[[1]](#footnote-1)

Cloud Instance Automation

Chris Ballenger, Blaine Brewer , Tom Gianelle, Venkat Kasarla

*Abstract*— Comparative analysis of Cloud-based methods for automating end-to-end process to train and deploy machine learning models. Our design was derived from “The Resource Allocation Optimization Problem for Cloud Computing Environments.” The researcher Victor Yim, spent countless hours manually building Cloud instances for his report. We wanted to see what it takes to automate that process.

# INTRODUCTION

O

UR goals were to simplify the process by automating many of the steps for configuring machine learning development environments and provide a simple framework for comparing cloud provider costs.

In “The Resource Allocation Optimization Problem for Cloud Computing Environments”, a research paper that explores the process of walking the tightrope of cost versus performance while deploying a cloud model, the researcher’s main challenge was creating each compute instance, setting up environments, and training/testing models to obtain results. The researcher then needed to determine which cloud provider to select, based upon ease of deployment, maintenance cost and operating expense. Whether you are working within the constraints of this research problem or you are an analyst trying to make business critical decisions for your company with limited time, the nuances and complexity involved in productionizing a learning model can add unnecessary time to your deployment. A more ideal strategy would employ automation to productionize a learning model so the analyst can concentrate on more important things that add value to the research or business.

Below is an overview of the initial two main objectives (automation and cost comparison) documented in this paper:

1. Automation of machine learning (ML) end-to-end workflow. The automation of creating an ML environment supports everything from training a model to the deployment of a web service. Figure 1 shows the overall structure of the design from a high level. We provided automation for creating both a dedicated compute instance and serverless method for model deployment. We believe by automating this design we can provide a reusable deployment base to quickly build, train and deploy our machine learning models.

A screenshot of a cell phone

Description automatically generated

Figure 1

1. For our research project, we also wanted to explore options available to instantaneously fetch up-to-date pricing of the relevant resources we used in our development. In a production environment, this would allow the user to make a cost-conscious decision on which platform and service they should use to deploy their model. Both Google Cloud and AWS offer resource pricing data to their customers via a RESTful API, where the response is delivered in a convenient JSON format. In order to make the data more readable for potential users, we developed an R process to query the data, reformat the response into a columnar format, and push that data to an Azure SQL database for future use.

# Cloud Comparison

In order to learn how to make Cloud Computing work for us, as Data Scientists, we explored what it takes to train and deploy within the cloud. For our project, we have selected Amazon and Google’s IaaS and SaaS platforms, which includes server-based and serverless configurations. Figure 2 breaks down the resources used in each configuration.

A screenshot of a cell phone

Description automatically generated

Figure 1

The only requirement we set for ourselves is the ability to automate the entire process by leveraging each services API. Amazon has an extensive SDK library for Python called boto3 and a SageMaker library for our serverless implementation. Google Cloud’s API is accessed through REST requests, which proved to be less intuitive than the GCP command line tool that Google offers. We chose to leverage the command line tool to automate our process. Although not ideal for a true production application, our development was done in Jupyter Notebook, where python executes each shell request, in order to better demonstrate the steps of our automation.

Our GCP and EC2 solutions provided the most flexibility, with lesser cost. We were able to create an instance and mount our cloud storage bucket using fuse (s3fs-fuse or gcs-fuse). This configuration would give the analyst the ability to terminate their compute instance at will without losing critical data. Comparing the GCP and AWS compute options, some additional setup was required in the Google platform, as GCP did not have any images that came preinstalled with any Machine Learning stack, like Conda or MiniConda. The gap between the two offerings was easily overcome with the use of startup scripts, which both providers allow you to utilize. We were able to install all required packages and mount our cloud storage bucket all using a startup bash script. The startup script solution ensures nearly identical starting points, despite leveraging two different providers with two different operating systems. One hurdle we could not overcome was being able to install the required python packages during EC2’s startup process. Our resolution was to execute a secondary package-installation-script after the EC2 instance was initialized.

To showcase the versatility of Amazon’s cloud offerings, we decided to build similar processes using their machine learning service. We selected SageMaker for its out of the box solutions to train and deploy without needing to configure your own server. Leveraging SageMaker to perform these tasks can be done from your own personal computer and through AWS’s SageMaker Jupyter Lab notebook. There is no benefit from using your personal computer or AWS’s notebook, as all the processing is performed in the cloud. SageMaker comes prepackaged with basic and advanced Machine Learning libraries, for example: Tensorflow, MXNet, and Linear Regression. Sagemaker also gave us the ability to BYO model by creating custom containers using docker. The benefit of using docker containers is that you are able to prepackage an validated environment for your model to run. Although this configuration may be easier to use when starting from scratch, this was not ideal in the development of our process because of the added cost of using SageMaker. Still, for ground up development, it is reasonable to conclude that an analyst could produce a production ready model, much quicker, without the added headache of configuring the environment.

