Followup Seminars - Report

Benchmarking the performance of recent serverless technologies

José Donato*, Marco Vieira†
Departamento de Engenharia Informática
Universidade de Coimbra
*donato@student.dei.uc.pt, †mvieira@dei.uc.pt

***Abstract*—Web technologies are constantly evolving. Serverless Computing is an ambitious web technology that has emerged in recent years and has been the focus of several studies. It is ambitions because Serverless aims to replace the traditional architecture where a server is always running waiting for requests. However, its stateless approach and design characteristics raise some concerns that need to be studied and the primary one is performance. In this paper we study several technologies (both serverless and traditional) normally used to support web applications. We propose a preliminary performance benchmark that allows comparing recent serverless technologies (such as Netlify functions, Next.js API routes and Cloudflare Workers) and traditional technologies (such as Express.js). Results show that all technologies tested provide high up-time and the serverless technologies do not fall behind the technologies based on traditional server architectures.**

***Index Terms*—Serverless Computing,Performance Benchmark**

I. Introduction

Serverless computing (also called Functions as a Service or FaaS) has been a hot topic for the past few years as it aims to replace the traditional servers that are the standard and world-wide used. Serverless proposes an alternative approach regarding how applications run in the cloud: instead of constantly running as tradition servers, serverless consists of stateless functions that run on containers in the cloud and only wake up on demand, i.e., these containers only turn on when they are called. Since the containers are by default *sleeping* (i.e., not running), serverless functions can be an efficient alternative in terms of energy consumption (and consequently, cost). However, if they are by default *asleep*, before the request is processed we need to wait until the containers turn on. This process is called cold boot and it is the primary reason of serverless performance limitations.

This study focus on **assessing the performance of serverless and non-serverless technologies**. Therefore, we propose a **simple performance benchmark that allows comparing alternative technologies** that can be used to develop either serverless or traditional applications.

We can define benchmark as "standard tools that allow evaluating and comparing different systems or components according to specific characteristics (performance, dependability, security, etc)" [1]. For our benchmark we focus on performance and up-time. Therefore, we choose to perform several throughput tests, i.e., the number of items processed per unit time [2]. We are also interested in up-time, i.e., the percentage of time the system is up [2].

There are several studies that compare different serverless technologies [3], [4], [5] or the same serverless technology deployed on different serverless providers [6]. We focus on recent technologies that were not considered yet but are already gaining popularity. This benchmark currently supports three different technologies but can be easily extended to support (as we will see in the Section II). The currently supported technologies are Next.js API Routes [7], Netlify Functions [8] and Express.js [9] (we dive deep into the technologies used in Section II).

We developed the same application using these three different technologies and several scripts to assess them (more detail in Section II). All the source code is available on github [10] and the application with the results is live on benchmarking.vercel.app/sa [11].

As for the results, regarding the tests performed we observed that the serverless technologies behave similarly to applications that follow the traditional architecture. All technologies tested (both serverless and non-serverless) should be viable solutions when the goal is to implement simple web Application Programming Interfaces (or APIs) with high up-time and that may involve expensive computations (e.g., image processing).

The remainder of this document is structured as follows. Section II describes the performance benchmark and explains all its necessary phases. Section III presents the experiences made and their results. Section IV we discuss the results and compare the different technologies under study. Section V enumerates the limitations of this work and why they exist. Finally, Section VI concludes the work. In this section we also consider what could be added to this study in the future.

II. Benchmark Description

Our approach is described in Figure 1. Our benchmark is divided into in three different processes. Therefore, the remainder of this section is divided into three subsections. In each subsection we through the processes and the necessary tools to implement them are explained.

The following list itemizes a brief description about each benchmark process:

- **Application Development** (represented with yellow in the diagram): The first process of the benchmark. This

