AGH University of Science and Technology

Faculty of Computer Science, Electronics and Telecommunications

Networks Evolution

Serverless Mobile Computing

Jan Ściga, Wiktor Tekiela, Christian Białek, Mateusz Zdyrski,
Marcin Karcz

Cracow, 28/05/2023

Contents

1 Introduction

Nowadays, many companies give consideration whether they should migrate their resources to a cloud or stay with a traditional computing. Both models have their advantages and disadvantages, therefore choosing which option best suits one needs is a difficult task. Each application has its own set of requirements, a workload, as well as a specific type of traffic traversing towards it. Factors such as latency, scalability, compliance requirements and cost, should be carefully examined before making such a crucial decision. In this paper, we investigate if the implementation of the Domain Name System (DNS) with a serveless computing is able to provide some benefits and where such a solution might face some inconveniences.

In the traditional way, different network components are hardware-based. It means that specialized software is run on the proprietary hardware providing network functions such as routing, load balancing or firewalling. The main drawback of such a solution is that it requires dedicated devices, which have to be physically wired, properly configured and then constantly maintained. It poses the risk of creating difficult to manage and highly unscalable infrastructure. Consequently, it can become quite expensive and time consuming making the capital and operational expenditures (CapEx and OpEx) very high. However, despite forementioned difficulties, traditional way of deploying network components is still used in many environments, which require meeting specific performance and security measures.

Such a situation often applies to the Domain Name System (DNS), which is a critical component in today's network infrastructure. The process of DNS lookup translates domain names into IP addresses and is directly or indirectly used by a vast majority of devices. Therefore, it plays a crucial role in providing a good user experience, the faster the domain name is resolved the better. Hence, it is important to ensure that DNS operates without any disruption, with a special regard to performance and security. Currently, DNS servers are usually implemented on bare metal environments. It enables the full control over such a server in terms of both hardware and software, allowing organizations to customize it for their needs.

However, with the rapid growth of virtualization, the natural course of events was to implement different network functions in this technology. Such a recent solution is so-called the Network Function Virtualization (NFV) [1], which decouples a specialized software from a proprietary hardware. In other words, software-based implementations of network functions called VNFs (Virtual Network Functions), which are usually implemented as containerized applications or serverful virtual machines, can be run on commodity hardware, thus dedicated one is not required anymore. Examples of such VNFs may involve, among others, virtual routers, firewalls, load balancers, DNS and DHCP servers. What is more, NFV orchestration system provides dynamic deployment of such VNFs according to the current needs as well as plenty of possibilities of network management. Created on-demand network resources make the process of scaling up much faster and more efficient. This solution enables service providers to build highly scalable and flexible systems. Automatic deployments, scaling as well as better management speeds up the whole process of launching new services and is able to significantly reduce costs and complexity of network infrastructures.

One step further from the virtualization is the cloud computing. Worth mentioning is the fact that the virtualization is the foundation of the cloud computing. The former enables to run many virtual machines (VMs) as virtual resources on a single physical server, while the latter provides virtualized computing resources on demand as a service over a network (e.g. the internet). As it is in the title of this paper, we mainly focused on a *Serverless Cloud Computing* execution model, where the *Mobile* refers to the network function being implemented. In this model, the cloud provider is responsible for all of the underlying infrastructure such as servers, operating systems, data storage and runtime environments as well as provision them dynamically when needed. This enables developers not to worry about forementioned details, but to focus on providing the desired application. In order to implement the solution for this study, we will especially make use of a Function as a Service (FaaS) model, which utilizes so-called Serverless functions. These functions are usually stateless, self-contained units of code written in programming language of choice (if supported) and can be run in response to an API call or in general to a predefined set of events (commonly used in an Event-Driven Architecture [2]). Serverless cloud computing model will be discussed in more detail in Section 2.

As for our system under test, we have chosen to implement the Domain Name System (specifically DNS-over-HTTPS) as a serverless function. There are different types of DNS servers, however, we implemented only the Authoritative one [3], which is a last stop of the dns query and contains records for the specific domain. DNS is perceived as a request-based function, so in order to obtain a proper response, a specific request must be sent. This raises a question if this functionality could be implemented with the use of FaaS model and if it is even worth it. Our main motivation was that there are not a lot of articles on this topic of virtualized DNS. What is more, some research examined other network functions like NAT, DHCP [4] and MME [5] from LTE core network, but only suggested that DNS could be a potential next network function that could be successfully virtualized. This prompted us to examine this field further. In order to make our results more reliable and valuable, we decided to perform tests on two independent cloud providers: Amazon (AWS Lambda [6]) and Google (GCP Cloud Functions [7]). As for the external database, we have chosen MongoDB [8] provided as a external cloud cluster.

While Function-as-a-Service (FaaS) provides many advantages, there are also some potential drawbacks, which possibly may result in implemented product being unprofitable. In this research, we want to test those limitations and check whether it is worth to further develop this solution. The purpose of this paper is to implement and test the functionality of Authoritative DNS over Serverless Computing model together with an external database. We perform a series of simulations and investigate how particular parameters impact the performance. On this basis we draw some conclusions. At the end, we provide cost analysis as well as compare *Serverless* to *Serverful* Computing models.

The rest of this paper is structured as follows. Section 2 explains the Serverless Computing and FaaS model in depth. Section 3 describes the used technologies and the implemented solution. Section 4 presents conducted simulations with proper conclusions. Section 5 briefly concludes the paper.

2 Serverless Mobile Computing

As stated in the introduction, the DNS functionality was implemented with the use of serverless computing, more specifically using the Function as a Service (FaaS) model. In this section, we would like to explore this topic in more detail and explain what it actually entails.

Generally, the cloud computing can be categorized into a couple of different models such as Software as a Service (SaaS), Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) [9]. The main difference between them is the control over underlying infrastructure given to the customer, as opposed to pre-configured resources and tools offered by the cloud provider. In order to have a better understanding where the FaaS model is placed, the brief comparison of others is provided:

- In the *Software as a Service (SaaS)* model, the cloud provider manages and maintains the entire infrastructure with the application running on it, which is accessible through the internet.
- In the *Platform as a Service (PaaS)* model, the cloud provider provides a so-called platform, where customers can deploy, develop, manage and maintain their applications.
- In contrast to previous models, the *Infrastructure as a Service (IaaS)* provides the greatest control over cloud-hosted computing resources such as virtual servers, storage and networking.

Figure 1 presents a comparison of the above-mentioned models as well as a traditional one with specified responsibilities.

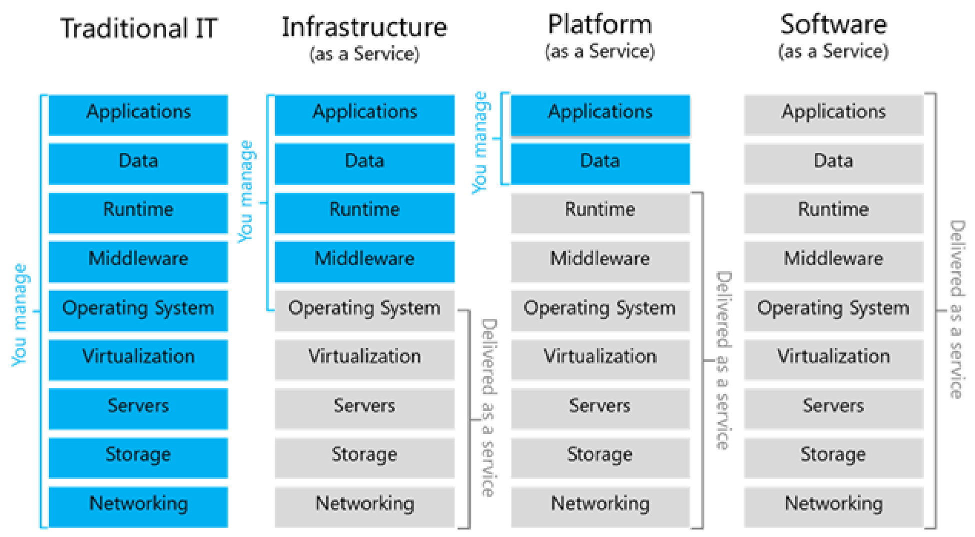


Figure 1: The comparison between different cloud computing models and the traditional one [10]

The Serverless computing can be considered as a yet another cloud computing model. According to the Figure 1, in this model the customer manages only the *Applications (Functions)* layer. In this paper, we mainly focus on the Function as a Service (FaaS), which is often mistakenly referred to the Serverless computing as a whole, however it is just a specific implementation of it [11]. Serverless computing itself is a much broader term than only FaaS. It mainly refers to the situation where a cloud provider is responsible for managing all of the underlying infrastructure and its automatic scaling based on the current demand. This means that the customers do not have to waste their time to configure, manage and maintain resources, hence they are able to focus more on building the actual applications (products).

