An Overview on Serverless Computing, its Opportunities and its Challenges

Nicolò Monaldini

February 2025

Abstract

Serverless computing has gained importance in both academia and industry due to its cost-efficiency, automated scalability, and virtually unlimited resource availability, offering a novel pay-as-you-go model compared to traditional cloud computing solutions. The serverless paradigm not only eases end-users of the burden of infrastructure management, but its cost model enables users to pay only for resources consumed, while the provider bears the cost of idle resources. However, the stateless nature of the services offered and communication needs of related executions are problems that need to be addressed, as they could pose performance issues and limit the applications of the paradigm. This survey provides an overview of the current advancements in serverless computing, examining its opportunities, challenges, and potential solutions.

1 Introduction

Traditional cloud computing paradigms such as *Infrastructure-as-a-service* (IaaS) offer on-demand access to computing resources. However, the end-users need to configure the machines and deploy autonomously. They also need to reserve resources based on the peak usage of their applications, and this exclusive allocation results in a strong under utilization of data centers by only about 10% on average [1]. Therefore, serverless computing has gained traction as a way to address these criticalities, and many large technology companies offer serverless platforms. While there are slight differences among the services offered on the market, the fundamental idea is similar: a pay-as-you-model that enables end-users to run their services transparently from the underlying infrastructure. The serverless platforms take care of all the system administration tasks, including configuration, deployment, and scaling management, while the users focus on the functionalities they intend to provide.

Serverless makes use of the *Function-as-a-service* (FaaS) paradigm, allowing end-users to easily deploy services to external providers. In this paradigm, an application is sliced into functions that are then sent to a platform; their execution is event-driven, meaning each function runs in response to an event, and the provider manages their scheduling. The functions are not hidden from the provider: it knows information such as the external dependencies and the runtime environment, and produces logs about the executions as well. Specifically, the following data are necessary to define a function, as explained in the implementation provided by McGrath et al. [2]:

- **Function identifier**: a randomly generated Global Unique Identifier to identify and locate function's resources
- **Language runtime**: it specifies the language of the function's code
- **Memory size**: the maximum memory a function's container can consume
- **Code blob URI**: a URI identifying an archive containing the function's code

that will be executed

While serverless and FaaS are often used interchangeably, they represent different concepts: FaaS is a paradigm used in serverless computing to enable function isolation and invocation, delegating their execution to an external provider, and while it is essential to serverless computing, *Backend-as-a-Service* (BaaS) is a fundamental component of serverless as well. BaaS is a cloud service model originally used for mobile and web development, used to outsource backend aspects of these services, such as hosting and database management. It has been grown beyond just mobile-centric services to encompass serverless as well, since it provides overall backend support for this type of service. As stated in Berkley View [3], serverless is defined as *FaaS + BaaS*. A serverless system is made of various parts:

- A **controller** manages the execution of functions; it is event-driven and triggered by events created consequently to users' actions, such as REST-ful HTTP queries or adding new data to a database.
- A **database** contains users' functions code and outputs of completed executions.
- Functions are executed in **sandboxed containers**. After the execution, the output is saved to the database. Logfiles are produced and sent to the provider, to then be used to optimize further executions of the function [4].

An example of an asynchronous invocation in serverless computing is depicted in Figure 1. Based on the status of containers, two types of startups are defined:

- **Warm start**: it is the case where a sandbox from a previous execution that just concluded is reused
- **Cold start**: it is the case where a new sandbox is started upon the function's execution, or it was previously hibernated by the provider to decrease its energy consumption

A cold start incurs in a higher delay, which is dominated by the setup of the software environment (e.g. libraries loading), and can take up to tens of seconds [3]. Copik et al. [5] measured the execution of an empty C++ function with various providers, resulting in startup delays in the order of tens or hundreds of milliseconds depending on the provider. Moreover, the provider takes cares of the scheduling, allocating computer resources to various customers; therefore, there is always a risk of queuing delays in case of high demand for the service.

