Serverless web application for risk identification and risk assessment in a financial asset

Víctor Bonilla Pardo
vb00293
vb00293@surrey.ac.uk

***Abstract*— This document describes the cloud system which provides a user the capability of identifying the average value at risk (VAR) for a determined trading strategy over an asset of choice. The system architecture and implementation are discussed. Also, an experiment is described in which the results of changing relevant parameters are showed. Finally, an analysis of the scalable services running cost and the satisfaction of the requirements are presented.**

Keywords—Cloud Computing, Serverless, trading, risk analysis.

I. Introduction

The cloud system presented in this document implement an on-demand self-service application to execute a trading strategy based in a simple moving average and further identify the average value at risk (VAR) for each trading signal over a selected financial asset, generating and storing so a risk analysis. A web application is published permitting broad network access to any consumer. Also, the system shows rapid computing elasticity by offering a user the capability of executing part of the VAR calculations in an on-demand cluster of EC2 instances to parallelize and speed up the overall execution.

Two points of view can be distinguished when describing the characteristics of the cloud system:

A. Developer

From the point of view of a developer, the system offers on-demand functionalities executed by a pool of resources making the system capable of scaling if demanded. All the resources usage is automatically measured by the provider to control and optimize the system.

The system is divided into two subsystems differing in the service model:

1. The back-end system developed under the *Function as a Service (FaaS)* model which completely abstracts the developer from managing servers or containers for each function [1].
2. The front-end system developed under the *Platform as a Service (PaaS)* model avoiding managing underlying infrastructure by defining a container with supported technologies by the provider [2].

The whole system was deployed under the *public cloud* model using a combination of providers [2]. This allows a developer to take advantage of the benefits from the best services from each provider: Amazon Web Services (AWS) was used for the back-end system and Google Cloud Platform (GCP) was used for the front-end system.

B. User

On the other side, from the user's perspective, the system offers an on-demand self-service application available over the network and accessed through a web browser. Also, it provides the user with the capability of elastically running part of the calculations in a cluster of parallel resources.

In addition, the user can review previously created risk analyses stored in the cloud system.

II. Final Architecture

A. Major System Components

The system is compound of five different components. For clarity, these are grouped into back-end and front-end components:

1) Back-end components

a) Amazon API Gateway

Amazon API Gateway enables the system to create, publish, maintain and monitor a scalable and stateless REST API [3]. This component serves two types of resources following the REST principles: assets and risk analyses. It provides the functionality to browse assets; and browse and create risk analyses. In concrete, the API publish four HTTP methods to access the functionalities mentioned.

b) AWS Lambda

AWS Lambda executes the code of the system without managing infrastructure [4]. It computes most of the functionalities exposed in the REST API. Specifically, seven lambda functions are implemented in the system.

c) Amazon Elastic Compute Cloud (Amazon EC2)

Amazon EC2 provides scalable computing capacity [5]. It enables the system to fit the requirements of the VAR functionality. If demanded, a disposable cluster of EC2 instances is launched to parallel a heavy VAR calculation. Amazon EC2 was chosen over Amazon ECS and Amazon EMR because of its quick development process in order to get the functionality deployed, including the software package executed in the instances. However, in this case, Amazon ECS or Amazon EMR would be more reliable services as a cluster of instances to parallel jobs because of the management of the services by the provider and the approach of running parallel tasks/jobs with containers/instances.

d) Amazon Simple Storage Service (Amazon S3)

Amazon S3 is a service to store and retrieve any amount of data at any time, from anywhere on the web [6]. It enables the system to store all the information needed for the correct execution of the functionalities. Four categories of data are stored for system usage: the financial assets data, generated risk analyses, the resources used by the EC2 cluster and its disposable