On the other hand, FaaS refers rather to a method of building applications as functions, which are triggered by some kind of events, such as API calls, database events or scheduled actions. Instead of deploying the entire application, the developers can partition the code into multiple stateless functions, where each one is responsible for executing a specific task. As most technologies, this solution also introduces some trade-offs. If implemented knowingly, this model can offer a lot of benefits, the main of them are:

- Reduced costs. The Function as a Service (FaaS) utilizes a so-called pay-as-you-go model, where the customers pay only for the resources they actually utilize on the granularity of the time needed to execute certain functions. It means that, only the operational time is what the users are charged for as opposed to the model, where resources are allocated in advance and have to be paid for not matter if are being utilized.
- Automatic resource scaling. When the demand increases/decreases, the architecture is automatically scaled up or down horizontally. This means that the new instances of the function are created when needed by the vendor. Usually, there is no limit of maximum number of instances (the higher number the more has to be paid). Such an approach allows to avoid creating emergency plans, which are only useful in times of increased traffic, as well as reduces operational costs.
- Request isolation. It means that every incoming request or function invocation is executed in a separate, isolated environment. It is related to above-mentioned automatic scaling due to the fact that per request there is a new instance of the function provisioned. This process increases security and reduces potentials conflicts.

This solution also introduces some specific characteristics, which can be considered as limitations. In order to fully understand this model of cloud computing, the drawbacks are also thoroughly presented and discussed. The main disadvantages are as follows:

- Function cold start. Essentially, it means that newly created instances of functions introduce a delay towards warm ones. The process of creating a new function consists of two main steps, which are code downloading and starting the new environment. It can be affected by a number of factors, such as what size is the function itself, what technology it is written in or how fast the cloud provider is able to provision needed resources. As a result, the first function invocation can take a little longer, therefore increasing the overall latency.
- Separate data storage. By the definition, serverless functions are supposed to be stateless. Sometimes, developers have to store some needed information to properly run their application. In such a scenario, they are compelled to use an external database (usually also hosted in the cloud), which can introduce some of its limitations and restrictions, such as storage capacity, latency or privacy and security issues.
- Limited control and vendor lock-in. In the serverless concept, the cloud provider manages and maintains the entire underlying infrastructure. In some scenarios, it can be considered as a drawback for the customers, who lack of a control over it and have to fully rely on the vendor. What is more, there is a risk of potential so-called vendor lock-in, which refers to the situation, where the customer is heavily dependent on the specific provider. It can result in harder to achieve and costly migration to the environment of a different provider.

Another important topic to mention is the high availability [12] of the cloud solutions (about 99.95%). Cloud providers invest heavily in highly distributed and redundant infrastructure, as well as develop automatic fault tolerance systems by utilizing various methods of monitoring and recovery mechanisms. The resources are widespread across many data centers in different locations, which allows to efficiently reroute traffic to a healthy instance of the function or just to the new one in case of any problems. High availability can also be achieved in the traditional computing, however it might need a more proactive approach (sometimes even manual intervention) and in case of provisioning redundancy in such a scenario, the unused hardware might be underutilized in the event of low demand.

In the figure 2 an exemplary FaaS model architecture is presented with the use of AWS (Amazon Web Services) as the cloud provider. As can be seen, it consists of a set of AWS Lambda functions, which each one of them is able to scale up or down depending on the current workload. Worth noticing is the external database being utilized, which is needed due to the fact that functions are typically stateless by design. When it comes to the communication with such a application, the client sends its requests to the so-called API Gateway. This external service is responsible for, among others, proper routing of the incoming traffic to appropriate resources, load balancing as well as providing authentication and authorization.

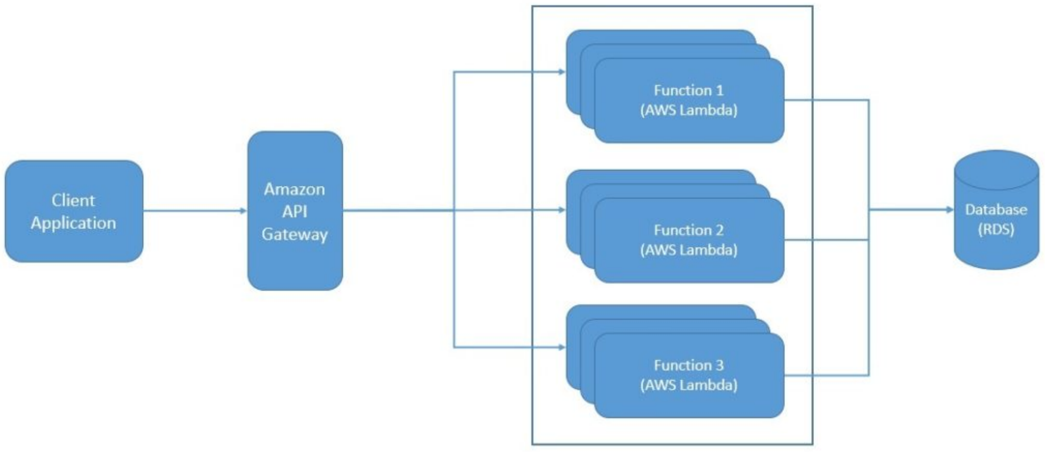


Figure 2: An example of the FaaS model architecture with the use of the AWS as the cloud provider [13]

Considering all of the presented advantages and disadvantages, the question is what would be their impact on the performance and operation on our DNS solution implemented as a function. In this paper, we performed many simulations in order to determine if such an approach is worth considering. We measured such parameters as latency, memory and cpu usage, errors, responsiveness as well as an overall cost of this solution. Firstly, we examine how the function scaling performs under an increasing number of requests per second (Subsection 4.1). Then, we closely analyze the latency under the occurrence of cold starts (Subsection 4.2). We also measure the users influence, firstly by increasing the incoming packet load from them, then we simulate their concurrent utilization of resources (Subsection 4.3 and 4.4 respectively). Furthermore, we look into interference effect on performance across specific periods of time (Subsection 4.5) and CPU Utilization (Subsection 4.6). Due to the fact, that we utilize MongoDB as our external database, we also check how its specific characteristics, such as the number of records and utilized connections, affect our system (Subsection 4.7 and 4.8 respectively). At the end, we compare the serverless computing model to the traditional one (Subsection 4.9) and perform the detailed cost analysis (Subsection 4.10).

3 Tools and Implementation

In this section of the paper, we would like to focus on describing used technologies and the implementation of the proposed solution. Firstly, we will discuss why we decided to choose such tools and provide brief analysis of them. Then, the implemented method will be presented with its key characteristics specified.

3.1 Used technologies

In order to implement the proposed solution we had to consider various cloud providers and evaluate their offerings. As our FaaS environment providers, we decided to choose Amazon (AWS Lambda) and Google (Google Cloud Functions). It can certainly be said that they are the most mature cloud providers, who offer a lot of services such as computing, data storage, networking and many more. Both providers are the leaders in terms of the market dominance, which makes them even more desirable to examine in the context of our scenario. Due to their high market share, they provide high data center coverage, which enabled us to pick the closest options. Surprisingly, in the case of Amazon we had to choose Frankfurt, as it does not provide any locations in Poland, while Google offers to pick the Warsaw. When it comes to the possible parameters to configure, both FaaS platforms are more or less the same and allow to manipulate such variables as the allocated memory to the function, previously mentioned region or the number of instances (to configure more sophisticated parameters, we usually have to additionally pay). As for the programming language needed to implement our function, we decided to choose Go or also called Golang [14] (version 1.19) as it is a fairly new language, which was specifically designed to build cloud-native solutions. It provides many benefits, especially regarding the ability to efficiently and quickly handle high traffic and process large amount of data. In order to conduct simulations and measure performance, we have chosen to use an open-source Apache JMeter tool [15] as our load generator. It enabled us to create various testing scenarios simulating realistic user behavior. It does so by supporting many activities, protocols (such as HTTP, HTTPS, FTP and more) as well as is able to generate traffic concurrently. To store our DNS records, we utilized external MongoDB database. Due to the fact that offered serverless MongoDB platform does not provide a free trial [16], we used the alternative cloud cluster.

3.2 Implementation

In this subsection, we would like to provide a comprehensive overview of the proposed solution. As stated in the introduction, we developed the Authoritative Domain Name System (DNS) as a serverless function, which serves definite information about an IP address of the requested domain in its zone. Considering other types of DNS, such as DNS resolvers or Root/Top Level Domain servers, we concluded that they would be either very expensive to develop or just inefficient. In order to implement our FaaS DNS, we based on DNS-over-HTTPS (DoH), its specification is defined in RFC8484 [17] titled *DNS Queries over HTTPS (DoH)*. It refers to the method of utilizing HTTPS protocol in transportation of DNS traffic, therefore imposing the unified security and encryption mechanisms commonly used in current web technologies. The RFC8484 document defines also how the Uniform Resource Identifier (URI) should be structured and how the DNS Message Format should be encapsulated in requests and responses of HTTPS protocol. As for HTTP methods, only GET and POST are possible. The example of such a GET request in our scenario is as follows:

GET `https://dnsServer.com?domain=es2023.pl`

As can be seen, the domain name to be resolved is defined as a query parameter. According to the RFC8484 these parameters should be encoded with a Base64URL, however in this research we sent it in plain text in order to simplify tests. Worth mentioning is the fact that, Cloudflare DoH also accepts requests with query parameters in plain text. The implementation of Authoritative Domain Name System imposes the necessity of using external database to store DNS records. Important to note is that the database was populated with the mock data in accordance to the particular scenario currently being tested. As mentioned before, mainly due to free trial, we utilized a shared version of MongoDB also implemented in its own cloud environment. It does not provide many sophisticated features, however still allows to perform some experiments. In this version, MongoDB cluster can handle only 500 connections, this limits possible number of function instances as every new one needs its own connection (the most expensive database configuration is able to handle around 120k, which means the maximum number of instances is also around this number). Due to this factor, our simulations regarding the database were performed under the 500 connections threshold. When it comes to conducting experiments in this research, we utilized Apache JMeter in order to test the performance of the proposed solution. This tool itself provides various parameters to configure, such as number of threads (users), ramp-up period in seconds (time from the start with 0 virtual users to reaching their maximum number), loop count and obviously API endpoint to be hit. According to the currently examined scenario, we created a suited test plan by properly adjusting those parameters in order to achieve the intended goal. Apache JMeter also enables to define desired sampler, which determines the type of requests we want to send, in our case we only used a HTTP sampler as it required by the proposed solution (DoH).