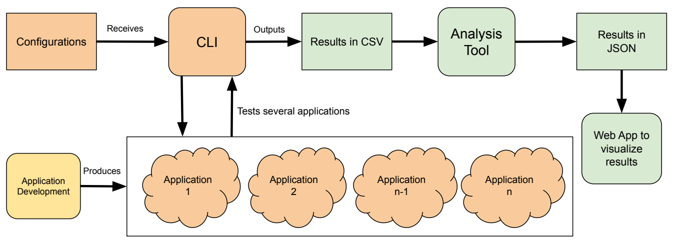


Fig. 1: Benchmark Architecture

phase consists of the development of the several applications that will be tested in the benchmark execution phase. In its subsection we explained what are the application features and why we choose those features. Also, we describe the technologies we choose.
- **Benchmark Campaign** (represented with orange in the diagram): Receives the configurations and is responsible for testing each application while collecting the metrics. In its subsection we enumerate the configurations, the metrics we collect and how command-line interface (CLI) we developed works. The campaign outputs the results in comma-separated values (CSV).
- **Result Analysis and Presentation** (represented with green in the diagram): This is the final process of our assessment. To sort the results from the Benchmark Campaign we developed a simple analysis tool that receives the results in CSV format, performs some analysis and converts them into JavaScript Object Notation (JSON). Finally, we can provide the JSON files to the web application we also implemented in order to display the results.

A. Application Development

The focus of our assessment is performance. Therefore, we tried to collect a set of features that despite being simple allowed to evaluate the performance (primarily the throughput) of serverless and non-serverless functions. This resulted in two features:

- **Dominant Color Endpoint**: An endpoint that receives a JPEG or PNG image and retrieves its dominant color, i.e., the color that appears more times in the image. We think this use case is representative because it involves expensive computations. The bigger the image, the more computations are needed. In code snippet 1 is presented the code we used to calculate the dominant color. The snippet navigates through all the images pixels (represented by matrix) and calculates the color that appears more times.

Listing 1: Compute image dominant color. Adapted from this source code.

```
function dominantColor(w, h, matrix) {
  const size = w * h
  const rgb = [0, 0, 0]
  let idx
  for (let y = 0; y < h; y++) {
    for (let x = 0; x < w; x++) {
      idx = (w * y + x) << 2
      rgb[0] += matrix[idx]
      rgb[1] += matrix[idx + 1]
      rgb[2] += matrix[idx + 2]
    }
  }
  return [Math.floor(rgb[0] / size), Math.floor(
      rgb[1] / size), Math.floor(rgb[2] / size)]
}
```

As we will see in Benchmark Campaign II-B, for more expensive workloads we submit bigger images. For the opposite effect, we use smaller images. This endpoint expects POST requests with the image in the body and is normally located at `/dominantColor`. Alongside the dominant pixel, this endpoint returns the duration it took to compute the pixel (which will be useful in Subsections II-B and II-C for collecting the metrics).
For our tests, we consider three different sizes for of the same JPEG image:
  - One small image: 640x959 pixels.
  - One medium image: 1920x2878 pixels.
  - One large image: 2400x3598 pixels.
- **Echo Endpoint**: Simple endpoint that does not receive anything and just returns a "Hello World" string. This endpoint expects GET requests and is normally located at /echo and its purpose is to assess the throughtput of each application.

As said in the introduction, we developed several applications and they are all open sourced in the github [10].

Each application has its own folder and a Dockerfile associated (i.e., a file that contains the information needed to build a Docker image [12]) to ease the process of running each application. We considered the following technologies to develop the applications under test:

- Serverless - recent technologies that are used to support web applications such as:
  - Next.JS API routes: "API routes provide a straightforward solution to build your API with Next.js" [7].
  - Netlify functions: "Serverless functions built into every Netlify account. No setup, servers, or ops required" [8].
  - Cloudflare Workers: "Simple FaaS on the edge of Cloudflare CDN" [13]. Cloudflare workers was not included in tests because this technology is not compatible with docker and when running locally it crashed as soon as the tests started.
- Traditional - also recent technologies to support web applications:
  - Express.js: "Node.js web application framework" [9].

As we will talk in Section V, the deployment of such technologies is hard without paying some money. Therefore, we deployed the production builds of each application using docker to a server that we own (its specifications are enumerated below). The commands listed in Snippet 2 were used to limit the amount of RAM (to 1GB) and CPU (to 1 CPU, i.e., if the machine has 4 CPUs, each application will have access to atmost 1 CPU) each application can have. These conditions were an attempt to simulate the amount of RAM and CPU they would have when deployed to Cloud Providers such as Vercel, Netlify or Heroku.

Listing 2: Docker commands to start applications

```
$ docker run -d -p 3003:3003 --memory="1g" --cpus="1
    " sa-express
$ docker run -d -p 8888:8888 --memory="1g" --cpus="1
    " sa-netlify
$ docker run -d -p 3001:3001 --memory="1g" --cpus="1
    " sa-next
```

Docker is running in a machine inside a HP DL360e Server. The machine contains 8GB ram and N cores of Intel Xeon E5-2450L and the operating system is Lubuntu 20.04.

