Serverless technologies comparison
Large Systems project

Sean Liao and Mar Badias

December 19, 2019

Abstract

Write abstrat here

1 Introduction

Public clouds are growing, and with it comes the latest push into serverless offerings. These come in many forms depending on the abstraction level, but they can largely be grouped into: long-lived containers, short-lived containers, functions as a service.

Long-lived containers or Platform as a Service (PaaS) represent the first generation of serverless technologies. These can be full-fledged, stateful applications, packaged in containers. The clouds will take these and run them for you on VMs. Auto-scaling and load balancing is usually offered, but fast startup times are not guaranteed. These should be considered an alternative UI to the underlying VMs, which will be reflected in the pricing model (charge for underlying VMs). Examples: AWS Elastic Container service, GCP App Engine, Azure App Services, Alibaba Container Service.

Functions as a Service (FaaS), currently the highest level of abstraction and the second generation of serverless technologies. Developers provide their application code for the clouds to compile, package, deploy, and run. These are short-lived and stateless, an instance may be started for every request and killed after it completes. Billing is only for the time it is running serving a request. Examples: AWS Lambda, GCP Cloud Functions, Azure Functions, Alibaba Function Compute, IBM Cloud Functions, Zeit Now.

Short-lived containers or Containers as a Service (CaaS). Is the newest technologies short-lived, stateless runtimes and a similar billing model. Where they differ from FaaS is that they introduce containers, giving developers control of the execution environment, allowing them to run languages or runtimes unsupported by FaaS. Examples: AWS Fargate, GCP Cloud Run, Azure Container Instance, Alibaba Elastic Container Instance.

Given the amount of services of this type in the current market we defined a selection criteria based on the most popular features [1] of serverless computing: dynamically managed runtimes, globally reachable HTTP(S) endpoint, pay only for usage and autoscaling. Having defined these characteristics, we selected the products that provide them offered by the top 5 cloud providers as of 2019: Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, Alibaba Cloud, and IBM Cloud. Additionally, we also included Zeit, a startup in the FaaS space popular for its streamlined experience. The products selected are:

- AWS Lambda (FaaS)
- GCP Functions (FaaS)
- GCP Cloud Run (CaaS)
- GCP App Engine (PaaS)
- Azure Functions (FaaS)
- IBM Cloud Functions (FaaS)
- Alibaba Cloud Function Compute (FaaS)
- Zeit Now (FaaS)

As mentioned in section before, the the pricing model for PaaS (charge for underlying

VMs) does not fit in the 'Pay only for usage' defined in our selection criteria. However GCP App Engine offers the possiblitily to scale your applications to 0 when it is not been used so it fits into our selection criteria.

Our reaserch will measure and analiyse its CPU performance, overhead and specially their capacity to scale when the workload increases. Scaling entails creating more instances, known as cold starts, and assigning new resources between them without influencing others' instances performance.

2 Related work

Serverless technologies have been subject of previous research. We can find good examples of PaaS analysis in [2] and [3], where the reaserchers analize Google App Enginee performance with CPU-intensive applications. CaaS have also been a topic of study, comparing them to long live containers or performance test among of different provides. We can find examples of this in [4] and [5]. Comparison between Faas and short-lived containers has also been analized but from a functionality point of view but oversimplificating it and not taking into account the performance [6][7][8].

FaaS on its own also has been subject of previous research. An excellent example is [9] where multiple serverless providers are continuously been benchmarked. Another example of a comparison of Faas providers is [10]. In [**?**] the researches dicuss the advantages of using cloud services and AWS Lambada for systems that require higher resilance. Finally, is necessary to mention [**?**], which fouses in its performance evaluation and also benchmarking the data transfer to storage and its lifetime of the major cloud functions providers.

Aldought serverless performance has been heavily analized, to the best of our knowledge their scaling and its consequences to performance and overhead have not been comprehensively analysed yet, which motivates our research.

3 Research question

Having defined the services that fullfill our selection criteria, we needed to answer the following question:

- **When one should be chosen over the others when all are available?**

For deciding which is preferable we will consider the following subquestions:

- **Does scaling out affect the performance of the services?** When the workload increases, serverless platforms should create more instances and assign more resources to them to provide an stable performance to the client. The stablest performance and overhead when incresing the workoad would be considered better.

- **Is platform overhead affected by parallel requests?** We plan to measure the excess of computation time introduced by the platform as it supposes extra time in client request. We will measure the overhead under low and heavy workload. Lower platform overhead will be considered more preferable, specially under instense workload.