Figure 3 presents the architecture of the proposed solution. In a typical scenario, the load is generated by Apache JMeter in the form of HTTP requests with certain, previously defined parameters and characteristics (e.g. number of different users hitting the endpoint). Such traffic goes to the API Gateway of the cloud provider (in our case it is either AWS or GCP), where it is distributed between provisioned resources. It means that per each request there is a new *go-dns* function initiated, which serves the incoming traffic (here the automatic scaling of resources take place). The *go-dns* function itself is a piece of code, which implements our Authoritative Domain Name System (DNS) solution. For every such a function the external MongoDB database connection is utilized, which is demonstrated in the figure.

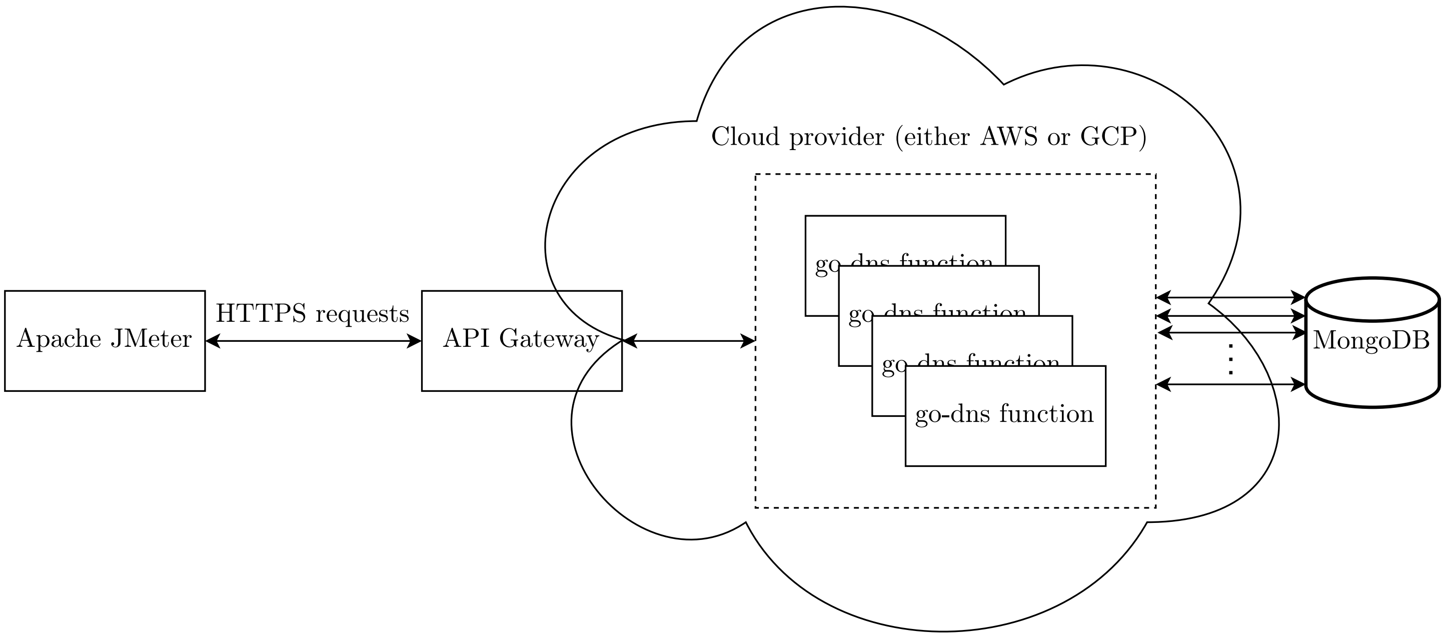


Figure 3: The architecture of the proposed solution including utilized testing tool

Such an architecture was prepared and deployed for each of the selected cloud provider and then thoroughly tested and compared to each other. The performed simulations are presented and concluded in Section 4. What is more we provide extensive cost analysis of such implemented architectures.

4 Simulations

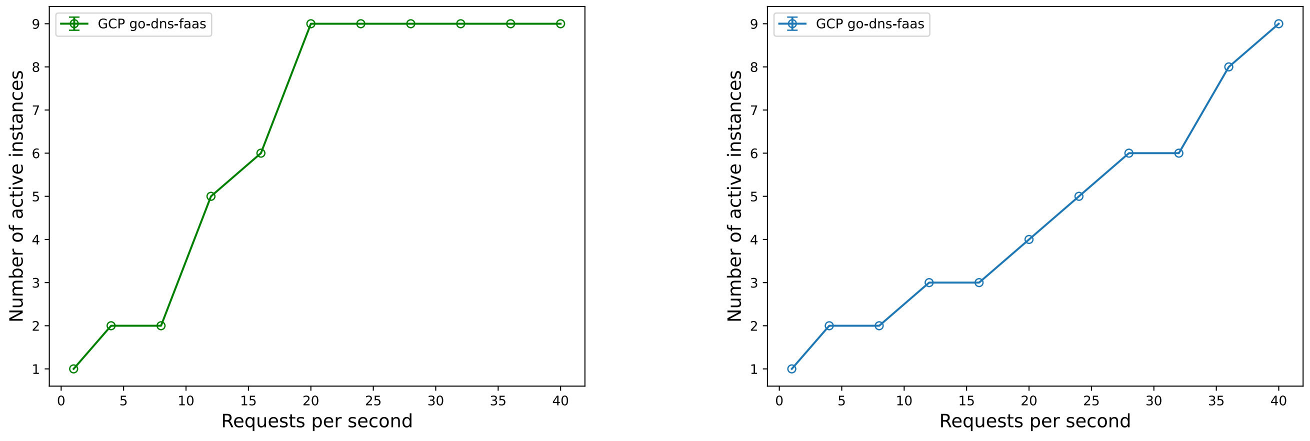
4.1 Function Scaling

Scaling serverless applications relies on constant adjustment of number of cloud functions that are running on the serverless platform. Instances of functions are added and removed dynamically as the demand for their usage grows and drops. In this simulation, we consider the number of instances in relation to the traffic volume controlled by the traffic generation rate.

Table 4: Parameters for function scaling simulation

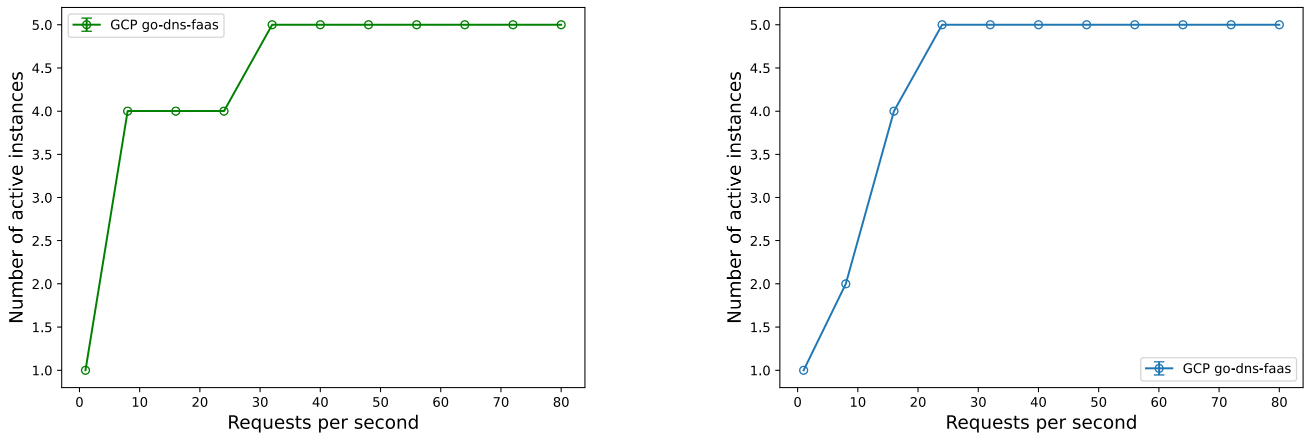
Scenario	Region name	Allocated memory	Concurrency	Max number of instances	Generation time
1	europe-central2	128 MB	1	100	5 min
2	us-central1	4 GB	1	10	5 min

Scenario description: In scenario no. 1, we consider a *go-dns* function deployed to the Google Cloud, specifically to the europe-central2 region configured with the possibility to handle one request, scaled up to 100 active instances. In the second case, a *go-dns* function is again deployed to the Google Cloud but to the us-central1 region, max number of instances is limited to 10. Comparison is made between two Google Cloud platform types of deployed functions: generation 1 (*gen1*) and generation 2 (*gen2*) that differ from each other with some specific technical parameters (i.e. allocated memory size).