In addition to the server-based platform, we also decided to use explore both GCP and AWS’ serverless options for our deployment. Although, there are many ways to leverage AWS’ Lambda Functions. For our EC2 design we chose a WSGI compatible web service and incorporated SageMaker’s existing functions. If you decide to use SageMaker, you will find a more conventional development platform to write your Lambda Function. In our design of the EC2, we had to save the model using the python function joblib and reference it through a Flask micro web server. Fortunately, a streamlined solution was developed to automate this deployment using Zappa. Zappa seamlessly displaced both Lambda and the Gateway API with the provided command line tool. The only thing we needed to provide was a lightweight Flask application.

For our Google solution, we utilized the cloud function as our serverless option for deployments. Cloud function triggered upon user’s HTTP request and returns the response. Our development and deployment workflow are automated with scripting and gcloud command line tools. we used packages as google-cloud-storage, gcsfs, flask in cloud function. Google Cloud Function works properly with the users who want simplicity in the tasks and operations.

# Build to Deploy Automation Solution

No matter the deployment method, our primary approach was to automate the creation of training and deployment of machine learning model using the very popular iris dataset (https://archive.ics.uci.edu/ml/datasets/iris). The method and data are of secondary importance though, as the focus of this project was the methodology to build and deploy that model. In the following sections we will take a in-depth look into the individual processes with some code snippets to highlight the steps taken. For both cloud providers, in order to replicate our steps, it is first necessary to setup a cloud account using the respective cloud consoles. In order for the following steps to work, project creation, API access, and security access is assumed.

In order to show the differences between each technology, we will break up the sections into 4 parts

1. Creating and pushing files to cloud storage
2. Configuring and initializing our instance
3. Training the model
4. Deployment

**Pre-Setup**

In order to setup your google cloud API or AWS API, you need to initialize the CLI console.  This can be done by running the following commands:

*AWS*: aws configure

*Google:* gcloud init

**AWS - EC2**

Each block of code will be used importing the following packages

import boto3

Import os

session = boto3.session.Session()

s3\_resource = boto3.resource("s3")

ec2\_resource = boto3.resource("ec2")

ec2\_client = boto3.client('ec2')

Create the S3 Storage

bucket\_name = ‘iris-train’

bucket\_response = s3\_connection.create\_bucket(

            Bucket=bucket\_name)

In order to upload files, we found the best option was to use the AWS CLI sync feature.  This will upload all files within a folder to the S3 bucket. We will set all our files private by using the command --acl private.

os.popen('aws s3 sync '+path\_to\_src+' '+path\_to\_s3+' --exclude "model/\*" --acl private').read()

The files contained in our storage bucket will be the python script to train and save our model, the dataset we would like to train on, and a list of required libraries conda will install.

Now that we have our files uploaded into the bucket, we will create our EC2 instance using a “user data” bash script that will be executed at start-up.  Our bash script will include code to update the yum repository, install and download fuse for s3, and mount our new bucket to the EC2 instance.  The following script was adapted from an example found here: <https://cloudkul.com/blog/mounting-s3-bucket-linux-ec2-instance/>

user\_data = [

**"#cloud-boothook",**

**"#!/bin/bash",**

        "yum update -q -y",

        "yum install automake fuse fuse-devel gcc-c++ git libcurl-devel libxml2-devel make openssl-devel -q -y",

# Additional Mounting Steps

    "mkdir -p /mys3bucket",

    "chown ec2-user:ec2-user /mys3bucket",

    "s3fs " + bucket\_name + " -o use\_cache=/tmp -o uid=500 -o mp\_umask=002 -o multireq\_max=5 -o allow\_other /mys3bucket",

 ]

Your bash script must start with the first 2 lines, #cloud-boothook and #/bin/bash and the script will be executed from root so some considerations may need to be made around user access permissions.

Once our startup script has finalized, our next step is to initialize our EC2 instance.  You will need to provide the name of your PEM key, and your security group id. We recommend creating an SSH specific security group.  The below example uses the Miniconda image and t2.medium size machine.

instances = ec2\_resource.create\_instances(

    TagSpecifications=[{  'ResourceType': 'instance', 'Tags': [ {'Key': 'Name', 'Value': ‘iris-ec2’}] }],

    ImageId= ‘ami-062c42cbecc1d5ec0’ #miniconda,

    MinCount=1,

    MaxCount=1,

    InstanceType=”t2.medium”,

    KeyName=‘PROVIDE YOUR KEY.pem’,

    SecurityGroupIds=security\_group, #Bring your own!