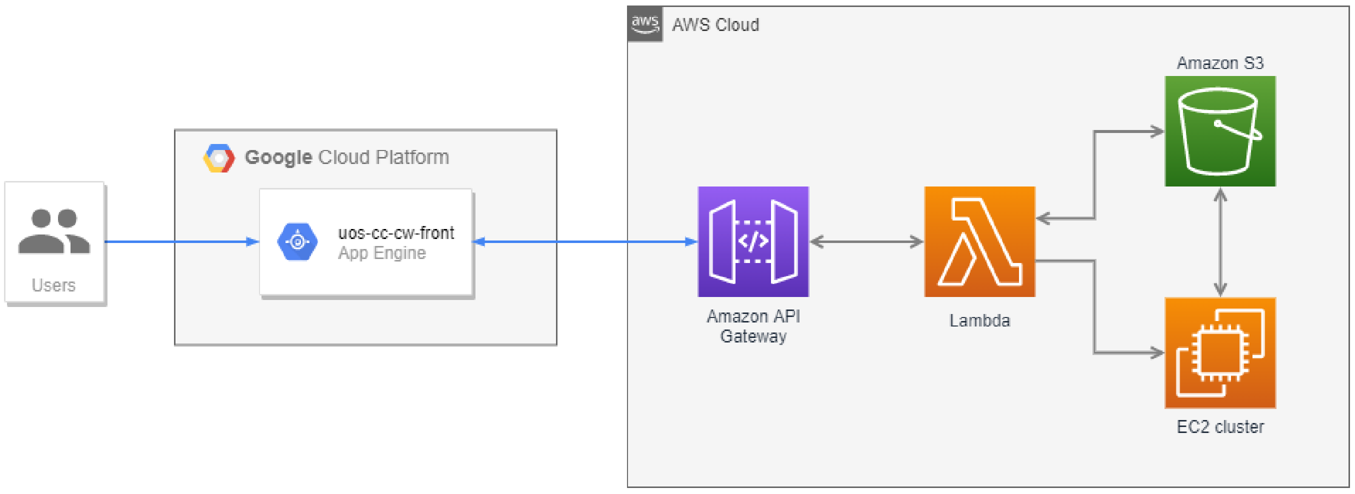


Figure 1. High-level diagram of the major components of the cloud system.

results. Amazon S3 was chosen over Amazon DynamoDB, Amazon RDS or Amazon Aurora because of its simplicity for accessing and storing data, no necessity for managing infrastructure, cheaper cost given the system usage and no requirements for advanced data storage functionalities.

2) Front-end components

a) Google App Engine (GAE)

GAE is a serverless platform for developing and hosting web applications at scale [7]. It provides the system with a platform for hosting the web application which interacts with the REST API allowing the user being capable of executing the functionalities discussed previously.

B. *System Component Interactions*

As previously mentioned, four types of data objects exist in the system storage: financial assets, EC2 software package, risk analyses (result of executing the trading strategy and calculation of VAR) and EC2 temporal results for each instance for VAR calculations. Two of them are required before the start of the system for its correct running, specifically at least one financial asset and the EC2 software package.

The interactions described are grouped by functionality. For clarity, communication of data/information *between major components* is indicated with labels in parenthesis and the seven Lambda functions are listed for context purposes:

- ListS3ObjectKeysWithPrefixFunction (A)
- BrowseAssetsFunction (B)
- BrowseRiskAnalysesFunction (C)
- BrowseRiskAnalysisFunction (D)
- CreateRiskAnalysisFunction (E)
- AnalyseRiskFunction (F)
- AssessRiskFunction (G)

1) Create a risk analysis fully calculated in Lambda

Before rendering the web page, the App Engine requests the financial assets available calling the proper HTTP GET method of the REST API in API Gateway (1-App Engine to API Gateway). This method without parameters, triggers the Lambda function *B* (2-API Gateway to Lambda) which triggers the function *A* to list the assets from Amazon S3 (3-S3 to Lambda). The list of assets is sent from *A* to *B* and from *B* through the API Gateway (4-Lambda to API Gateway) to App Engine in an HTTP response (5-API Gateway to App Engine).

Once the assets are listed to the user, this specify six parameters requested to create a risk analysis: selected asset, moving average period, VAR period, number of Monte Carlo samples for the VAR simulation, a choice of the scalable services to use for VAR (in this case, Lambda) and the number of resources to be used for VAR (in this case, one resource).

After submitting the html form, the parameters are collected by App Engine which send them using the proper HTTP POST method in the API (6-App Engine to API Gateway). The Lambda function *E* is executed with the parameters (7-API Gateway to Lambda). This function delegates the identification of trading signals and the calculation of VAR triggering two other Lambda functions:

The Lambda function *F* that identifies the trading signals and calculates each signal profit/loss is executed synchronously with the selected asset and specified moving average as parameters. In order to generate a new id for the risk analysis, function *F* invoke without parameters the function *C* which, in turn, invokes function *A* returning a list of available risk analyses to *C*, and in turn to *F* (8-S3 to Lambda). After the calculations and creation, function *F* stores the results to the system storage (9-Lambda to S3) and returns the id of the created risk analysis to function *E*.

