type-2 (hosted) hypervisors run on top of an operating system. These hypervisors are consumer-grade products that provide a convenient way for end users to run systems or programs that would otherwise not be supported by their hardware or OS. Examples of type-2 hypervisors include QEMU [102] and Oracle VirtualBox [94].

2.2.2 Containers

Containerization is an OS-level virtualization technique. The host operating system's kernel is responsible for the allocation of resources. Containers virtualize the OS: they give the containerized process the impression of having the entire machine at hand, while actually being constrained and limited regarding resource utilization by the host kernel [17].

From the running application's point of view, the execution platform behaves as if it were bare metal. However, its allocated resources are actually a virtualized subset of the host's hardware resources.

Containers constitute a lightweight isolation mechanism that relies on the kernel isolation capabilities of the host system, as shown in figure 2. Namely, under Linux:

- `chroot` changes the apparent root directory for a given process tree. It allows a container to operate on a virtual `/` directory that could be located anywhere on the host's filesystem;
- `cgroups` create hierarchical groups of processes and allocates, limits, and monitors hardware resources for these groups: I/O to and from block devices, accesses to the CPU, memory and network interfaces;
- `namespaces` are an abstraction layer around global system resources, such as networking or IPC. Processes within a namespace have their own isolated instances of these resources.

The idea behind containers is to sandbox an application's execution in a process isolated from the rest of the system. That process is bootstrapped from an image which contains all dependencies needed to either build and/or execute the application.

Among the container ecosystem, Docker [36] in particular has seen important traction since its inception in 2013. Docker was instrumental in specifying industry standards for container formats and runtimes through the Open Container Initiative (OCI) [69].

The OCI specification is a Linux Foundation [120] initiative to design an open standard for containers. It defines container image specifications – guidelines on how to create an OCI image with its manifest, filesystem layers, and configuration – and runtime specifications regarding how to execute application bundles as they are unpacked on the host OS.

This is how that specification translates in the case of Docker:

- `dockerd` is the daemon that provides both the Docker Engine application programming interface (API) and console line interface (CLI), capable of building distributable images representing the initial state of future containers. It is the high-level interface through which the user can either programmatically or interactively manage networking, storage, images and containers lifecycle;
- `containerd`, a Cloud Native Computing Foundation (CNCF) [41] initiative, handles containers lifecycle (hypervision, execution) and manages images (push and pull to and from image registries), storage and networking by establishing links between containers namespaces;

- `runc` implements the Open Container Initiative specification and contains the actual code allowing a container's execution. It creates and starts containers, and stops their execution.

2.3 From monoliths to microservices

Cloud computing saw the birth of new development techniques. "Cloud-native development" means building applications for the cloud from the ground up, with scaling capacities in mind [37] [43].

When an application grows, there are two ways of making room for new requests by scaling:

- **Vertically**: attaching more hardware resources to the servers that support the application. It may mean moving data to new, more powerful servers, thus impacting application availability;
- **Horizontally**: increasing the server count for running the application. It may mean introducing a load balancing mechanism to route requests and responses between users and the multiple instances of an application, thus impacting complexity.

A monolithic application is built as a single unit. There is no decoupling between the services it exposes, as they are all part of the same codebase [126]. When scaling for a monolith, adding more resources (scaling vertically) does not solve the problem of competing priorities inside the application: when the popularity of a monolithic application increases, some parts of the codebase will be solicited more than others, but the strain will not be distributed across the application. On the other hand, spinning up more instances of the monolith (scaling horizontally) can prove cost ineffective, as not all parts of an application suffer from load spikes at the same time: in figure 3, an increase in authentication requests means scaling the infrastructure for the whole application.

The Twelve-Factor App methodology, a set of guidelines for building cloud-native applications, recommends "[executing] the app as one or more stateless process" [131]. This is called a microservices architecture – arranging an application as a collection of loosely-coupled services. Each of these services runs in its own process, communicates with the others through message-passing and can be deployed to scale independently on heterogeneous servers to meet service level objectives: in figure 4, an increase in authentication requests can be absorbed by scaling the infrastructure for the authentication microservice alone.

However, a microservices infrastructure implies a complex layer of centralized management, which either drives costs in operations or in DevOps teams. It relies on long-running backend services (databases, message buses, etc.) that also have to be monitored and managed. In IaaS and PaaS settings, developer experience in particular is not satisfying: cloud deployment comes with a burden of systems administration [62]. Microservices alone do not solve the deployment problem:

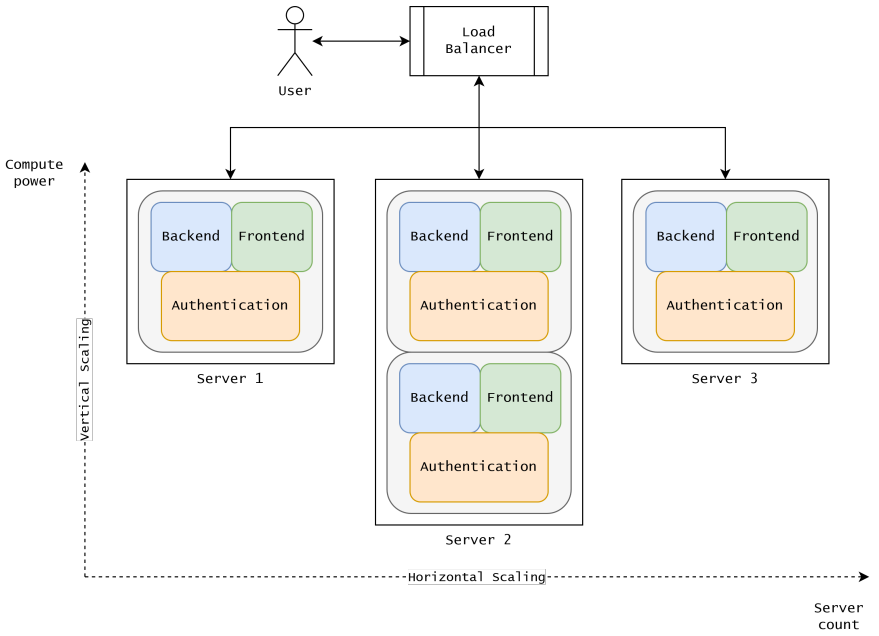

Fig. 3 Scaling out a monolith-architectured application requires replicating the monolith on multiple servers

writing container images recipes does not add value to the product, as it relates to operational logic rather than business logic.

Continuous Integration (CI) and Continuous Delivery or Deployment (CD) are foundational building blocks of the DevOps culture – the idea that, at any given stage of an application's lifecycle, its codebase is in a working state [68]. This can be achieved through the automation or running unit and integration test suites (CI), and automated deployment of the main code trunk to a staging (or pre-production) environment with heavy use of monitoring and reporting [109]. It allows developers to ensure no regression is introduced by the addition of new features.

Continuous practices align neatly with the microservices architecture, where incremental modifications to the application as a whole can be deployed as microservices updates. Popularity of both DevOps practices and the microservice architecture led to transformations in the cloud landscape. To some extent, the microservice architecture can be mapped to a cloud programming and service model, *Function as a Service* [60].

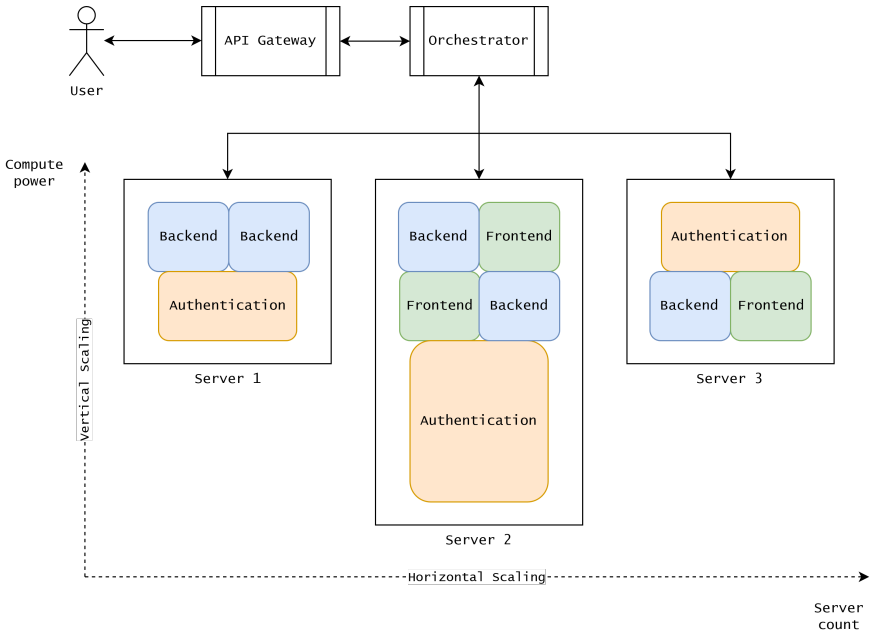

Fig. 4 Scaling out a microservices-architectured application allows distributing and replicating each microservice independently

3 Serverless, a new paradigm

In this section, we introduce the serverless model of programming for, and deploying applications to the cloud. We will go through essential characteristics of serverless platforms and highlight the tradeoffs that both service providers and application developers have to consider when targeting serverless infrastructures. This section also proposes a description of serverless offerings in public cloud solutions, and open source platforms for private cloud architects.

3.1 Characteristics of serverless platforms

Serverless refers at the same time to a programming and a service model for cloud computing. In a serverless architecture, developers design their applications as a composition of stateless functions. Stateless (or "pure", side-effect free) means that the outcome of the computation depends exclusively on the inputs [20]. These functions take a payload and an invocation context as input, and produce a result that is stored in a persistent storage tier. Their execution is triggered by an event that can be described as the notification of an incoming message, be it an HTTP request,

a cron job, a file upload, etc. As such, serverless is an event-, or demand-driven model [106].

The aforesaid design is illustrated by an example serverless application in figure 5: when the user's request hits the serverless platform's API gateway, it triggers the execution of different functions according to the requested HTTP endpoint – these functions are not *daemons* listening for events (e.g. on an opened socket), they are executed on demand.

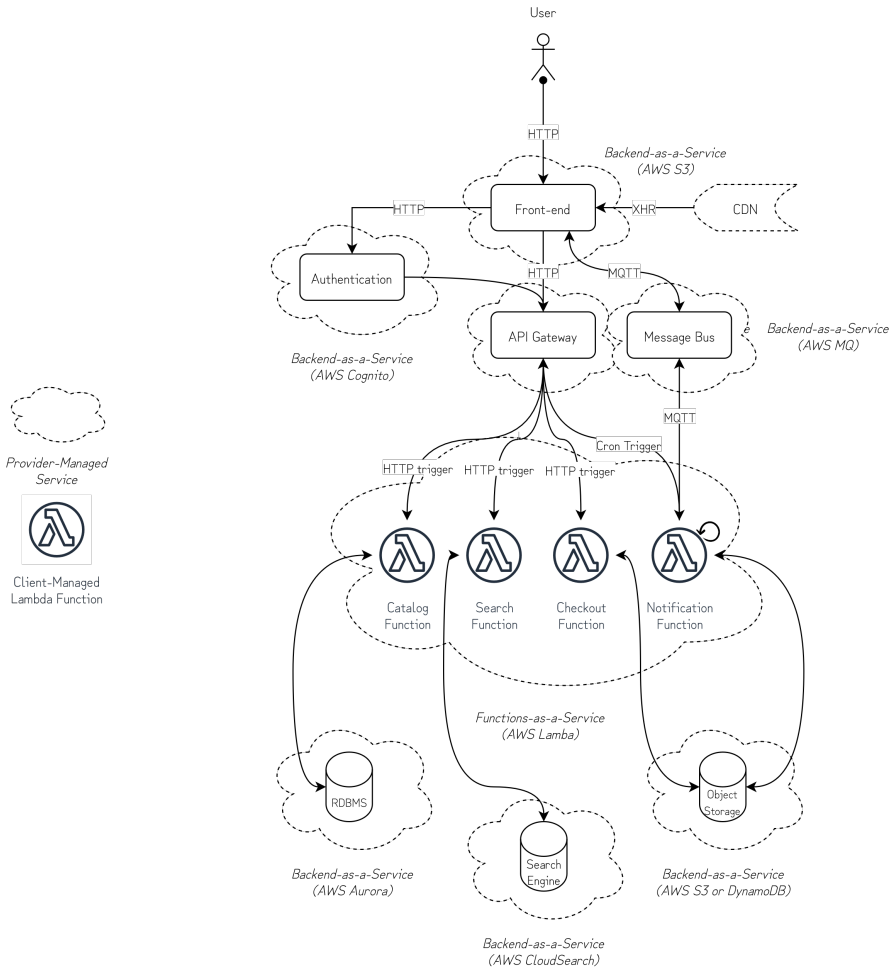

Fig. 5 Fictional reference architecture for a serverless e-commerce web application deployed in the Amazon Web Services ecosystem

In commercial offerings, serverless is usually referred to as *Function as a Service* (FaaS). It has been believed that functions as an abstraction over cloud computing are part of a first generation of serverless offerings, and might change later on [54].