B. Benchmark Campaign

The Benchmark Campaign is structured as the diagram 2. The crucial part of this phase is the tool that we develop to

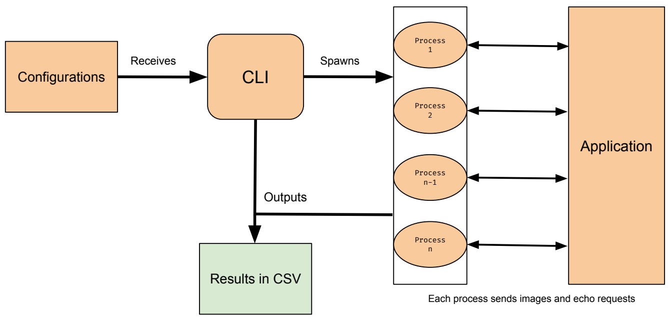


Fig. 2: Benchmark Campaign

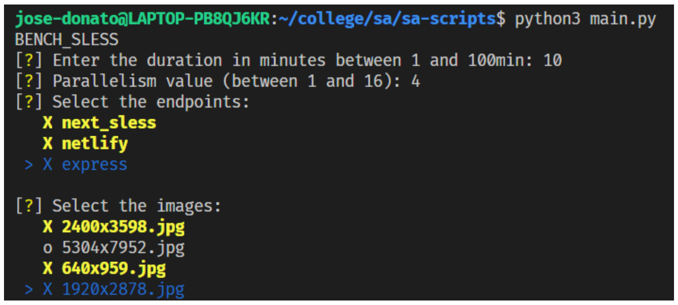
automate the submission of requests (to both endpoints referred in last subsection) of each application. The tool receives a set of configurations, spawns N processes (depending on the configurations) that are responsible for sending requests to each application. While the requests are being sent and received, the metrics are being collected and in the final, the tool outputs all the results in a CSV formatted file.

This tool is a command-line interface (CLI) developed with Python. Initially, the tool was developed in JavaScript but turn out to be an enormous mistake because Node.JS (engine to run JavaScript applications outside the browser) is single-threaded. Therefore, it was impossible to implement true parallelism in our tests, i.e., to perform several requests at the exact same time. This is one of the benchmark requirements because we need to perform requests to the same application to evaluate its scalability capabilities.

Before running the tool, the applications must be running and we need to provide their endpoints in this configuration file.

The tool receives a set of configurations that we can see in Figure 3.

- **Duration**: the duration in minutes the tool will be performing requests for each application (and for each



```
jose-donato@LAPTOP-PB8QJ6KR:~/college/sa/sa-scripts$ python3 main.py
BENCH_SLESS
[?] Enter the duration in minutes between 1 and 100min: 10
[?] Parallelism value (between 1 and 16): 4
[?] Select the endpoints:
   X next_sless
   X netlify
 > X express

[?] Select the images:
   X 2400x3598.jpg
   o 5304x7952.jpg
   X 640x959.jpg
 > X 1920x2878.jpg
```

Fig. 3: CLI Configurations

image). If we choose 10 minutes, the tool will send requests during 10 minutes for each selected image for each selected application (10 times the number of images per application). As for the echo endpoint, the duration the tool will be sending requests will be a third of this value (10/3 minutes per application).
- **Parallelism value**: it is the number of requests the tool will be sending at the same time (illustrated in figure 2). The range presented in Figure 3 (1 and 16) is calculated by querying the number of logical processors of the machine in which the tool is running. This was possible using the multiprocessing python package. With this, we were able to achieve parallelism. If we choose 2, for example, during the duration configured above, the tool will be sending requests from two different processors.
- **Endpoints**: the applications that the tool will be testing. The values that appear here are provided in a configuration file that lists the applications and their two endpoints 3.

