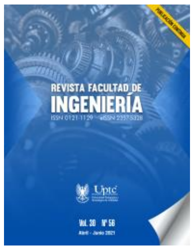
**Revista Facultad de Ingeniería**
Journal Homepage:
https://revistas.uptc.edu.co/index.php/ingenieria



# Towards Digital Transformation: Serverless Function Deployment Strategies

Armando Cabrera-Silva[1]
José Carrillo-Verdún[2]
Patricio Martínez-Palacios[3]
Daniel-Alejandro Guamán-Coronel[4]

**Received:** March 12, 2021 **Accepted:** April 05, 2021 **Published:** April 20, 2021

**Citation:** A. Cabrera-Silva, J. Carrillo-Verdún, P. Martínez-Palacios, D.-A. Guamán-Coronel, "Towards Digital Transformation: Serverless Function Deployment Strategies," *Revista Facultad de Ingeniería*, vol. 30 (56), e12776, 2021. https://doi.org/10.19053/01211129.v30.n56.2021.12776

## Abstract

Digital transformation is a great asset for companies that evolve and drive their activities towards new ways where technology is a great ally, in this evolution, cloud computing plays a key role for transformation. Each company measures its performance through a business model enabled and managed in the cloud and considers the customer experience to differentiate its strategy and opt for

[1] M. Sc. Universidad Politécnica de Madrid (Madrid, España). armando.cabrerasilva@alumnos.upm.es. ORCID: 0000-0002-3465-2440
[2] Ph. D. Universidad Politécnica de Madrid (Madrid, España). jcarrillo@fi.upm.es
[3] Universidad Técnica Particular de Loja (Loja, Ecuador). jpmartinez1@utpl.edu.ec. ORCID: 0000-0002-2390-3291
[4] M. Sc. Universidad Técnica Particular de Loja (Loja, Ecuador). daguaman@utpl.edu.ec. ORCID: 0000-0002-2681-565X

technological solutions that make them different from the market competitors. As a strategic tool for digital transformation, moving from technological solutions from a local environment to one of serverless computing is the next step in the evolution of software. This allows software engineers to focus on coding for services, microservices, or functions to meet time-to-market without thinking too much on the complexity to implement and maintain the infrastructure. In this work, we propose a strategy to build and implement functions using a set of serverless runtimes provided by different Cloud Service Providers (CSPs). For the validation, a guided experimentation is carried out in three scenarios, considering the performance of the workload in each runtime and the average execution time of each CSP, which are monitored through analysis and visualization tools. The performance value associated with each CSP allows defining a serverless computing (FaaS) deployment strategy.

**Keywords:** cloud computing; cloud serverless platforms; digital transformation; functions-as-a-service; serverless; serverless runtimes.

**Hacia la transformación digital: Estrategias de despliegue de funciones Serverless**

## Resumen

En la actualidad, la transformación digital es un gran activo para las empresas que evolucionan e impulsan su actividad hacia nuevas formas donde la tecnología es un gran aliado. En esta evolución, la computación en la nube juega un rol clave para dicha transformación. Cada empresa mide su desempeño a través de un modelo de negocio habilitado y gestionado en la nube y toma en cuenta la experiencia del cliente para diferenciar su estrategia y optar por soluciones tecnológicas que los diferencien de los competidores del mercado. Como herramienta estratégica para la transformación digital, pasar de soluciones tecnológicas desde un ambiente local a uno de computación sin servidor es el siguiente paso en la evolución del software. Esto permite a los ingenieros de software concentrarse en el código de los servicios, microservicios o funciones para cumplir con el *time to market* sin pensar mucho en la complejidad para implementar y mantener la infraestructura. En este trabajo, proponemos una estrategia para construir e implementar funciones utilizando un conjunto de *runtimes* sin servidor provistos para diferentes *Cloud Service Providers (CSP).* Para la validación, se realiza una experimentación guiada bajo tres escenarios teniendo en cuenta el rendimiento de la carga de trabajo en cada runtime y el tiempo promedio de ejecución de cada CSP, los cuales se monitorean a través de herramientas de análisis y visualización.

**Palabras clave:** computación en la nube; computación sin servidor; funciones como servicio; plataformas en la nube sin servidor; *serverless runtimes*; transformación digital.

## Rumo à transformação digital: estratégias de implantação de recursos sem servidor

### Resumo

Atualmente, a transformação digital é um grande trunfo para empresas que evoluem e direcionam a sua atividade para novos caminhos onde a tecnologia é uma grande aliada. Nessa evolução, a computação em nuvem desempenha um papel fundamental nessa transformação. Cada empresa mede seu desempenho por meio de um modelo de negócios habilitado e gerenciado na nuvem e leva em consideração a experiência do cliente para diferenciar sua estratégia e optar por soluções tecnológicas que os diferenciam dos concorrentes de mercado. Como uma ferramenta estratégica para a transformação digital, passar de soluções de tecnologia de um ambiente local para um de computação sem servidor é o próximo passo na evolução do software. Isso permite que os engenheiros de software se concentrem no código para serviços, microsserviços ou funções para atender o tempo de colocação no mercado sem se preocupar muito com a complexidade de implementar e manter a infraestrutura. Neste trabalho, propomos uma estratégia para construir e implementar funções usando um conjunto de tempos de execução sem servidor fornecidos para diferentes Cloud Service Providers (CSPs). Para a validação, é realizada uma experimentação guiada em três cenários, levando em consideração o desempenho da carga de trabalho em cada runtime e o tempo médio de execução de cada CSP, os quais são monitorados por meio de ferramentas de análise e visualização.

**Palavras-chave:** computação em nuvem; computação sem servidor; funciona como um serviço; plataformas de nuvem sem servidor; tempos de execução sem servidor; transformação digital.