Serverless does not mean that servers are no longer used to host and run applications – much like PaaS, from a customer's perspective, serverless refers to an abstraction over computing resources that allows engineers to leave out thinking of the actual servers supporting their applications. Thanks to auto-scaling mechanisms, they do not have to consider the optimal number of instances needed to run their workloads in a capacity planning fashion. Serverless platforms are designed to handle scaling requirements and address fluctuations in demand, therefore freeing customers from the burden of having to define explicit scaling strategies. In a layered vision of cloud deployments, the SPEC Research Group [115] presents a FaaS reference architecture that shows that serverless development allows developers to be as close as possible to business logic [123].

Developing for serverless platforms requires re-thinking an application's architecture. Indeed, long-running backend servers are relegated to *serverful* solutions that provide always-on servers [80], such as IaaS offerings.

As Shafiei et al. [108] pointed out in their 2022 survey on serverless computing, there is no formal definition for the concept of serverless computing, although we can identify various essential differences between the serverful and serverless models (summed up in table 1).

A major difference between PaaS and FaaS is that FaaS achieves scaling to zero: providers only bill customers when their application actually uses hardware resources, i.e. when functions are executed on the platform. That is made possible because, in the FaaS paradigm, applications are designed as a collection of short-running microservices.

Backend as a Service (BaaS) solutions are commercial, managed offerings of backend services, made available to application developers through a unified API [104]. Backend software usually consist in stateful, long-running services that cannot be scaled down to zero. In order to maintain a consistent pricing model, providers must offer these services in the same pay-as-you-go manner as they do serverless functions. These third party services constitute the backbone infrastructure of serverless applications by handling the state of the functions deployed by developers, through e.g. key-value stores or file storage; providing authentication to application endpoints; allowing communications between functions using message buses; etc. Figure 5 shows possible dependencies between serverless functions and BaaS software: the example application relies on provider-managed authentication, message bus, relational database, search engine and object storage, and is accessed by the user through the provider's API gateway. This situation introduces a high degree of coupling between the application and vendor-specific services, potentially tying developers to their initial choice of service provider.

Serverless allows reduced developer overhead by abstracting away server management, while enabling providers to share physical resources at a very fine granularity, thus achieving better efficiency. The fine level of granularity presented in the FaaS

model enables the provider to offer perfect elasticity: scaling out and scaling in are event-driven, in a typical *pay-as-you-go* pricing model.

This abstraction makes it desirable for providers to deploy code in multiple geographic zones. This fail-over mechanism guarantees availability in case of outage in one deployment zone and decreases the risk of function failure cascading through the application [117].

Furthermore, as function instances are spun up on-demand by the provider, the concurrency model offered by FaaS platforms means that an application's performance can scale linearly with the number of requests [81].

Table 1 Comparison of key characteristics in serverless and serverful service models

Characteristic	*Serverful* (IaaS, PaaS)	*Serverless* (FaaS)
Provisioning	Customer responsibility	Fully **managed** (*i.e.* by the provider)
Billing	Pay for **provisioned** resources	Pay for **consumed** resources
Scaling	Customer responsibility	**Auto-scaling** built in
Availability	Depends on **provisioned** resources	Code runs in multiple high availability zones
Fault tolerance	Depends on deployment strategy	Backend services are fully **managed** and retries are guaranteed
Concurrency	Depends on **provisioned** resources	Virtually **infinite**

Various authors ([54], [123], [108], [63]) already consider serverless to be the future of cloud deployment. However, FaaS adoption seems to be stalling among cloud developers [76], and the Cloud Native Computing Foundation (CNCF) even reports decreasing figures [29].

In the next section, we provide insights regarding workloads for which serverless is particularly desirable. In section 4, we provide an overview of technical challenges that are still holding serverless back from becoming the go-to cloud subscription model.

3.2 Suitable workloads

In their 2018 white paper [95], the Cloud Native Computing Foundation (CNCF) – a Linux Foundation initiative supported by more than 800 industrial members involved in cloud services – identifies characteristics for serverless use cases, including:

- "Embarrassingly parallel" workloads: asynchronous and concurrent, with little to no communication and no synchronization between processes;
- Infrequent with unpredictable variance in scaling requirements, i.e. event-driven or interactive jobs rather than batch jobs;

- Stateless and ephemeral processes, without a major need for instantaneous start time.

We can argue that these conditions are too restrictive for general computing: for example, it implies that long-lived jobs cannot be mapped to FaaS deployments. Providers have introduced mechanisms such as Step Functions or Durable Functions ([7], [86], [48]) to implement serverless workflows: an orchestration function maintains state across the application's stateless functions to create stateful workflows [20].

Developers are already deploying part of their applications' logic to serverless functions. According to a survey [96] led in 2018 by the Serverless framework publisher among a panel of their users, examples of such logic include data transformation pipelines, high-availability alerting platforms, ETL (*Extract, Transform, Load*, batch data manipulation) tools, media transcoding, etc. These applications are a subset of computer programs that produce output that only depends on the program's input: they apply purely functional transformations to data.

Problems that are conveniently split up in batches of sub-tasks would also benefit from the virtually infinite level of concurrency offered by the serverless model [46]. In [1], the authors identify use cases for serverless in massive scale video encoding, linear algebra and parallel compilation.

As there are fundamental similarities between a microservices-architectured application and an application devised for FaaS deployment [60], full-fledged applications can be designed with FaaS in mind. Figure 5 gives an example architecture for an e-commerce web application. The application's business logic comprises three serverless functions (catalog, search and checkout) that are triggered by user navigation, and one function (notification) that is scheduled to run periodically. As these functions are spun up and down according to the application's load, state has to be stored in persistent storage, i.e. a relational database and an object store that are both managed by the provider. The application further relies on provider-managed services: its search engine is powered by a BaaS solution, as is the notification function and the authentication mechanism.

Using AWS Lambda in 2022, this kind of application could scale from zero resources used to 150 TB of RAM and 90 000 vCPUs in less than two seconds [58], allowing for timely reaction to load variations.

3.3 Tradeoffs in serverless deployments

When we consider serverless as a programming model, the immediate benefit is a reduced development cost for teams leaning on BaaS offerings: instead of implementing in-house backend services such as authentication or notifications, developers merely introduce boilerplate in their codebase so as to connect frontend applications to their cloud provider's BaaS APIs. However, that degree of coupling means developpers can find themselves locked-in in a vendor-specific environment, ultimately losing control over deployment costs [13].

From a customer's point of view, FaaS coupled with managed BaaS achieve perfect scaling. Customers are charged at a fine grain, only when resources are actually used, and for the exact duration of execution. From a provider's point of view, increased tenancy creates an opportunity to achieve better resource multiplexing, allowing for increased efficiency and thus greater profits. This auto-scaling mechanism has a cost in terms of latency: spinning up new sandboxes for incoming requests can create situations of *cold starts* where initialization times dominate execution times of functions [61]. Serverless auto-scaling has further implications in terms of throughput: since function state has to be persisted in disaggregated storage, applications that display patterns of extensive communications between functions can suffer from the shipping time of data to compute nodes [89].

A side effect of the FaaS service model is the increase in job count per customer. While the rise in concurrency, density and resources usage is a selling point for both FaaS providers and customers, these short-lived jobs have to be isolated from each other to prevent the leaking of secrets across customers, or at the scale of a single application comprising multiple individual functions [124].

Table 2 Considerations regarding the FaaS service model

Pros	*Cons*
Reduced **development costs** for teams leaning on BaaS offerings	Can we map **any application** to the FaaS architecture?
Reduced **costs in operations** thanks to fully managed infrastructure	Risks of **vendor lock-in** due to high degree of coupling with BaaS offerings
Perfect scaling allows billing granularity close to actual use of resources	Increase in **latency** due to cold starts, and decreased **throughput** from communications through slow storage to handle statefulness
Providers can achieve **better efficiency** in resources multiplexing leading to increased profits	Massive multitenancy might involve **security** threats

To build on the model's strengths, customers and providers have to consider the tradeoffs that are associated with serverless deployments. Table 2 summarises key takeaways when using serverless to deploy applications to the cloud.

3.4 Description of current FaaS offerings

In this section, we provide an overview of current FaaS offerings from public cloud providers and open source solutions for private cloud.

Table 3 presents a summary of major FaaS offerings regarding their pricing models and properties; including Alibaba Function Compute [3], Amazon Web Services Lambda [6], Microsoft Azure Functions [83], Google Cloud Functions [47], IBM Cloud Functions [57] and Oracle Cloud Functions [93].

Table 4 presents a summary of self-hostable FaaS platforms regarding their project status and adoption, and corporate backers; including Apache OpenWhisk [11], Fission [100], Fn [92], Knative [42] and OpenFaaS [128].

3.4.1 Commercial solutions

To position each offering among the commercial offerings, we chose to compare them in terms of:

- Pricing model: the manner in which a customer can expect to be billed for product usage;
- Properties: the limits imposed by the provider regarding resource usage.

Table 3 Cloud customers are faced with a diversity of FaaS offerings

	Pricing model		**Properties**			
Service	**free quota per month**	**pay-as-you-go cost**				
	# of invocations / compute resources [GB s]	1M requests / 1 GB s compute [USD]	code size	memory	execution time	payload size
Alibaba Function Compute	1 000 000 / 400 000	20 / 0.000 016 384	500 MB	3 GB	24 h	128 kB (request), 6 MB (response)
AWS Lambda	1 000 000 / 400 000	0.2 / 0.000 016 666 7	10 GB	10 240 MB	15 min	6 MB (synchronous), 256 kB (asynchronous) for requests and responses
Azure Functions	1 000 000 / 400 000	0.2 / 0.000 016	N/A	1.5 GB	10 min	100 MB (request)

Continued on next page

Table 3 Cloud customers are faced with a diversity of FaaS offerings (Continued)

Service	Pricing model		Properties			
	free quota per month	**pay-as-you-go cost**				
	# of invocations / compute resources [GB s]	1M requests / 1 GB s compute [USD]	code size	memory	execution time	payload size
Google Cloud Functions	2 000 000 / 400 000	0.4 / 0.000 002 5	500 MB	8 GB	9 min	10 MB for requests and responses
IBM Cloud Functions	5 000 000 / 400 000	N/A / 0.000 017	48 MB	2048 MB	60 s	5 MB for requests and responses
Oracle Cloud Functions	2 000 000 / 400 000	0.2 / 0.000 014 17	N/A	2048 MB	5 min	6 MB

[a] billed per 10 000 requests (for USD 0.02)

3.4.2 In the open source community

To measure a project's status and adoption, we chose two indicators publicly available at GitHub [84]:

- GitHub "stars" indicate how many GitHub users chose to keep track of a project;
- Contributors are people who pushed 10 or more *git commits* (modifications to the codebase) to the repository.

Table 4 Open source solutions allow cloud providers to devise their own FaaS offering

	Project status and adoption	**Corporate backer**
Apache OpenWhisk	5.8k GitHub stars, 34 contributors ($\geq$ 10 commits)	IBM (Apache Foundation)
Fission	7.3k GitHub stars, 10 contributors ($\geq$ 10 commits)	Platform9
Fn	5.3k GitHub stars, 21 contributors ($\geq$ 10 commits)	Oracle
Knative	4.5k GitHub stars, 55 contributors ($\geq$ 10 commits)	Google
OpenFaaS	22.2k GitHub stars, 13 contributors ($\geq$ 10 commits)	VMware

In this section, we introduced FaaS as a deployment and a programming model, available for both public and private cloud. Serverless sparked interest in academia and industry, as a solution for customers to reduce their development and operation costs, and for providers to maximize resource usage. However, despite attractive pricing with extensive free plans in commercial offerings, and a various panel of open source solutions targeted at the major cloud orchestrators, FaaS has not yet become the go-to cloud subscription model: some challenges are still open and have to be addressed before serverless can become ubiquitous. Those are described in the next section.