Listing 3: Endpoints configuration file available in content.py

```
endpoints = {
    "next_sless": {
        "url": "http://
            SERVER_IP_RUNNING_APPLICATIONS:3001/
            api/dominantColor",
        "echo": "http://
            SERVER_IP_RUNNING_APPLICATIONS:3001/
            api/echo"
    },
    "netlify": {
        "url": "http://
            SERVER_IP_RUNNING_APPLICATIONS
            :8888/.netlify/functions/
            dominantColor",
        "echo": "http://
            SERVER_IP_RUNNING_APPLICATIONS
            :8888/.netlify/functions/echo"
    },
    "express": {
        "url": "http://
            SERVER_IP_RUNNING_APPLICATIONS:3003/
            api/dominantColor",
        "echo": "http://
            SERVER_IP_RUNNING_APPLICATIONS:3003/
            echo"
    }
}
```

From all the endpoints, we can choose which applications the tool will test (normally, all of them).

- **Images**: contains all the images available to send to each application. The images are located in the repository inside the folder assets/. As said previously, there are three different images and for the tests we choose all three. Depending on how many images we choose, the tool will send each one to each application during the duration we specified in the first configuration.

After the configurations are provided, the images selected are loaded and the tests start. A estimated

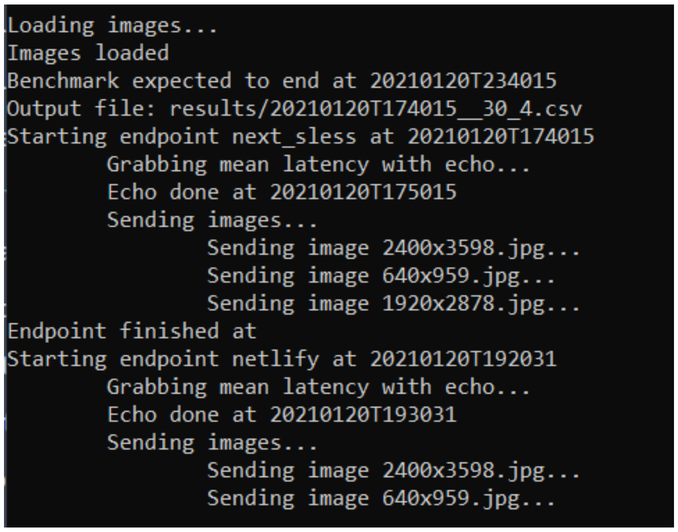

```
Loading images...
Images loaded
Benchmark expected to end at 20210120T234015
Output file: results/20210120T174015__30_4.csv
Starting endpoint next_sless at 20210120T174015
        Grabbing mean latency with echo...
        Echo done at 20210120T175015
        Sending images...
                Sending image 2400x3598.jpg...
                Sending image 640x959.jpg...
                Sending image 1920x2878.jpg...
Endpoint finished at
Starting endpoint netlify at 20210120T192031
        Grabbing mean latency with echo...
        Echo done at 20210120T193031
        Sending images...
                Sending image 2400x3598.jpg...
                Sending image 640x959.jpg...
```

Fig. 4: Benchmark Campaign in progress

finish time is calculated $(currenttime + duration * numberofimages * numberofapplications + (duration * numberofapplications)/3)$. Snippet 4 calculates when the benchmark is expected to end.

Listing 4: Estimated time of benchmark finish. Source code from bench.py file

```python
now = datetime.now()
total_duration = duration*len(endpointsKeys)*len(
    images) + (duration*len(endpointsKeys) / 3)
estimated = now + timedelta(minutes=total_duration)
print("Benchmark expected to end at " + estimated.
    strftime("%Y%m%dT%H%M%S"))
```

During the benchmark execution, two set different metrics are collected:

- Echo Function Duration: the time in milliseconds that takes for the request to leave our tool, reach /echo endpoint and to receive back the "Hello String". It is important to note that his value can be heavily influenced by the network latency. To get the time the function perf_counter() from python time module was used. Since this function returns a clock with the highest available resolution is ideal to measure this short duration.

Listing 5: Echo Function Duration. Snippet from bench.py file

```python
start = perf_counter()
res = requests.get(endpointUrl) #echo endpoint
end = perf_counter()
#end - start  * 1000 = duration in milliseconds
```

- Image Function Duration: the time in milliseconds that each application takes to calculate the dominant color. As explained in Subsection II-A, the endpoint returns this value. Contrary to the previous metric, the network latency does not matter here because the computation is done during the function execution at the application side. Since the applications we developed were using JavaScript, we used the function performance.now() from perf_hooks Node.js module which is similar to perf_counter() from python.