## I. Introduction

Today, digital transformation is enabling enterprises to increase capacity and integrate IT infrastructure components to improve and innovate processes, applications, and services used by customers and business partners [1]. This involves the inclusion of cultural philosophies such as DevOps [2] practices and tools that improve collaboration and automate deployment processes and continuous delivery. One of the trends in software development involves the decomposition of the system into functional pieces that are developed and deployed independently through environments that allow continuous integration and continuous deployment (CI/CD) [3]. Small functional pieces are known as services, microservices, and Function as a Service (FaaS). A service is considered an abstract resource that represents a specific business capability [4].

At the software level, services expose their functionality through interfaces and ports to be consumed by applications or as input to compose other services according to business needs [5]. Microservices as architecture allow the composition of an application through a set of small, individual, and independent services that run in their process and communicate through lightweight mechanisms, where each service is dedicated to solving a single business capability [6]. Function as a Service (FaaS), as one of the categories of cloud computing, provides a platform that allows each small piece of code (function) to run, manage itself, and use computational resources only when required [7], simplifying the development process, especially when performing virtualizations by reducing execution times and operational management through cloud service providers [1].

In this paper, we propose to evaluate the execution time of three functions written in Python programming language. We first benchmark Cloud Service Providers (CSP) such as AWS Lambda, Google Cloud Functions, Azure Functions or IBM Cloud Functions [9] that support FaaS implementation. Then each function is written following the programming model proposed by the CSPs and using runtimes, supported libraries, and triggers that allow evaluating the execution of the function in "cold start" and "warm start". Finally, to obtain results, real-time data storage and analysis tools are configured to visualize the 300 invocations performed by each function in the CSPs. Works such as [10] exposed AWS Lambda and Amazon MXNET [11] as Deep learning frameworks to evaluate the platform capacity. Cold start and warm start are used for function execution, and metrics such as execution time and performance to evaluate horizontal scaling.

The functions under study SqueezeNet [12], ResNet [13], ResNet-50 [14] are used in computer vision work. In [15] the performance of AWS, Microsoft Azure, Google, and IBM Cloud and the ability to execute compiled files containing mathematical functions coded in NodeJS are evaluated. Two suites of services are used for benchmarking, the first for automated deployment of functions and the other for the orchestration of parallel tasks that enable the execution of such mathematical functions. In [7] the performance of AWS, Azure, IBM, Google and OpenWhisk is evaluated when executing three functions: Fast Fourier Transformation, Matrix multiplication and Sleep function. The execution results show which service provides better performance comparing metrics such as execution time, and performance per resource consumption.

## II. Methodology

The goal of this work is to evaluate the execution time, and performance of three functions written in Python for different contexts (Table 1). For the evaluation, experiments are designed using programming models proposed by CSP. Python 3.6 and Python 3.7 are used to code the functions. The execution of the functions is performed in cold start and warm start [16] using AWS Lambda, Microsoft Azure Functions, Google Cloud Functions, and IBM Cloud Functions. Finally, tools such as Influxdb and Grafana are used for real-time analysis and monitoring of the functions.

**Table 1.** Experimentation functions and requirements.

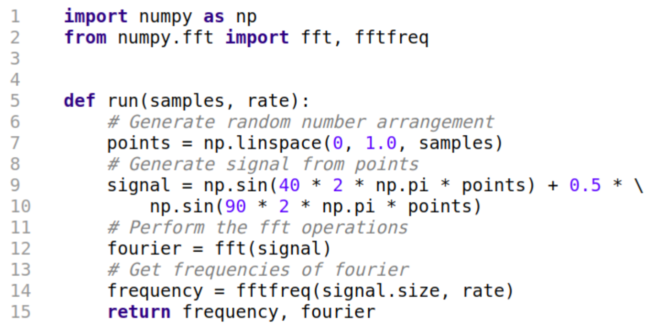
| Function type | Context | Description | Requirements | |
|---|---|---|---|---|
| | | | Services | Libraries |
| Mathematics | Scientific-Mathematics | Performs operations to obtain the Fast Fourier transform | Not applicable | Numpy |
| Machine | Artificial | Performs object recognition | Not applicable | MXNET |

| Function type | Context | Description | Requirements | |
|---|---|---|---|---|
| | | | Services | Libraries |
| Learning | Inteligence | in images | | |
| General | General | Performs data backup operations | AWS Dynamo DB, Azure Mongo DB, Google Datastore, IBM Cloudant | Boto3 PyMongo DataStore Cloudant |

### *A. Experimental Functions*

This section describes the 3 functions used to evaluate the execution time and performance of the runtime of each platform. Each function is coded separating the business logic from the entry point to the function. That is, generic code is created for the four platforms and the function entry point is developed based on the programming model specified for each CSP.