(a) GCP Function Generation 1 (b) GCP Function Generation 2

Figure 5: Simulation results for scenario 1



(a) GCP Function Generation 1 (b) GCP Function Generation 2

Figure 6: Simulation results for scenario 2

Conclusions: With the volume growth of the generated traffic, the number of active instances also arises to maintain the capability of the DNS service to handle incoming requests. Despite the fact that the GCP function versions contain some differences between each other, these simulation shows that at the end, the same number of instances have been created, although the functions are scaled with different dynamics. Expansion or decrease of active instances can be limited by setting the *max* or *min* number of function instances. Number of instances created in the second simulation for the same load is smaller due to the function more powerful capabilities (memory, number of vCPUs) in the second configuration.

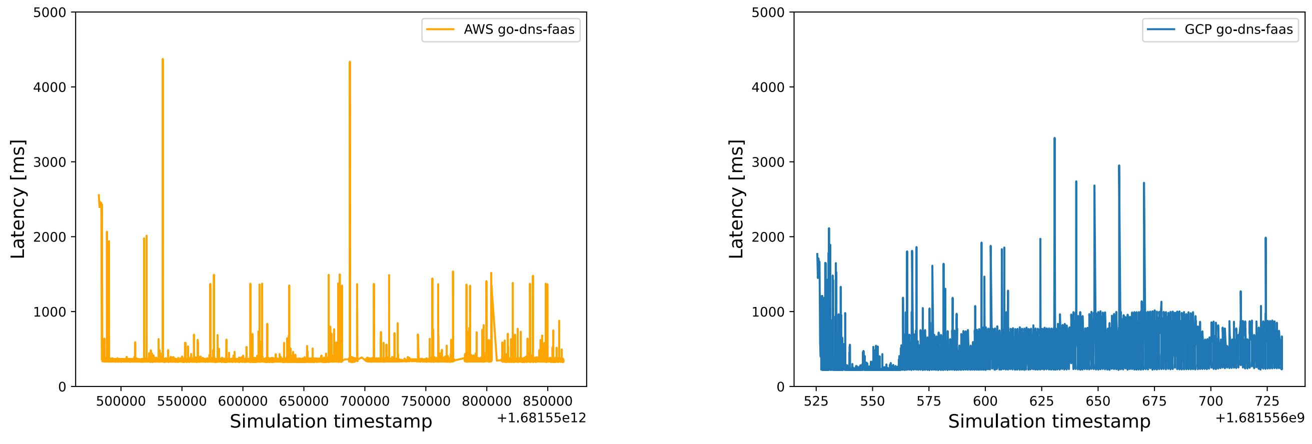
4.2 Cold Start

Results obtained from the simulations presented in subsection 4.1 allowed us to observe that the DNS serverless service scaling relies on increasing the function instances actively responding to the traffic incoming to the cloud. The vital question coming from this experiment is how the constant scaling affects the overall performance of the deployed function. In this simulation, we review the response time generated by *go-dns* with emphasis on the moments when so called *cold starts* occur.

Table 7: Parameters for cold start simulation

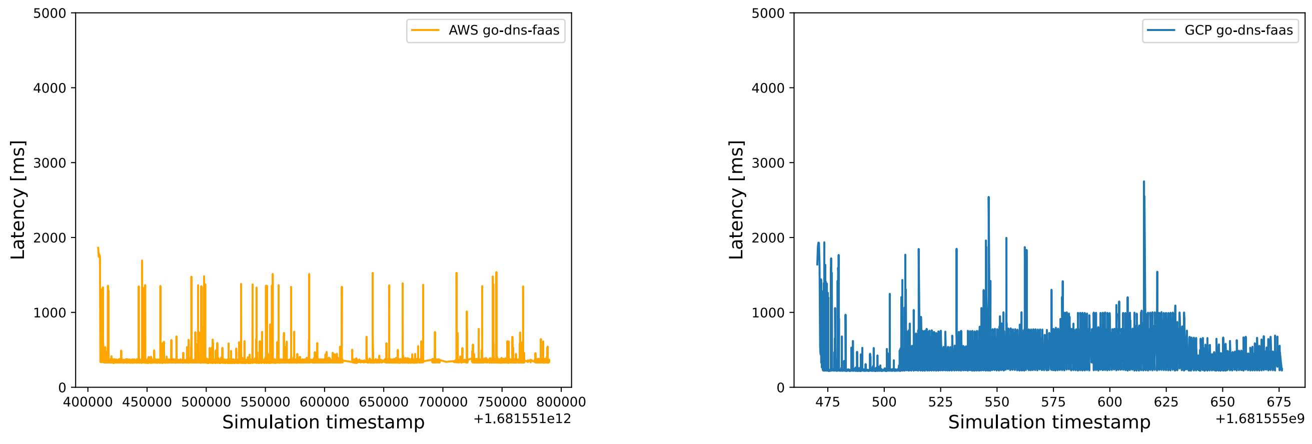
Scenario	Region name	Allocated memory	Generation time	Range number of concurrent users
1	eu-north1	128 MB	6 min	(0,100)
2	eu-north1	2 GB	6 min	(0,100)

Scenario description: In this simulation, we configured two scenarios for a load test between two functions differentiated with allocated memory size (128MB and 2GB respectively). However, this time we conduct a stress test between two serverless platforms: GCP Functions and AWS Lambda. For the possible fair assessment of these frameworks, functions were deployed to the same regions, configured with the same parameters and examined within the same time intervals.



(a) AWS Lambda go-dns (b) GCP Function go-dns

Figure 8: Simulation results for scenario 1



(a) AWS Lambda go-dns (b) GCP Function go-dns

Figure 9: Simulation results for scenario 2

Conclusions: Conducted simulations proved that the problem of cold starts indeed exists in the serverless frameworks. Furthermore, high peaks visible in the first scenario implies that cold starts last longer due to the lower number of resources assigned (proportionally to the memory allocation). Another observation is the negligibly small frequency of cold starts in AWS. If one request per 5000 comes with bigger latency, it should not be destructive to the overall performance of the DNS. Moreover, comparison between Lambda and GCP function proves that the cold start appears in both cases and stands as a challenge for both frameworks. Finally, experiments suggest that AWS Lambda is more stable when the amount of generated load increases that can be visible in the lower average latency in the whole simulation.

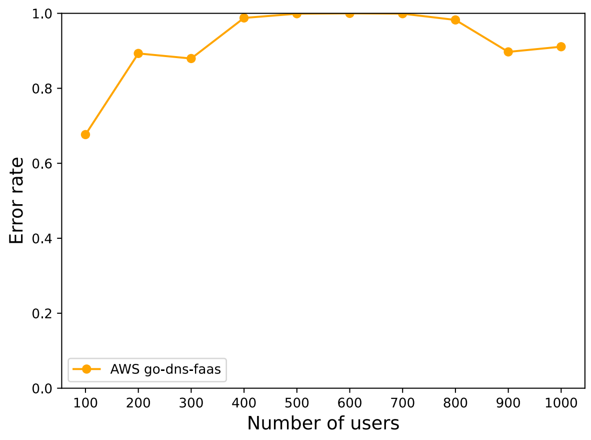
4.3 Service Availability

Despite the fact that the cold start is still a real challenge for the serverless platforms, one of the conclusion that came out from the previous experiment performed in the subsection 4.2 was that the cold start does not have to impact the DNS availability as it occurs too rarely in AWS to be an obstacle for the potential user of the serverless function. For this reason, we decided to conduct an experiment that will measure the *go-dns* availability under common factor of congestion in modern networks - large number of concurrent users trying to get use of the DNS function deployed in the cloud environment.

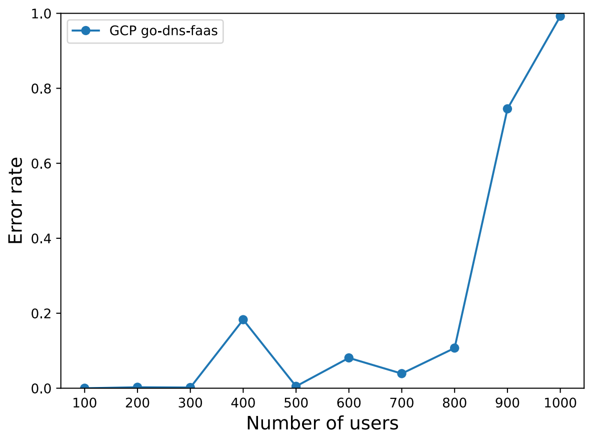
Table 10: Parameters for service availability simulation

Scenario	Region name	Allocated memory	Generation time	Max no. of instances
1	eu-north1	128 MB	2 min	AWS (10), GCP (100)
2	eu-north1	2 GB	2 min	AWS (10), GCP (100)

Scenario description: In this simulation, we perform a service availability analysis to test the *go-dns* function availability under increasing packet load from the customer side. We are aware of the fact that some of these errors might not be a defect of the cloud framework or the DNS function itself, but the application installed on the host side that we are using for the traffic generation. However, we believe that testing these frameworks in similar conditions, and with the same application, bring reliable simulation comparison and conclusions about the researched topic. In both scenarios, we consider the number of threads (users) in comparison to the error rate. Scenarios are distinguished with the allocation memory size.