1.1 Methodology

Serverless computing is a broad and complex field that is closely linked to various topics, such as performance, security or caching. While some publications focus on specific areas by reviewing the latest academic literature in those fields, this work adopts a broader approach in order to provide a comprehensive overview from multiple perspectives. It first analyzes surveys and articles to cover critical topics closely connected to serverless computing and, additionally, it explores innovative and notable studies addressing individual challenges, ultimately offering a holistic perspective on serverless computing.

2 Cost-effectiveness

Serverless computing appears more expensive than traditional cloud computing at a first analysis: the per-minute execution cost of an AWS Lambda function is 7.5 times as expensive compared to an AWS t3.nano VM. As Schleier et al. observe [6], however,

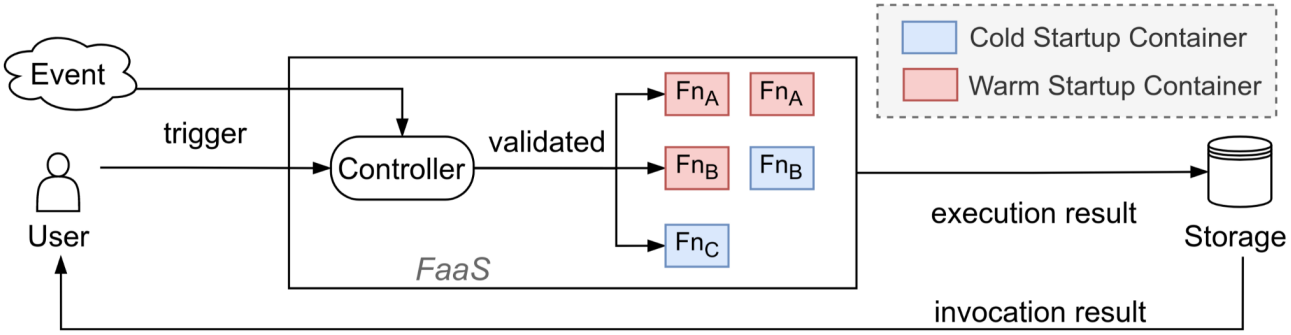

Figure 1: *Example of an asynchronous invocation. Source: [1].*

serverless providers include in the price the cost of redundancy for ensuring availability, auto-scaling, monitoring and logging, which are functionalities that are under the responsibility of developers in traditional cloud computing approaches (which this work refers to as "serverful"), or have to be purchased as separate services. Another benefit of serverless computing is its cost model, which follows a pay-as-you-go pricing structure. In this model, end-users are charged only for the resources they actually consume, while the cloud provider covers the costs of any idle resources, as illustrated in Figure 2. Even with dynamic resource allocation, serverful computing operates on a reservation model, requiring computing resources to be allocated based on the application's peak usage, which leads to a very large underutilization of resources. As Castro et al. [6] point out, in practice, customer realize saving when moving their applications to serverless. The price reductions and the increased functionalities offered by cloud providers could threaten their revenues; however, as noted by the Jevons Paradox [7], lower prices and fewer customer responsibilities can stimulate demand, offsetting the price decrease and potentially boosting providers' earnings.

3 Security and Privacy

Since serverless computing is inherently event-driven and the services provided by the platforms are cloud-based, authentication becomes a critical challenge: only authorized applications should be able to trigger events that start function executions on the cloud, otherwise a freeloader could take advantage of the available resources of the victims [4]. A token-based authentication strategy could be employed, where specific tokens are included in requests, as AWS Lambda implements[1]; however, these tokens could be extracted and reused if the message is sent through an unsecured network. An effective solution would be to use SSL/TLS protocols [8, 9], but it may be beyond the capabilities of low-energy IoT devices.