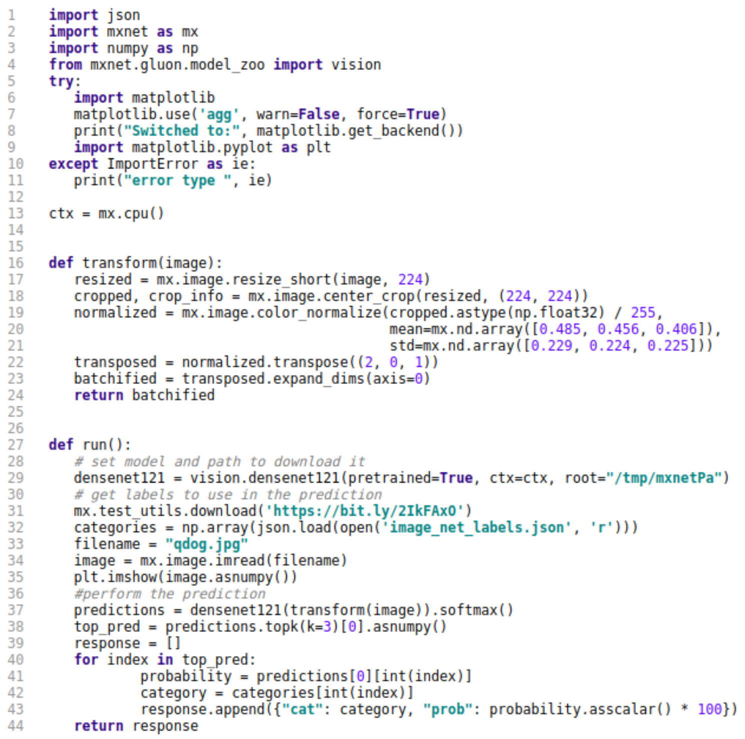
***1) Mathematical Function.*** This function uses mathematical procedures to calculate The Fast Fourier Transform (FFT). The implementation of this type of algorithm within a function becomes relevant by eliminating or simplifying a large number of repetitive processes, achieving results with lower execution costs. The code of the function implemented in Python shows the main procedures to find the frequency and magnitude of a wave, which is generated from numbers obtained at random. The resulting values are stored in a vector. In the function, the variable sample corresponds to data sent from the input function and the rate related to the space between samples. Explaining the function code, in lines 9 to 10, the random signal is generated; in line 12, the FFT is obtained; and in line 14, the FFT frequencies are obtained (Fig. 1).



```
1    import numpy as np
2    from numpy.fft import fft, fftfreq
3
4
5    def run(samples, rate):
6        # Generate random number arrangement
7        points = np.linspace(0, 1.0, samples)
8        # Generate signal from points
9        signal = np.sin(40 * 2 * np.pi * points) + 0.5 * \
10           np.sin(90 * 2 * np.pi * points)
11       # Perform the fft operations
12       fourier = fft(signal)
13       # Get frequencies of fourier
14       frequency = fftfreq(signal.size, rate)
15       return frequency, fourier
```

**Fig. 1.** Fast Fourier Transform function code.

***2) Machine Learning Function.*** The purpose of this function is to perform procedures that simulate machine learning. For this purpose, Apache MXNet is used as a flexible and efficient library for Deep Learning. Additionally, the concept of convolutional neural networks is also applied in this function for the processing and recognition of objects in images. The code implemented in the function shows the use of pre-trained models taken from MXNET's own set of models (See Fig. 2). Explaining the code, it can be seen that in line 10 the Python-pk module, which must be installed on the host operating system, is deactivated. The function proposes 2 methods, the first one called transform (line 16 to 24) is in charge of preparing the image to be used by the second method called run (line 27 to 44). In the run method, the image prediction is performed using a training model provided by MXNet and this model is specified in line 35. In lines 37 to 44, the download of the classification labels needed for the prediction is performed. It is worth mentioning that the implementation of this function could not happen on the IBM Cloud Functions platform. In a first attempt, we tried to implement it by creating an implementation package that exceeds the limits of the platform (maximum 48MB, 53MB obtained). In a second attempt, we tried to implement it using Docker, however, the incompatibility of MXNet to run on Linux Alpine did not allow the correct implementation of this feature.



```python
import json
import mxnet as mx
import numpy as np
from mxnet.gluon.model_zoo import vision
try:
   import matplotlib
   matplotlib.use('agg', warn=False, force=True)
   print("Switched to:", matplotlib.get_backend())
   import matplotlib.pyplot as plt
except ImportError as ie:
   print("error type ", ie)

ctx = mx.cpu()


def transform(image):
   resized = mx.image.resize_short(image, 224)
   cropped, crop_info = mx.image.center_crop(resized, (224, 224))
   normalized = mx.image.color_normalize(cropped.astype(np.float32) / 255,
                                         mean=mx.nd.array([0.485, 0.456, 0.406]),
                                         std=mx.nd.array([0.229, 0.224, 0.225]))
   transposed = normalized.transpose((2, 0, 1))
   batchified = transposed.expand_dims(axis=0)
   return batchified


def run():
   # set model and path to download it
   densenet121 = vision.densenet121(pretrained=True, ctx=ctx, root="/tmp/mxnetPa")
   # get labels to use in the prediction
   mx.test_utils.download('https://bit.ly/2IkFAxO')
   categories = np.array(json.load(open('image_net_labels.json', 'r')))
   filename = "qdog.jpg"
   image = mx.image.imread(filename)
   plt.imshow(image.asnumpy())
   #perform the prediction
   predictions = densenet121(transform(image)).softmax()
   top_pred = predictions.topk(k=3)[0].asnumpy()
   response = []
   for index in top_pred:
           probability = predictions[0][int(index)]
           category = categories[int(index)]
           response.append({"cat": category, "prob": probability.asscalar() * 100})
   return response
```