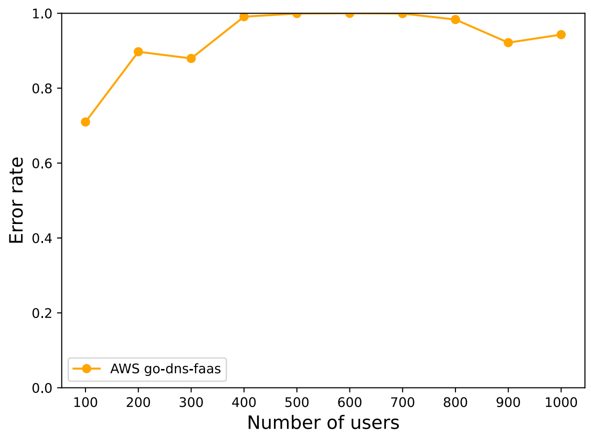


(a) AWS Lambda go-dns

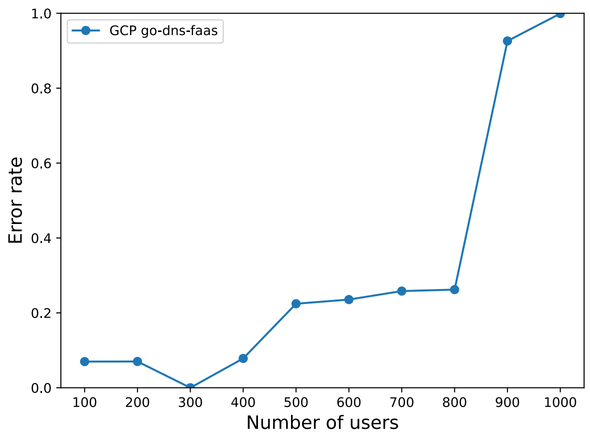


(b) GCP Function go-dns

Figure 11: Simulation results for scenario 1



(a) AWS Lambda go-dns



(b) GCP Function go-dns

Figure 12: Simulation results for scenario 2

Conclusions: Conducted experiments show the thresholds above which both frameworks become unavailable for selected parameters (AWS - 400 users for max 10 instances, GCP - 1000 users for max 100 instances). However, in this simulation we didn't catch any significant difference between scenarios. This suggests a strong undeterministic behavior of the cloud frameworks and the need of examinating their performance in the long term horizont that will be covered in the simulation presented in subsection 4.5. Another fact is the need of proper configuration of max number of instances in serverless.

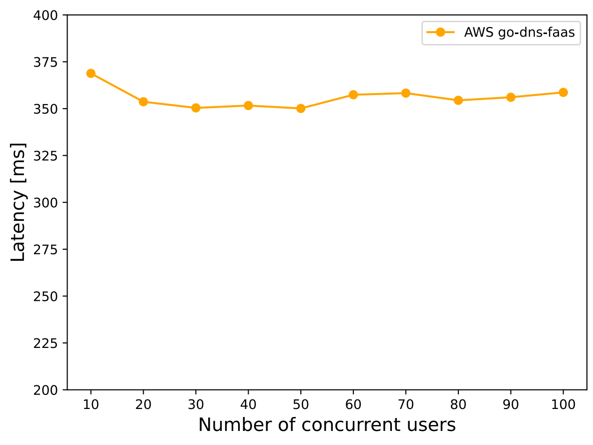
4.4 Latency With User Concurrency

Experiments related with cold start analysis in subsection 4.2 proved that the cold start is still problematic in some cases for cloud serverless framework providers and stands for a challenge especially at the beginning of the simulation while in the further processing behaves differently depending on the function scaling which was observed in the simulation of active instances in subsection 4.1. In this simulation, we measure the relationship between average latency of the requests and the number of active users to check how serverless frameworks are acting under different load generated by the users towards the DNS, and inspect the role of the cold start that should be visible especially at the beginning of the diagrams.

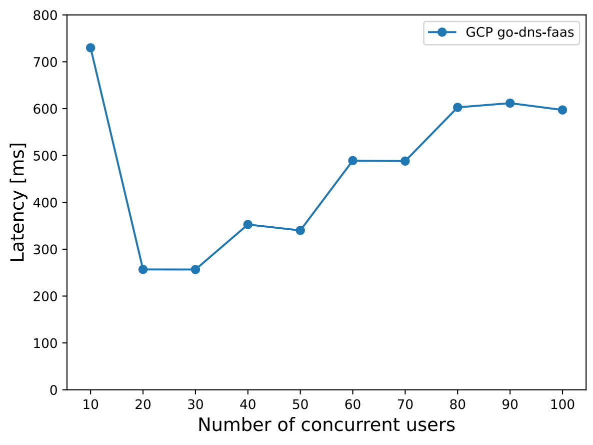
Table 13: Parameters for latency with concurrency simulation

Scenario	Region name	Allocated memory	Generation time	Range number of concurrent users
1	eu-north1	128 MB	6 min	(0,100)
2	eu-north1	2 GB	6 min	(0,100)

Scenario description: Scenarios are configured in full accordance with the parameters from subsection 4.2 and are focused on latency assessment under the increasing number of concurrent users represented by the active system threads that are trying to send the DNS request to the function deployed in AWS Lambda or GCP Function cloud frameworks.

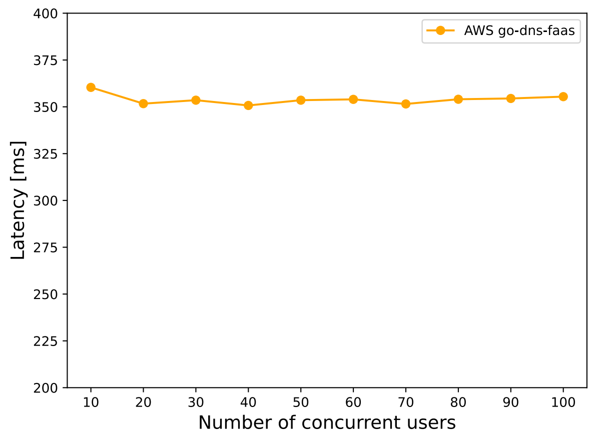


(a) AWS Lambda go-dns

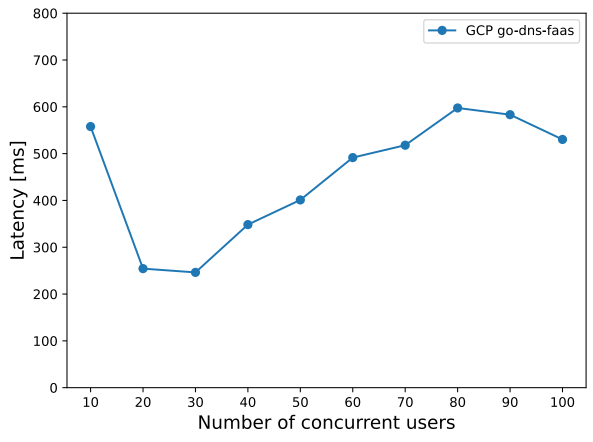


(b) GCP Function go-dns

Figure 14: Simulation results for scenario 1



(a) AWS Lambda go-dns



(b) GCP Function go-dns

Figure 15: Simulation results for scenario 2

Conclusions: All the conducted experiments showed the importance of considering cold start while using serverless platforms. Nevertheless, its role is different in both cloud solutions presented in this work. In AWS we observe a small cold start peak at the beginning of each simulation (especially noticeable in scenario 1) and increasing number of users is not too visible as the curve remains constant after the first cold start is done. On the other hand, cold start in GCP function is much more noticeable in both scenarios and the impact of increasing number of users is also more visible because of the curve that is gradually increasing along with the number of concurrent users that are trying to get access to DNS service.

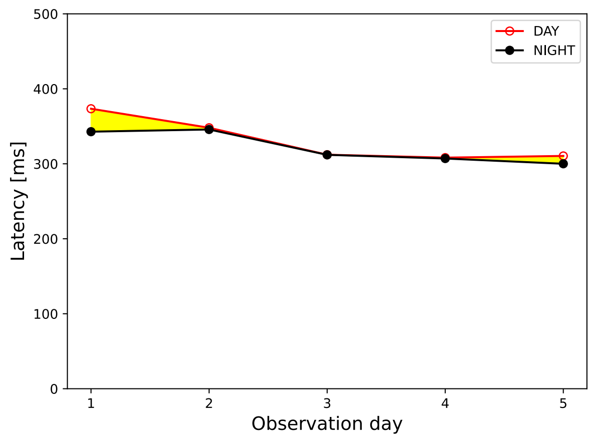
4.5 Interference Effect on Performance

Simulations that were performed so far, delivered variety of consequences or problems that are inevitable while using serverless platforms. However, none of them considered the overall strategy of cloud platforms on how serverless functions are executed. Containers that are in charge of performing the computation related with DNS are isolated from individual users. However, different VMs belonging to users that exist in the same region might lead to some interference results. Another factor causing anomalies or fluctuations are the scale updates or system outages. Therefore, in this series of simulations, we focus on the cloud systems performance in a broader time horizon to detect some interference patterns.

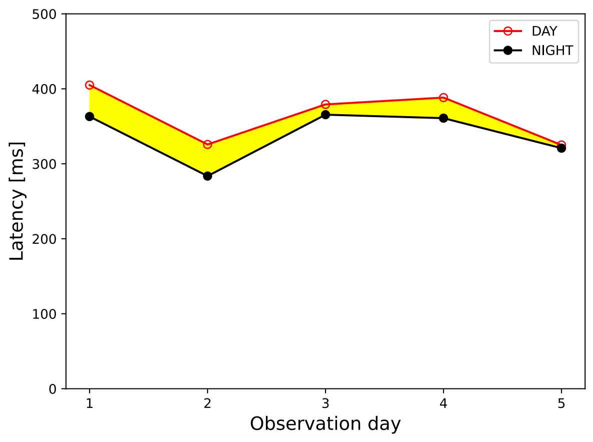
Table 16: Parameters for interference effect simulation

Scenario	Region name	Allocated memory	Generation time	User range	Day time	Night time
1	eu-north1	128 MB	8 min	(0,80)	11:00 - 17:00	22:00 - 01:00
2	eu-north1	2 GB	8 min	(0,80)	11:00 - 17:00	22:00 - 01:00

Scenario description: Both scenarios were configured for testing under increasing traffic load, arising from 0 to 80 threads (users) in 8 minutes. Application was deployed to the AWS and GCP clouds and is differentiated with the allocated memory size (128 MB or 2 GB). Measurements have been done during different day and night time as stated in Table 16.