4 Problems addressed in the literature

In order to achieve flexibility and performance comparable to PaaS or IaaS solutions, FaaS providers need to tackle major problems that hinders the progress of serverless in becoming the norm in cloud computing. Serverless is a lively topic in cloud computing and many authors are contributing toward mitigating these setbacks: the number of published papers around serverless almost doubled between 2019 and 2020 [53]. The following sections will describe each problem and provide a set of state-of-the-art solutions that have been proposed in the literature. A real challenge in addressing these shortcomings is to avoid "serverful" solutions to the problem of dynamic allocation of resources, i.e. allocating additional stable resources that purposefully do not scale to zero [54].

4.1 Cold start delays and frequency

As serverless containers must spend a minimum amount of time in an idle state, they are spun up and down very frequently as compared to PaaS containers or IaaS VMs. Each time a function is called and has to be scaled from zero, the container or virtual machine hosting the function's code has to go through its initialization phase:

this is called a ”cold start” [73]. Cold starts can incur latency penalties, aggravated by delays snowballing during composition of functions in the context of complex applications [88].

Functions are typically invoked in bursts – the AWS Lambda execution model can maximize concurrency by instantiating a function in hundreds to thousands of sandboxes across different geographical locations [8]. Minutes after handling a request, a function's sandbox is freed from the execution node; moreover, future new instances are not guaranteed to be created on the same node. This leads to situations in which a function's environment is not cached on the node. Code and associated libraries have to be fetched and copied to the filesystem again, resulting in cold start latency.

A ”naive” approach would consist in pre-allocating hardware resources in order to keep a pool of function containers ready for new requests. This is not acceptable [75] as it strides away from the possibility of scaling to zero.

Vahidinia et al. [122] propose a comprehensive study of the position and strategies of various commercial FaaS offerings regarding cold start. While serverless computing suggests spinning up disposable instances of functions to handle each incoming request, the authors note that commercial actors such as AWS, Google and Microsoft all re-use execution sandboxes to some extent, keeping them running during a timeout period in order to circumvent latency costs incurred by cold starts.

4.1.1 Reducing initialization times

Different approaches can be implemented by cloud providers to shrink the initialization time of function sandboxes. This is a crucial work as function invocations follow mostly unpredictable patterns [110]. This section explores contributions from the literature that focus on bridging the gap in latency between serverless and serverful models.

Sandbox optimization approach

In [89], the authors propose Lambada to address the cold start problem in the context of distributed data analytics by batching the invocation of workers in parallel. They identify a bottleneck in the invocation process of new workers: in their evaluation, they show that invoking 1000 AWS Lambda workers takes between 3.4 to 4.4 seconds. In their contribution, each worker is responsible for invoking a second generation of sandboxes, which will in turn invoke a next generation of workers until the scaling process is complete. This technique allows to spawn several thousands of workers in under 4 seconds.

In [77], the authors propose LightVM to put VM boot time in the same ballpark as containers. The authors show that instantiation times grow linearly with image size: creating a sandboxed environment for an application to run is an I/O-bound operation. By redesigning Xen control plane and using lightweight VMs that include a minimal environment needed to run the sandboxed application, they achieve boot times comparable to the performances of the `fork/exec` implementation in Linux.

In [2], the authors propose that full-blown isolation mechanisms such as containers are needed to isolate workloads among customers, while at the granularity of a single application, processes are enough to isolate functions. In SAND, they implement an isolation mechanism on top of Docker that enables resources-efficient, elastic, low-latency interactions between functions.

In [1], the authors present Firecracker, which grew to become the *de facto* virtualization technology for serverless, being used at AWS Lambda. They tackle the tradeoff in isolation versus performances by introducing lightweight VMs (or MicroVMs) in lieu of containers as a sandboxing mechanism for serverless workloads. Firecracker achieves boot times under 125 ms by replacing QEMU with a custom implementation of a virtual machine monitor that runs on top of KVM and allows to create up to 150 MicroVMs per second and per host with a 3% memory overhead.

Snapshotting approach

In [38], the authors argue that startup overhead in virtualization-based sandboxes is caused by their application-agnostic nature. Indeed, they show that the application initialization latency dominates the total startup latency. In Catalyzer, the authors show that sandbox instances of one same function possess very similar initialization states, and thus present a snapshotting solution that allows restoring a function instance from a checkpoint image, effectively skipping the application's initialization phase when scaling from zero. They build a solution based on Google's gVisor [49] that consistently outperforms state-of-the-art technologies such as Firecracker [1], HyperContainer and Docker by one order of magnitude.

In [121], the authors present vHive, a benchmarking framework for serverless experimentation that allows them to show that high latency can be attributed to frequent page faults during sandboxes initialization, with very similar patterns among executions of a same function – 97% of the memory pages being identical across invocations of studied functions. They propose REAP to create images of a sandbox memory layout that enable prefetching pages from disk to memory, eagerly populating the guest memory before function execution and thus avoiding the majority of page faults at initialization time. This technique allows a cold start delay speedup of 3.7 times on average.

In [112], the authors propose Faaslets, a new isolation mechanism based on software-fault isolation provided by WebAssembly. Faaslets allow restoring a function's state from already initialized snapshots. These snapshots are pre-initialized ahead of time and can be restored in hundreds of milliseconds, even across hosts. Fasslets take advantage of the WebAssembly memory model: linear bytes arrays can be copied without a lengthy (de)serialization phase.

Caching approach

In [91], the authors argue that while serverless allows costs savings through increased elasticity and developer velocity, lengthy container initialization times hurt latency performances of deployed applications. They identify bottlenecks in Linux primitives involved in containers initialization, with package dependencies being the major culprit in I/O operations during sandboxing. They propose SOCK

as a container system optimized for serverless tasks that builds on OpenLambda and rely on a package-aware caching system, and show that their solution offers speedups up to 21 times over Docker.

In [44], the author show conceptual similarities between object caching and function keep-alive, allowing them to devise policies that reduce cold start delays. Building on that analogy, they propose a keep-alive policy that is essentially a function termination (or eviction) policy. By keeping functions warm as long as possible (i.e. as long as server resources allow it), FaasCache manages to double the number of serviceable requests in their OpenWhisk-based implementation.

The ability for serverless platforms to scale a function to zero replicas in order to avoid billing customers for idle resources is a key difference with regards to traditional cloud service models. Seeking techniques that minimize the impact of a cold start on function latency is a critical research topic, as prohibitive initialization times hinder the potential for FaaS to compete with PaaS platforms.

4.2 Data communications overhead

In FaaS offerings, functions are non-addressable: composition is done through storing results in a stateful slow storage tier that is usually not collocated with the computing tier.

As functions of the same application cannot share memory or file descriptors to achieve IPC, they have to establish communication through message-passing interfaces, introducing overhead when data need to flow through the application.

This problem is particularly concerning when data-hungry applications have to work with cold data, i.e. data that are sparsely accessed and therefore not cached, and stored on lower performing storage such as hard drives located on remote nodes. In [61], the authors present LambdaML, a benchmarking platform that enables comparing performances of distributed machine learning model training across IaaS and FaaS offerings. They find out that using FaaS for ML training can be profitable as long as the models present reduced communication patterns.

In [89], the authors show that FaaS can be profitable when running sporadic, interactive queries on gigabytes to a terabyte of cold data. By providing serverless-specific data operators with Lambada, they achieve interactive queries on more than 1 TB of Amazon S3-stored data in approximately 15 seconds, which is in the same ball park as commercial Query-as-a-Service solutions.

In [105], the authors argue that serverless offerings lack an in-memory, per-application data caching layer that would allow auto-scaling and work transparently. Faa$T can form a strongly consistent distributed caching layer when multiple instances of an application are spun up, with the latest caching node vanishing as the application scales down to zero, effectively enabling billing on the basis of effective resources usage. The experimentations show that Faa$T can improve performance

of varying applications by 57% on average while being 99% cheaper than serverful alternatives.

SAND [2] introduces a hierarchy in communication buses. In SAND, an application's functions are always deployed on the same node. A node-local bus serves as a shortcut for inter-function communications, allowing for fast sequential execution. A global bus, distributed across nodes, ensures reliability through fault tolerance. In SAND, the local bus achieves almost 3x speedup for message delivery compared to the global bus.

As serverless functions are ephemeral by nature, and given the isolation mechanisms rolled out by providers to meet privacy and security objectives, minimizing the overhead of inter-function communication seems to be a two-fold problem: on the one hand, serverless platforms need both domain-specific solutions that factor in the characteristics of the data that are fed to and returned by the functions; on the other hand, there is room for general improvements in the field of distributed caches.

4.3 Durable state and statefulness

"State", or "local state", refers to data usually read and written from and to variables or disk by a process during its execution. FaaS offers no guarantees as to the availability of such storage across multiple executions. That is why serverless functions are referred to as "stateless": data that need to be persisted have to be stored externally, and functions should be idempotent in order to prevent state corruption.

Furthermore, FaaS offerings present arbitrary limitations including a function's execution time, payload size and allocated memory (cf. table 3). This is a problem when designing "real world" applications that consist of long-lived jobs, and/or that comprise functions that need to communicate or synchronize, e.g. to pass on intermediate results depending on transient state.

Given the ephemeral nature of function instances, one must be aware of fault tolerance and data consistency in their application when they deploy to FaaS.

In [133], the authors address I/O latency in the context of serverless function composition, where an application is divided into multiple functions that may run concurrently on different nodes while accessing remote storage. They propose HydroCache, a system that implements their idea of Multisite Transactional Causal Consistency (causal consistency in the scope of a single transaction distributed across multiple nodes). They observe improvements up to an order of magnitude in performance while achieving consistency. HydroCache outperforms state-of-the-art solutions such as Anna [132] and ElastiCache [5].

In [97], the authors argue that serverless database analytics would allow data analysts to avoid upfront costs by achieving elasticity. However, as these kinds of workloads are by nature unpredictable, cloud providers tend to have difficulty in provisioning adequate resources, leading to solutions that are elastic but sometimes suffer minutes of latency during scaling phases. They present Starling, a query

execution engine built on FaaS: three stage of functions (Producers, Combiners and Consumers) can scale independently to handle datasets stored on remote cold storage. Their evaluation shows that Starling is cost-effective on moderate query volumes (under 120 queries per hour on a 10 TB TPC-H dataset), while showing good latency results for ad-hoc analytics on cold data in Amazon S3 and being able to scale on a per-query basis.

In serverless, scaling from zero when activity returns after an idle period is usually event-driven. This poses a problem when no hardware resources are immediately available to resume workloads, inducing high latency. In [99], the authors investigate proactive auto-scaling for their serverless Azure SQL database offering. The contribution focuses on the prediction of pause and resume patterns in order to avoid the latency issue when resuming activity, and to minimize resources reclamation in the first place when idle periods are short. Using samples from thousands of production databases, they found that only 23% of databases are unpredictable, and trained machine learning models on three weeks of historical data to build a prediction system. The approach has been successfully used in production at Azure, achieving 80% of proactive resumes and avoiding up to 50% less pauses.

In [116], the authors build on the Anna KVS [132] to propose a stateful FaaS platform. Cloudburst achieves low-latency mutable state and communication with minimal programming effort. Leveraging Anna's capabilities, they provide essential building blocks to allow statefulness in an FaaS context: direct communication between functions, low-latency access to shared mutable state with distributed session consistency, and programmability to transparently implement Cloudburst consistency protocols. In their evaluation against real-world applications, Cloudburst outperforms both commercial and state-of-the-art solutions by at least an order of magnitude while maintaining auto-scaling capabilities.

4.3.1 Distributed data stores

Event-driven invocation implies that functions of a single application are not always executed on the same node, thus these functions cannot make use of shared memory or inter-process communications. Moreover, given the nature of serverless offerings that allow scaling to zero, functions are not always in an execution state and as such are not network addressable. Given these constraints, developers have to rely on increased communications through slow storage such as S3 buckets to handle statefulness within their applications.