- **Which one has lower cold start latencies?** When a new instance receives its first request, the response time increases because this instance must be created. When recieving multiple parallel request, more instances will be created possibliy introducing a major latencie/overhead.As answering the client request needs to be done as fast as possibles, lower cold start overherad/latency will be considered preferable.

4 Methods

As mentioned in section 1, we want to test the products offered by the top cloud providers as of 2019. In order to answer the reseach questions we used used image processing as our test workload, a real world [11] use case. Specifically we will be testing image resizing (thumbnail creation).
We aim to use the same code for all platforms (excluding API adapters) and our choice of language is Python because is one of the few languages that all platforms support. We will send images as HTTP requests to the various platforms to be resized. The code deployed on the platforms will be responsible for resizing and sending the image back to the client. Image processing was selected as it represents a common workload that, without hardware accelerators, relies heavily on CPU performance. It also does not require access to external resources, such as databases, which while also a common workload, introduce too much variability.

Each service was configured as similar as possible. We the instance image was set to 128 MB and the highest possible version of Python was used. Also, we choose a location as closest as possible to Amsterdam. In the table 1 we can see the products that have been use with their configuration/
During these experimients, the following variables will be measured:

- Success ratio. How many request answer with HTTP code 200.

- Compute performance or resizing time. How much time does the platform need to do the resizing opetation. This has been measured in the code deployed in the platforms and sent back in the response headers.

- Overall performance or client time. How much does it take for each services to performe a request from the client's point of view.

- Platform overhead. Defined as the extra time introduced by the platform to handle the request and deliver it to our code. It will be calculated using the overall performance minus the compute performance

Type	Platform	Product	Web Framework	Instance Size	Pyhon runtime	Location of the DC
FaaS	AWS	Lambda	Custom	128 MB	3.8	London, UK
FaaS	GCP	Functions	Flask	128 MB	3.7	St. Ghislain, BE
FaaS	Azure	Functions	Custom	128 MB	3.7	NL
FaaS	IBM Cloud	Functions	OpenWhisk	128 MB	3.7	London, UK
FaaS	Alibaba Cloud	Function Compute	Custom	128 MB	3.6	Frankfurt, DE
FaaS	Zeit	Now	Python http.server	128 MB	3.6	Brussels, BE
CaaS	GCP	Cloud Run	Python http.server	128 MB	3.8	St. Ghislain, BE
PaaS	GCP	App Engine	Flask	128 MB	3.8	St. Ghislain, BE

Table 1: Serverless services that will be tested in this reaserch.

- Instance ID. Each instance has a unique ID which will be added as a variable in to the headers response.
- Time of the day we perform the request. In order to be able to observe the variablity of the results over time.

The tests will consist of two parts. First, we will do 10 request sequentially, which we will call *single* test, and, after 20 minutes to ensure that the previous part has finished, we will test again 10 sequentally requests but executing 50 of them in parallel, which we will call *multi* test .These will be performed duing four days, non stop, with one test every hour. Using the variables mentioned before, the following data can be obtained:

- We can observe the behaviour of all services with a high and low workload.
- Using the instance ID is possible to determine the number of instances created and if the execution is a cold start. Also, it they have been reused for executing several requests.
- The client time minus the resizing time shows the overhead introduced by the network and the platform. As we know which request supose a cold stard, we can also appreciate the overhead differences between a cold start and an already iniciated instance.
- Finally, with the time we can express the mentioned variables and verify if they change over time.

Service	Total request	Success rate	Cold starts ratio	Parallel instances	StDev (parallel inst.)
AWS Lambda	47000	1	0.097	50.402	3.858
GCP Functions	46972	0.999	0.062	36.348	6.231
Azure Functions	46997	1	0.005	3.598	0.680
IBM Cloud Functions	47000	1	0.097	50.685	5.256
Alibaba Cloud Function Compute	43501	0.926	0.169	81.043	4.427
Zeit Now	46995	1	0.08	41.663	4.962
GCP Cloud Run	47000	1	81.163	33.575	
GCP App Engine	47000	1	0.049	29.402	5.985

Table 2: Results multi test.

Service	Total request	Success rate	Cold starts ratio	Parallel instances	StDev (parallel inst.)
AWS Lambda	940	1	0.098	1.022	0.209
GCP Functions	940	1	0.027	1.022	0.209
Azure Functions	940	1	0.099	1.022	0.209
IBM Cloud Functions	940	1	0.099	1.022	0.209
Alibaba Cloud Function Compute	940	1	0.098	1.022	0.209
Zeit Now	940	1	0.098	1.022	0.209
GCP Cloud Run	940	1	0.098	1.011	0.104
GCP App Engine	940	1	0.001	1.011	0.104

Table 3: Results single test.