    UserData=user\_data

)

After the instance is created you will need to execute the below command, which will install the required python packages.

for requirement in `cat /mys3bucket/requirements.txt` ; do  conda install --yes ${requirement} ; done

After the necessary packages have been installed, we can now train our model by executing our python script, seen below.

cd mys3bucket/; python train.py

Now that the python script has finished executing, a python pickle file will automatically be saved to your S3 drive. The pickle file is the model artifact that can be used with other data to produce predictions.  The EC2 instance can now be terminated.

Although the preceding steps seem daunting, in reality, the SDK abstracts the complexity of building a machine learning model in the cloud.  We did have trouble installing the required packages through the startup scripts which forced us to connect to the server as a stopgap.  If we were to continue development of our automation solution, we would devote more time to solving the package installation problem, which would eliminate the gap in automation.

Finally, we deployed our newly trained iris dataset model.  Our model does require additional python development using a Flask web application. We chose Flask for its WSGI compatibility with Lambda, which allowed Lambda and our API Gateway to connect seamlessly. Assuming a Flask application has been built and the correct S3 bucket referenced, the following code will load the model response to the API.

response = S3.get\_object(Bucket=BUCKET\_NAME, Key=key)

model\_str = response['Body'].read()

model = joblib.load(BytesIO(model\_str))

For our deployment, we will use Zappa.io, which can be installed through python’s pip command.  It is recommended to install this package to the Lambda virtual environment.

From your project folder:

virtualenv lambda

source lambda/bin/activate

pip install flask zappa sklearn numpy scipy

At this point we have a new folder named lambda within the project/folder location.  This will include all the required packages lambda needs to run the application. The next step is to initialize the zappa workflow by using the following steps:

From your console:

1. Initialize: zappa init
   1. It is recommended to use default values
2. Deploy: zappa deploy dev
3. Test Model: curl -d '{"data":[[4, 2.1,1,0.4], [6.2, 1,3,2]]}' -X POST http:\\YOUR-AWS-URL

Using the steps defined above, a user is able to customize the configuration of their deployment environment to suit their needs.

**AWS SageMaker**

At this time, AWS does not offer a command line interface (CLI) which can be used to create a SageMaker instance. We utilized the AWS SageMaker built-in Linear Learner function and SageMaker’s Elastic Container Repository for custom prediction model.

The initial step to configure and create the storage bucket is identical to the step taken in the previous section. Creating an S3 bucket for input and output storage are prerequisites to run the SageMaker code referenced below.

An important note regarding setting SageMaker trainer parameters: When using a small dataset where *mini\_batch\_size* needs to be set, *feature\_dim* must be equal to the number of feature variables. Also, *role* needs to be an IAM role that has explicit access to run SageMaker and access to S3. Additionally, the *container* can be either a built-in AWS SageMaker container or an acceptable customer container (e.g. ECR Docker containers) for running machine learning training code.

# Set the training model parameters

linear = SageMaker.estimator.Estimator(container,

role,

train\_instance\_count=1, #arg

train\_instance\_type='ml.c4.xlarge', #arg

output\_path=output\_location,

SageMaker\_session=sess)

linear.set\_hyperparameters(feature\_dim=4,

predictor\_type='regressor',

mini\_batch\_size=50,

normalize\_data=False)

#Train

linear.fit({'train': s3\_train\_data})

Once training and any testing of model is complete, the final step is to deploy the model and make it an accessible on-demand SageMaker Endpoint.

# Create Model Endpoint

linear\_predictor = linear.deploy(initial\_instance\_count=1,

instance\_type='ml.m4.xlarge')

In order to finalize the deployment, we created a custom Lambda function that invokes the SageMaker endpoint. For SageMaker deployment we used the serverless.com utility, provides a convenient CLI used to create the AWS API and Lambda function for invoking SageMaker endpoint.

# Install the serverless cli

npm install -g serverless

cd project\folder

virtualenv lambda

source lambda/bin/activate

The following script is used to execute the Lambda Function:

# Invoke endpoint

runtime= boto3.client('runtime.SageMaker')

def predictHandler(event, context):

response = runtime.invoke\_endpoint(EndpointName=ENDPOINT\_NAME,

ContentType='text/csv',

Body=payload)

result = json.loads(response['Body'].read().decode())

As mentioned in the AWS documentation, SageMaker provides a powerful platform for building, training, and deploying machine learning models into a production environment on AWS. By combining this powerful platform with the serverless capabilities of Amazon’s Simple Storage Service (S3), Amazon API Gateway, and AWS Lambda, it is possible to transform a SageMaker endpoint into a web application that accepts new input data, potentially from a variety of sources, and presents the resulting output to an end user.