There are hard challenges in scaling a database to zero: coming up with a database design that allows *true* serverless is an ongoing engineering and research effort. Microsoft recently proposed auto-scaling capabilities in their Azure SQL database [99]. In 2022, Cloudflare introduced D1 [32], which is based on SQLite.

Indeed, serverless applications are often deployed alongside a key-value store that scales much more naturally than a database, as key-value stores (KVS) are essentially stateless and thus can be distributed across nodes [65]. Given that KVS systems are central to serverless statefulness, implementing consistent, efficient and elastic

KVS is a lively research subject. However, industry-grade storage systems were not designed with serverless properties in mind, resulting in impaired elasticity and thus costs that grow faster than linearly with the infrastructure's size, and inconsistent performance depending on the scale.

In [132], the authors set out to design a KVS for any scale: the store should be extremely efficient on a single node and has to be able to scale up elastically to any cloud deployment. Their design requirements include partitioning the key space (starting at the multi-core level to ensure performance) with multi-master replication to achieve concurrency; wait-free execution to minimize latency and coordination-free consistency models to avoid bottlenecks during communications across cores and nodes. Using a state-of-the-art data structure, lattices, Anna can efficiently merge state in an asynchronous (or wait-free) fashion. The evaluation shows that Anna outperforms Cassandra by a 10x factor when used in a distributed setting, across four 32-core nodes in different geographical locations.

In [65], the authors argue that existing store services have objectives orthogonal or contradictory to those of a serverless KVS: they sacrifice performance or cost for durability or high availability of data. In particular, they find that these systems are inherently not suitable for intermediate (or "ephemeral") data in the context of inter-functions communications, as they require a long-running agent to achieve communication among tasks. The authors present Pocket, a distributed data store designed for intermediate data sharing in the context of serverless analytics, with sub-second response times, automatic resource rightsizing and intelligent data placement across multiple storage tiers (DRAM, Flash, disk). This is achieved by dividing responsibilities between three planes that scale independently: a control plane that implements data placement policies, a metadata plane that allows distributing data across nodes, and the data storage plane. When evaluated against Redis for MapReduce operations on a 100 GB dataset, Pocket shows comparable performance while saving close to 60% in cost. It is also significantly faster than Amazon S3, with a 4.1x speedup on ephemeral I/O.

4.3.2 Ephemeral storage

Cloud storage is devised as a tiered service: data is disaggregated across fast, but costly medium and slow, but cheap medium, according to frequency of use, size, age, etc.

Table 5 A simplified overview of media choice in tiered infrastructures

Capacity	TB			GB	
Addressability	Block			Byte	
Consideration	Cost			Data	
Latency	s	ms	µs	µs	ns
Data	Cold	Warm	Hot	Hot	Mission critical
Medium	Tape	HDD	SSD (Flash)	NVRAM	DRAM

Intel Optane are persistent memory (PM) modules that target a tier in-between Flash SSDs and DRAM: their latency and bandwidth are slightly worse than DRAM, but they offer SSD-level capacities of non-volatile memory at an affordable price ([19], [59], [18]).

In [25], the authors aim at delivering a key-value storage engine that would take advantage of persistent memory (PM, or NVM for non-volatile memory) technology to achieve greater performance than on spinning disks or Flash memory. They focus on write-intensive, small-sized workloads: indeed, previous studies ([12], [90]) have shown that Memcached pools in the wild are mainly used to store small objects, e.g. 70% of them are smaller than 300 bytes at Facebook. Moreover, serverless analytics exchange short-lived data and thus are very write-intensive, while object stores have historically been used as a read-dominated caching layer. Building on these observations, and characteristics specific to PM devices, they present FlatStore, a KVS engine with minimal write overhead, low latency and multi-core scalability. As persistent memories present fine-grain addressability and show low latency as compared to HDDs and SSDs, the authors designed FlatStore for minimal batching of write operations so as to avoid contention. When benchmarked on Facebook data with tiny (1-13 bytes) to large (> 300 bytes) items, evaluation shows that FlatStore performs 2.5 to 6.3 times faster than state-of-the-art solutions.

Statefulness is a major problem for serverless platforms. Service providers are deploying a variety of BaaS software to bridge the gap between traditional service models and FaaS and allow developers to deploy their full applications to their serverless offerings. Serverless functions present intrinsically disaggregated storage and compute, as they are deployed on-the-fly to multiple geographic zones, on hardware resources that are dynamically allocated by the provider. They need a means to operate on data that is fast enough, offers consistency guarantees, and scales in coherence with the pay-as-you-go pricing model. There is room for research in the field of distributed data stores, and using emerging non-volatile memory to accelerate throughput.

4.4 Hardware heterogeneity

Cloud customers are expected to book different resources depending on their applications' needs, be it a specific CPU architecture, hardware accelerators, ad-hoc storage... A striking example is distributed machine learning, in which many GPUs are used to speed up the training of models – furthermore, cloud providers are starting to generalize access to specialized hardware such as TPUs [28] in VMs.

Manual selection of hardware resources ("instance type"), expected from customers in IaaS offerings such as Amazon EC2, does not make sense in the serverless paradigm. Hardware acceleration should be decided by the provider on a per-application or per-request basis. To date, that possibility is not available in FaaS offerings such as AWS Lambda.

In [61], the authors set out to compare IaaS and FaaS configurations for machine learning training on Amazon Web Services offerings (resp. EC2 and Lambda). They propose an implementation of FaaS-based learning, LambdaML, and benchmark it against state-of-the-art frameworks running on EC2 instances. They measured that serverless training can be cost-effective as long as the model converges quickly enough so that inter-function communications do not dominate the total run time. Otherwise, an IaaS configuration using GPUs will outperform any FaaS configuration, yielding better performance while being more cost-effective.

In [14], the authors explore multitenancy in FPGAs to achieve higher board usage rate. They propose BlastFunction, a scalable system for FPGA time sharing in a serverless context. Their implementation relies on three building blocks: a library that allows transparent access to remote shared devices, a distributed control plane that monitors the FPGAs to achieve time sharing, and a central registry that handles allocating the boards to each compute node. This design allows reaching higher utilization rates on the boards and thus processing a higher number of requests, especially under high load, although at the cost of a 36% increase in latency due to the added concurrency.

In [35], the authors focus on a financial services use case and propose FPGAs to decrease end-to-end response time and increase scalability in a microservices architecture. The application they studied is computationally intensive and has real-time characteristics. They propose CloudiFi, a cloud-native framework that exposes hardware accelerators as microservices through a RESTful HTTP API. CloudiFi allows offloading workloads to hardware accelerators at a function level. An evaluation of the application's performance under CloudiFi shows 485x gains in response time when using network-attached FPGAs against a vanilla configuration.

ARM and RISC CPUs, GPUs and FPGAs are increasingly used in datacenters to address demand for performance, power efficiency and reduced form factor. In [56], the authors argue that since these heterogeneous execution platforms are usually collocated with a general purpose host CPU, being able to leverage their characteristics by migrating workloads could yield significant performance gains. They propose Xar-Trek, a compiler and run-time monitor to enable execution migration across heterogeneous-ISA CPUs and FPGAs according to a scheduling policy. Xar-Trek involves limited programming effort: the application is written once and compiled for different targets thanks to the Xilinx toolchain, without necessary high-level synthesis annotations to guide the compiler. Xar-Trek's runtime system, a user space online scheduler, is able to determine if a migration is effective and to proceed to migrate selected functions that benefit the most from acceleration. Evaluation on machine vision and HPC workloads finds out that as long as the workloads are dominated by compute-intensive functions, Xar-Trek always outperforms vanilla configurations, with performance gains between 26 and 32%.

Even when heterogeneous hardware is collocated on the same node, they are usually interconnected through PCI-Express buses managed by the host's CPU. Communications are achieved with message-passing interfaces that introduce bandwidth and latency costs. In [125], the authors present FractOS, a distributed operating system for heterogeneous, disaggregated datacenters. FractOS allows decentralizing

the execution of applications: instead of relying on the CPU to pass on control and data from one execution platform to another, FractOS provides applications with a library that allows direct communications between devices, thanks to an underlying controller that catches system calls and provides direct device-to-device functionality. When benchmarked on a face verification application that leverages GPUs to accelerate computations, their solution shows a speedup of 47% in execution time and an overall network traffic divided by 3.

With the exponential progression and growing interest in the field of machine learning, demand for hardware accelerators in the cloud has never been so prevalent. Commercial serverless offerings are lagging behind traditional IaaS in that regard, as none offer access to GPUs, TPUs nor FPGAs. Furthermore, dynamically allocating such hardware to accelerate select tasks has potential for providers to improve their resource usage and energy consumption.

4.5 Isolation and security

In order to achieve resource pooling, cloud providers rely on virtualization technologies so as to isolate cusomters' workloads. Furthermore, they offer various models of service ranging from IaaS to FaaS, all of which call for different sandboxing techniques providing a different balance between performance and isolation.

The usual tradeoff happens between the robustness of hypervision-based isolation (VMs) where each sandbox runs a separate OS, and the performance of OS-level virtualization (containers) where sandboxes all share the host's kernel. Ideally, cloud providers should not have to sacrifice one of these two essential characteristics. Efforts have been made to reduce virtualization overhead in order to decrease startup times and reduce the performance gap between these two techniques [77].

4.5.1 MicroVMs

In [1], the authors identify numerous challenges to devise an isolation method specifically suitable for serverless workloads in the context of AWS Lambda – Firecracker must provide VM-level security with container-level sandboxing density on a single host, with close to bare-metal performances for any Linux-compatible application. Firecracker overhead should be small enough so that creating and disposing of sandboxes would be fast enough for AWS Lambda ($\leq 150\,ms$), and the manager should allow over committing hardware resources with sandboxes consuming only the resources it needs. With Firecracker, the authors present a new Virtual Machine Monitor (VMM) based on Linux KVM to run minimal virtual machines (or MicroVMs) that pack an unmodified, minimal Linux kernel and user space. Thanks to sandbox pooling, Firecracker achieves fast boot times and high sandbox density

on a single host, for any given Linux application. It has been successfully used in production in AWS Lambda since 2018.

In [10], the authors study the differences in host kernel functionality usage across Linux Containers (LXC), Firecracker MicroVMs and Google's gVisor secure containers. gVisor sandboxes are `seccomp` containers: they are restricted to 4 system calls, namely `exit`, `sigreturn`, and `read` and `write` on already opened file descriptors. Extended functionality relies upon a Go-written user space kernel called Sentry that intercepts and implements system calls, and manages file descriptors. This prevents direct interaction between the sandboxed application and the host OS. While effectively achieving secure isolation, gVisor's design is complicated and adds overhead: the authors find that gVisor has the largest footprint in CPU and memory usage, with the slowest bandwidth for network operations.

In [129], the authors argue that the virtualization ecosystem lacks a solution tailored for isolation at the granularity of a single function. They present virtines, a lightweight VM isolation mechanism, and Wasp, a type-2, minimal library hypervisor that runs on GNU/Linux and Windows. Virtines are programmer-guided: annotations at function boundaries allow the compiler to automatically package subsets of the application in lightweight VMs with a POSIX-compatible runtime. Wasp works in a client-server fashion: the runtime (client) issues calls to the hypervisor (server) that determines if each individual request is allowed to be serviced according to an administrator-defined policy. In their evaluation with a JavaScript application, the authors find this design introduces limited overhead of 125 µs in boot time compared to baseline, while effectively achieving fine-tunable isolation for selected functions at almost no programmer effort.

4.5.2 Unikernels

The idea behind unikernels is to provide OS functionality as a library that can be embedded in an application sandbox so as to avoid packing and booting a full-fledged operating system to run the application, and to eliminate costly context switches from user space to kernel space. In [67], the authors present Unikraft, a Linux Foundation initiative. Unikraft aims at making the porting process as painless as possible for developers who want to run their applications on top of unikernels. Resulting images for different applications (nginx, SQLite, Redis) come close to the smallest possible size, i.e. Linux user space binary size, with very limited memory overhead during execution (< 10 MB of RAM) and fast boot times in the milliseconds range. Unikraft-packaged applications achieve 1.7 to 2.7x performance improvements compared to traditional Linux guest VMs.