5 Results

After performing the mentioned experiments we obtained the following results:

5.1 Overall performance

The table shows. Parallel instances say they are parallel duirng a single test cycle. Overall performence does not change during time (time of the day does not matter)

5.2 Overhead

In figure XX we can find one plot for each service we have analized. They show the overhead of single vs muntiple and the other cold vs warm.

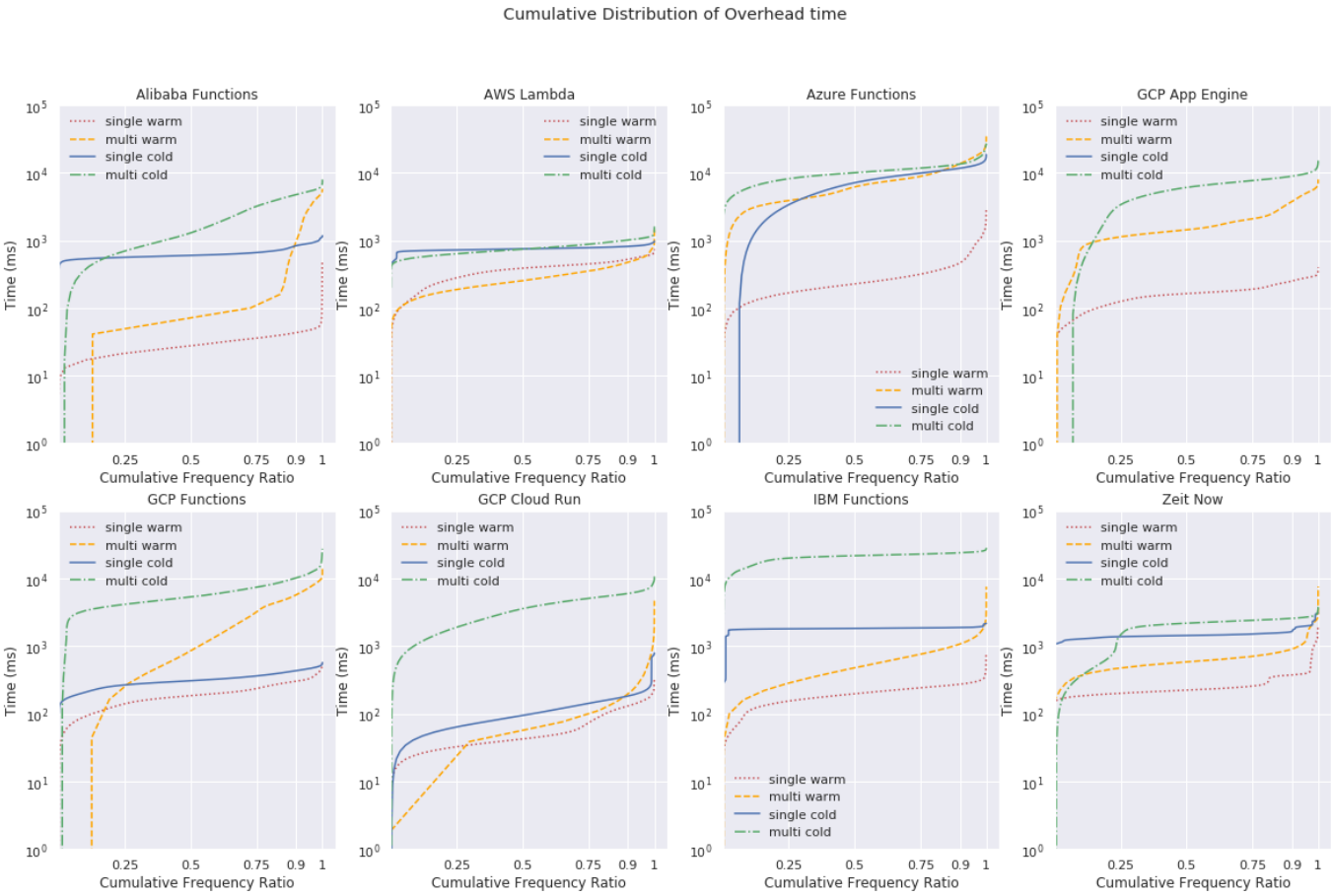

Figure 1: Acumulative distribution of overhead time

5.3 Compute performance

In figure XX we can see strictly the time of resizing the image in each test(multi vs single).