The Lambda function *G* that calculates VAR is executed asynchronously with the newly risk analysis id, VAR period, Monte Carlo samples, the choice of scalable service and number of parallel resources. This invokes function *D* to retrieve the new risk analysis from the system storage (10-S3 to Lambda) and adds the VAR calculation results to the previously created risk analysis storing it by rewriting the object in Amazon S3 (11-Lambda to S3). While Lambda function *G* is running, the Lambda function *E* returns the new risk analysis id through API Gateway (12-Lambda to API Gateway) to App Engine (13-API Gateway to App Engine) in an HTTP response with the id in the *location* header.

Consequently, App Engine redirects the web browser to the newly created analysis URI with the returned id from function *E* as a path parameter.

2) Create a risk analysis partly calculated in parallel resources

The interactions in this functionality are the same to the ones described in the previous functionality except for the VAR calculations, so in this section, the interactions described take place when the Lambda function *E* invokes the function G with the newly risk analysis id, VAR period, Monte Carlo samples, the choice of scalable service and *R* number of parallel resources.

As previously, the function *G* invokes function *D* to retrieve the new risk analysis from the system storage (10-S3 to Lambda), but this time the function *G* only calculates part of the VAR calculation, specifically the mean and standard deviation of the probability distribution for the prices $[p_{t-v}, p_t]$ being p_t the price of the signal evaluated and v the VAR period. After computing it, the results are stored in the system storage (11-Lambda to S3) and function *G* launch an EC2 cluster of *R* number of EC2 instances with the location of the partially calculated VAR, *R* number of parallel resources and the number of Monte Carlo samples for each instance in the cluster (12-Lambda to EC2) before ending.

Each EC2 instance automatically set up the environment and retrieve from Amazon S3 the software package to complete the VAR calculation (13-S3 to EC2). The EC2 instance runs the software reading the uncomplete risk analysis from the system storage (14-S3 to EC2) and completing the VAR calculation.

A master instance is designated by design in the code to merge the parallel results. Therefore, the workers store their results temporarily in the system storage (15-EC2 to S3) and the master instance read them (16-S3 to EC2), merge them, store the final result and delete the temporal results from the system storage (15-EC2 to S3).

The following interactions are similar to the interaction 12 and onwards in the previous section.

3) Show the results of a risk analysis

A method in App Engine is fired with the risk analysis id as path parameter requesting the data to the back-end system. Therefore, App Engine sends a request containing the risk analysis id to the proper HTTP GET method in API Gateway (1-App Engine to API Gateway). This triggers the Lambda function *D* with the id as parameter (2-API Gateway to Lambda) which retrieves from the system storage the requested risk analysis (3-S3 to Lambda). Function *D* compute the summary metrics for the risk analysis and returns it to API Gateway (4-Lambda to API Gateway) and, in turn, to App Engine (5-API Gateway to App Engine).

4) Show the available risk analyses

Before rendering the page, the App Engine requests the risk analyses available calling the proper HTTP GET method without parameters of the REST API in API Gateway (1-App Engine to API Gateway). Following the same scheme of interactions when browsing the financial assets, the Lambda function *C* is triggered without parameters (2-API Gateway to Lambda) which triggers the function *A* to list the analyses from Amazon S3 (3-S3 to Lambda). The list of analyses is sent from *A* to *C* and from *C* through the API Gateway (4-Lambda to API Gateway) in an HTTP response to App Engine (5-API Gateway to App Engine).

III. Implementation

The public address of the web application is: https://uos-cc-cw-front.ew.r.appspot.com/

A. *Implementation process*

The overall cloud system was mainly implemented with Python. The implementation had two phases.

In the first iteration, the system was developed with back-end functionality for identifying trading signals and computing profit/loss; and with front-end functionality to list assets and risk analyses, create a risk analysis (without simulating VAR) and browse the details of a risk analysis.

In the second iteration, functionality for VAR calculation was added to the back-end system including scaling to parallelise the work if required. Also, functionality to present the VAR results was added to the front-end system.

The back-end system is hosted in the *us-east-1* Amazon region. It was developed with the *AWS Serverless Application Model* (AWS SAM) framework taking advantage of the capabilities of the *Infrastructure as Code* developing model [8]. The resulting template defines the API Gateway methods and Lambda functions allowing persistency of the components and deploying with a few commands. The Amazon S3 bucket was created and provisioned manually earlier.