In [22], the authors present a high-density caching mechanism that leverages unikernels and snapshotting (see 4.1.1) to speed up deployments. They argue that serverless functions are good candidates for caching: as they usually are written in high-level languages that execute in interpreters, their startup path mainly consists in initializing this interpreter and associated dependencies, which can be shared across different sandboxes. The snapshotting mechanism benefits from the unikernel

memory layout, where all functionalities (ranging from filesystem, to network stack, to user application) are combined into a single flat address space. The implement this mechanism in SEUSS to achieve caching over 16x more unikernel-based sandboxes in memory than Linux-based containers. Furthermore, deployment times drop from hundreds of milliseconds to under 10 ms, and platform handling of bursts of requests dramatically improves under high-density caching, leading to reduced numbers of failed requests.

In [118], the authors present Unikernel-as-a-Function (UaaF), a single address space, library OS aimed at deploying serverless functions. UaaF builds on the observation that cross-function invocations are slow in serverless deployments that rely on network-based message passing interfaces (see 4.2); furthermore, Linux guests suffer from memory usage overhead in sandboxes and their startup latency is not satisfying (see 4.1). The authors investigate using VMFUNC, an Intel technology for cross-sandboxes function invocations that do not incur latency when exiting from a VM to the hypervisor. It effectively enables remote function invocation, thus giving hardware-supported, secure IPC capabilities to serverless functions. They also propose a new programming model for serverless functions: *session* and *library* functions, with the former being ”workflow” (or skeleton) functions and the latter being actual code, uploaded by customers and possibly shared across applications. In their evaluation, the authors implement UaaF with three unikernels (Solo5, MirageOS and IncludeOS) and show that inter-function communication in UaaF is three orders of magnitude lower than native Linux IPC. Their programming model allows for reduced memory overhead and initialization times in several milliseconds thanks to shared functions.

FaaS workloads are orders of magnitude shorter-lived than workloads in traditional offerings. As such, relying on virtualization techniques that were not built for serverless is suboptimal: initialization times may not meet latency requirements when scaling from zero; sandbox sizes may be too high to cache in memory given the increase in multitenancy; isolation might be too weak to collocate different customers' jobs. This assessment sparked interest in research around unikernels and MicroVMs, while commercial providers developed their own approaches such as Firecracker for AWS, or gVisor for Google Cloud.

4.6 Programming model and vendor lock-in

As shown in figure 5, FaaS applications tend to rely heavily on BaaS offerings to benefit from costs savings associated to their capability to scale to zero. This tie-in introduces a risk of lock-in with vendor-specific solutions that might not be available across commercial offerings, or available as off-the-shelf, open source software.

Furthermore, some providers will use prohibitive egress bandwidth pricing [55] so as to deter their customers from moving data to a competitor.

Another aspect of that problem is the difficulty to develop, test and debug FaaS applications locally [119]. At the very least, developers will have to simulate the API gateway in order to run test suites; if their application makes use of vendor-specific storage solutions or communication buses, developers will have to deploy similar solutions or mock the specificities of these BaaS building blocks, e.g. their API and performance characteristics.

This amounts to non-negligible engineering efforts and indeed, deploying a full-fledged serverless infrastructure for staging might offset the operations cost benefits of choosing serverless for production. Entry level [33] and seasoned [87] engineers alike report having trouble with tooling, testing and the general ecosystem and added complexity of developing FaaS applications.

In [98], the authors observe that the disaggregation of storage and compute resources in FaaS limits the development of applications that make heavy use of shared mutable state and synchronize a lot between iterations. Indeed, state does not persist between invocations of a same function (see 4.3), and message passing for inter-function communications induces high overhead (see 4.2). In particular, they focus on machine learning algorithms (k-means clustering and logistic regression). They present Crucial, a framework aiming at supporting the development of stateful serverless applications. Crucial provides applications with a shared memory layer that guarantees durability through replication, with strong consistency guarantees. Crucial programming model is annotation-based, allowing programmers to port a single-machine, multi-threaded application to a FaaS platform with minimal involvement. Evaluation against a Spark cluster over a 100 GB dataset shows that Crucial running on AWS Lambda introduces very small overhead, enabling it to outperform Spark by 18 to 40% in performance at similar cost.

In [136], the authors acknowledge that the serverless programming model is challenging for developers. They have the responsibility to correctly partition their code into stateless units of work, to manage coordination mechanisms to achieve a microservices architecture, and to implement consistency models for state retention in case of failures. The complexity might deter customers from deploying general-purpose applications that would greatly benefit from the level of parallelism offered by serverless providers. They present Kappa, a Python framework for serverless applications. Kappa provides a familiar API that achieves checkpointing (by periodically storing the application's state so that the program can resume in case of timeout), concurrency (by supporting spawning tasks, waiting on futures, and cross-function message passing), and fault tolerance (by ensuring idempotent state restoration when resuming from checkpoints). Kappa applications can be deployed to any serverless platform, as the framework requires no change on the server side. In their evaluation, they implement five applications with Kappa and results indicate that the checkpointing mechanism works well when functions time out a lot, with less than 9% response time overhead under heavy (15 seconds) timeout duration, and a maximum of 3.2% with a more reasonable 60 seconds timeout period.

In order to limit the increase in latency when scaling from zero, the container or VM images that support serverless applications are usually made as lean and lightweight as possible. This deters developers from including monitoring or debug-

ging tools, making it very hard to inspect a serverless function at runtime. In [119], the authors present VMSH, a mechanism that allows attaching arbitrary guest images to running lightweight VMs in order to instrument it for development or debugging purposes. Evaluation done on KVM – although VMSH is built as a hypervisor-agnostic solution – shows that guest side-loading adds no overhead to the original VM guest, successfully slashing the tradeoff between no-frills, lightweight VMs and functionality.

There is a clear tradeoff in providing sandboxes as small as possible to minimize storage and memory costs in serverless platforms, while shipping adequate tools for developers to build, test, distribute and deploy their functions. Furthermore, the programming model based on stateless functions shed light on a new challenge: provider-side and developer-side tooling for stateful FaaS is needed to enable the serverless deployment of legacy and future applications that make use of long-running services and data persistence.

5 Perspectives and future directions

The previous section provided an overview of contributions linked with technical challenges in serverless computing. In this section, we introduce some future directions for research in the field. We present problems investigated in works from the cloud, system and database communities. We argue that contributions building on these insights would have the potential to strengthen serverless platforms for a broader recognition of the serverless paradigm.

5.1 Service Level Agreements

In 2011, Buyya et al. [21] advocated for SLA-oriented resource allocation in cloud computing at the dawn of the microservices era. They identified reliability in utility computing as a major challenge for the next decades: even with reserved resources in traditional service models, the growing complexity of customer applications made meeting Service Level Agreements (SLAs) a hard albeit inescapable problem for cloud providers.

Latency, throughput and continuity of service are difficult to guarantee in cloud computing when using unreserved resources [34]. Due to the transient nature of function sandboxes in serverless computing, auto-scaling platforms face a similar problem of dynamic allocation of resources. However, being able to offer SLAs to customers and meet Quality of Service (QoS) commitments as a provider is necessary for wide-scale adoption of the serverless service model [39].

In [23], the authors argue that serverless auto-scaling platforms are challenged by bursty workloads. In their work, they highlight the importance of workload

characterization to rightsize the amount of reserved VMs needed to meet SLAs. When the number of incoming requests drives up the concurrency level in reserved VMs, and makes task latency go past the acceptable threshold negotiated via SLA, they rely on a serverless platform to accommodate for extra tasks and maintain performance. While that framework managed to keep most of response times under the target threshold, the authors still see an incompressible number of violations caused by cold start delays on the serverless platform.

In [27], the authors argue that the task model in serverless computing and the infrastructure view in auto-scaling platforms are inadequate to address customers' needs in terms of service level. Indeed, auto-scalers base their allocation decisions upon generic metrics such as query per second (QPS) that do not reflect application-specific characteristics and do not take into account the heterogeneity of the hardware resources at hand. They propose a framework in which their application metrics (such as request execution time) are fed to the auto-scaler in order for it to allocate resources according to user-specified service level objectives, such as target latency. However, observed response times are non-deterministic due to cold start delays, and user-defined target latencies are subject to violations in an auto-scaling scenario.

In order to meet per-user QoS requirements, the auto-scaling platforms should take into account the characteristics of heterogeneous hardware resources, and SLAs should be negotiated on a per-request basis rather than on a per-function basis. We believe that auto-scaling policies based on workload and platform characterization could be implemented to minimize the impact of cold start latency and allow serverless platforms to meet SLAs with unreserved, heterogeneous resources.

5.2 Energy efficiency

Power usage in cloud computing is a crucial challenge: in 2010, datacenters totaled between 1.1 and 1.5% of global electricity use [66], and projections for 2030 show that these figures could go up from 3 to 13% of global electricity use [9]. With serverless becoming an increasingly popular service model for the cloud, and many authors considering serverless as the future of cloud computing, there is an opportunity for cloud providers to implement energy policies at scale.

To be efficient in terms of cost and energy consumption, an auto-scaling platform should be able to rightsize the allocated resources in a serverless cloud infrastructure, while being responsive enough to accommodate workload changes without impacting end users with spikes in latency. This highlights a tradeoff between energy and performance: oversubscribing resources can help ensure low latency on function invocation, but will result in higher energy consumption.

Multitenancy helped slow down the growth in server count in datacenters [79]. With promises of massive collocation of short-lived jobs, serverless seems to be a promising direction for cloud infrastructures looking to reduce their energy footprint.

Workload consolidation is a technique that consists in maximizing the number of jobs on the fewest number of nodes [24]. This allows for dynamic power management:

nodes that are not solicited can then be powered off, and nodes that observe moderate load can be slowed down, i.e. via CPU throttling [72].

One fundamental problem in the serverless paradigm is the intrinsic data-shipping architecture [4] [26]. Since function sandboxes are deployed on nodes in various geographic regions to achieve load balancing and availability, serverless platforms ship up to terabytes of data from the storage nodes to code which size can range from kilobytes to megabytes within the compute nodes.

Storage functions allow small units of work to be executed directly on the storage nodes [135], achieving 14% to 78% speedup against remote storage. Storage functions do not question the physical disaggregation of storage and compute resources that is instrumental in cloud computing, while effectively limiting data movement between nodes and as such reducing energy consumption in a datacenter.

Computational storage is a means to offload workloads from the CPU to the storage controller [15]. When dealing with large amounts of data, such techniques can help decrease data transfers, improve performance and reduce energy consumption. While these technologies are not yet ready for production use, they provide interesting research opportunities for the serverless community.

These techniques could be implemented in serverless platforms to yield further gains in cloud energy consumption. It implies taking into account the diversity of user applications and the heterogeneity of requests and hardware resources.

5.3 AI-assisted allocation of resources

In the serverless paradigm, it is the provider's responsibility to rightsize the allocation of hardware resources so that their customers' workloads are executed in time. Dynamically allocating appropriate hardware resources for event-driven tasks in a heterogeneous infrastructure is a hard problem that may hit a computational complexity barrier at scale, with online scheduler producing sub-optimal solutions [74]. Artificial intelligence (AI) techniques can help overcoming such a challenge.

Some authors expect AI-driven autonomic computing to become the norm in future systems [45]. The idea of autonomic computing is to build self-managed and self-adaptive systems that are resilient to an extremely changing environment at scale [101]. Such systems can be implemented with machine learning (ML) in a cost-effective way, using models that do not require extensive human intervention for supervision.

In [107], the authors show that reinforcement learning (RL) can achieve appropriate scaling on a per-workload basis, resulting in improved performance as compared to baseline configuration. In their contribution, they propose an RL model that effectively determines and adjusts the optimal concurrency level for a given workload.

Serverless platforms require reactive resources allocation and scheduling of tasks under SLAs with per-request QoS requirements [51]. Machine learning techniques

[4] Moving data from where they are stored to where they need to be processed.

can help achieve QoS requirements in traditional cloud computing paradigms [114], and have been used to enforce virtual machine consolidation [111]. Resources management and optimization using AI and ML could further help taking advantage of the heterogeneity of hardware resources in a cloud infrastructure.

6 Conclusion

Serverless is an emerging paradigm in cloud computing, implemented in Function-as-a-Service offerings. Customers devise their applications as compositions of stateless functions and push their code to the serverless provider's function repositories. When an event triggers the execution of a serverless function, the function is reactively instantiated in a virtualized environment.