6 Discussion

6.1 Scaling

From Table 2 we observed elevated failure rates for Alibaba Function Compute. Their documentation states they have a limit of 50 instances per service, and even though our client places a strict limit of 50 inflight requests at any point in time, their scheduler might not be keeping up.
Also from Table 2 we can see the average number of unique instances per test cycle. Ideally, this number would be 50, to match the number of parallel requests we are sending.
On the low end, Azure Functions is notable for only starting 4 instances. This may be from a documented limitation of only starting at most 1 instance per second with HTTP

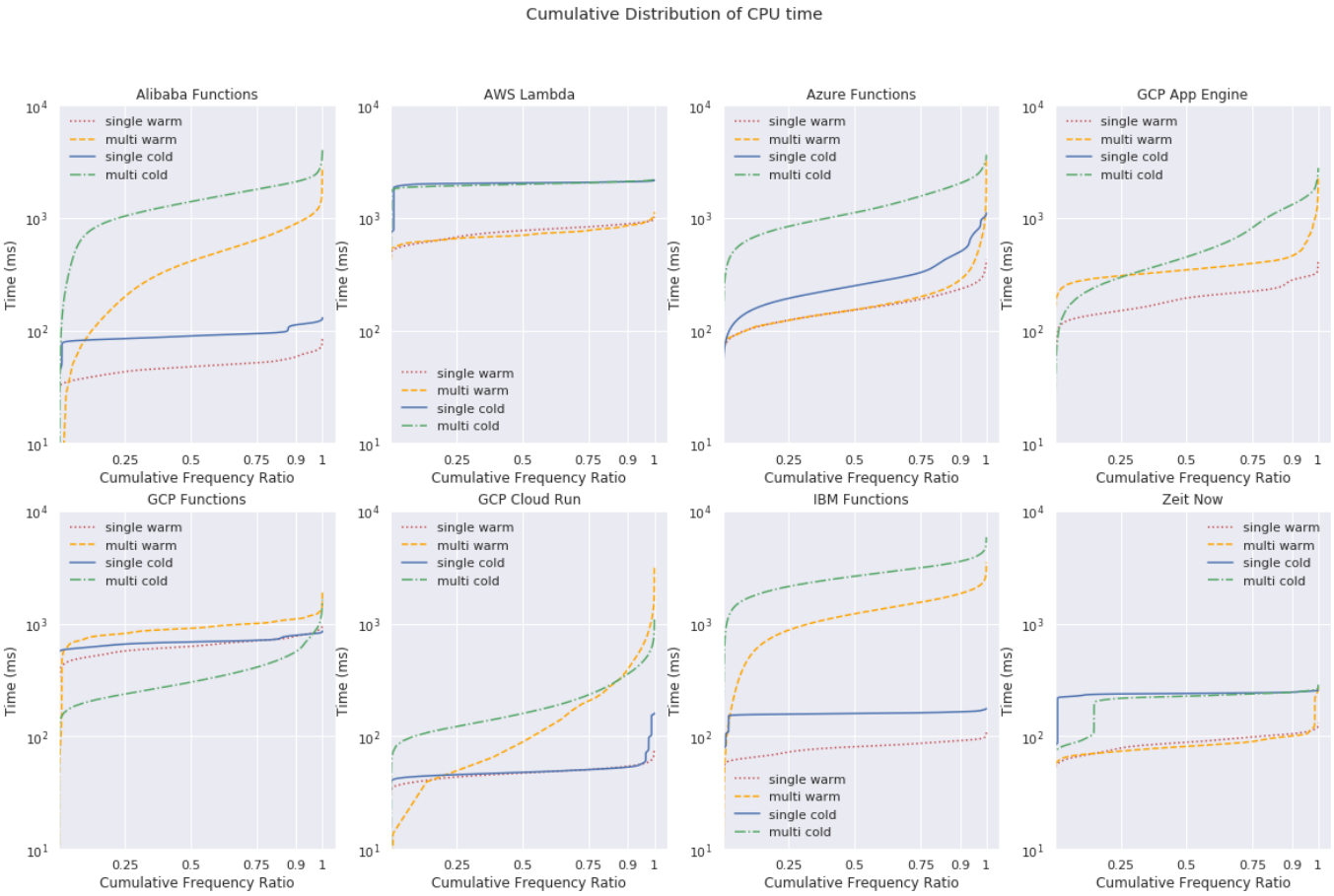

Figure 2: Acumulative distribution of overhead time

triggers. Their load balancers still hold the requests in queue, but they appear to be reluctant to start new instances even as sufficient time has passed.
At the high end, we configured GCP Cloud Run to accept up to 5 concurrent requests per instance, expecting it to utilise the available memory and CPU more efficiently. Unfortunately we are unsure if this negatively impacted their scheduler as we observed a less efficient reuse of instances and greater variability in the number of instances started.