To design the API Gateway methods, the Representational state transfer (REST) software architectural style was used resulting in two resources (assets and riskAnalyses) and four HTTP methods:

- /assets GET
- /riskAnalyses GET
- /riskAnalyses POST
- /riskAnalyses/{id} GET

Regarding the Lambda functions, dependencies such as *pandas*, *boto3* and *numpy* were used, as well as native Python libraries [9, 10, 11, 12]. To help developing the code, documentation from the AWS services and the dependencies, as well as Stack Overflow posts were used.

The most challenging part to implement was the VAR functionality. From this phase, the scaling algorithm to parallelise the VAR work followed the master-workers model for a cluster and used the system storage for data integrity when communicating the data/information between the major components of the architecture. To ensure the model, each instance in the cluster is assigned a generated id: the instance with id 0 is the master. All the instances *including the master* compute a parallel result. The additional functionality of the master is to wait for all the parallel results, merge and delete them, and store the final result. An example of the workflow and interactions is described in the section II. *Final Architecture*, subsection B. *System Component Interactions*, subsection 2) *Create a risk analysis partly executed in parallel resources*.

The front-end system was developed reusing code from lab 1 and lab 3 [13, 14]. Dependencies such as Flask, gunicorn, pandas, matplotlib and native Python libraries were used [15, 16]. To help developing the code, documentation from the dependencies and Stack Overflow posts were used. Also, the Jinja documentation and the HTML and CSS posts from w3schools.com were used to develop the HTML templates and CSS style [17, 18, 19].

The web app presents three paths with four different HTTP methods:

- / GET
- /analyses GET
- /analyses POST
- /analyses/{id} GET

Also, it handles two errors:

- When a non existing path is called, the web app renders an error template.
- When a HTTP 500 error happen, the web app returns a message.

B. *Tests against the code*

Tests can be divided in two groups:

1) Back-end tests

The tests executed weren't programmatic. Some of the manual tests were carried out locally (running a mock environment) and others in the AWS console to ensure the correct functionality of the lambda functions and the API Gateway methods. For both, AWS Lambda and API Gateway, the tests were executed with JSON events as argument. Also, some of the API Gateway tests were executed with an HTTP client.

2) Front-end tests

Also, the tests executed weren't programmatic. The tests were carried out manually browsing the web app running both locally and deployed in GAE.

IV. Results

The following experiment is executed to observe the effects of changing levels from two considered factors: moving average period (MA) and VAR simulations. The data used is *Bitcoin*. The levels considered for MA are [10, 50, 100] (in days) and for VAR simulations are [10,000; 100,000; 1,000,000] (integer).

Comparing the results in Table I, the *Profit/Loss* doesn't change if the MA factor remains the same. The biggest profit is achieved when MA equals *10* with 17666799.16 units. Regarding VAR simulations for MA *10*, it can be determined that VAR at 95% and 99% tend to 3164 and 3163 respectively since results from bigger simulations are more precise [20].

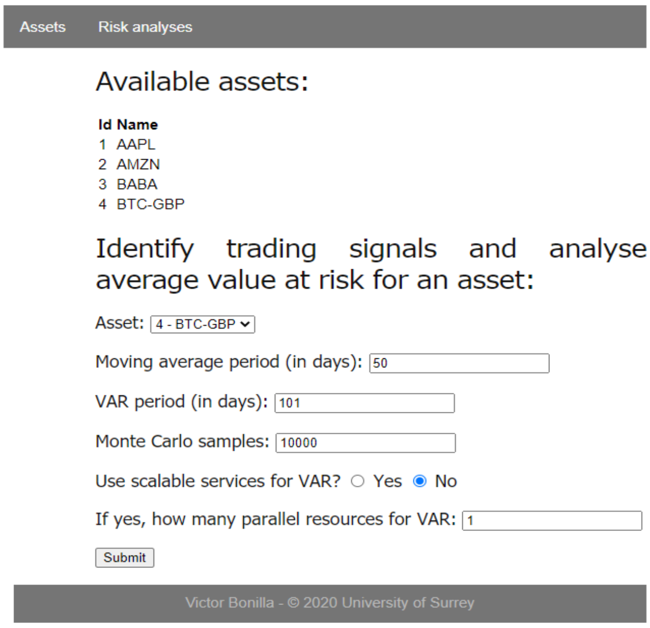
It can be observed a decrease in *Profit/Loss* when increasing the MA period meaning that shorter MA periods work better for this trading strategy.