Using IoT devices in serverless environments is not uncommon: due to the auto-scalability property of serverless computing, it can easily handle large bursts of requests, which frequently occur when IoT systems monitor environmental conditions that can generate numerous events (consider, for example, the case of an earthquake detection network that might rapidly generate extensive data whenever seismic activity is detected). IoT applications might also be very privacy-sensitive: an example that Shafiei et al. [4] point out is a use case in healthcare, where IoT devices transfer patient data: in

[1] `https://docs.aws.amazon.com/apigateway/`

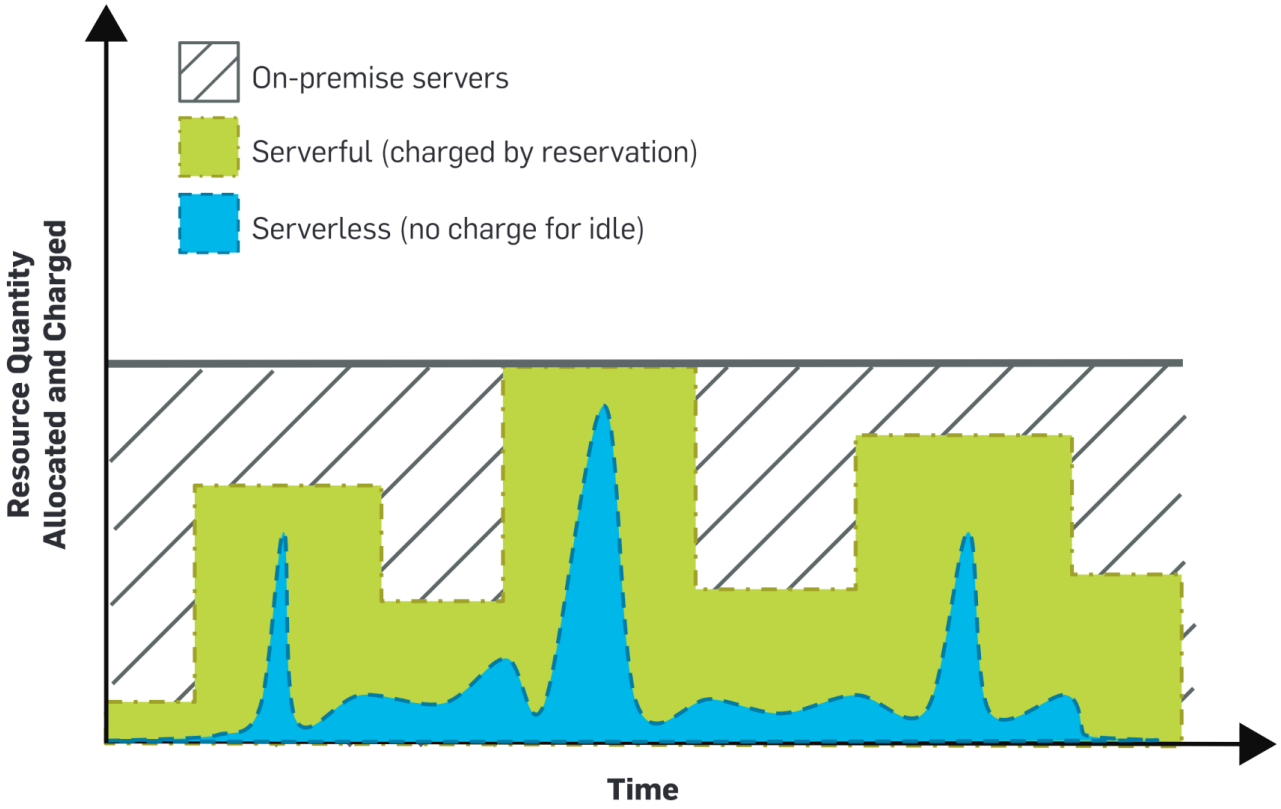

Figure 2: *Advantages of serverless computing's cost model. Source: [6].*

this scenario, an attacker could extract sensitive information about patients only by having access to the name of the function called and its parameters.

Authorization is another central aspect: certain functions might be restricted to be called only by certain users or functions. To tackle this issue, AWS Lambda uses a role-based approach, where every function is associated with a particular set of permissions and users have a role that define the functions they are allowed to execute. User executions can be isolated, allowing functions to run within the caller's context, as depicted in Figure 3.