It is a radical shift in the cloud computing landscape: while traditional offerings such as IaaS or PaaS are based on the reservation of stable resources, FaaS providers propose a demand-driven service model.

This paradigm allows customers to benefit from pay-per-use pricing models to the granularity of a function invocation. On the other hand, serverless providers can maximize the collocation of jobs on nodes and achieve better resource usage.

By freeing customers from the constraint of manual rightsizing of cloud resources, the serverless service model pledges to ease the auto-scaling of applications. Thanks to an event-driven, on-demand resource allocation mechanism, customers can benefit from significant reductions in operations costs as they do not have to pay for idle hardware anymore.

However, current serverless solutions present non-negligible limitations that constrain the model's adoption to specific use cases. This paradigm is based on a programming model in which developers have to design their applications as compositions of pure (stateless, idempotent) functions that cannot rely on side effects. This constitutes a considerable engineering effort.

As with the microservices architecture, serverless software rely on message-passing communications between functions. These functions being non-network addressable in current serverless offerings, these communications have to go through slow storage. This leads to important costs in performances when functions have to synchronize, sometimes outweighing the benefits offered by the virtually infinite level of parallelism in the serverless paradigm.

Furthermore, scaling from zero presents a frequent risk of increased latency during application wake-up, as the provider has to allocate hardware resources and instantiate the application's sandboxes before answering to the event. Serverless providers usually pre-allocate some resources so as to avoid these cold starts, which comes with a cost in resources multiplexing.

Finally, hardware accelerators are not yet available in commercial serverless offerings. With increasing demand in GPUs and FPGAs for massively parallel tasks, such as machine learning training or big data analytics, customers have to turn to

conventional cloud offerings such as IaaS if they want to benefit from heterogeneous hardware resources.

For serverless to impose itself as a serious contender among the cloud computing offerings, providers need to be able to guarantee some sort of quality of service through service level agreements. Characterizing the customers' workloads and taking into account the heterogeneity of both the infrastructure and the requests is crucial to that matter, and has the potential to boost both performance and reliability.

Cloud providers have a responsibility to tackle the problem of energy consumption in datacenters. To that end, serverless could prove to be an efficient service model if adequate techniques for workload consolidation are proposed and implemented. Providers could make the most of their new responsibility to allocate resources by devising power-off or slow-down strategies in serverless infrastructures.

AI-assisted resource management appears to be a promising research direction for serverless computing. Indeed, as interactive workloads exhibit hard-to-predict patterns, cloud providers could take advantage of ML models to guide allocation and scheduling decisions in auto-scaling platforms.

References

1. Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.M.: Firecracker: Lightweight Virtualization for Serverless Applications. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), pp. 419–434. USENIX Association, Santa Clara, CA (2020). URL https://www.usenix.org/conference/nsdi20/presentation/agache
2. Akkus, I.E., Chen, R., Rimac, I., Stein, M., Satzke, K., Beck, A., Aditya, P., Hilt, V.: SAND: Towards High-Performance Serverless Computing. In: Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '18, p. 923–935. USENIX Association, USA (2018). DOI 10.5555/3277355.3277444. URL https://dl.acm.org/doi/10.5555/3277355.3277444
3. Alibaba: Alibaba Function Compute (2022). URL https://www.alibabacloud.com/product/function-compute
4. Almeida Morais, F.J., Vilar Brasileiro, F., Vigolvino Lopes, R., Araujo Santos, R., Satterfield, W., Rosa, L.: Autoflex: Service Agnostic Auto-scaling Framework for IaaS Deployment Models. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, pp. 42–49. IEEE, Delft (2013). DOI 10.1109/CCGrid.2013.74. URL https://doi.org/10.1109/CCGrid.2013.74
5. Amazon Web Services: Amazon ElastiCache (2022). URL https://aws.amazon.com/elasticache/
6. Amazon Web Services: AWS Lambda (2022). URL https://aws.amazon.com/lambda/
7. Amazon Web Services: AWS Step Functions (2022). URL https://aws.amazon.com/step-functions/
8. Amazon Web Services: Lambda function scaling (2022). URL https://docs.aws.amazon.com/lambda/latest/dg/invocation-scaling.html
9. Andrae, A., Edler, T.: On Global Electricity Usage of Communication Technology: Trends to 2030. Challenges **6**(1), 117–157 (2015). DOI 10.3390/challe6010117. URL https://doi.org/10.3390/challe6010117
10. Anjali, Caraza-Harter, T., Swift, M.M.: Blending Containers and Virtual Machines: A Study of Firecracker and gVisor. Proceedings of the 16th ACM SIGPLAN/SIGOPS International

Conference on Virtual Execution Environments (2020). DOI 10.1145/3381052.3381315. URL https://doi.org/10.1145/3381052.3381315
11. Apache: Openwhisk (2022). URL https://openwhisk.apache.org/
12. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. SIGMETRICS Perform. Eval. Rev. **40**(1), 53–64 (2012). DOI 10.1145/2318857.2254766. URL https://doi.org/10.1145/2318857.2254766
13. Baarzi, A.F., Kesidis, G., Joe-Wong, C., Shahrad, M.: On Merits and Viability of Multi-Cloud Serverless. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 600–608. ACM, Seattle WA USA (2021). DOI 10.1145/3472883.3487002. URL https://doi.org/10.1145/3472883.3487002
14. Bacis, M., Brondolin, R., Santambrogio, M.D.: BlastFunction: An FPGA-as-a-Service System for Accelerated Serverless Computing. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 852–857. IEEE, Grenoble, France (2020). DOI 10.23919/DATE48585.2020.9116333. URL https://doi.org/10.23919/DATE48585.2020.9116333
15. Barbalace, A., Do, J.: Computational Storage: Where Are We Today? In: CIDR, p. 6 (2021). URL http://cidrdb.org/cidr2021/index.html. Conference on Innovative Data Systems Research 2020, CIDR 2020 ; Conference date: 11-01-2021 Through 15-01-2021
16. Baude, B.: Basic security principles for containers and container runtimes (2019). URL https://www.redhat.com/sysadmin/basic-security-principles-containers
17. Bentaleb, O., Belloum, A.S.Z., Sebaa, A., El-Maouhab, A.: Containerization Technologies: Taxonomies, Applications and Challenges. The Journal of Supercomputing **78**(1), 1144–1181 (2022). DOI 10.1007/s11227-021-03914-1. URL https://doi.org/10.1007/s11227-021-03914-1
18. Boukhobza, J., Olivier, P.: Flash Memory Integration. ISTE Press - Elsevier (2017). URL https://www.elsevier.com/books/flash-memory-integration/boukhobza/978-1-78548-124-6
19. Boukhobza, J., Rubini, S., Chen, R., Shao, Z.: Emerging nvm: A survey on architectural integration and research challenges. ACM Trans. Des. Autom. Electron. Syst. **23**(2) (2017). DOI 10.1145/3131848. URL https://doi.org/10.1145/3131848
20. Burckhardt, S., Chandramouli, B., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S., Zhu, X.: Netherite: Efficient Execution of Serverless Workflows. Proc. VLDB Endow. **15**(8), 1591–1604 (2022). DOI 10.14778/3529337.3529344. URL https://doi.org/10.14778/3529337.3529344
21. Buyya, R., Garg, S.K., Calheiros, R.N.: SLA-oriented resource provisioning for cloud computing: Challenges, architecture, and solutions. In: 2011 International Conference on Cloud and Service Computing, pp. 1–10. IEEE, Hong Kong, China (2011). DOI 10.1109/CSC.2011.6138522. URL https://doi.org/10.1109/CSC.2011.6138522
22. Cadden, J., Unger, T., Awad, Y., Dong, H., Krieger, O., Appavoo, J.: SEUSS: Skip Redundant Paths to Make Serverless Fast. In: Proceedings of the Fifteenth European Conference on Computer Systems, pp. 1–15. ACM, Heraklion Greece (2020). DOI 10.1145/3342195.3392698. URL https://doi.org/10.1145/3342195.3392698
23. Chahal, D., Palepu, S., Mishra, M., Singhal, R.: SLA-aware Workload Scheduling Using Hybrid Cloud Services. In: Proceedings of the 1st Workshop on High Performance Serverless Computing, pp. 1–4. ACM, Virtual Event Sweden (2020). DOI 10.1145/3452413.3464789. URL https://doi.org/10.1145/3452413.3464789
24. Chaurasia, N., Kumar, M., Chaudhry, R., Verma, O.P.: Comprehensive survey on energy-aware server consolidation techniques in cloud computing. The Journal of Supercomputing **77**(10), 11682–11737 (2021). DOI 10.1007/s11227-021-03760-1. URL https://doi.org/10.1007/s11227-021-03760-1
25. Chen, Y., Lu, Y., Yang, F., Wang, Q., Wang, Y., Shu, J.: FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (2020). DOI 10.1145/3373376.3378515. URL https://doi.org/10.1145/3373376.3378515

26. Chikhaoui, A., Lemarchand, L., Boukhalfa, K., Boukhobza, J.: Multi-objective Optimization of Data Placement in a Storage-as-a-Service Federated Cloud. ACM Transactions on Storage **17**(3), 1–32 (2021). DOI 10.1145/3452741. URL https://doi.org/10.1145/3452741
27. Cho, J., Tootaghaj, D.Z., Cao, L., Sharma, P.: SLA-Driven ML Inference Framework for Clouds With Heterogeneous Accelerators. p. 13 (2022)
28. Cloud, G.: Cloud tpu vms are generally available (2022). URL https://cloud.google.com/blog/products/compute/cloud-tpu-vms-are-generally-available
29. Cloud Native Computing Foundation: New SlashData report: 5.6 million developers use Kubernetes, an increase of 67% over one year (2021). URL https://www.cncf.io/blog/2021/12/20/new-slashdata-report-5-6-million-developers-use-kubernetes-an-increase-of-67-over-one-year/
30. Cloud Native Computing Foundation: Kubernetes (2022). URL https://kubernetes.io/
31. Cloud Native Computing Foundation: Kubevirt (2022). URL http://kubevirt.io/
32. Cloudflare: Announcing D1: our first SQL database (2022). URL https://blog.cloudflare.com/introducing-d1/
33. D., J.: Baby's First AWS Deployment (2022). URL https://blog.verygoodsoftwarenotvirus.ru/posts/babys-first-aws/
34. Dartois, J.E., B. Ribeiro, H., Boukhobza, J., Barais, O.: Cuckoo: Opportunistic MapReduce on Ephemeral and Heterogeneous Cloud Resources. In: 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), pp. 396–403. IEEE, Milan, Italy (2019). DOI 10.1109/CLOUD.2019.00070. URL https://doi.org/10.1109/CLOUD.2019.00070
35. Diamantopoulos, D., Polig, R., Ringlein, B., Purandare, M., Weiss, B., Hagleitner, C., Lantz, M., Abel, F.: Acceleration-as-a-µService: A Cloud-native Monte-Carlo Option Pricing Engine on CPUs, GPUs and Disaggregated FPGAs. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD), pp. 726–729. IEEE, Chicago, IL, USA (2021). DOI 10.1109/CLOUD53861.2021.00096. URL https://doi.org/10.1109/CLOUD53861.2021.00096
36. Docker Inc.: Docker (2022). URL https://www.docker.com/
37. Dragoni, N., Lanese, I., Larsen, S.T., Mazzara, M., Mustafin, R., Safina, L.: Microservices: How To Make Your Application Scale. In: A.K. Petrenko, A. Voronkov (eds.) Perspectives of System Informatics, vol. 10742, pp. 95–104. Springer International Publishing, Cham (2018). DOI 10.1007/978-3-319-74313-4_8. URL https://doi.org/10.1007/978-3-319-74313-4_8
38. Du, D., Yu, T., Xia, Y., Zang, B., Yan, G., Qin, C., Wu, Q., Chen, H.: Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting. In: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 467–481. ACM, Lausanne Switzerland (2020). DOI 10.1145/3373376.3378512. URL https://doi.org/10.1145/3373376.3378512
39. Elsakhawy, M., Bauer, M.: FaaS2F: A Framework for Defining Execution-SLA in Serverless Computing. In: 2020 IEEE Cloud Summit, pp. 58–65. IEEE, Harrisburg, PA, USA (2020). DOI 10.1109/IEEECloudSummit48914.2020.00015. URL https://doi.org/10.1109/IEEECloudSummit48914.2020.00015
40. Fehling, C., Leymann, F., Retter, R., Schupeck, W., Arbitter, P.: Cloud Computing Patterns. Springer Vienna, Vienna (2014). DOI 10.1007/978-3-7091-1568-8. URL https://doi.org/10.1007/978-3-7091-1568-8
41. Foundation, C.N.C.: Cloud native computing foundation (2022). URL https://www.cncf.io/
42. Foundation, C.N.C.: Knative (2022). URL https://knative.dev/
43. Fowler, M., Lewis, J.: Microservices (2014). URL http://martinfowler.com/articles/microservices.html
44. Fuerst, A., Sharma, P.: FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 386–400. ACM, Virtual USA (2021). DOI 10.1145/3445814.3446757. URL https://doi.org/10.1145/3445814.3446757