**Google Compute**

In order to automate the Google Cloud Platform (GCP) process, we leveraged the gcloud command line tool. For demonstration purposes, the following functions were executed with python in Jupyter Notebook, but a user interface could easily be constructed where push-button control could drive the model deployment.

Our first step, after installation of the Google Cloud Shell, is to access our operating system commands with the following script:

import os

So we can generalize the locations of our files, we will create a variable with the location of our local python script and csv files.

file\_source = 'c:/Users/blaine.brewer/Documents/Python/CloudComputingBCTV/'

Our next step is to create a Google Cloud bucket, where we will store our model data. In the following command, the p variable allows you to specify which project to associate the bucket with and the c and b variables allow us to specify the default class of Standard and the default bucket policy of off.

os.system('gsutil mb -p cloudcomputingbctv -c standard -l us-central1 -b off gs://bctv-storage')

Now that the bucket has been created, we need to upload our python script , training csv, and testing csv data. Using the following commands, we can load the 3 separate files from our local machine to our bucket in the cloud.

os.system('gsutil cp ' + file\_source + 'model.py gs://bctv-storage')

os.system('gsutil cp ' + file\_source + 'iris\_train.csv gs://bctv-storage')

os.system('gsutil cp ' + file\_source + 'iris\_test.csv gs://bctv-storage')

Now that the bucket contains the necessary files, we need to create our Google Cloud compute instance in order to run the model. Although there are several options to push our files to the instance, to allow for a fully automated process, we will use a startup script that will execute when the instance is initialized. The startup script will be tasked with creating a folder on the instance where we will mount the bucket and store the output, update apt-get package manager, install the GCSFuse package, and execute the mount using the GCSFuse utility. The startup script will be stored on our local machine as bctv-startup-script.sh and that script can be seen below.

sudo mkdir /tmp/bucket

sudo mkdir /tmp/output

apt-get update

export GCSFUSE\_REPO=gcsfuse-`lsb\_release -c -s`

echo "deb http://packages.cloud.google.com/apt $GCSFUSE\_REPO main" | tee /etc/apt/sources.list.d/gcsfuse.list

curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

apt-get update

apt-get install gcsfuse -y

gcsfuse -o nonempty bctv-storage /tmp/bucket

Once we have the files organized and our startup script ready, we can initialize the google compute instance with the following command. In this command, we make reference to the zone where the compute instance will be created, the project that the compute instance will be associated with and the bash command that will be executed at startup.

os.system('gcloud compute instances create bctv-compute --zone us-central1-a --project cloudcomputingbctv --metadata-from-file startup-script=' + file\_source + 'bctv-startup-script.sh')

Now that the instance has been created and the bucket mount has been performed, our next step is to run our python script, which will create our model, programmatically. We will perform this step by sending an ssh command to our instance specifying to run our python file.

os.system('gcloud compute ssh bctv-compute --project cloudcomputingbctv --zone us-central1-a --command=\"sudo python3 /tmp/bucket/model.py\"')

In our python script, we specify where the prediction and performance output should be saved so we can download the output from the compute instance with the following commands.

os.system('gcloud compute scp bctv-compute:/tmp/output/predictions.csv ' + file\_source + 'predictions.csv --zone us-central1-a --project cloudcomputingbctv')

os.system('gcloud compute scp bctv-compute:/tmp/output/prediction\_accuracy.txt ' + file\_source + 'prediction\_accuracy.txt --zone us-central1-a --project cloudcomputingbctv')

Utilizing Cloud Functions and the flask framework, users can get the prediction by posting in the request at the cloud function url.