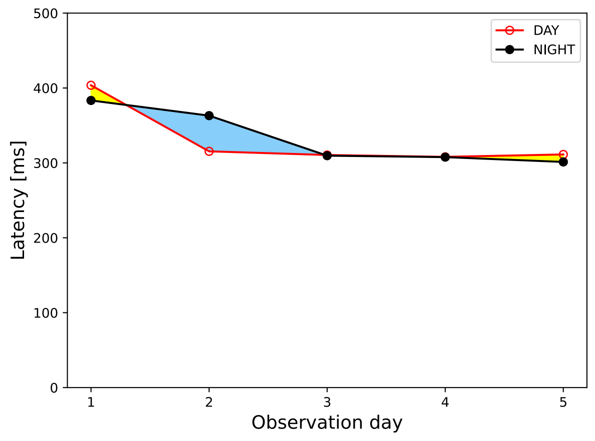


(a) AWS Lambda go-dns

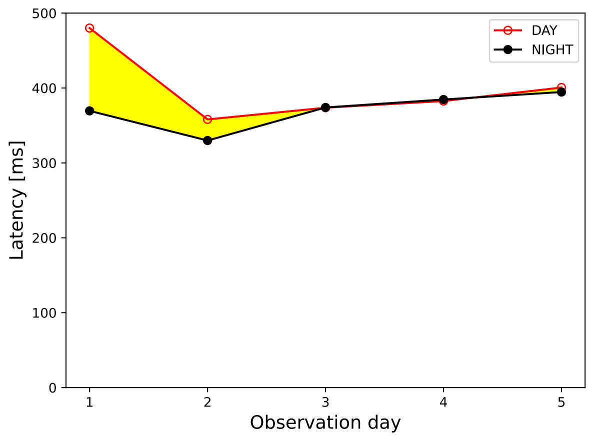


(b) GCP Function go-dns

Figure 17: Simulation results for scenario 1



(a) AWS Lambda go-dns



(b) GCP Function go-dns

Figure 18: Simulation results for scenario 2

Conclusions: As we can see on all diagrams, latency during the night is generally below or equal to the day level. It may suggest that the lower network traffic intensity during the night and larger in the day impacts the overall performance of the cloud framework. This information is crucial as serverless functions are billed per 1ms of execution time, and it will affect the final price for hosting the function. Another thing to consider is that some critical application can be sensitive to such latency fluctuations, and this kind of performance might be unacceptable for their usage via serverless platforms.

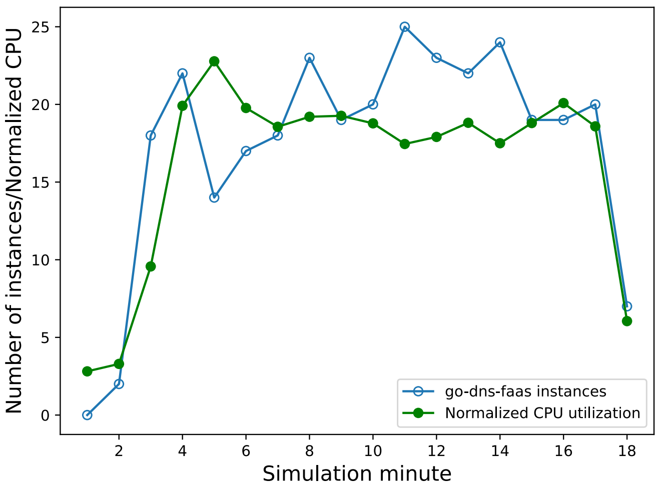
4.6 CPU Utilization

Although in serverless computing code developer should not care about how the provider is dealing with the growing number of existing containers, virtual machines creation and all the hardware issues that are behind the cloud infrastructure, CPU utilization rate is very popular metric that is used to evaluate serverless platforms performance as it can be the information for cloud provider to increase the number of active functions in case of the growth of the network traffic. In this simulation, we are trying to assess how the CPU utilization rate is changing and correlate it with number of instances creation.

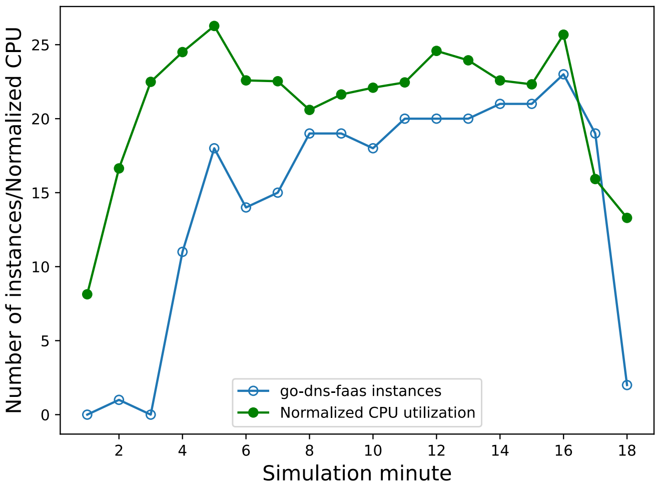
Table 19: Parameters for CPU utilization simulation

Scenario	Region name	Allocated memory	Generation time	User number range
1	eu-north1	128 MB	15 min	(0,150)
2	eu-north1	2 GB	15 min	(0,150)

Scenario description: Exactly as in the previous simulations, both of the scenarios were configured in the same manner but with the different size of allocated memory for the single function (128 MB in the first scenario, 2 GB in the second scenario) to evaluate the impact of the amount of hardware resources assigned to the *go-dns* in both cases. Experiments have been set up for GCP Functions deployment to consider its function scaling politics. As an output, normalized CPU utilization with amount of active go-dns instances are produced to compare its correlation.



(a) Scenario 1



(b) Scenario 2

Figure 20: Simulation results for CPU utilization simulation

Conclusions: In this simulation, we observed a strong correlation between the CPU utilization and the number of active instances in GCP Functions framework. Moreover, we had to normalize the CPU metric by multiplying it 1000 times in the second configuration when 100 times in first configuration which proves assigning larger amount of resources in second scenario, so these CPU resources are used less intensively comparing to the first configuration. Initially, we planned to do the same test for AWS Lambda, but the possibility to track the number of functions in time domain in detail is more limited.

4.7 Database Analysis

Another important factor which is able to impact serverless DNS performance is a number of records stored in an external database. In this simulation, we want to present differences in average latency times for various number of stored records.

Table 21: Parameters for database simulation

Scenario	Region name	Allocated memory	Targeted load	Duration	Number of records
1	eu-central1	256 MB	100 RPS	3 min	10^3
2	eu-central1	256 MB	100 RPS	3 min	10^4
3	eu-central1	256 MB	100 RPS	3 min	3×10^5

Scenario description: The main purpose of the whole simulation was to check if our serverless DNS deployments (with 100 as a limited number of instances) would be able to handle targeted load in the unit of the requests per second (RPS), regardless of the number of stored records. Table 21 represents configurations of the scenarios. Each simulation was independently performed for AWS Lambda and GCP Functions. To gather credible results, simulations were repeated three times and the average of those runs was calculated.