Listing 6: Image Function Duration. Source code from Express Application.

```javascript
const { performance } = require("perf_hooks")
....
app.post('/api/dominantColor', async (req, res)
    => {
    const start = performance.now()
    const [data, base64] = req.body.content.
        split(',')
    if (data === 'data:image/jpeg;base64') {
    const rawImageData = jpeg.decode(
        base64ToArrayBuffer(base64))
    const mean = meanRgba(rawImageData.width,
        rawImageData.height, rawImageData.data).
        toString();
    const end = performance.now()
    const duration = end - start
    return res.status(200).send(`${mean};${
        duration};jpeg;${rawImageData.width}x${
        rawImageData.height}`)
  }
  ....
}
....
```

As we can see in Snippet 6, the usage of performance.now() is similar to perf_counter.now() in python. The only difference is that the JavaScript version returns the high resolution timestamp in milliseconds instead of seconds. Therefore, we do not need to multiply this value by 1000.

We believe that although they are simple, these two metrics allows us to compare the different applications (and furthermore serverless against non-serverless) in terms of performance.

Finally, the tool writes all the outputs to a CSV formatted file that will be analyzed in the following Subsection.

C. *Result Analysis and Presentation*

This last phase is related to the output of the Benchmark Campaign. The snippet 7 is the example content that one of the output files can have.

Listing 7: Example of results' output file. All CSV outputs can be found on results/ folder.

```
next_sless,echo,268.9282999999705,success
...
next_sless,image
    ,3325.2480999981344,2591.9733029976487,jpeg,1920
    x2878,success
```

```
...
next_sless,image
    ,498.69630000102916,347.68583200126886,jpeg,640
    x959,success
...
netlify,echo,49.6642999969481,success
...
express,image,3694.3297000034363,2863.7091179937124,
    jpeg,1920x2878,success
...
express,image,49232.63100000005,-,-,-,failure
...
```

The rows presented are all valid rows that can appear in these files. First column identifies the application, second column identifies whether it is related to the echo test or image test:

- If is related to **echo test**: third column is the metric Echo Function Duration (in milliseconds) and last column tells whether the test was successful or not.
- Otherwise, is related to **image test**: third column indicates the duration from the request left our assessment tool until the response was received (this value will not be used for our tests), forth column indicates the metric Image Function Duration (in milliseconds), fifth column indicates the image type and sixth column the image dimensions.

Finally, any test can fail the last column of both tests tell whether it was successful or not (last row of Snippet 7 is an example of an image test that failed).

The results' countless lines do not mean much alone. Computations and data processing are needed to transform the several numbers in something noticeable. Therefore, we made another Tool (called Analysis Tool in diagram 1) and the source code can be find analysis.py. This tool was also written in python3 and uses pandas, a python data analysis library. Our analysis tool receives all the results in CSV and calculates the average time and total number of successful requests for each type of test. The average time is calculated by dividing the sum of all metric values (of course it depends whether it is a echo test or image test) by the number of requests.

Listing 8: Hypothetical result to explain how Average Time is calculated

```
next_sless,echo,100,success
next_sless,echo,120,success
next_sless,echo,110,success
express,image,3000,1000,jpeg,1920x2878,success
express,image,3200,1200,jpeg,1920x2878,success
```

Let's imagine the list 8 represented one CSV result of our tests. For the echo test, next_sless application would have an average time of $(100+120+110)/3 = 110$ millisseconds and a total of 3 requests. For the 1920x2878 (large) image, express application would have an average time of $(1000+1200)/2 = 1100$ millisseconds and a total of 2 requests.

This average value that we can conclude from our metrics is important because given two applications A and B, if A has a lower average time than B for the same test it means that application A processed more requests than application B in the same period of time (i.e., application A had a higher throughput than application B). Therefore, we can conclude already that the lower the average the better the application performed in a certain test.

Analysis Tool output is a JSON file that contains an array with the size of number of applications tested (for example, if we tested three applications, the array would have three elements). Each position in the array is an object that contains the information of certain application (an example of such object can be seen in Listing 9). Inside the object, there are three keys:

- **application**: identifies the application
- **echo**: information related to the echo test (i.e., total number of successful requests and the mean value)
- **image**: array with each image tested. In each position there is an object with information related to that image test (i.e., image size, total number of successful requests and the mean value).