**Fig. 2.** Machine Learning function code.

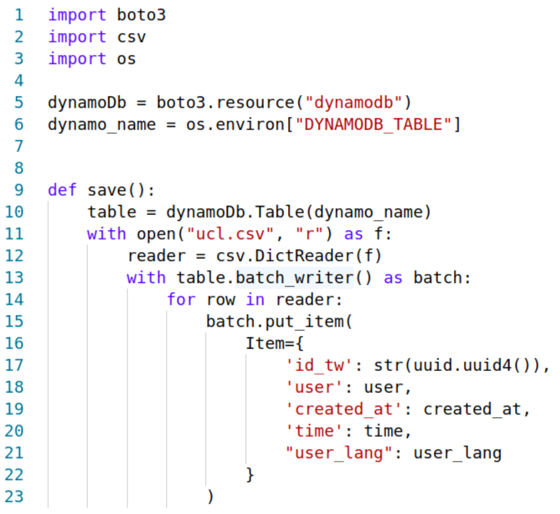
***3) General Context Function.*** The purpose of this function is to perform the processing of a file with CSV extension and store each of the records in a NoSQL key-value database. However, given the different NoSQL services that each cloud provider provides, the functions intended to be executed on the platform are coded according to the specifications of each of them. Therefore, 4 variants of this general context function are created with the objective that they interact uniquely and exclusively with the service provided by the CSP. Each of these variants implements batch insertion mechanisms, a method that proves to be optimal compared to record-by-record insertion. The variants of the general function coding according to the supported FaaS platform (Fig. 3-6) are:

**- AWS Lambda**. AWS offers a Software Developer Kit available for Java, C#, PHP, Python that allows the use of the Python SDK called Boto3 (Fig. 3). In lines 13 to 23, we visualize the record insertion operation in DynamoDB by implementing the table.batch_writer( ) method, which controls the batch size and record insertion. It is important to mention that within the AWS DynamoDB service, the default configuration was modified to move from the free layer to the on-demand layer to adjust the number of computational resources required to execute the function.

**- Azure Functions.** This function uses MongoDB, which is part of the CosmosDB service provided by Microsoft Azure (Fig. 4). Line 27 shows the implementation of MongoDB's collection.bulk_write method to perform bulk insertion of records. However, this method requires making use of a list object which is defined in line 15 to store in memory the records to be inserted into the repository. In the CosmosDB service, the default values of MongoDB API and AWS DynamoDB related to the Request Units - RU/s are adjusted from 400 to the upper limit corresponding to 10,000 RU/s to improve the performance of this service.

**- IBM Cloud Functions.** IBM Cloud provides the NoSQL database service called Cloudant. This service, like MongoDB, requires the configuration of a list object within the database.bulk_docs method to store the records. Such functionality is accessible through the specification made in line 1 from Cloudant import CouchDB (Fig. 5).

**- Google Cloud Functions.** Google Cloud Platform provides a NoSQL service called Google Cloud DataStore (Fig. 6). Line 1 visualizes the use of the library available for Python under the same name of the service. This service allows the insertion of records in batches, however, the batch size must be controlled by the same code taking into account that a maximum of 500 records are allowed in each insertion (see lines 21 to 27).

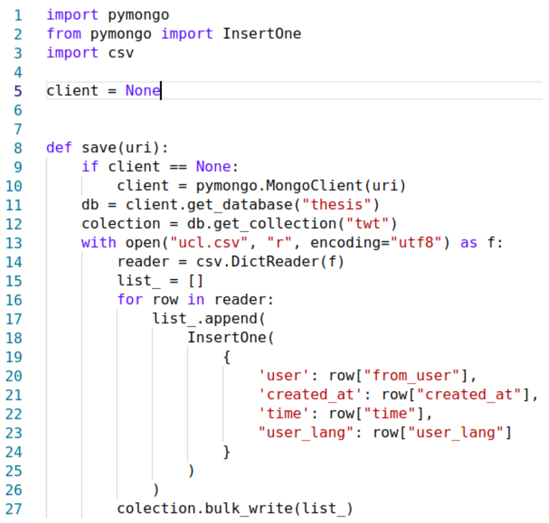


```python
import boto3
import csv
import os

dynamoDb = boto3.resource("dynamodb")
dynamo_name = os.environ["DYNAMODB_TABLE"]


def save():
    table = dynamoDb.Table(dynamo_name)
    with open("ucl.csv", "r") as f:
        reader = csv.DictReader(f)
        with table.batch_writer() as batch:
            for row in reader:
                batch.put_item(
                    Item={
                        'id_tw': str(uuid.uuid4()),
                        'user': user,
                        'created_at': created_at,
                        'time': time,
                        "user_lang": user_lang
                    }
                )
```

**Fig. 3.** AWS Lambda.

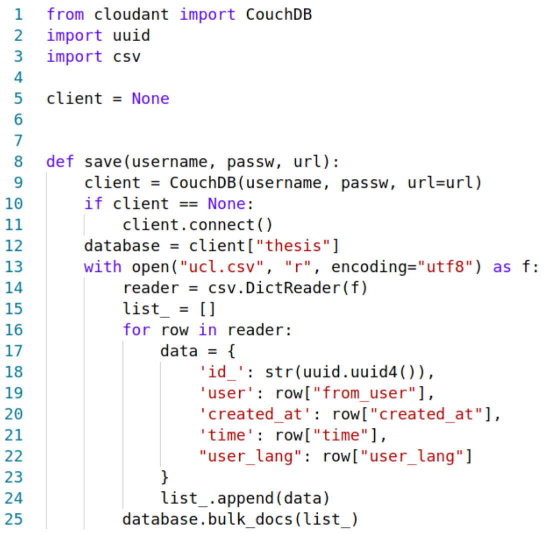


```python
import pymongo
from pymongo import InsertOne
import csv

client = None


def save(uri):
    if client == None:
        client = pymongo.MongoClient(uri)
    db = client.get_database("thesis")
    colection = db.get_collection("twt")
    with open("ucl.csv", "r", encoding="utf8") as f:
        reader = csv.DictReader(f)
        list_ = []
        for row in reader:
            list_.append(
                InsertOne(
                    {
                        'user': row["from_user"],
                        'created_at': row["created_at"],
                        'time': row["time"],
                        "user_lang": row["user_lang"]
                    }
                )
            )
        colection.bulk_write(list_)
```