45. Gill, S.S., Xu, M., Ottaviani, C., Patros, P., Bahsoon, R., Shaghaghi, A., Golec, M., Stankovski, V., Wu, H., Abraham, A., Singh, M., Mehta, H., Ghosh, S.K., Baker, T., Parlikad, A.K., Lutfiyya, H., Kanhere, S.S., Sakellariou, R., Dustdar, S., Rana, O., Brandic, I., Uhlig, S.: AI for next generation computing: Emerging trends and future directions. Internet of Things **19**, 100514 (2022). DOI 10.1016/j.iot.2022.100514. URL https://doi.org/10.1016/j.iot.2022.100514
46. Golec, M., Ozturac, R., Pooranian, Z., Gill, S.S., Buyya, R.: iFaaSBus: A Security- and Privacy-Based Lightweight Framework for Serverless Computing Using IoT and Machine Learning. IEEE Transactions on Industrial Informatics **18**(5), 3522–3529 (2022). DOI 10.1109/TII.2021.3095466. URL https://doi.org/10.1109/TII.2021.3095466
47. Google: Google Cloud Functions (2022). URL https://cloud.google.com/functions/
48. Google: Google Workflows (2022). URL https://cloud.google.com/workflows/
49. Google: gVisor (2022). URL https://gvisor.dev/
50. Greenberger, M., of Management, S.S., of Technology. School of Industrial Management, M.I.: Management and the Computer of the Future, pp. 220–248. Published jointly by M.I.T. Press and Wiley, New York (1962). URL https://archive.org/details/managementcomput00gree/page/220/
51. Gujarati, A., Elnikety, S., He, Y., McKinley, K.S., Brandenburg, B.B.: Swayam: Distributed autoscaling to meet SLAs of machine learning inference services with resource efficiency. In: Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, pp. 109–120. ACM, Las Vegas Nevada (2017). DOI 10.1145/3135974.3135993. URL https://doi.org/10.1145/3135974.3135993
52. Handaoui, M., Dartois, J.E., Boukhobza, J., Barais, O., d'Orazio, L.: ReLeaSER: A Reinforcement Learning Strategy for Optimizing Utilization Of Ephemeral Cloud Resources. In: 2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), pp. 65–73. IEEE, Bangkok, Thailand (2020). DOI 10.1109/CloudCom49646.2020.00009. URL https://doi.org/10.1109/CloudCom49646.2020.00009
53. Hassan, H.B., Barakat, S.A., Sarhan, Q.I.: Survey on Serverless Computing. Journal of Cloud Computing **10**(1), 39 (2021). DOI 10.1186/s13677-021-00253-7. URL https://doi.org/10.1186/s13677-021-00253-7
54. Hellerstein, J.M., Faleiro, J.M., Gonzalez, J., Schleier-Smith, J., Sreekanti, V., Tumanov, A., Wu, C.: Serverless Computing: One Step Forward, Two Steps Back. In: 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings. www.cidrdb.org (2019). URL http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf
55. Holori: Holori GCP Pricing Calculator (2022). URL https://holori.com/gcp-pricing-calculator/
56. Horta, E., Chuang, H.R., VSathish, N.R., Philippidis, C., Barbalace, A., Olivier, P., Ravindran, B.: Xar-Trek: Run-Time Execution Migration among FPGAs and Heterogeneous-ISA CPUs. In: Proceedings of the 22nd International Middleware Conference, pp. 104–118. ACM, Québec city Canada (2021). DOI 10.1145/3464298.3493388. URL https://doi.org/10.1145/3464298.3493388
57. IBM: IBM Cloud Functions (2022). URL https://cloud.ibm.com/functions/
58. Ionescu, V.: Scaling containers on AWS in 2022 (2022). URL https://www.vladionescu.me/posts/scaling-containers-on-aws-in-2022/
59. Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y.J., Wang, Z., Xu, Y., Dulloor, S.R., Zhao, J., Swanson, S.: Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. ArXiv **abs/1903.05714** (2019). URL https://dblp.uni-trier.de/rec/journals/corr/abs-1903-05714.html
60. Jia, Z., Witchel, E.: Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '21, p. 152–166. Association for Computing Machinery, New York, NY, USA (2021). DOI 10.1145/3445814.3446701. URL https://doi.org/10.1145/3445814.3446701

61. Jiang, J., Gan, S., Liu, Y., Wang, F., Alonso, G., Klimovic, A., Singla, A., Wu, W., Zhang, C.: Towards Demystifying Serverless Machine Learning Training. Proceedings of the 2021 International Conference on Management of Data (2021). DOI 10.1145/3448016.3459240. URL https://doi.org/10.1145/3448016.3459240
62. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., Shankar, V., Carreira, J., Krauth, K., Yadwadkar, N.J., Gonzalez, J.E., Popa, R.A., Stoica, I., Patterson, D.A.: Cloud Programming Simplified: A Berkeley View on Serverless Computing. CoRR **abs/1902.03383** (2019). URL http://arxiv.org/abs/1902.03383
63. Khandelwal, A., Kejariwal, A., Ramasamy, K.: Le Taureau: Deconstructing the Serverless Landscape & A Look Forward. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 2641–2650. ACM, Portland OR USA (2020). DOI 10.1145/3318464.3383130. URL https://doi.org/10.1145/3318464.3383130
64. Kivity, A., Kamay, Y., Laor, D.: Kvm: The Linux Virtual Machine Monitor. In: In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07, p. 8 (2007). URL https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf
65. Klimovic, A., wen Wang, Y., Stuedi, P., Trivedi, A.K., Pfefferle, J., Kozyrakis, C.E.: Pocket: Elastic Ephemeral Storage for Serverless Analytics. login Usenix Mag. **44** (2018). DOI 10.5555/3291168.3291200. URL https://www.usenix.org/conference/osdi18/presentation/klimovic
66. Koomey, J.G.: Growth in data center electricity use 2005 to 2010. Analytics Press for the New York Times (2011). URL https://www.koomey.com/post/8323374335
67. Kuenzer, S., Bădoiu, V.A., Lefeuvre, H., Santhanam, S., Jung, A., Gain, G., Soldani, C., Lupu, C., Teodorescu, Ş., Răducanu, C., Banu, C., Mathy, L., Deaconescu, R., Raiciu, C., Huici, F.: Unikraft: Fast, Specialized Unikernels the Easy Way. In: Proceedings of the Sixteenth European Conference on Computer Systems, pp. 376–394. ACM, Online Event United Kingdom (2021). DOI 10.1145/3447786.3456248. URL https://doi.org/10.1145/3447786.3456248
68. Leite, L., Rocha, C., Kon, F., Milojicic, D., Meirelles, P.: A survey of devops concepts and challenges. ACM Comput. Surv. **52**(6) (2019). DOI 10.1145/3359981. URL https://doi.org/10.1145/3359981
69. Linux Foundation Projects: Open Container Initiative (2022). URL https://opencontainers.org/
70. Linux Foundation Projects: Xen Project (2022). URL https://xenproject.org/
71. Linux Kernel: KVM (2022). URL https://www.linux-kvm.org/page/Main_Page
72. Liu, N., Li, Z., Xu, J., Xu, Z., Lin, S., Qiu, Q., Tang, J., Wang, Y.: A Hierarchical Framework of Cloud Resource Allocation and Power Management Using Deep Reinforcement Learning. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 372–382. IEEE, Atlanta, GA, USA (2017). DOI 10.1109/ICDCS.2017.123. URL https://doi.org/10.1109/ICDCS.2017.123
73. Lloyd, W., Vu, M., Zhang, B., David, O., Leavesley, G.: Improving Application Migration to Serverless Computing Platforms: Latency Mitigation with Keep-Alive Workloads. In: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), pp. 195–200. IEEE, Zurich (2018). DOI 10.1109/UCC-Companion.2018.00056. URL https://doi.org/10.1109/UCC-Companion.2018.00056
74. Lopes, R.V., Menasce, D.: A Taxonomy of Job Scheduling on Distributed Computing Systems. IEEE Transactions on Parallel and Distributed Systems **27**(12), 3412–3428 (2016). DOI 10.1109/TPDS.2016.2537821. URL https://doi.org/10.1109/TPDS.2016.2537821
75. Mackey, K.: Fly Machines: An API for Fast-booting VMs (2022). URL https://fly.io/blog/fly-machines/
76. Magoulas, R., Swoyer, S.: Cloud Adoption in 2020 (2020). URL https://www.oreilly.com/radar/cloud-adoption-in-2020/
77. Manco, F., Lupu, C., Schmidt, F., Mendes, J., Kuenzer, S., Sati, S., Yasukata, K., Raiciu, C., Huici, F.: My VM is Lighter (and Safer) than Your Container. In: Proceedings of the

26th Symposium on Operating Systems Principles, SOSP '17, p. 218–233. Association for Computing Machinery, New York, NY, USA (2017). DOI 10.1145/3132747.3132763. URL https://doi.org/10.1145/3132747.3132763

78. Marshall, P., Keahey, K., Freeman, T.: Elastic Site: Using Clouds to Elastically Extend Site Resources. In: 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 43–52. IEEE, Melbourne, Australia (2010). DOI 10.1109/CCGRID.2010.80. URL https://doi.org/10.1109/CCGRID.2010.80
79. Masanet, E., Shehabi, A., Lei, N., Smith, S., Koomey, J.: Recalibrating global data center energy-use estimates. Science **367**(6481), 984–986 (2020). DOI 10.1126/science.aba3758. URL https://doi.org/10.1126/science.aba3758
80. Matei, O., Skrzypek, P., Heb, R., Moga, A.: Transition from Serverfull to Serverless Architecture in Cloud-Based Software Applications. In: R. Silhavy, P. Silhavy, Z. Prokopova (eds.) Software Engineering Perspectives in Intelligent Systems, vol. 1294, pp. 304–314. Springer International Publishing, Cham (2020). DOI 10.1007/978-3-030-63322-6_24. URL https://doi.org/10.1007/978-3-030-63322-6_24
81. McGrath, G., Brenner, P.R.: Serverless Computing: Design, Implementation, and Performance. In: 2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW), pp. 405–410. IEEE, Atlanta, GA, USA (2017). DOI 10.1109/ICDCSW.2017.36. URL https://doi.org/10.1109/ICDCSW.2017.36
82. Mell, P., Grance, T.: The NIST Definition of Cloud Computing. National Institute of Standards and Technology Special Publication 800-145 (2011). DOI 10.6028/NIST.SP.800-145. URL https://doi.org/10.6028/NIST.SP.800-145
83. Microsoft: Azure Functions (2022). URL https://azure.microsoft.com/products/functions/
84. Microsoft: GitHub (2022). URL https://github.com/
85. Microsoft: Introduction to Hyper-V on Windows (2022). URL https://learn.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/
86. Microsoft: What are durable functions? (2022). URL https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview
87. Mitchell, B.: After 5 years, I'm out of the serverless compute cult (2022). URL https://dev.to/brentmitchell/after-5-years-im-out-of-the-serverless-compute-cult-3f6d
88. Mohan, A., Sane, H., Doshi, K., Edupuganti, S., Nayak, N., Sukhomlinov, V.: Agile cold starts for scalable serverless. In: 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19), p. 6. USENIX Association, Renton, WA (2019). URL https://www.usenix.org/conference/hotcloud19/presentation/mohan
89. Müller, I., Marroquín, R., Alonso, G.: Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 115–130. ACM, Portland OR USA (2020). DOI 10.1145/3318464.3389758. URL https://doi.org/10.1145/3318464.3389758
90. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling memcache at facebook. In: 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pp. 385–398. USENIX Association, Lombard, IL (2013). URL https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala
91. Oakes, E., Yang, L., Zhou, D., Houck, K., Harter, T., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18), pp. 57–70. USENIX Association, Boston, MA (2018). URL https://www.usenix.org/conference/atc18/presentation/oakes
92. Oracle: Fn (2022). URL https://fnproject.io/
93. Oracle: Oracle Cloud Functions (2022). URL https://www.oracle.com/cloud/cloud-native/functions/