Moreover, security concerns such as replay attacks are significant [4]: in this type of attacks, the attacker does not know the content of a message, but is interested in the effects of its transmission; in the context of serverless, this could lead to resource exhaustion attacks in the case of multiple malicious function calls that result in a usage beyond the service-level agreement with the provider. Despite this, as Marin et al. [11] highlight, the serverless paradigm is more resistant to (Distributed) Denial-of-Service attacks compared to other approaches, as it enjoys elasticity provided by its auto-scaling feature. Rather than the disruption of the service, a more common consequence of attacks is a heavy financial load on the application owner.

On the bright side, serverless can act as a security enabler under some points of view [11]:

- It increases difficulty in performing attacks: serverless functions are short-lived, and this imposes time limits on the adversaries to retrieve data or to reach high privileged accounts to perform further damage.
- It relieves developers of some security concerns: security becomes a shared concern between the developers and the providers. Providers take care of many

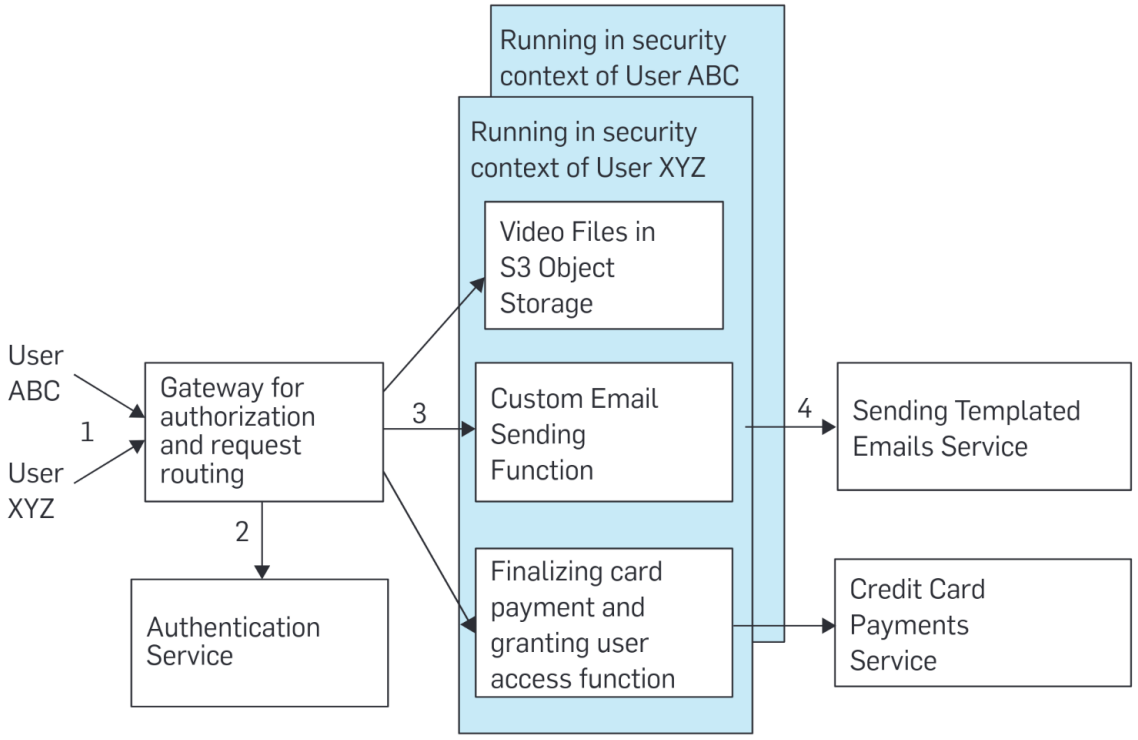

Figure 3: *Authorization and isolated execution in serverless. Source: [10].*

security tasks, especially related to the infrastructure; however, developers still need to the heavily involved in security matters, securing their workflows by preventing functions' vulnerabilities, limiting execution privileges and securing the data on the cloud.