**Fig. 4.** Azure Functions.

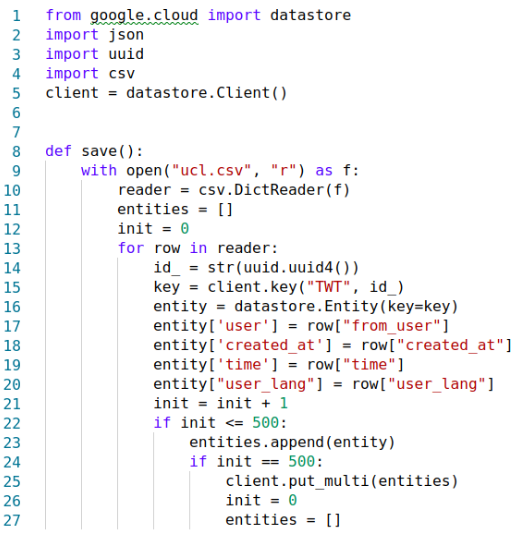


```python
from cloudant import CouchDB
import uuid
import csv

client = None


def save(username, passw, url):
    client = CouchDB(username, passw, url=url)
    if client == None:
        client.connect()
    database = client["thesis"]
    with open("ucl.csv", "r", encoding="utf8") as f:
        reader = csv.DictReader(f)
        list_ = []
        for row in reader:
            data = {
                'id_': str(uuid.uuid4()),
                'user': row["from_user"],
                'created_at': row["created_at"],
                'time': row["time"],
                "user_lang": row["user_lang"]
            }
            list_.append(data)
        database.bulk_docs(list_)
```

**Fig. 5.** IBM Cloud Functions.



```python
from google.cloud import datastore
import json
import uuid
import csv
client = datastore.Client()


def save():
    with open("ucl.csv", "r") as f:
        reader = csv.DictReader(f)
        entities = []
        init = 0
        for row in reader:
            id_ = str(uuid.uuid4())
            key = client.key("TWT", id_)
            entity = datastore.Entity(key=key)
            entity['user'] = row["from_user"]
            entity['created_at'] = row["created_at"]
            entity['time'] = row["time"]
            entity["user_lang"] = row["user_lang"]
            init = init + 1
            if init <= 500:
                entities.append(entity)
                if init == 500:
                    client.put_multi(entities)
                    init = 0
                    entities = []
```

**Fig. 6.** Google Cloud Functions.

## *B. Experimental Configurations*

The experiment scenario is designed to obtain runtime related metrics by performing 300 invocations of each function. The invocations of the CSPs are performed through multiprocess scheduling from a local device. To avoid processing overhead, a time delay is implemented between each of the 300 invocations regardless of the completion of function processing in the serverless runtime [8]. The total execution time for each function is obtained by subtracting the completion time from the processing start time in each serverless runtime.

***1) Experimental Environment.*** The experimentation environment consists of 3 components: (1) deployment and configuration of the functions on serverless platforms; (2) invocation towards the HTTP endpoints generated by the HTTP triggers for each function; and (3) analysis and monitoring tools to store and visualize the results of each execution in a Local and Cloud Environment (Fig. 7). In the Cloud Environment, the functions are hosted together with their main dependencies and in the Local Environment the invocations are performed and the results of the invocations are stored. This scheme also shows the interaction of the client with each platform, which starts with a request to invoke the function; after finishing the execution, the platform uses a response that contains the execution time of the function. These values are stored locally within InfluxDB and queried through the Grafana graphical interface. Within the Local Environment, the invocations are managed by a script that coordinates their simultaneous sending to the CSPs and the respective time delay established between each invocation.

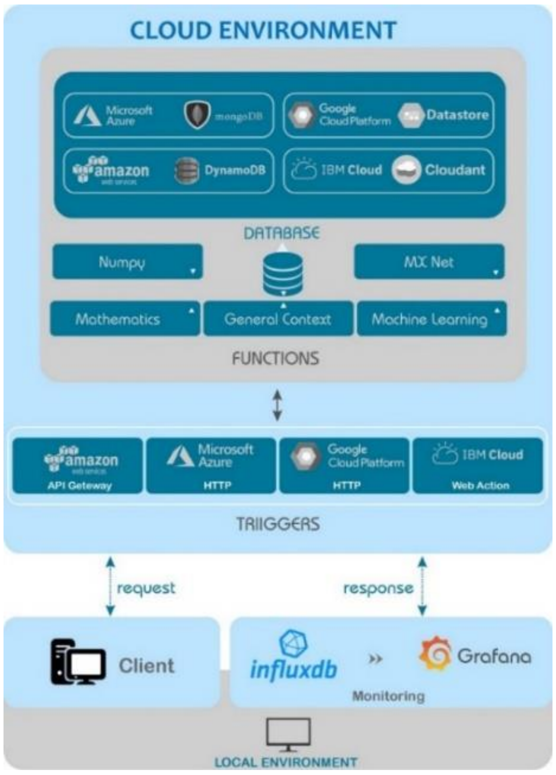


**Fig. 7.** Cloud Environment.

There are platform-specific characteristics that condition the execution of the functions (runtime, execution time, memory, etc.). Table 2 shows the values configured in the platforms, the Python runtime version, the maximum execution time of each function, the deployment regions used in each CSP, and the memory allocated to each function. From these, it is highlighted that in Azure Functions the allocation is dynamically performed up to 1536MB limits as the platform sees fit.