94. Oracle: VirtualBox (2022). URL https://www.virtualbox.org/
95. Owens, K.: CNCF WG-Serverless Whitepaper v1.0. Tech. rep., Cloud Native Computing Foundation (2018)
96. Passwater, A.: 2018 Serverless Community Survey: huge growth in serverless usage (2018). URL https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/
97. Perron, M., Fernandez, R.C., DeWitt, D.J., Madden, S.: Starling: A Scalable Query Engine on Cloud Functions. Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (2020). DOI 10.1145/3318464.3380609. URL https://doi.org/10.1145/3318464.3380609
98. Pons, D.B., Artigas, M.S., París, G., Sutra, P., López, P.G.: On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures. Proceedings of the 20th International Middleware Conference (2019). DOI 10.1145/3361525.3361535. URL https://doi.org/10.1145/3361525.3361535
99. Poppe, O., Guo, Q., Lang, W., Arora, P., Oslake, M., Xu, S., Kalhan, A.: Moneyball: Proactive Auto-Scaling in Microsoft Azure SQL Database Serverless. In: VLDB, pp. 1279–1287. ACM (2022). DOI 10.14778/3514061.3514073. URL https://www.microsoft.com/en-us/research/publication/moneyball-proactive-auto-scaling-in-microsoft-azure-sql-database-serverless/
100. Project, F.: Fission (2022). URL https://fission.io/
101. Puviani, M., Frei, R.: Self-Management for Cloud Computing. Science and Information Conference p. 7 (2013). URL https://ieeexplore.ieee.org/document/6661855?arnumber=6661855
102. QEMU team: QEMU (2022). URL https://www.qemu.org/
103. Reeve, J.: Kubernetes: A Cloud (and Data Center) Operating System? (2018). URL https://blogs.oracle.com/cloud-infrastructure/post/kubernetes-a-cloud-and-data-center-operating-system
104. Roberts, M.: Serverless Architectures (2018). URL https://martinfowler.com/articles/serverless.html
105. Romero, F., Chaudhry, G.I., Goiri, Í., Gopa, P., Batum, P., Yadwadkar, N.J., Fonseca, R., Kozyrakis, C.E., Bianchini, R.: Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications. Proceedings of the ACM Symposium on Cloud Computing (2021). DOI 10.1145/3472883.3486974. URL https://10.1145/3472883.3486974
106. Schleier-Smith, J., Sreekanti, V., Khandelwal, A., Carreira, J., Yadwadkar, N.J., Popa, R.A., Gonzalez, J.E., Stoica, I., Patterson, D.A.: What Serverless Computing is and Should Become: The next Phase of Cloud Computing. Commun. ACM **64**(5), 76–84 (2021). DOI 10.1145/3406011. URL https://doi.org/10.1145/3406011
107. Schuler, L., Jamil, S., Kuhl, N.: AI-based Resource Allocation: Reinforcement Learning for Adaptive Auto-scaling in Serverless Environments. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid), pp. 804–811. IEEE, Melbourne, Australia (2021). DOI 10.1109/CCGrid51090.2021.00098. URL https://doi.org/10.1109/CCGrid51090.2021.00098
108. Shafiei, H., Khonsari, A., Mousavi, P.: Serverless Computing: A Survey of Opportunities, Challenges, and Applications. ACM Comput. Surv. (2022). DOI 10.1145/3510611. URL https://doi.org/10.1145/3510611. Just Accepted
109. Shahin, M., Ali Babar, M., Zhu, L.: Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices. IEEE Access **5**, 3909–3943 (2017). DOI 10.1109/ACCESS.2017.2685629. URL https://doi.org/10.1109/ACCESS.2017.2685629
110. Shahrad, M., Fonseca, R., Goiri, Í., Chaudhry, G., Batum, P., Cooke, J., Laureano, E., Tresness, C., Russinovich, M., Bianchini, R.: Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider p. 14 (2020). URL https://www.usenix.org/conference/atc20/presentation/shahrad

111. Shaw, R., Howley, E., Barrett, E.: Applying Reinforcement Learning towards automating energy efficient virtual machine consolidation in cloud data centers. Information Systems p. 21 (2022). DOI 10.1016/j.is.2021.101722. URL https://doi.org/10.1016/j.is.2021.101722
112. Shillaker, S., Pietzuch, P.: FAASM: Lightweight Isolation for Efficient Stateful Serverless Computing. USENIX Association, USA (2020). DOI 10.5555/3489146.3489174. URL https://dl.acm.org/doi/abs/10.5555/3489146.3489174
113. Singhvi, A., Balasubramanian, A., Houck, K., Shaikh, M.D., Venkataraman, S., Akella, A.: Atoll: A Scalable Low-Latency Serverless Platform. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 138–152. ACM, Seattle WA USA (2021). DOI 10.1145/3472883.3486981. URL https://doi.org/10.1145/3472883.3486981
114. Soni, D., Kumar, N.: Machine learning techniques in emerging cloud computing integrated paradigms: A survey and taxonomy. Journal of Network and Computer Applications **205**, 103419 (2022). DOI 10.1016/j.jnca.2022.103419. URL https://doi.org/10.1016/j.jnca.2022.103419
115. SPEC Research Group: SPEC Research Group (2022). URL https://research.spec.org/
116. Sreekanti, V., Wu, C., Lin, X.C., Schleier-Smith, J., Faleiro, J.M., Gonzalez, J.E., Hellerstein, J.M., Tumanov, A.: Cloudburst: Stateful Functions-as-a-Service. Proc. VLDB Endow. **13**, 2438–2452 (2020). DOI 10.14778/3407790.3407836. URL https://doi.org/10.14778/3407790.3407836
117. Taibi, D., El Ioini, N., Pahl, C., Niederkofler, J.: Patterns for Serverless Functions (Function-as-a-Service): A Multivocal Literature Review:. In: Proceedings of the 10th International Conference on Cloud Computing and Services Science, pp. 181–192. SCITEPRESS - Science and Technology Publications, Prague, Czech Republic (2020). DOI 10.5220/0009578501810192. URL https://doi.org/10.5220/0009578501810192
118. Tan, B., Liu, H., Rao, J., Liao, X., Jin, H., Zhang, Y.: Towards Lightweight Serverless Computing via Unikernel as a Function. In: 2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS), pp. 1–10. IEEE, Hang Zhou, China (2020). DOI 10.1109/IWQoS49365.2020.9213020. URL https://doi.org/10.1109/IWQoS49365.2020.9213020
119. Thalheim, J., Okelmann, P., Unnibhavi, H., Gouicem, R., Bhatotia, P.: VMSH: Hypervisor-Agnostic Guest Overlays for VMs. In: Proceedings of the Seventeenth European Conference on Computer Systems, pp. 678–696. ACM, Rennes France (2022). DOI 10.1145/3492321.3519589. URL https://doi.org/10.1145/3492321.3519589
120. The Linux Foundation: Linux Foundation (2022). URL https://www.linuxfoundation.org/
121. Ustiugov, D., Petrov, P., Kogias, M., Bugnion, E., Grot, B.: Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 559–572. ACM, Virtual USA (2021). DOI 10.1145/3445814.3446714. URL https://doi.org/10.1145/3445814.3446714
122. Vahidinia, P., Farahani, B., Aliee, F.S.: Cold Start in Serverless Computing: Current Trends and Mitigation Strategies. In: 2020 International Conference on Omni-layer Intelligent Systems (COINS), pp. 1–7. IEEE, Barcelona, Spain (2020). DOI 10.1109/COINS49042.2020.9191377. URL https://doi.org/10.1109/COINS49042.2020.9191377
123. van Eyk, E., Iosup, A., Abad, C.L., Grohmann, J., Eismann, S.: A SPEC RG Cloud Group's Vision on the Performance Challenges of FaaS Cloud Architectures. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pp. 21–24. ACM, Berlin Germany (2018). DOI 10.1145/3185768.3186308. URL https://doi.org/10.1145/3185768.3186308
124. Vaquero, L.M., Rodero-Merino, L., Morán, D.: Locking the Sky: A Survey on IaaS Cloud Security. Computing **91**(1), 93–118 (2011). DOI 10.1007/s00607-010-0140-x. URL https://doi.org/10.1007/s00607-010-0140-x

125. Vilanova, L., Maudlej, L., Bergman, S., Miemietz, T., Hille, M., Asmussen, N., Roitzsch, M., Härtig, H., Silberstein, M.: Slashing the Disaggregation Tax in Heterogeneous Data Centers with FractOS. In: Proceedings of the Seventeenth European Conference on Computer Systems, pp. 352–367. ACM, Rennes France (2022). DOI 10.1145/3492321.3519569. URL https://doi.org/10.1145/3492321.3519569
126. Villamizar, M., Garces, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., Gil, S.: Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In: 2015 10th Computing Colombian Conference (10CCC), pp. 583–590. IEEE, Bogota, Colombia (2015). DOI 10.1109/ColumbianCC.2015.7333476. URL https://doi.org/10.1109/ColumbianCC.2015.7333476
127. VMware: ESXi (2022). URL https://www.vmware.com/products/esxi-and-esx.html
128. VMware: Openfaas (2022). URL https://www.openfaas.com/
129. Wanninger, N.C., Bowden, J.J., Shetty, K., Garg, A., Hale, K.C.: Isolating Functions at the Hardware Limit with Virtines. In: Proceedings of the Seventeenth European Conference on Computer Systems, pp. 644–662. ACM, Rennes France (2022). DOI 10.1145/3492321.3519553. URL https://doi.org/10.1145/3492321.3519553
130. Weissman, C.D., Bobrowski, S.: The Design of the Force.Com Multitenant Internet Application Development Platform. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, pp. 889–896. ACM, Providence Rhode Island USA (2009). DOI 10.1145/1559845.1559942. URL https://doi.org/10.1145/1559845.1559942
131. Wiggins, A.: The Twelve-Factor App (2017). URL https://12factor.net/
132. Wu, C., Faleiro, J.M., Lin, Y., Hellerstein, J.M.: Anna: A KVS for Any Scale. 2018 IEEE 34th International Conference on Data Engineering (ICDE) pp. 401–412 (2018). DOI 10.1109/TKDE.2019.2898401. URL https://doi.org/10.1109/TKDE.2019.2898401
133. Wu, C., Sreekanti, V., Hellerstein, J.M.: Transactional Causal Consistency for Serverless Computing. In: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, pp. 83–97. ACM, Portland OR USA (2020). DOI 10.1145/3318464.3389710. URL https://doi.org/10.1145/3318464.3389710
134. Yalles, S., Handaoui, M., Dartois, J.E., Barais, O., d'Orazio, L., Boukhobza, J.: Riscless: A reinforcement learning strategy to guarantee sla on cloud ephemeral and stable resources. In: 2022 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 83–87 (2022). DOI 10.1109/PDP55904.2022.00021. URL https://doi.org/10.1109/PDP55904.2022.00021
135. Zhang, T., Xie, D., Li, F., Stutsman, R.: Narrowing the Gap Between Serverless and its State with Storage Functions. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 1–12. ACM, Santa Cruz CA USA (2019). DOI 10.1145/3357223.3362723. URL https://doi.org/10.1145/3357223.3362723
136. Zhang, W., Fang, V., Panda, A., Shenker, S.: Kappa: A Programming Framework for Serverless Computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing, SoCC '20, p. 328–343. Association for Computing Machinery, New York, NY, USA (2020). DOI 10.1145/3419111.3421277. URL https://doi.org/10.1145/3419111.3421277
137. Zomer, J.: Escaping privileged containers for fun (2022). URL https://pwning.systems/posts/escaping-containers-for-fun/