Listing 9: Example object inside results JSON array. All the JSON outputs can be found on results_json/ folder.

```json
{
    "application": "next_sless",
    "echo": {
        "total": 8473,
        "mean": 51.03107600614179
    },
    "image": [
        {
            "size": "640x959",
            "total": 3330,
            "mean": 308.17142929579256,
        },
        {
            "size": "1920x2878",
            "total": 492,
            "mean": 2630.7996119227473,
        },
        {
            "size": "2400x3598",
            "total": 320,
            "mean": 4637.17616056874,
        }
    ]
}
```

After processing all the CSV results to JSON results with the Analysis Tool we can proceed to the final step of our benchmark which is providing them to the Web Application to easily visualize them and finally compare the different technologies.

This web application was developed using Next.js and Recharts. The source code can be found on webapp folder and the application is live on benchmarking-web.vercel.app/sa.

III. Experiments & Overall Results

We performed the tests with different set of configurations. The differences in configurations between each run are listed

TABLE I: Set of benchmark runs

Experience ID	**Configuration**	
-	**Duration (min)**	**Parallelism**
e-1-1	1	1
e-5-1	5	1
e-10-1	10	1
e-20-1	20	1
e-30-1	30	1
e-30-2	30	2
e-30-4	30	4
e-30-8	30	8
e-30-16	30	16

in Table I (we varied the value of parallelism and duration). We started with a duration of 1 minute and increased until we reached 30 minutes always with the value of parallelism. Then, we fixed the duration at 30 minutes and started varying the parallelism value (from 1 to 16, in base powers of 2 i.e., 1, 2, 4, 8 and 16).

At first, we were testing with the CLI tool and the applications running in the same machine. We came to the conclusion that this approach was not representative and the results would not be interesting. Moreover, the tests were realized with the client machine running the CLI positioned 200 kilometers away from the server running the applications (server specs were presented back in Section II).

Figures 5, 6, 7, 8 are simple line charts from our web application and each chart contain the results of our several runs. In the X Axis we can see the matching experience ids from table I and in Y Axis the Average Time in millisseconds. It is important to note the lower the value the better a certain application behaves (this is true for any of our tests). These charts show how the different applications behave when increasing the duration and/or the parallelism value.

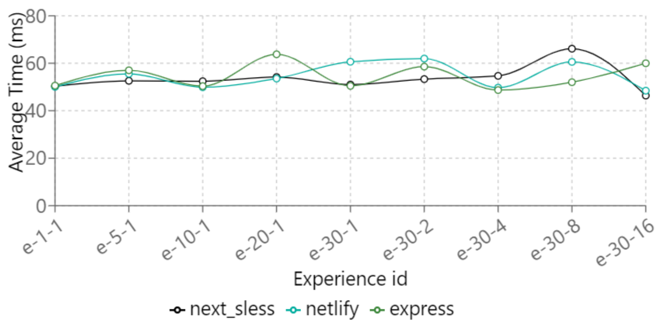


Fig. 5: Line chart for Echo test

Table II presents a summary of all runs. It contains the average value of each test in all runs for each application.

All the charts and tables are also available in our web application benchmarking.vercel.app/sa. We strongly recommend its visualization for a more interactive analysis.

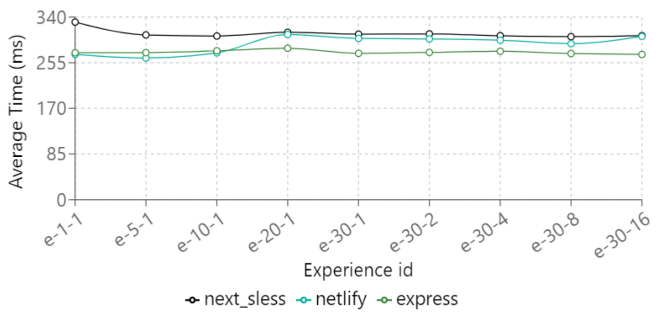


Fig. 6: Line chart for Small image test

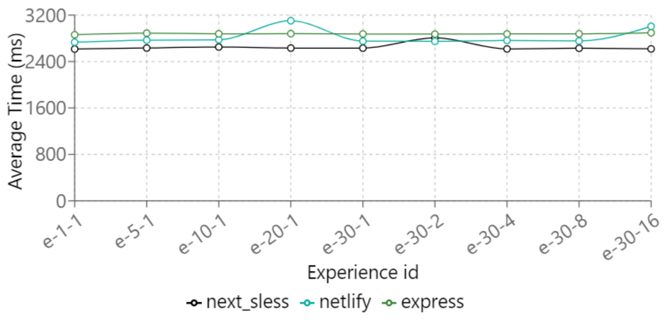


Fig. 7: Line chart for Medium image test

IV. Results Discussion

In all tests we did not have a single failure, i.e., all the requests returned the expected output without timing out. Thus, the charts and table presented in the last Section only contain successful requests.