**Table 2.** Cloud Platforms Configuration.

| | AWS Lambda | Azure Cloud Functions | Google Cloud Functions | IBM Cloud Functions |
|---|---|---|---|---|
| Runtime | Python 3.7 | Python 3.6 | Python 3.7 | Python 3.7 |
| Memory | 3008 | Dynamic | 2048 | 2048 |
| Time | 15 | 10 | 9 | 10 |
| NoSQL Service | On Demand layer | 10.000 Rus | Default | Free Layer |
| Region | us-east-1 | us-east | us-central1 | Dallas |

Table 3 shows the number of invocations that the client performs on the platform. It also shows the size of the image files whose format is .jpg. These files are used for the Machine Learning function to perform object recognition on images. In addition, the size of the file with extension .CSV and the number of records it contains − which are processed in the general context function − are shown. Finally, the table also shows the size of the registers configured for the variable samples and that are required for the mathematical function to simulate a digital signal randomly generated by the function itself.

**Table 3.** Initial Conditions for Implementation.

| Function | Number of invocations | Samples (records) | File size |
|---|---|---|---|
| Mathematics | 300 | 20097152 | Not applicable- |
| Machine Learning | | 10,000 | 8349 kB |
| General Context | | Not applicable | 15.6 kB |

## III. Results

The results and metrics obtained by performing the experiments with 300 invocations for each function are shown below:

### *A. Mathematical Function*

The execution times of the mathematical function that obtains The Fast Fourier Transform (FFT) are presented in Fig. 8, where it is visualized that the AWS platform provides better performance than Azure Functions when performing 300 invocations of the function. The period spent is approximately 6 hours. The requests were performed simultaneously on each platform with a 60 seconds difference between each request.

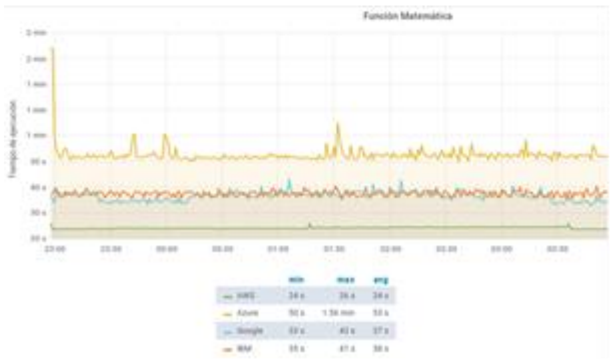


**Fig. 8.** Results of mathematical function execution.

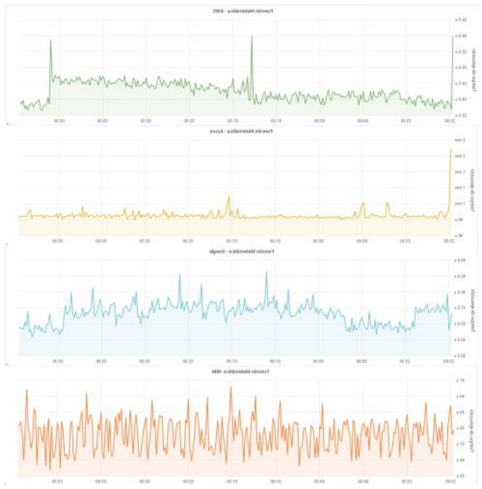


**Fig. 9.** Performance per platform when invoking the mathematical function.

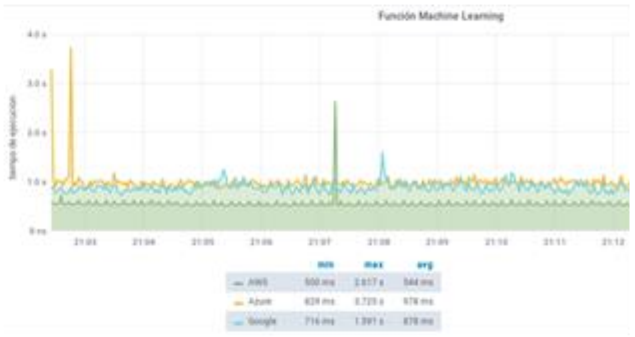
Fig. 9 shows the individual performance per platform according to the provider of service. For this purpose, the time invested to execute the mathematical function is considered; as part of the results, AWS Lambda presents a certain degree of homogeneous performance in contrast to the rest of the platform, since there is no evidence of variations in the performance peaks. However, if we visualize its performance individually, there is a constant variation in the time used to perform the 300 executions of the function. There is relative similarity with the times obtained using IBM Cloud Functions and Google Cloud Functions. However, the minimum times of these 2 platforms are above the maximum times achieved by AWS Lambda, accentuating the fact of presenting constant variations with respect to the execution times in the 3 platforms (AWS Lambda, IBM Cloud Functions and Google Cloud Functions). To establish a benchmark with respect to execution time, Table 4 shows the most representative times (minimum, maximum, and average) obtained on each platform. It can be seen that the longest execution time in AWS Lambda is well below the shortest execution times of the other platforms. The relative similarity in the average time spent in the execution of this function in the IBM and Google platforms is also evident; however, there is a 1 second difference, therefore, in Google there is better use and provisioning of computational resources. Finally, we can highlight the high consumption of time invested by Azure Functions, a platform that invests about 2 times more average time in the execution compared to the average obtained in AWS Lambda.