6.2 Warm vs Cold Starts

An often cited concern of using truly on-demand compute services is the cold start, when the platforms have to start up a new instance to handle increases in load. This is such a concern that some some platforms have specific features built to keep your instances warm, such as AWS Provisioned Concurrency and Azure Functions Premium Plan.
Our results in Figure 2 show that, as expected, in most cases, single concurrency cold starts are slower than warm starts and the start times are highly consistent. Cold start times are much more variable at higher concurrency levels, with the exception of AWS

Lambda which appears unaffected.
What is unexpected is in Figure 3, that that cold starts affect CPU performance. These instances consistently perform worse on a cold start, even as they are reused for future requests. Further testing would be needed to identify the root cause of thus.

6.3 Technologies

GCP App Engine is the oldest technology being compared. Under the hood it appears to be an NGINX reverse proxying Python apps through Gunicorn. Its scaling is controlled by CPU utilization, and in our case a heavy workload pushes that up and limits the concurrent requests a single instance can handle. While it can scale to zero instances, each instance has a high (15 minute) startup fee resulting in higher costs for short spikes in traffic.
The current generation of Functions as a Service are built on a wide variety of technologies. AWS and GCP have publicly stated that they use their own VMs, Firecracker and gVisor respectively, to isolate workloads. The results from Figure 3 show they provide a highly consistent environment even under load. Zeit Now uses AWS datacenters for their serverless offering. They behave similarly to AWS Lambda and we believe that is what they use, but with the largest machine type.
Alibaba Functions, Azure Functions, and IBM Functions all appear to be implemented on Docker containers, as either their developer tooling or documentation hint at the possiblitily of customizing the runtime in not very well documentated ways. They exhibit similar characteristics of degraded performance under load.
GCP Cloud Run is a fully managed Kubernetes runtime and the only service we tested that exposed the Docker runtime directly. As expected, single concurrency performance is consistent but drops when more requests are routed to a single instance. As the industry moves to converge on Kubernetes as a common interface and runtime, we expect more fully integrated products in this space in the future. While other platforms provide container runtimes or hosted Kubernetes, they are at a lower level in the stack and don't provide the full functionality we were looking for. Zeit also used to provide a Docker runtime, but dropped it in a strategic pivot in 2018.

6.4 Research Questions

(answer research questions)

References

[1] CloudFlare. What Is Serverless Computing? — Serverless Definition. `https://www.cloudflare.com/learning/serverless/what-is-serverless/`.

[2] Prodan R. Sperk M. Ostermann S. Evaluating High-Performance Computing on Google App Engine. IEEE Software (Volume: 29, Issue: 2, March-April 2012).

[3] M. Wojcik P. BubakM H. Malawski, M. Kuzniar. How to Use Google App Engine for Free Computing. IEEE Internet Computing (Volume: 17, pp. 50-59, 2013).

[4] Michael Wittig. ECS vs. Fargate: What's the difference? https://cloudonaut.io/ecs-vs-fargate-whats-the-difference/, 2019.

[5] Y.C. Tay ; Kumar Gaurav ; Pavan Karkun. A Performance Comparison of Containers and Virtual Machines in Workload Migration Context. https://ieeexplore.ieee.org/document/7979796.

[6] Mike Chan. Containers vs. Serverless: Which Should You Use, and When? https://www.thorntech.com/2018/08/containers-vs-serverless/.

[7] Philipp Muns. Serverless (FaaS) vs. Containers - when to pick which? https://serverless.com/blog/serverless-faas-vs-containers/.

[8] Chad Arimura. Functions vs Containers. https://medium.com/oracledevs/containers-vs-functions-51c879216b97.

[9] Bernd Strehl. Serverless Benchmark. https://serverless-benchmark.com/.

[10] Maciej Malawski; Kamil Figiela; Adam Gajek; Adam Zima. Benchmarking Heterogeneous Cloud Functions. https://www.icsr.agh.edu.pl/~malawski/CloudFunctionsHeteroPar17InformalProceedings.pdf.

[11] Amazon Web Services. Square Enix Case Study. https://aws.amazon.com/solutions/case-studies/square-enix/.