Liverpool John Moores University

**Analysis of Tabular Data Using Local Deployment vs. Cloud Services,**

**with Data Pipelines Optimisation for Cloud Deployment.**

A final project submitted in satisfaction of the requirements

for the degree Master of Science.

in

Data Science

by

Marcin Majeran

Committee in charge:

Professor Iain Steele, Dr. Joao Da Silva Bento

2024

Copyright

Marcin Majeran, 2024

All rights reserved

# TABLE OF CONTENTS

[1. TABLE OF CONTENTS 2](#_Toc175998981)

[2. ABSTRACT 3](#_Toc175998982)

[3. INTRODUCTION 4](#_Toc175998983)

[3.1. Context 4](#_Toc175998984)

[3.2. Literature Review 4](#_Toc175998985)

[3.3. Data 5](#_Toc175998986)

[4. METHODOLOGY 6](#_Toc175998987)

[4.1. Design 6](#_Toc175998988)

[4.1.1. Deployments 6](#_Toc175998989)

[4.1.2. Functions 7](#_Toc175998990)

[4.1.3. Measurements – performance indicators 8](#_Toc175998991)

[4.2. Implementation 9](#_Toc175998992)

[4.2.1. Local 10](#_Toc175998993)

[4.2.2. Cloud 10](#_Toc175998994)

[4.2.3. Containerized 11](#_Toc175998995)

[4.2.4. Vertex 11](#_Toc175998996)

[5. RESULTS 12](#_Toc175998997)

[5.1. Data pull – data\_grab() 12](#_Toc175998998)

[5.1.1. Execution time 13](#_Toc175998999)

[5.1.2. Memory used 15](#_Toc175999000)

[5.1.3. Average CPU usage 16](#_Toc175999001)

[5.2. ML – evaluate() 18](#_Toc175999002)

[5.2.1. Execution time 18](#_Toc175999003)

[5.2.2. Memory used 19](#_Toc175999004)

[5.2.3. Average CPU usage 21](#_Toc175999005)

[5.3. Boost versions 22](#_Toc175999006)

[5.4. Correlations 25](#_Toc175999007)

[5.5. Other Factors 27](#_Toc175999008)

[5.5.1. Cost 27](#_Toc175999009)

[5.5.2. Ease of setup and use 28](#_Toc175999010)

[6. CONCLUSIONS 30](#_Toc175999011)

[6.1. General conclusions 30](#_Toc175999012)

[6.2. Further work 31](#_Toc175999013)

[7. SELF-EVALUATION 32](#_Toc175999014)

[8. REFERENCES 33](#_Toc175999015)

# ABSTRACT

Organizations across various sectors often grapple with the challenges of managing, analyzing, and deriving insights from vast amounts of numerical data. Due to its constantly increasing sheer amounts, large-scale cloud solutions needed to be applied. To address these challenges, this project explores the integration of cloud computing technologies, applied through different types of cloud deployments, with data science methodologies.

The primary aim of this research is to evaluate the feasibility and effectiveness of deploying data science tools for tabular data analysis in both local and cloud environments. Through a comparative analysis focused on Google Cloud Platform (GCP), this study assesses key performance metrics such as execution time, memory and CPU usage, processing speed, cost-effectiveness, and ease of use of each deployment.

The results obtained and described in this document indicate that cloud-based solutions, offer significant advantages over their local equivalent, in terms of memory and CPU management, reliable performance not affected by consecutive executions, scalability and accessibility. However, this study also compares in-cloud deployments between themselves. Results show that basic Cloud Shell computing isn’t efficient enough for computationally demanding tasks, and there are several pros and cons of Containerized and/or Vertex AI, solutions, which user has to leverage when choosing a desired service. Solutions in containers carry a substantial cost-quality value, but can be difficult to setup, slower and slightly less reliable than the expensive Vertex AI solution with high-end configuration.

# INTRODUCTION

## Context

The initial idea behind this project’s motivation was to observe, whether a direct ‘translation’ of a locally built solution to the exemplary data science problem, gains any advantages across different cloud-based deployment types. The general setup structure envisioned a progressive rise in the deployment’s ‘cloud contained’ factor – which can be explained as the estimate of what part of the solution utilizes available cloud tools and services, to what extent, and how ‘deep’ in the cloud infrastructure the deployment is. To further elaborate on this aspect – as an example, the basic cloud infrastructure, using the GCP’s Cloud Shell as the working environment is considered very ‘shallow’ as a cloud solution. On the other side of this spectrum is a Vertex deployment, which is built within Google’s Vertex AI instance, using its own, dedicated cores, memory and using proprietary features – this solution will be considered as a ‘deep’, integrated and self-contained cloud deployment. Each solution has been setup in a way that allows key metrics to be tracked, and further used for a comparative analysis.