**Table 4.** Time Distribution Calculation.

| Platform | Minimum time | Maximum Time | Average |
|---|---|---|---|
| AWS Lambda | 24 s | 26 s | 24 s |
| Azure Functions | 50 s | 93.6 s | 53 s |
| Google Cloud Functions | 33 s | 43 s | 37 s |
| IBM Cloud Functions | 35 s | 41 s | 38 s |

### *B. Machine Learning Function*

The execution times of the Machine Learning function in AWS, Azure, and Google are presented in Fig. 10. It is important to note that in each one the request is made for the 3 platforms to simultaneously execute the function code with 1 second difference between each of the requests. Finally, it can be seen that AWS has better performance with respect to function execution.



| | min | max | avg |
|---|---|---|---|
| AWS | 500 ms | 2.617 s | 544 ms |
| Azure | 628 ms | 3.725 s | 976 ms |
| Google | 716 ms | 1.391 s | 878 ms |

**Fig. 10.** Machine Learning function execution times.

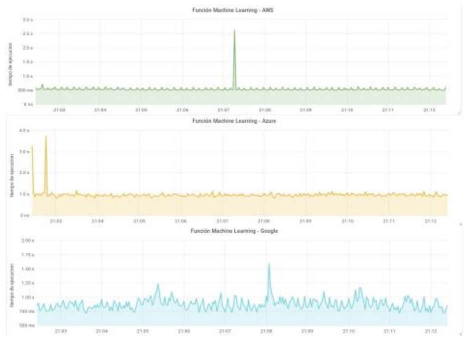


**Fig. 11.** Performance per platform when invoking Machine Learning function,

Fig. 11 shows the different performance values in terms of execution time (Y-axis) that have been employed by each platform at certain stretches of the request period (X-axis). It can be concluded that Azure Functions runtime spends more time compared to AWS Lambda and GoogleCloud Functions. Google presents marked fluctuations in the peaks with high and low performance in FaaS platform presented during the period of request, denoting an unstable and unpredictable behavior. However, it is also established that despite the behavior found in Google Cloud Functions, this platform improves the average performance obtained in Azure Functions but remains relatively far from that achieved by AWS Lambda. Table 5 presents, in summary, the most significant times obtained during the execution of this function. The platform provided by Google executes this function with lower time consumption compared to Azure Functions which needs a high time range. However, AWS Lambda still evidences the best performance by presenting less time to obtain the same results compared to Google and Azure platforms.

**Table 5.** Distribution of calculation times.

| Platform | Minimum time | Maximum Time | Average |
|---|---|---|---|
| AWS Lambda | 500 ms | 2.617 s | 544 ms |
| Azure Functions | 829 ms | 3.725 s | 978 ms |
| Google Cloud Functions | 716 ms | 1.591 s | 878 ms |

It is worth mentioning the implementation of this function has not been possible in the IBM Cloud Functions platform. In a first attempt we tried to implement it by creating an implementation package that exceeds the limits of the platform (maximum 48MB, obtained 53MB), in second instance we tried to implement it using Docker, however, the incompatibility of MXNet to run on Linux Alpine truncated the correct implementation of this function.

### *C. General Context Function*

Regarding the general context function, the results obtained from the execution times of the function in the 4 platforms are shown in Fig. 12, where it is evident that IBM provides the best performance.

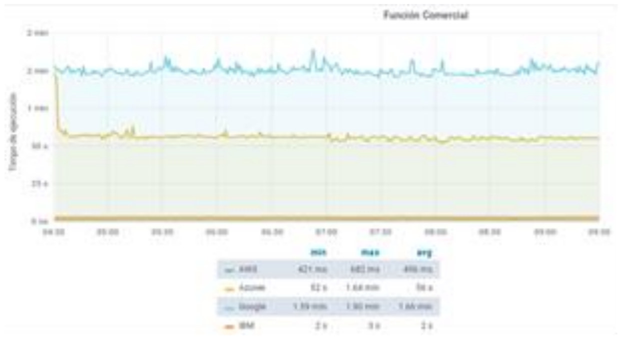


**Fig. 12.** General context function.

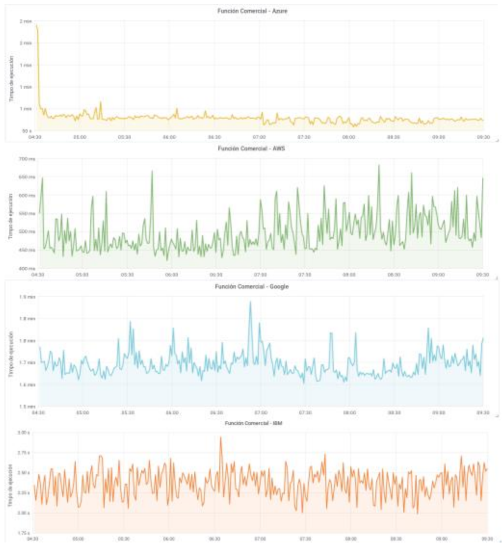


**Fig. 13.** Performance per platform when invoking the general function.

Fig. 13 shows the results of the execution of this function segmented by CSP. The unstable or unpredictable behavior of each platform can be observed. Despite this, better stability is observed when executing this function on the Microsoft Azure platform. Similarly, a marked difference is shown in the AWS and IBM platforms with those of Azure and Google, where although this function is subject to the operation of NoSQL services, the execution times obtained with IBM Cloud are lower than those of Azure Functions (optimized NoSQL service) and Google Cloud Functions. However, these times are high compared to those of AWS Lambda (optimized NoSQL service); notwithstanding, they can be considered acceptable bearing in mind that no modifications were made to IBM Cloudant. Another noteworthy aspect lies in the time invested by Google Cloud Functions where the fact of finding limitations concerning the size of the batches sent can become a variable that directly affects the optimal execution of the function. In addition, the low performance of Azure Functions when executing this function is visualized. Finally, despite the modifications made to its NoSQL service, the average execution time of this function is above those obtained by the rest of the platforms. Table 6 summarizes the most relevant times during the execution of the functions on each platform. This table shows that while in AWS Lambda the insertion of 10000 records is performed in less than half a second, Google Cloud Functions takes about 2 minutes to perform the same operation. In addition, the average time consumption of the executions in IBM Cloud Functions denotes better performance when compared to the times presented in Azure Functions, although they were executed on the free layer of IBM's NoSQL service.