4 Data Caching

Applications across various domains frequently rely on querying databases to retrieve and manipulate data essential for their functionalities, which in the case of serverless platforms are cloud databases. Querying cloud databases may result in bottlenecks due to the latencies of persistent storage. Serverful systems use multiple level of caches, but this approach can be challenging to implement in serverless environments, given the stateless nature of the functions. Since a lack of caching makes the serverless services infeasible in real-world scenarios, there were various attempts to implement caching, mainly in-memory and distributed on multiple nodes. Amazon developed ElastiCache[2], an in-memory cache and data store, but it is at least 700 times more expensive compared to Amazon S3 (Amazon's cloud storage service[3]) [4]; therefore, open source alternatives such as InfiniCache [12] were also implemented. The implementation of a distributed cache is not straightforward[4]: a first idea that might come to mind is caching data between requests in an in-memory hashmap; however, the size of functions is limited to some gigabytes by serverless providers, and every instance of the same function would need to keep in the memory potentially the same data (for

[2] https://aws.amazon.com/elasticache/
[3] https://aws.amazon.com/s3/
[4] An in-depth explanation of the problem can be found at https://mikhail.io/2020/03/infinicache-distributed-cache-on-aws-lambda/

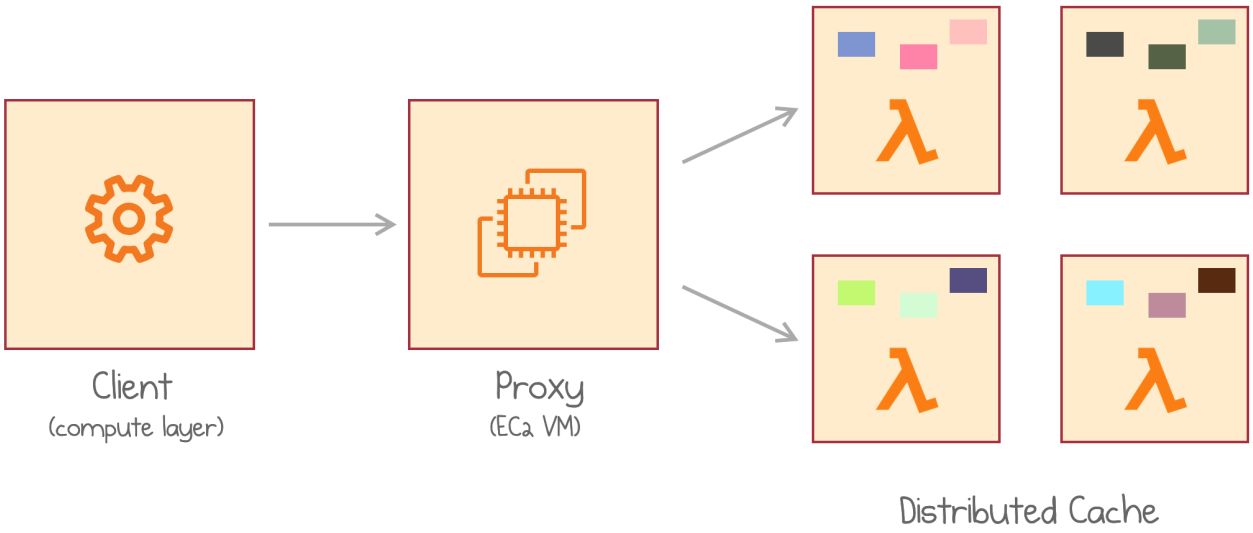

Figure 4: *InfiniCache workflow. Source: www.mikhail.io.*

example, in the case of parallel requests). Another option would be to implement a simple distributed caching system by moving the cache in multiple functions (units of execution) that contains a portion of the data. A big upside of this approach would be the caching pay model: cost is charged only when cache functionalities are used (i.e. function instances are running), while cluster-based services charge for memory capacity whether cached objects are accessed or not. However, serverless functions are not designed for stateful use cases: they have limited resources (both CPU and memory-wise) and can run for some minutes at most. Moreover, providers such as AWS Lambda reserve the right to terminate idle functions at any time. InfiniCache tackles these problems with a component, called **proxy**, that manages multiple functions that hold slices of data. Clients communicate with the proxy, which orchestrates the caching transparently from them, as depicted in Figure 4. Since InfiniCache is implemented on AWS Lambda, where the provider can claim idle instances at any time, it implements a backup mechanism for cache functions, creating redundancy with multiple functions containing the same data. In this way, if the proxy has to retrieve data in the cache but the primary instance containing the requested data was shut down, it can reroute the request to its backup. On AWS Lambda the functions cannot receive TCP or UDP connections; therefore, to allow communication with the proxy, the proxy's IP and port are passed as parameters during the function invocation and the function itself starts a TCP connection towards the proxy during initialization, as shown in Figure 5.