In conclusion, shorter MA periods with large VAR simulations work better for this trading strategy.

TABLE I. Results for *Bitcoin*

MA	*VAR*	*Profit/Loss*	*95%*	*99%*
10	10,000	17666799.16	3165.33	3164.35
10	100,000	17666799.16	3164.59	3163.35
10	1,000,000	17666799.16	3164.71	3163.17
50	10,000	13249204.28	3262.09	3259.69
50	100,000	13249204.28	3262.23	3260.36
50	1,000,000	13249204.28	3262.31	3260.23
100	10,000	11431735.63	2917.44	2911.34
100	100,000	11431735.63	2917.25	2913.59
100	1,000,000	11431735.63	2917.21	2913.85

Two screenshots are presented (1) showing the user interface with the inputs for one of the tests, and (2) the results for that test.



Available assets:

Id	Name
1	AAPL
2	AMZN
3	BABA
4	BTC-GBP

Identify trading signals and analyse average value at risk for an asset:

Asset: 4 - BTC-GBP

Moving average period (in days): 50

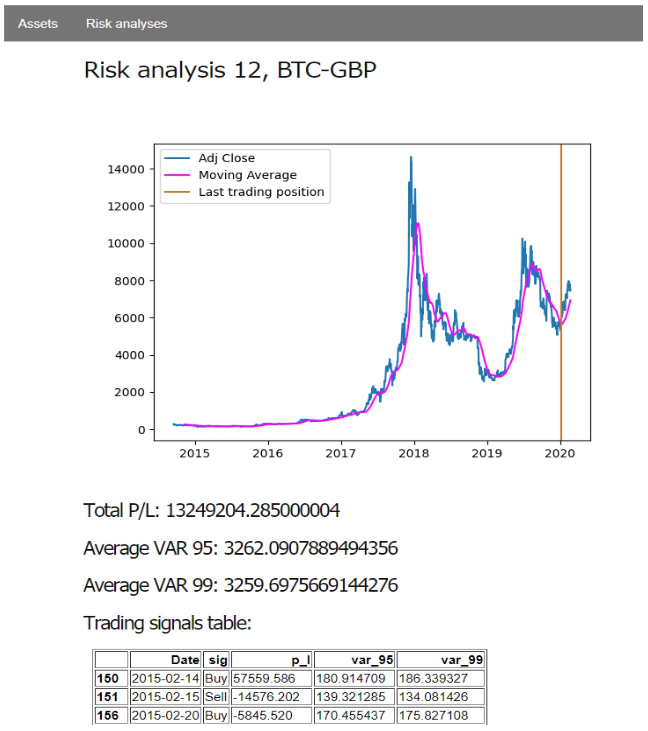
VAR period (in days): 101

Monte Carlo samples: 10000

Use scalable services for VAR? Yes [✓ No]

If yes, how many parallel resources for VAR: 1

Figure 2. User interface with inputs for test [50, 10,000].



Risk analysis 12, BTC-GBP

Total P/L: 13249204.285000004

Average VAR 95: 3262.0907889494356

Average VAR 99: 3259.6975669144276

Trading signals table:

	Date	sig	p_l	var_95	var_99
150	2015-02-14	Buy	57559.586	180.914709	186.339327
151	2015-02-15	Sell	-14576.202	139.321285	134.081426
156	2015-02-20	Buy	-5845.520	170.455437	175.827108

Figure 3. Results of test [50, 10,000] from the experiment.

V. Satisfaction of Requirements

TABLE II. Requirements

#	*C*	*Description*
i.	M	As previously described, GAE, AWS Lambda and Amazon EC2 are used in the system.
ii.	M	As previously described, a front-end system presents a form to create an analysis and web pages for the results.
iii.	P	The chart only shows a vertical line indicating the last trading signal of the strategy.
iv.	M	The data of a risk analysis is presented in the corresponding web page.
v.	M	The VAR code is run in a combination of AWS Lambda and Amazon EC2 (if requested). The choice of Amazon EC2 was discussed in a previous section.
vi.	M	As previously described, the EC2 resources are launched from AWS Lambda and switched off automatically.
vii.	M	As previously described, the web application presents a form to create a risk analysis with all the required parameters.
viii.	M	The web application allows the user to run an analysis with a single click.
ix.	M	As previously described, the analyses results are stored in Amazon S3.
x.	M	As previously described, the data provided 'live' in Amazon S3.