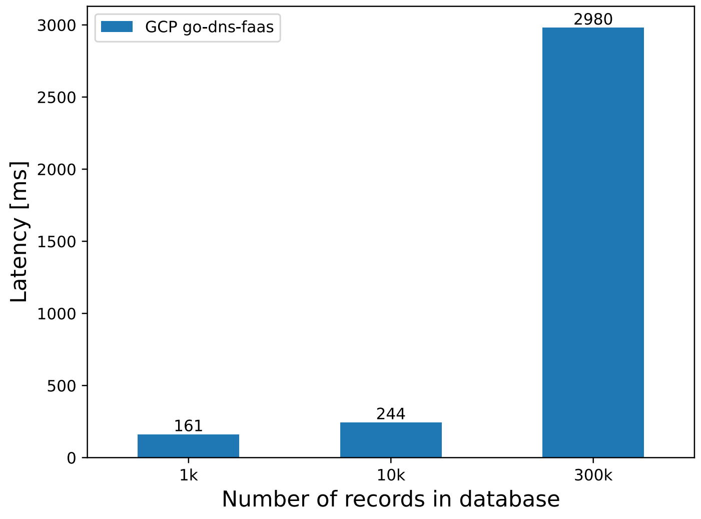


Figure 22: GCP Function go-dns

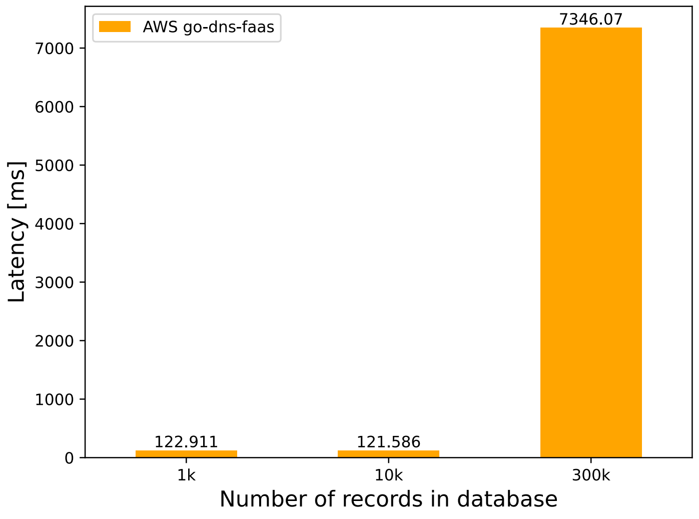


Figure 23: AWS Lambda go-dns

Conclusions: Figure 22 represents results for GCP Functions and figure 23 for AWS Lambda solution. As we can see, response latency is directly connected with the number of stored records. In the case of thousand and ten thousand entries, both deployments were able to handle offered load, with a little bit better performance of AWS Lambda solution. However, for 300 000 records we have drastically exceeded the capacity of proposed configuration for given number of stored records, which results with huge average latency spike for both deployments. Interestingly, the tendency of lower latency for AWS Lambda solution has not been preserved and GCP Functions deployment achieved almost 2.5 better performance. According to the gathered results, potential owners of serverless DNS should not select deployment configuration based only on average load, but also take estimated maximal records number under consideration.

4.8 External Connections Usage

Despite the fact that cloud providers share a lot of metrics to keep monitor running applications, there are also some other external metrics for serverless application which user should consider before running it. For example, in our work, MongoDB cluster was used as an external service for the *go-dns* function. However, as the limitation for the free usage is 500 connections at most, it poses a question on how the number of registered connections changes during the work of serverless application and whether memory of the allocated function is impacting the final price for MongoDB database cluster.

Table 24: Parameters for External Connections Usage simulation

Scenario	Region name	Allocated memory	Generation time	User number range
1	eu-north1	128 MB	10 min	(0,200)
2	eu-north1	16 GB	10 min	(0,200)

Scenario description: This simulation was configured with increasing traffic load from 0 to 200 threads (users) with generation time set to 10 minutes. Similarly to previous tests, deployed functions have different size of allocated memory (128 MB in scenario 1 and 16 GB for scenario 2) and constant, maximal number of function instances (100). Tests have been conducted only for GCP Functions. The metric which is describing number of active MongoDB connections was taken directly from MongoDB Atlas (web based application for managing MongoDB clusters).

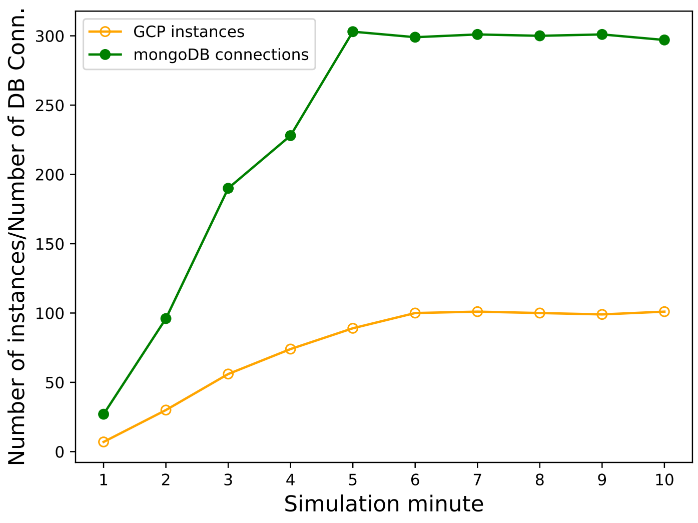


Figure 25: Scenario 1

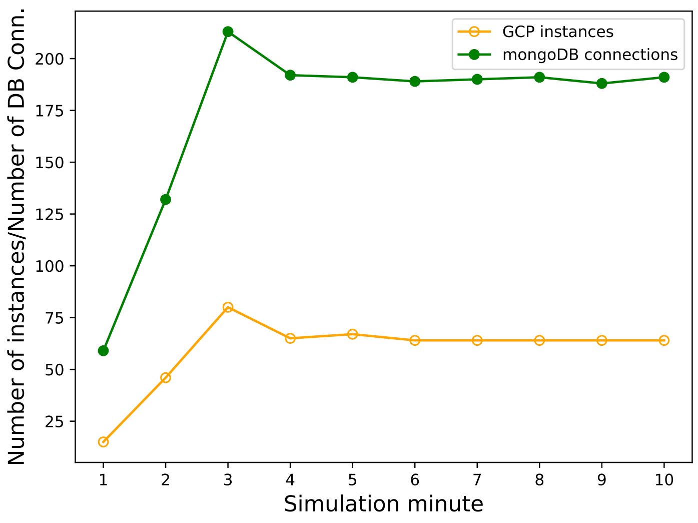


Figure 26: Scenario 2

Conclusions: Conducted experiments show that the number of created instances impacts a number of external MongoDB database connections and need to be considered when these connections are billed independently of the cloud provider. In the second scenario, we observed a lower number of created connections in comparison to the first scenario because of lower number of created *go-dns* instances as more powerful resources have been assigned to each of them. There is still one question to answer about this scenario. If we look closely at figures 25 and 26 we will see that the number of active MongoDB connections is almost three times higher than the number of active GCP function instances. We have investigated this problem closely, and we have encountered an interesting thread on the official MongoDB forum [18], which is also describing the problem with additional connections. According to one of the thread responses, during connecting to MongoDB cluster, connection is active to each collection (structure similar to table in relational databases). In the case of our simple MongoDB cluster, we had three collections, one of which was storing DNS records and the rest were created automatically and filled with sample data during cluster creation. Due to that factor, potential owners of faas-dns should reduce the number of collections to minimize costs of external record storage.

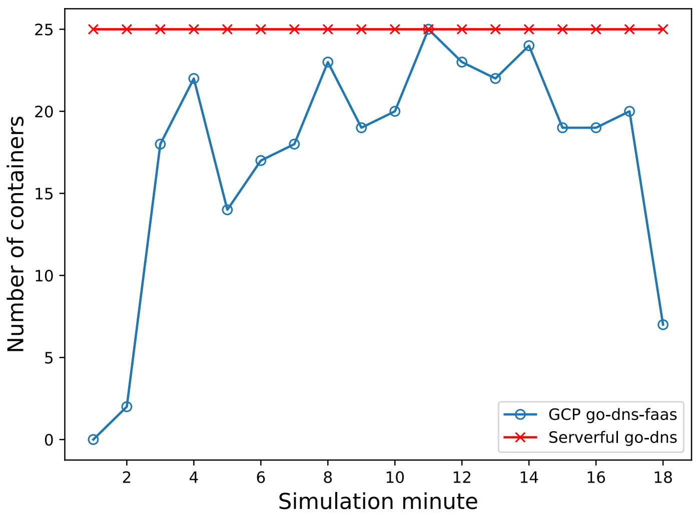
4.9 Serverless vs Traditional Computing

Last but not the least, simulation will be about very simple and practical comparison between Serverless Computing and Serverful Computing. As we know, in the serverless computing resources are assigned dynamically to the user needs and the volume of its network traffic that is coming to the function deployed in the cloud framework. On the other hand, serverful idea acquires constant number of active containers and hardware resources assigned to the user. For this reason, results from the CPU simulation 4.6 were reused for a brief comparison between amount of containers in serverless and serverful computing.

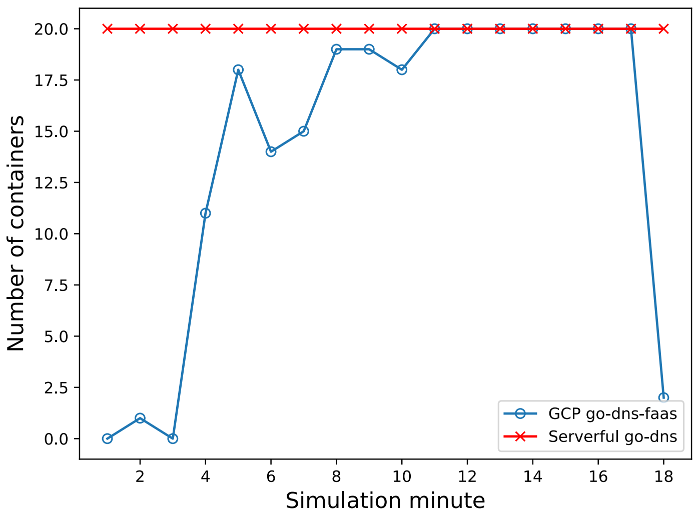
Table 27: Parameters for Serverless vs Traditional computing simulation

Scenario	Region name	Allocated memory	Generation time	User number range
1	eu-north1	128 MB	15 min	(0,150)
2	eu-north1	2 GB	15 min	(0,150)

Scenario description: Presented scenarios acquired deploying *go-dns* function to the GCP Functions framework with a different memory allocation size (128 MB in Scenario no.1 and 2 GB in Scenario no.2). Time was configured to 15 min and the network traffic was generated increasingly from 0 to 150 threads (users) to observe changes in number of containers.