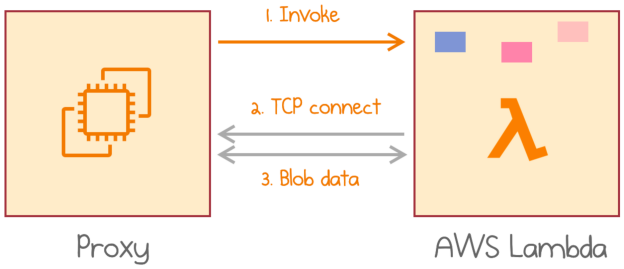

Figure 5: *Invocation of a function and connection to the proxy. Source: www.mikhail.io.*

5 Performance

While the serverless paradigm can bring the elasticity to allocate idle resources and even reduce computational costs, as shown in Section 2, its performance cannot match cluster

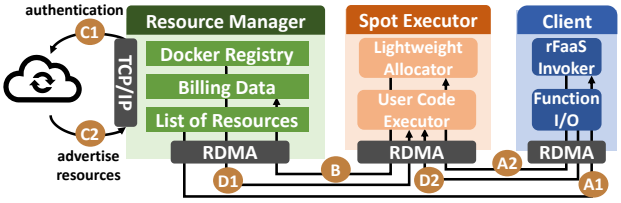

Figure 6: *Architecture of rFaaS: clients acquire leases (A) and invoke functions (D), spot executors manage execution environments, resource managers interact with data center resources (C) and manage billing (B). Source: [5].*

allocations and coarse-grained cloud executions. Serverless poses some limitations to High-Performance Computing applications, and, more in general, to latency sensitive applications, since many platforms make use of centralized scheduling with a high number of parties involved in functions executions that increase execution latencies. Moreover, inefficient network transport protocols are used, limiting the use of serverless for latency-sensitive applications. To limit these issues and make serverless a feasible option for the aforementioned applications, rFaaS [5], a Remote Direct Access Memory (RDMA) accelerated FaaS platform, aims to reduce execution latencies by limiting the overhead on the fastest available network.

rFaaS introduces various components, as depicted in Figure 6. **Spot executors** are entities that provide hardware and software resources to clients to support the execution of serverless functions; each spot executor is assigned a part of the available resources. A spot executor includes:

- **User code executors**, executor processes responsible for the execution of client code
- A **lightweight allocator**, which is responsible for connecting new clients (A2), managing code executors and the processes being executed, and removing processes that are idle or that are exceeding specific time limits

When a client wants to execute code it connects to a spot executor which initializes an isolated execution context with a user code executor, that can be implemented as a bare-metal process or a Docker container (D1). The client can then leverage the RDMA capabilities of the service to write functions headers and payloads directly to the code executor's memory (D2). Spot executors are managed by entities called **resource managers**: each one is assigned a group of spot executors and interacts with data center resources (C), assigning them to spot executors when they become available. Resource managers also handle billing data for the clients (B). An important concept introduced by rFaaS to speedup consecutive invocations is **leasing**: clients acquire leases on executor processes, and leverage them to perform consecutive executions on warmed up containers. Leases allow to avoid the initialization costs of reliable RDMA connections and limit the interaction between clients and resource managers: the latter is involved only once to acquire a lease (A1), and not every time a function is invoked. The separation of concerns between resource managers and spot executors allows to reduce the critical path of invocations by removing a centralized gateway, ultimately reducing the number of parties involved in transmitting functions data.