VI. Costs

The costs showed in Table III are calculated out of the *Free-tier discounts.* It is calculated for the scalable services using run timing information and AWS pricing for *us-east-1*.

The lambda cost is determined by the duration of the function and the GB/s of memory used, plus a fee per call [21].

The EC2 cost is determined by the run timing and the amount of data in and out from the instance [22]. However, the data transfers are small and made to Amazon S3 into the same region, so no cost is added.

TABLE III. Cost results

MA	*VAR*	*Lambda Time Billed (ms)*	*EC2 Time Billed (s)*	*Cost*
10	10,000	95400	0	0.356531425
10	100,000	194400	0	0.76973905
10	1,000,000	61800	115	0.2204861472
50	10,000	69600	0	0.252760675
50	100,000	158800	0	0.625335175
50	1,000,000	57200	137	0.2013908278
100	10,000	82300	0	0.3058963
100	100,000	171200	0	0.6759703
100	1,000,000	59200	128	0.2106517028

In conclusion, it is clearly seen in every MA group that running analyses with parallel resources is significantly cheaper than only using AWS Lambda, even for small VAR simulations.

References

[1] L. Gillam, "FaaS (PaaS) overview", Lecture Week 4/5, COMM034-Cloud Computing, 2019-2020, University of Surrey.

[2] P. Mell, T. Grance (2011), "The NIST Definition of Cloud Computing" (U.S. Department of Commerce, Washington, D.C.), NIST Special Publication 800-145 (SP 800-145).

[3] Amazon Web Services, Inc., "What is Amazon API Gateway?", Developer Guide, Amazon API Gateway, AWS Documentation, 2020.

[4] Amazon Web Services, Inc., "What is AWS Lambda?", Developer Guide, AWS Lambda, AWS Documentation, 2020.

[5] Amazon Web Services, Inc., "What is Amazon EC2?", Developer Guide, Amazon EC2, AWS Documentation, 2020.

[6] Amazon Web Services, Inc., "Getting started with Amazon Simple Storage Service", Developer Guide, Amazon Simple Storage Service (S3), AWS Documentation, 2020.

[7] Google Commerce Limitet, "Google App Engine Documentation", Documentation, App Engine, Serverless computing, Google Cloud, 2020.

[8] Amazon Web Services, Inc., "What Is the AWS Serverless Application Model (AWS SAM)?", Developer Guide, AWS Serverless Application Model, AWS Documentation, 2020.

[9] The pandas development team, "pandas documentation", Documentation, pandas, 2020.

[10] Amazon Web Services, Inc., "Boto3 documentation", Boto3 Docs 1.13.16, 2020.

[11] The SciPy community, "NumPy v1.18 Manual", Documentation, NumPy, 2019.

[12] Python Software Foundation, "The Python Standard Library", Documentation, 3.7, Python, 2020.

[13] COMM034 teaching team, "Google App Engine (using Python)", Lab Session 1, COMM034-Cloud Computing, 2019-2020, University of Surrey.

[14] COMM034 teaching team, "AWS Lambda: 'Function as a Service'", Lab Session 3, COMM034-Cloud Computing, 2019-2020, University of Surrey.

[15] Pallets, "User's Guide", Flask documentation, 1.1.x, Flask project, 2010.

[16] J. Hunter, D. Dale, E. Firing, M. Droettboom and the Matplotlib development team, "Documentation", 3.1.2, matplotlib, 2018.

[17] Pallets, "Jinja", Jinja, 2.11.x, Jinja project, 2007.

[18] Refsnes Data, "HTML Tutorial", HTML, w3schools.com, 2020.

[19] Refsnes Data, "CSS Tutorial", CSS, w3schools.com, 2020.

[20] Various authors, "Law of large numbers", Wikipedia, Wikimedia Foundation, Inc., 2020.

[21] Amazon Web Services, Inc., "AWS Lambda Pricing", Pricing, AWS Lambda, AWS, 2020.

[22] Amazon Web Services, Inc., "Amazon EC2 Pricing", Pricing, Amazon EC2, AWS, 2020.