## Literature Review

The shift towards cloud computing has become increasingly prevalent across various sectors, with a significant percentage of businesses now managing their data in the cloud. According to the Colorlib report (Rok Krivec, 2024) -  **94% of enterprises** worldwide use cloud computing to perform the data-oriented operations and practices, such as data storage, manipulation or machine learning. These adopted strategies can be categorized as IaaS cloud architectures (Infrastructure as a Service), which are the most flexible and comprehensive among other types of cloud services (examples are: AWS, Microsoft Azure, GCP) (Patel Hiral B, 2021). This widespread adoption is driven by the numerous advantages that cloud computing offers, particularly in the context of data science. This project focuses on identifying potential **computational** advantages by using performance metrics such as execution time, memory and CPU usage.

One of the most obvious benefits of using cloud deployments is the cost reduction. It has been proven that maintaining an on-premises private computing platform is more expensive, in early stages, than migrating and maintaining the same workload into the cloud system (Per Bondenson 2021). On the other hand, another study (Cameron Fisher, 2018) shows that despite initial low setup/maintenance costs of cloud deployment, in the long run the accumulative nature of cloud pricing makes the on-premises solutions significantly cheaper. Nevertheless, the scalability is another important factor worth taking into the account in all local vs cloud comparisons. The ease with which the user can scale up (or down) deployments on the cloud platform is far more appealing than the struggle one can experience when trying to expand the local solution, to accommodate more memory or processing power.

Despite these benefits, there are challenges associated with cloud adoption, but also data security and governance. While cloud providers offer robust security features, such as encryption and identity management, the complexity of managing data across distributed cloud environments can lead to increased risks. According to researchers **48% of cloud-stored data** is sensitive (Rok Krivec, 2024). However, many businesses are willing to accept these risks in exchange for the scalability and flexibility that cloud computing provides, especially given the cost and performance advantages. Standard practices ensuring the security of a workload have been applied, but are outside of this project’s scope, and won’t be discussed in detail.

In conclusion, the literature indicates that cloud computing is not only a viable but often a superior option for data science applications, particularly for organizations seeking to improve efficiency and reduce costs. However, it also introduces new set of challenges and problems that need to be addressed when deploying in the cloud.

## Data

To choose the dataset suitable for the analysis in this project, few factors needed to be considered. Firstly, the data had to allow predictions/classifications to be made, as supervised machine learning task will allow to precisely assess whether the algorithm performed well or not. Secondly, data needed to consist of a significant number of features and rows, that would be sufficient to make the model’s training complex enough to benchmark performance of various deployments and capture the potential differences between them. Lastly, chosen dataset had to be easily processed by Google’s Vertex AI service, which can process Image, Video, Text and Tabular data. After prior research and some tries with different datasets, the HAR (Human Activity Recognition) data, precisely **Human Activity Recognition with Smartphones** by UCI dataset turned out to be a perfect fit . Combined training and test data contribute to **10299** unique rows of **561-feature vectors with time and frequency domain variables** (+ labels). Each of those vectors represents the measurement taken on one of 30 volunteers, within age bracket of 19-48 years. Data has been collected through the app installed on **Samsung Galaxy S II** worn on the waist and consist of **6** different activities (**labels**): WALKING, WALKING\_UPSTAIRS, WALKING\_DOWNSTAIRS, SITTING, STANDING, LAYING. Accelerometer and gyroscope data has been processed beforehand including Fourier Transform with 50% overlap, Butterworth low-pass filter and cutoff filter, to prepare the data before using.

# METHODOLOGY

## Design

A diagram of a cloud server

Description automatically generated

Diagram 1. showing project’s complete structure. All relationships between deployments and separate services are shown as arrows.

### Deployments

This study follows the design made of 4 distinct parts, where each part is a separate, standalone environment, with purpose of solving the same data science problem. To ensure that this project captivates the difference between truly unrelated environments, where each one is on a different ‘level’ of how ‘deep’ in the cloud infrastructure it is, these themselves must be independent. In compliance with this rule, 4 separate deployment types have been derived. In this section, the general, high-level description of those deployments and their properties can be found. Technicalities and more details are provided in the **4.2 Implementation** section.

* Local – on-premises solution, code is executed on local machine using available hardware. For purposes of this solution data is stored in locally hosted object storage server – to emulate the general, very simple data pipeline that pulls the data from cloud storage service.
* Cloud – the solution contained and executed on Google Cloud Platform in a Cloud Shell. Data is stored in a Cloud Storage bucket.
* Containerized – this deployment is built as a Docker container image, deployed and executed as a Job in GCP Cloud Run service. Data is stored in a Cloud Storage bucket.
* Vertex – solution which utilizes Google’s proprietary ML platform, Vertex AI. Deployment uses Vertex instance to run the program and employs a ‘mounted bucket’ feature, which allows to directly connect an instance to the GCP Cloud Storage bucket and access it with minimal latency.