To reduce the startup delays, alongside warm and cold executions, rFaaS introduces the concept of **hot invocations**, a sandbox state where, after a request is completed, the threads continue to active poll for invocation requests. Busy polling decreases the invocation latency since threads do not enter blocked states waiting for RDMA events. This operation comes at the cost of occupying the CPU, and so it should be considered as active computation time; the executors can switch back to normal warm mode after a configurable amount of time to free the CPU.

The use of hot containers decreases the

overall overhead to about 350 nanoseconds on top of the fastest available network transmission. While this implementation of a RMDA-accelerated serverless platform requires specific hardware and lacks some features offered by cloud providers, such as distributed storage on the cloud, it is a great attempt to adapt serverless computing to scenarios with strict latency constraints.

6 Scheduling

Section 5 described an alternative architecture that can optimize scheduling, but most providers use a different architecture where functions scheduling is transparent from the end users. However, these providers can still optimize scheduling based on many parameters. Shafiei et al. [4] provide an in-depth analysis of the various scheduling policies available.

6.1 Resource Consumption Patterns

There is various information that is useful to know for the provider to ensure proper scheduling. Understanding data consumption patterns of tasks that are executed is crucial; for instance, if multiple CPU-intensive functions run on the same physical machine, it may lead to resource contention. Similarly, tasks that transfer large amounts of data over the network could introduce communication delays if they are co-located on the same machine. Therefore, a resource-aware scheduling can take these applications needs into account to limit the concurrent high demand on shared hardware resources.

6.2 Reducing Startup Overhead

Delays in serverless applications have multiple causes, including cold container startups, remote database queries and dependency installations, as explained in Sections 4 and 5. Providers can aim to mitigate these issues by incorporating predictive and speculative scheduling. In fact, serverless applications rarely involve only one function invocation, but they are mostly made by chain of invocations, where multiple functions are called concurrently, or a function invokes other functions. Providers could prepare warm containers based on known or anticipated invocation patterns, reducing startup overhead. Functions can often install dependencies as well, and batching together executions that require to download the same dependencies can further reduce overheads.

6.3 Data Locality

While functions are stateless, it is not uncommon to depend on external data sources to perform the computations; therefore, scheduling strategies increasingly prioritize data locality. By placing functions closer to the data they process, providers can significantly reduce data transfer latencies, improving overall applications' performance.

7 Statistics on Serverless Usage

Despite the leading serverless platform (AWS Lambda) was launched in 2015, serverless is still trending and growing. Datadog stated in its last report[5] that AWS Lambda grew by 3% from 2022 to 2023, while Azure and Google Cloud grew by 6% and 7% respectively. Python and Node.js are the leading languages used in serverless environments, and for good reasons: the average startup time of functions written in Java (which is the third leading language) is two times longer compared to Python or Node.js. The reason is that startup time is influenced by the memory allocated to a function, and more than 25% of all Java functions allocate

[5] https://www.datadoghq.com/state-of-serverless/

more than 1024 MB per function, compared to only about 10% of functions written in Python and Node.js.

7.1 Functions Marketplaces

Functions marketplaces offer pre-built serverless functions to be easily deployed; the most notable example is the AWS Serverless Application Repository[6], that counts more than 1100 functions. As Shafiei et al. [4] point out, the competition on these platforms could result in an increase of functions quality, ultimately in favor of the customers.

Spillner [13] provides a quantitative analysis of the functions in the AWS Serverless Application Repository. Although the report dates back to 2019, when the repository had roughly half of the functions it has today, the conclusion are partially analogous to the ones in the Datadog report cited above, particularly about the dominance of Python and Node.js as the leading programming languages for serverless.