(a) Scenario 1



(b) Scenario 2

Figure 28: Simulation results for Serverless vs Traditional Computing

Conclusions: Both diagrams present an GCP advantage over serverful solution. The benefit is visible as the difference between serverful red line and the serverless blue line, so the benefits will be particularly visible in case of spiky workload. Additionally, the red line represents a perfectly allocated number of resources for serverful solution that is difficult to predict before traffic generation. However, we must admit that the serverful solution has also its advantages as it does not have to deal with the serverless challenges (i.e. cold start, additional overhead, security issues, vendor lock-in or cost uncertainty).

4.10 Cost Analysis

Probably the biggest advantage of serverless DNS is pay-as-you-go architecture, in which environment providers are billing clients for used resources only. In this section, we have compared and estimate costs of serverless DNS for AWS Lambda and Google Cloud Platform Functions. Estimations of monthly billings were done with the usage of official *billing calculators* provided by AWS [19] and Google [20]. Moreover, costs of maintaining external MongoDB record storage were also included.

First of all, few assumptions were introduced:

- average number of requests per seconds equals 1000 (2678400000 monthly)
- total number of records equals around 10 000 entries
- no additional features like: concurrency inside single function, minimal number of active instances
- as an environment location, the closest datacenter to Cracow was selected

In the case of the GCP, clients are billed for the amount of data transferred out of the function instance each time it is invoked (responses). According to our simple simulations, average response size was around 350 bytes. On the other hand, AWS requires a separate service called *API Gateway* to trigger function via HTTP - such service is billed per request number (2678400000 in our case). Further costs related to function triggering are called *API costs*.

To show the impact of better function configuration, cost analysis was prepared for two different configurations with 512 MB and 2 GB of the assigned memory. The last factor, which is necessary to estimate costs, is average function execution time. To gather that information, another simulation was performed. The metric of average execution time was taken directly from the cloud providers. The results are visible in the Table 29.

Table 29: Average execution time for given configuration

Provider	Memory	Generation time	RPS	Avg. execution time	Location
AWS Lambda	512 MB	3 min	50	36 ms	eu-central1 (Frankfurt)
GCP Functions	512 MB	3 min	50	46 ms	eu-central2 (Warsaw)
AWS Lambda	2 GB	3 min	50	22 ms	eu-central1 (Frankfurt)
GCP Functions	2 GB	3 min	50	28.5 ms	eu-central2 (Warsaw)

Finally, we are assuming that in the worst scenario, each request will be handled by a single instance and according to the conclusions from 4.8, our database will contain only a single collection which stores DNS records. According to this factor, MongoDB cluster which will be able to handle 1000+ simultaneous connections is required – M10 MongoDB Cluster Tier with monthly pricing around 53$.

Table 30 represents results of the cost comparison for proposed configurations. First of all, monthly costs of invocations number are much lower in the case of AWS, while billings of RAM usage are almost four times lower for GCP functions. On the other hand, in the Google serverless platform, clients are also paying for the CPU usage. The biggest difference is for the cost of the HTTP API maintenance, where costs of GCP are 28 times lower. Costs of the outbound data in the GCP equals 0.12$ [21] per 1 GB, while AWS API Gateway costs 1.20$ (per 1 million requests) for first 300 million requests and 1.08$ (per 1 million requests) for the next ones (Frankfurt as a location) [22].

Table 30: Monthly cost comparison

Provider	Invocations cost	RAM cost	CPU cost	API costs	Database cost	Monthly cost
AWS (512 MB)	535.48$	796.85$	X	2928.67$	53$	4314$
GCP (512 MB)	1070.56$	214.21$	1377.11$	106.68$	53$	2821.56$
AWS (2 GB)	535.48$	1957.50$	X	2928.67$	53$	5474.65$
GCP (2 GB)	1070.56$	532.94$	2989.51$	106.68$	53$	4752.69 $

Table 31 shows cost of virtual private servers, which are an alternative to cloud computing environments. Testing on this hardware hasn't been performed, however, based on the data offered by the service providers, we can assume that both servers will be able to handle this traffic. OVH Cloud has more beneficial offer for requirements of tested configuration. Basing on Cinebench score, VPS from OVH Cloud offers equal or better performance comparing to AWS 2GB and GCP 2 GB config, but it's 99 percent cheaper than GCP and 129 percent cheaper than AWS. Considering the case where the DNS server is under constant high load, the costs are much higher than the standard server. However, taking into account the scalability of cloud solutions and billing per query, the DNS FaaS solution may be a better choice. Each case should be considered separately in terms of profitability, because much depends on traffic fluctuations and the average number of requests per day.

Table 31: Monthly cost comparison of VPS

Provider	vCPUs	RAM	Monthly cost
OVH Cloud [23]	256	2 TB	2.386$
Atman [24]	320	2 TB	5.846$

5 Conclusions

In conclusion, our implementation of the Authoritative DNS as the FaaS solution and the extensive testing we conducted on both AWS and GCP platforms have provided valuable insights into its performance and cost-efficiency. By leveraging the serverless architecture, we observed significant benefits in terms of scalability and resource allocation. We examined how the system dynamically allocated resources to handle increasing traffic, which showed that our Faas DNS solution effectively scaled in response to varying workloads. Another important aspect we investigated was the occurrence of cold starts, which can introduce latency and impact the overall responsiveness of serverless functions. It is important to note that properly tuning and selecting the parameters for the serverless functions, such as allocated memory or maximum number of instances, can have a positive impact on its overall performance. Furthermore, we explored what restrictions or limitations are imposed by the external database. We mainly considered two important factors: the number of records stored in the database and the number of external connections established. Both of these parameters have shown that they should not be ignored. The first proved that the higher number of records causes higher latency, while the latter presented the number of established connections, which can impact the operational expenses. Furthermore, our comparison between serverless and serverful approaches revealed the advantages of the former in terms of resource utilization, which can achieve better results during traffic spikes. The performed cost analysis presents the benefits of pay-as-you-go model and emphasize how significant is the proper optimization and workload management. Therefore, it is important to remember that the cost-effectiveness of the presented implementation of FaaS DNS can vary depending on the specific situation and requirements. While serverless model offers a lot of benefits, it is essential to assess the specific DNS traffic patterns and demand in every individual case. Such a particular scenario, where the proposed FaaS DNS is especially advantageous is when facing sudden workload spikes, fluctuations or the traffic being very unpredictable. By leveraging our solution in such moments, a higher level of elasticity and cost-efficiency may be obtained, especially comparing to the traditional approach in which redundant resources can be underutilized.

References

[1] Christian Tipantuña and Paúl Yanchapaxi. Network functions virtualization: An overview and open-source projects. In *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, pages 1–6, 2017.

[2] Alam Rahmatulloh, Fuji Nugraha, Rohmat Gunawan, and Irfan Darmawan. Event-driven architecture to improve performance and scalability in microservices-based systems. In *2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)*, pages 01–06, 2022.

[3] Dns server types. `https://www.cloudflare.com/learning/dns/dns-server-types/`.

[4] Marco Savi, Alessandro Banfi, Alessandro Tundo, and Michele Ciavotta. Serverless computing for nfv: Is it worth it? a performance comparison analysis. In *2022 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops)*, pages 680–685, 2022.

[5] Sonika Jindal and Robert Ricci. Mme-faas cloud-native control for mobile networks. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 152–157, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Aws lambda. `https://aws.amazon.com/lambda/`.

[7] Google cloud functions. `https://cloud.google.com/functions/`.

[8] Mongodb documentation. `https://www.mongodb.com/docs/`.

[9] Mohammad Ubaidullah Bokhari, Qahtan Makki Shallal, and Yahya Kord Tamandani. Cloud computing service models: A comparative study. In *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, pages 890–895, 2016.

[10] Cloud service models. `https://dachou.github.io/2018/09/28/cloud-service-models.html`.

[11] Mohit Sewak and Sachchidanand Singh. Winning in the era of serverless computing and function as a service. In *2018 3rd International Conference for Convergence in Technology (I2CT)*, pages 1–5, 2018.

[12] Norman Kong Koon Kit and Michal Aibin. Study on high availability and fault tolerance. In *2023 International Conference on Computing, Networking and Communications (ICNC)*, pages 77–82, 2023.

[13] Introduction to serverless architecture. `https://www.baeldung.com/cs/serverless-architecture`.

[14] Go documentation. `https://go.dev/doc/`.

[15] Apache jmeter documentation. `https://jmeter.apache.org/usermanual/index.html`.

[16] Mongodb pricing. `https://www.mongodb.com/pricing`.

[17] Paul E. Hoffman and Patrick McManus. DNS Queries over HTTPS (DoH). RFC 8484, October 2018.

[18] Mongodb forum. `https://www.mongodb.com/community/forums/t/high-number-of-connections-and-opcounters-without-anyone-using-the-cluster/13190/2`.

[19] Aws pricing calculator. `https://calculator.aws/#/addService`.

[20] Google cloud pricing calculator. `https://cloud.google.com/products/calculator`.

[21] Google cloud functions pricing. `https://cloud.google.com/functions/pricing#networking`.

[22] Amazon api gateway pricing. `https://aws.amazon.com/api-gateway/pricing/`.

[23] Ovh cloud vps. `https://www.ovhcloud.com/pl/bare-metal/high-grade/hgr-hci-6/`.

[24] Atman vps. `https://www.atman.pl/uslugi/przetwarzanie-danych/serwery-dedykowane/serwery-dedykowane-zamow-online//`.