This is the first conclusion that we came across. All three technologies are robust enough to perform expensive computations for image processing (regarding the image test) or for serving simple APIs (regarding echo test) even when the number of simultaneous requests were increased (by increasing the parallelism value). Therefore, any service where high availability is a main requirement both supported serverless applications (i.e., Next.js API routes and Netlify functions) traditional application (i.e., Express) can be used.

The four different tests presented similar results. Since the applications had similar average values we conclude that their performance is similar. This may have happened due to several reasons:

TABLE II: Results of all Runs

Application ID	**Tests Averages (ms)**			
-	**Echo**	**Small Image**	**Medium Image**	**Large Image**
express	54.6083	274.6407	2878.2962	4522.5559
next_sless	53.4500	309.3885	2648.0012	4617.0934
netlify	54.4844	289.5480	2822.5176	4847.6377

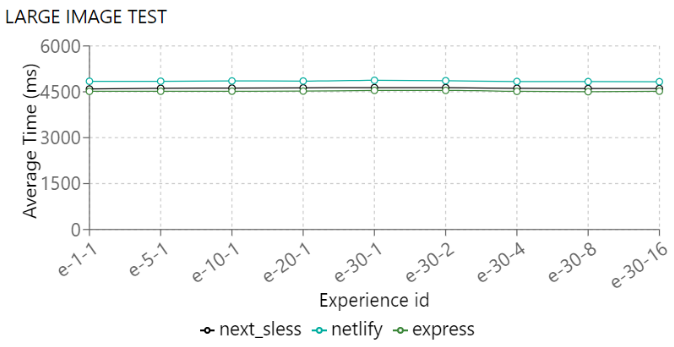


Fig. 8: Line chart for Large image test

- All the applications are being served with exact same conditions.
- The applications are all node.js based. In future work VI, we propose to add more applications which are not related to node.js (for example, applications developed with python)
- We only increased the parallelism to a maximum value of 16 which means that at most there are 16 simultaneous requests to each application. This value may be too low to start damaging the applications' performance.

By analysing the charts, the non-serverless technology (i.e., express application) was more consistent overall through all the tests than its competitors. However, looking at table II, we can also conclude that in some situations serverless technologies had better results. For example, in Echo test next_sless application outperformed the other two and in Medium image test both serverless technologies outperformed the traditional technology. Again, the possible reasons for this to happen were enumerated in the previous list.

According to these results, we can draw some conclusions. When developing Next.js web applications and there is a need for simple API endpoints (which can involve image processing), there is this good possibility of using Next.js API Routes instead of spinning up another Node.js instance with express for this purpose only. The same can be applied to use netlify functions for that purpose also. Since the results are so similar, in this situations may not be worth to have an additional Express API. In the other hand, we conclude that Express provides more stable and slightly better results for this purpose. Therefore, it can be a viable solution in case we can afford to have a node.js server in addition to our web application.

Regarding Cloudflare Workers, it is an interesting technology and the deployment to its Cloudflare platform was flawless. However, we hope to see more improvements when self-hosting it and official support for docker, for example.

Of course these conclusions are only valid for the conditions and environments similar to what we had in our study (e.g., running the applications using docker with similar hardware, processing similar image sizes, etc.).

V. Work Limitations

Since serverless is all about the cloud, during this study we faced several obstacles and had to look for alternatives to achieve minimally representative results. First, our applications were all being served directly from cloud providers. We had Next.js API Routes deployed in Vercel (which is the company behind Next.js), Netlify functions deployed in Netlify, Cloudflare Workers in Workers service and Express deployed in Heroku. However, this had several problems:

- Different cloud providers give the applications different hardware, making really hard to assess all the applications in a fair way.
- Because we are assessing the throughput of the applications, the CLI we developed send a lot of requests. Also, we were using the cloud providers' free plans which made it impossible to do some kind of testing because we ended up exceeding the plans' limits.