This set of 4 deployments serves as an entire project structure, remotely managed and stored in a (private) Github repository. Each solution consists of the same code, only altered to correctly communicate with a corresponding data source. This will bring additional information on how different settings for data storage affect the execution performance.

### Functions

In general, each deployment has two instructions, that need to be fulfilled to successfully finish the program run with a solution (list of classified labels). These instructions are contained in python functions:

* *data\_grab()* – responsible for:
  + connecting to the data source (this will vary based on the deployment type),
  + downloading train.csv and test.csv files from the data source,
  + saving both files as Pandas DataFrames, concatenating them and returning as a single DataFrame.

This function has been designed to test multiple aspects of each run i.e. how well will different solutions handle various protocols for getting the data from a remote data source, downloading data from previously connected data source, and finally reading and saving it as a single data structure.

The second function is:

* *evaluate()* – which takes care of:
  + fitting the train data to the provided model,
  + predicting labels for the test data,
  + calculating the best cross validation test accuracy score and returning model fitted with data.

This python function simply tests machine learning performance of each solution. Even if CPU usage is being measured for both functions, it’s the *evaluate()* where this metric can be tested sufficiently, under high demand. Such set of two main instructions will serve as a ‘base’ for taking measurements. Additionally, each deployment type shares some additional instructions. These can be summarized as:

After *data\_grab()*, before *evaluate()*:

* Separating explanatory/response variables, and saving them as separate DataFrames,
* 80:20 train-test-split,
* Defining Random Forest Classifier model,
* Setting up ad Grid Search to be used as a default model.

The idea behind selecting Random Forest Classifier for the model used in the testing, is that with conjunction with Grid Search, (designed to search through 1-9 pre-defined random states of a grid) it ensures that the computational task is sufficiently difficult to test the machines' memory and CPU usage. Because it builds several decision trees, Random Forest is computationally intensive, and Grid Search adds complexity by testing different configurations. Utilizing ‘random\_state’, allows each run to maintain high level of complexity without requiring the testing of extra hyperparameters, guaranteeing a thorough evaluation of system performance in various deployment scenarios, while preserving an acceptable runtime.

### Measurements – performance indicators

To quantify the results of this study 3 main measurements/performance indicators, have been selected:

* Execution time – measured in seconds (s), helps to determine whether runs from one deployment are faster (or slower) than runs executed using different solution.
* Memory used – expressed in Mega Bytes (MB), tells how much from available memory, the program run ended up using.
* CPU used – this performance indicator is using average value of percentage of total CPU usage during execution time, measured (almost) every 0.1 second, where only ‘ticks’ with positive values are taken into the account (to avoid including zeroes when CPU is not in use). Further breakdown of what % of all ticks was equal to 0 (CPU idle) will be provided in **4.2 Implementation** section.

In a single run, each of these 3 parameters will be tracked while executing *data\_grab()* and *evaluate()* function. This setup ensures that separate results are stored for both actions. Every successful run should generate 7 unique values structured as a row of data, with each column being:

* label – ‘name’/type of deployment, timestamp,
* data\_time,
* data\_memory,
* data\_cpu\_usage,
* ml\_time,
* ml\_memory,
* ml\_cpu\_usage.

For meaningful results, each deployment will be instructed to run a program for at least 100 times, with additional 100 when upgrade to the instance’s machine hardware is available (this is the case with **containerized** and **vertex** solutions). Upgraded runs will carry an additional ‘-boost’ postfix flag in ‘label’ variable.

## Implementation

Each implementation is coded in Python and its structure follows similar schema, having two folders: ‘results’ for storing measurements and ‘src’ which contains:

* config.py – where deployment’s global variables like labels, keys, or bucket names are stored,
* HAR.py – the main script which executes both *data\_grab()* and *evaluate()* functions. It uses *@profile* function decorators from *memory\_profiler* library to document execution time and memory usage.
* get\_results.py – supporting script, which accesses ‘mprofile.dat’ generated file and pulls the data into corresponding results file.
* HAR\_cpu\_read.py – 2nd main script which had to be introduced due to the conflict of *mempory\_profiler’s @profile* decorators and *multiprocessing* package, which is used to estimate average CPU usage per each **function** run.
* command.sh – instruction script used to execute all code in the correct order.

Because of *memory\_profiler* and *multiprocessing* libraries conflict, the second main script – HAR\_cpu\_read.py file had to be created. This fundamentally means that each measured run is, in reality, two different runs – one for which execution time and memory used are measured – and second one during which an average CPU usage is recorded. This, unfortunately, introduced unwanted complexity into the project and forced solution as described above. Also, due to this complication CPU usage will be treated as an independent variable from execution time and memory used.

Apart from the general file structure described above, every deployment has its own unique properties and storage solutions.