%%bash

BILLING\_ACCOUNT=$(gcloud beta billing accounts list --format 'value(name)')

echo "${BILLING\_ACCOUNT}"

PROJECT\_ID="cloudcomputingbctv-259223"

echo "${PROJECT\_ID}"

gcloud beta billing projects link $PROJECT\_ID --billing-account $BILLING\_ACCOUNT

gcloud services enable cloudfunctions.googleapis.com

GCF\_NAME="storagefn"

GCF\_ENTRY="get-pred"

GCF\_REGION="us-central1"

gcloud functions deploy $GCF\_NAME --entry-point $GCF\_ENTRY --region $GCF\_REGION --runtime python37 --trigger-http

Calling the Cloud Function Url:

curl -X POST https://us-central1-cloudcomputingbctv-259223.cloudfunctions.net/storagefn -H 'Content-Type: application/json' -d '{"data":[[4, 2.1,1,0.4], [6.2, 1,3,2]]}'

It is evident from the preceding configuration how a seamless integration of a model deployment could be developed on the Google Cloud platform. The method described in this section give the developer complete control of the configuration but there are still quite a few steps to take and one weakness we discovered was the lag time of the compute initialization. The latency at startup creates a wrinkle in our automation because the user or application would need to accurately guess the time needed to execute initialization before attempting to download the model artifacts. This is one reason why using a serverless approach might be advantageous in an integrated automation, like the one we are attempting.

**Google Cloud Functions (Serverless)**

The simplest attempt at our automation development was achieved with the Google Cloud Functions resource. Similarly to the Google Compute development, we first needed to create a bucket to store our model and input files. In this case however, it was required to write the python script as a function and name that python script main.py so that Google Function would recognize the file. Once we made this change, we could execute the following command to create our function where we specify an event which executes the function. In our case, changes to the bctv-storage bucket would trigger the execution of the function.

os.system('gcloud functions deploy run\_iris --runtime python37 --trigger-resource gs://bctv-storage --trigger-event google.storage.object.finalize --source c:/users/blaine.brewer/documents/python/cloudcomputingbctv')

Once the function executes, we could download our files to our local machine as we did in the previous section with the following command.

os.system('gcloud compute scp bctv-compute:/tmp/output/predictions.csv ' + file\_source + 'predictions.csv --zone us-central1-a --project cloudcomputingbctv')

os.system('gcloud compute scp bctv-compute:/tmp/output/prediction\_accuracy.txt ' + file\_source + 'prediction\_accuracy.txt --zone us-central1-a --project cloudcomputingbctv')

# Google Cloud Functions gives us the simplest solution to deploy and execute our python model, as we were able to run our model and produce our model output with only a few lines of code. The benefits to this method are that there is little room for error. In some of the other configurations, there are so many choices and so much complexity, that building the automated pipeline can be very time consuming and strenuous at times. Some limitations of the Google Cloud Functions approach are that debugging your model may not be as easy as running your python code inline on your instance. In this case, the user would need to reference the function’s log files to diagnose the issue. Additionally, the user is limited to coding their model in Node.js, Go, or Python, so the R developers are left hanging. These problems aside, the automatic scaling, ease of use, and serverless interface make Google Cloud Functions a clear winner in our use-case.Conclusion

We concluded that we were able to automate the end-to-end process seamlessly using only API commands. Having this end-to-end automated template will help with reducing time on setting up environments and allow more time on data analysis and research. In the comparative analysis, we reinforced that each of the provider’s solutions offers unique advantages and disadvantages. Although AWS offers a more intelligible API, these ease of use of Google Cloud Shell and the successful package install at instance startup provided the only top down fully automated solution. Using the analysis provided above, one could follow the steps we took to develop a user interface where an analyst could select a file on their local machine, specify a test and training dataset and, with the click of a button, push their model to the cloud and run their model with the provided data. With cloud offerings like the ones tested in our project, a Data Scientist seems to have truly unlimited resources at their disposal.

**Citations**

Columbus, Louis (2017) “Roundup of Cloud Computing Forecasts, 2017”

Available at: https://www.forbes.com/sites/louiscolumbus/2017/04/29/roundup-of-cloud-computing-forecasts-2017/#4e5caa0331e8

Yim, Victor and Fernandes, Colin (2018) "The Resource Allocation Optimization Problem for Cloud Computing Environments," SMU Data Science Review: Vol. 1 : No. 3 , Article 2.

Available at: https://scholar.smu.edu/datasciencereview/vol1/iss3/2

**References**

https://medium.com/@patrickmichelberger/how-to-deploy-a-serverless-machine-learning-microservice-with-aws-lambda-aws-api-gateway-and-d5b8cbead846

https://towardsdatascience.com/simple-way-to-deploy-machine-learning-models-to-cloud-fd58b771fdcf

https://linuxacademy.com/guide/14209-automating-aws-with-python-and-boto3/

https://cloud.google.com/sdk/docs/

https://codelabs.developers.google.com/codelabs/cpo200-startup-scripts/index.html

https://realpython.com/python-boto3-aws-s3/#creating-a-bucket

https://blog.ipswitch.com/how-to-create-an-ec2-instance-with-python

1. . [↑](#footnote-ref-1)