Afterwards, as we explained before, the applications under test ran in a server that we control using docker. Since we do not replicate all the mechanisms of the dedicated serverless cloud providers, our approach may not be fair because the serverless applications do not fall asleep (and consequently, there is no cold boot). We assume that we do not take into account this feature of serverless and is a limitation of our assessment and plan in Future work VI to add mechanisms to simulate the such features. In the other hand, this way we were able limit all applications to the same conditions and provide a fair comparison between at least the serverless applications.

We also acknowledge that the tests we have chosen are stateless (e.g., subsequent requests are never dependent on previous ones) which is favorable for serverless technologies. In the future work VI, we also propose to add more tests should be added that take into account some state in order to have a more complete comparison between serverless and traditional applications.

Finally, some results may be a little bit off because the client machine running CLI tool is my work machine. Therefore, at some moments I needed to work at the same time the tests were running and it can result in some minor fluctuations. However, this should have close to zero impact.

VI. Conclusions & Future Work

Despite the difficulties we faced, in this study we designed and implemented a performance benchmark between serverless and traditional technologies that are normally used to support web applications. We have shown that all technologies can provide high availability. Also, the tests performed showed that the serverless technologies do not fall behind the traditional tested application even outperformed it in some tests.

Therefore, we conclude that serverless applications can be a viable alternative in situations where the requirements are stateless, high up-time and may involve some image processing or simple API calls.

All serverless technologies studied are still at an early stage and should be monitored because they will continue to improve. Although Cloudflare Workers failed to integrate our tests it has a great potential since the serverless functions when deployed to Cloudflare Platform will be distributed through its CDN which has servers all around the world.

There are several aspects that can be improved in our work. As potential future work, we have the following suggestions:

- Our tests does not take into account operations that involve some kind of state and this favors serverless technologies over traditional ones. Moreover, more tests can be added to our benchmark.
- All the technologies tested performed similarly. By introducing more applications, namely applications that do not use JavaScript, in the benchmark we can assess whether all the applications being produced with JavaScript was the reason for this. Also, by adding more we would increase the representativeness of the benchmark and, consequently, assess more technologies.
- Improve the architecture where applications run (e.g., support cold boots when applications are inactive) in order to similar to how it works when deploying to serverless cloud providers. This can be done by using Kubernetes [14] or Docker Swarm [15], for example.

References

[1] M. Vieira, H. Madeira, K. Sachs, and S. Kounev, "Resilience Benchmarking," in *Resilience Assessment and Evaluation of Computing Systems*, K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, Eds. Berlin, Heidelberg: Springer, 2012, pp. 283–301. [Online]. Available: https://doi.org/10.1007/978-3-642-29032-9_14

[2] P. Killelea, *Web Performance Tuning: speeding up the web.* " O'Reilly Media, Inc.", 2002.

[3] R. Chard, T. J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, and K. Chard, "Serverless Supercomputing: High Performance Function as a Service for Science," *arXiv:1908.04907 [cs]*, Aug. 2019, arXiv: 1908.04907. [Online]. Available: http://arxiv.org/abs/1908.04907

[4] D. Jackson and G. Clynch, "An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. Zurich: IEEE, Dec. 2018, pp. 154–160. [Online]. Available: https://ieeexplore.ieee.org/document/8605773/

[5] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless Computing: An Investigation of Factors Influencing Microservice Performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2018, pp. 159–169.

[6] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, Jun. 2017, pp. 405–410, iSSN: 2332-5666.

[7] "API Routes: Introduction | Next.js." [Online]. Available: https://nextjs.org/docs/api-routes/introduction

[8] "Netlify Functions | Netlify." [Online]. Available: https://www.netlify.com/products/functions/

[9] "Express - Node.js web application framework." [Online]. Available: https://expressjs.com/

[10] J. Donato, "jose-donato/bench_sless." [Online]. Available: https://github.com/jose-donato/bench_sless

[11] ——, "https://benchmarking.vercel.app/sa." [Online]. Available: https://benchmarking.vercel.app/sa

[12] Docker, "Dockerfile reference | Docker Documentation," Jan. 2021. [Online]. Available: https://docs.docker.com/engine/reference/builder/

[13] "Cloudflare Workers®." [Online]. Available: https://workers.cloudflare.com/

[14] "Kubernetes." [Online]. Available: https://kubernetes.io/

[15] "Docker Engine swarm mode overview." [Online]. Available: https://docs.docker.com/engine/swarm/