8 Conclusion

Serverless computing represents a transformative shift in cloud computing, and porting applications to serverless environments would offer benefits such as cost-efficiency, automated scalability, and simplified infrastructure management. Its pay-as-you-go model has unlocked new possibilities for businesses and developers, allowing them to focus on innovation rather than operational complexities. However, this is not without challenges, since the stateless nature of the paradigm and its latencies compared to typical serverful implementations can limit its expansion, and need to be addressed.

Through this work, an overview of the topic from multiple perspectives was provided, highlighting not only the paradigm's potentialities but also the challenges serverless computing is facing; and previous innovative efforts that aim to limit or solve some limitations were analyzed. Addressing these limitations is crucial for serverless computing to mature into a more robust, adaptable, and widely applicable technology for the future of cloud services.

[6] https://serverlessrepo.aws.amazon.com/applications

References

[1] Zijun Li et al. "The Serverless Computing Survey: A Technical Primer for Design Architecture". en. In: *ACM Computing Surveys* 54.10s (Jan. 2022). arXiv:2112.12921 [cs], pp. 1–34. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3508360. URL: http://arxiv.org/abs/2112.12921.

[2] Garrett McGrath and Paul R. Brenner. "Serverless Computing: Design, Implementation, and Performance". In: *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. ISSN: 2332-5666. June 2017, pp. 405–410. DOI: 10.1109/ICDCSW.2017.36. URL: https://ieeexplore.ieee.org/document/7979855.

[3] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. en. arXiv:1902.03383 [cs]. Feb. 2019. DOI: 10.48550/arXiv.1902.03383. URL: http://arxiv.org/abs/1902.03383.

[4] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. "Serverless Computing: A Survey of Opportunities, Challenges, and Applications". en. In: *ACM Computing Surveys* 54.11s (Jan. 2022), pp. 1–32. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3510611. URL: https://dl.acm.org/doi/10.1145/3510611.

[5] Marcin Copik et al. *rFaaS: Enabling High Performance Serverless with RDMA and Leases*. en. arXiv:2106.13859 [cs]. May 2023. DOI: 10.48550/arXiv.2106.13859. URL: http://arxiv.org/abs/2106.13859.

[6] Johann Schleier-Smith et al. "What serverless computing is and should become: the next phase of cloud computing". en. In: *Communications of the ACM* 64.5 (May 2021), pp. 76–84. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3406011. URL: https://dl.acm.org/doi/10.1145/3406011.

[7] Blake Alcott. "Jevons' paradox". In: *Ecological Economics* 54.1 (July 2005), pp. 9–21. ISSN: 0921-8009. DOI: 10.1016/j.ecolecon.2005.03.020. URL: https://www.sciencedirect.com/science/article/pii/S0921800905001084.

[8] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The Secure Sockets Layer (SSL) Protocol Version 3.0*. Request for Comments RFC 6101. Num Pages: 67. Internet Engineering Task Force, Aug. 2011. DOI: 10.17487/RFC6101. URL: https://datatracker.ietf.org/doc/rfc6101.

[9] Eric Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. Request for Comments RFC 8446. Num Pages: 160. Internet Engineering Task Force, Aug. 2018. DOI: 10.17487/RFC8446. URL: https://datatracker.ietf.org/doc/rfc8446.

[10] Paul Castro et al. "The rise of serverless computing". en. In: *Communications of the ACM* 62.12 (Nov. 2019), pp. 44–54. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3368454. URL: https://dl.acm.org/doi/10.1145/3368454.

[11] Eduard 1 Marin et al. "Serverless computing: a security perspective". English. In: (Dec. 2022). Publisher: Springer Nature B.V. DOI: 10.1186/s13677-022-00347-w. URL: https://www.proquest.com/docview/2727498922?pq-origsite=primo&sourcetype=Scholarly%20Journals.

[12] Ao Wang et al. "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache". en. In: ().

[13] Josef Spillner. *Quantitative Analysis of Cloud Function Evolution in the AWS Serverless Application Repository*. en. arXiv:1905.04800 [cs]. May 2019. DOI: 10.48550/arXiv.1905.04800. URL: http://arxiv.org/abs/1905.04800.