**Table 6.** Distribution of calculation times.

| Platform | Minimum Time | Maximum Time | Average |
|---|---|---|---|
| AWS Lambda | 421 ms | 682 ms | 496 ms |
| Azure Functions | 52 s | 1.64 min | 56 s |
| Google Cloud Functions | 1.59 min | 1.90 min | 1.66 min |
| IBM Cloud Functions | 2 s | 3 s | 2 s |

## IV. Conclusions and Discussion

After performing the experiments with 300 invocations for each function on different platforms, it can be mentioned that AWS Lambda provides runtimes with higher stability that consume fewer resources, improving function execution times regardless of their context. Experimentation results also show instability on platforms such as Azure Functions with the Python runtime, which generates high execution times compared to the other platforms. It should be noted that the IBM service for NoSQL database (Cloudant) did not undergo modifications and it was possible to use the free layer, unlike the optimization performed on AWS, DynamoDB, and Azure CosmosDB-MongoDB API where performance can be improved by moving from the free layer to the standard layer. It is important to highlight the performance obtained by Google Cloud Functions when running AI context functions where, despite not having obtained the best execution time, it is close to AWS Lambda time. The light version of Linux (Linux Apline) implemented in IBM Cloud Functions containers (Apache Openwhisk) has incompatibility with some libraries, which ultimately prevents the use and exploitation of this platform.

As conclusions, serverless architectures have emerged as an alternative in terms of empowerment and innovation of new services. CSPs use different features, languages, and programming models within their FaaS platforms, which allow identifying variations in the availability of runtimes. For example, the addition of support for code written in Ruby and Go in AWS Lambda and Google, respectively.

In addition to status changes regarding the availability of Python in Azure Functions, or the extension of the memory allocation limit from 512MB to 2048 MB in IBM Cloud Functions. The different runtime options provided by FaaS platforms allow the selection of the programming language according to the needs of each component (function) of a serverless architecture, including the selection of libraries to satisfy code dependencies. At the programming language level, Python presents the best characteristics in terms of platform support, due to the existence of libraries that adapt to the limitations of these for the creation of experimentation functions. The results obtained when executing the experimentation functions have allowed comparing different behaviors on each platform, where AWS Lambda stands out because it spends less time executing the functions, unlike the rest of the analyzed platforms.

## AUTHOR'S CONTRIBUTION

**Armando Cabrera-Silva:** supervision, investigation, writing – original draft.
**José Carrillo-Verdún:** supervision, methodology, writing – review & editing.
**Patricio Martínez-Palacios:** experimental design, validation.
**Daniel-Alejandto Guamán-Coronel:** writing – original draft, validation.

## FUNDING

This article is the result of the research project "Construction of an Architectural Framework for the Management of Customer-Centric Digital Enterprises" developed jointly by the Department of Computer Science and Electronics of Universidad Técnica Particular de Loja - UTPL and the Faculty of Computer Science of the Universidad Politécnica de Madrid - UPM.

## REFERENCES

[1] M. Smith, P. R. Saunders, L. Lyons, *A Practical Guide to Microservices and containers*, 2018.

[2] W. Gottesheim, "Challenges, benefits and best practices of performance focused DevOps," in *Proceedings of the 4th International Workshop on Large-Scale Testing*, 2015, p. 3. https://doi.org/10.1145/2693182.2693187

[3] T. Kohlborn, A. Korthaus, T. Chan, M. Rosemann, "Identification and analysis of business and software services-a consolidated approach," *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 50–64,

2009. https://doi.org/10.1109/TSC.2009.6

[4] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, vol. 2nd. 2012.

[5] H. Lee, K. Satyam, G. Fox, "Evaluation of production serverless computing environments," in *IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 442–450. https://doi.org/10.1109/CLOUD.2018.00062

[6] J. Lewis, M. Fowler, "Microservices: a definition of this new architectural term," 2014. https://martinfowler.com/articles/microservices.html

[7] T. Back, V. Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," in *Lecture Notes in Computer Science*, Springer, 2018. https://doi.org/10.1007/978-3-319-99819-0_11

[8] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, P. Suter, "Serverless computing: Current trends and open problems," *Research Advances in Cloud Computing*, Springer, 2017, pp. 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

[9] D. Poccia, *AWS Lambda in Action: Event-driven serverless applications*. Manning Publications Co., 2016.

[10] V. Ishakian, V. Muthusamy, A. Slominski, "Serving Deep Learning Models in a Serverless Platform," in *IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 257–262.

[11] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," in Workshop on Machine Learning Systems, 2016. https://arxiv.org/abs/1512.01274

[12] B. Wu, F. Iandola, P. H. Jin, K. Keutzer, "Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2017, pp. 129–137. https://doi.org/10.1109/CVPRW.2017.60

[13] H. Lin, S. Jegelka, "Resnet with one-neuron hidden layers is a universal approximator," in *Advances in neural information processing systems*, 2018, pp. 6169–6178. https://doi.org/10.5555/3327345.3327515

[14] L. Wen, X. Li, L. Gao, "A transfer convolutional neural network for fault diagnosis based on ResNet-50," *Neural Computing and Applications*, vol. 32, pp. 6111–6124, 2020. https://doi.org/10.1007/s00521-019-04097-w

[15] K. Figiela, A. Gajek, A. Zima, B. Obrok, M. Malawski, "Performance evaluation of heterogeneous cloud functions," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 23, e4792. https://doi.org/10.1002/cpe.4792

[16] J. Manner, M. Endreß, T. Heckel, G. Wirtz, "Cold start influencing factors in function as a service," in *IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 181–188.