### Local

On-premise deployment utilizes MinIO object storage system which emulates cloud storage (MinIO is mainly compatible with AWS S3 solution but for purposes of project it’s being compared to Google Storage which is also supported). MinIO allows its users to create buckets which store data as objects – in this project train.csv and test.csv files are stored in a single bucket deployed on a standalone MinIO server. This data can be retrieved, as an object type, from server utilizing minio Python package.

**Average % of 0 ticks during CPU measure:** *data\_grab()*: 35%, *evaluate()*: 81%

This statistic stays fairly similar across all solutions, with Local being the biggest exception with relatively low % of 0s in *data\_grab()* CPU reads. In general, around 50% of all CPU reads were 0s for data pull, and around 80% for model training across all deployments. This means that during the active periods, the CPU is fully (if possible) occupied with the task, and any idle time is consistent, likely due to the CPU waiting for Input/Output operations or being in a low-power state when no tasks are ready to run. The consistency in idle time percentage indicates that the variability in execution time across different solutions is more about **how** they utilize their allocated CPU time rather than differences in CPU idling.

**Specifications:** CPU: Apple M1 3.2GHz 8 cores, RAM: 16GB

### Cloud

This solution is entirely stored (except for the data) within GCP’s Cloud Shell. Cloud Storage has been used to create a bucket (equivalent of local solution’s MinIO bucket), within which the data is stored. By using storage.Client object from *google.cloud* library and previously setting up valid credentials for GCP project, script can connect to the bucket and pull the data. Retrieved information is read as BLOBs (Binary Large OBjects), downloaded as text and then casted into csv format with StringIO function from io packaged – allowing the data to be loaded into pandas DataFrame and further processed by program. Each run in this solution has been executed while only single Cloud Shell terminal was active, to prevent any performance issues caused by this type of resource sharing.

**Average % of 0 ticks during CPU measure:** *data\_grab()*: 50%, *evaluate()*: 87%

**Specifications:** CPU: Intel(R) Xeon(R) 2.2GHz 4 cores, RAM: 16GB

What is worth noting at this point is that every GCP service used in this project is ran on a server using **US-west1 (Oregon)** as a Region (no specific Zone). This choice ensures that all Free-Tier services can be utilized to their fullest (available in Free-Tier). It’s also important as, in this case, Cloud Shell specification is determined by the region we’re hosting our project in. However, as there is no way of manually setting up a specific Region for Cloud Shell, GCP will assign a geographically closest, stable Region instead. Because of that, each run has been monitored to ensure that the same specification is used per every run.

### Containerized

Code structure of this deployment is also stored in a Cloud Shell, from which a Docker container image is built using *gcloud builds* command, and automatically saved in a specified GCP’s Artifact Registry – Google’s service where user can create repositories for container images. It allows them to be easily accessed and managed within GCP project. In this case, after image is built within Cloud Shell, and sent to the Artifact Registry, it can finally be executed by Cloud Run service as a Job. This deployment uses the same Cloud Storage bucket as the Cloud deployment. Due to the enclosed nature of this solution, all operations allowing the program to successfully pull, process data, and save the results, had to be put in the Dockerfile. Within that file, the container’s builder is instructed to activate GCP’s service account, to allow container communicating with Cloud Storage system. Due to the immutable nature of this deployment, all results are being stored in the Cloud Storage bucket for easier access.

**Average % of 0 ticks during CPU measure:** *data\_grab()*: 60%, *evaluate()*: 87%

**Specifications:** Intel(R) Xeon(R) 2.0GHz 4 cores, RAM: 2GB

**Boost:** Intel(R) Xeon(R) 2.0GHz 8 cores, RAM: 16GB

This deployment has added ‘Boost’ line of specification, as after collecting the data for the default machine the entire process has been repeated using the upgraded version. This approach will allow for more thorough analysis, by comparing the same solution type with different hardware specification. Initially, every default version of a deployment was run on the ‘recommended’ machine’s hardware. When boost version of a deployment is available, the chosen configuration will be trying to be as close as possible to the Local specification.

### Vertex

Solution utilizing Google’s proprietary Vertex AI service. This deployment runs on a Vertex AI instance, in which the default structure of the project is initialized. Storage is handled by GCP’s clever feature– bucket mounting. GCP allows users to mount a bucket onto the running instance (Vertex AI instance in this case) and access its contents directly. This significantly reduces latency and allows the program to execute *data\_grab()* faster. Since Vertex AI has been created with machine learning in mind, its hardware has been precisely selected to process ML requests. Unfortunately, GPU’s offered as part of Vertex AI available processing units couldn’t be applied in this project as *scikit-learn.RandomForestClassifier* doesn’t support GPU acceleration and library isn’t directly supported by Vertex AI (which supports specifically Tensorflow and Keras models), so in this case it can only utilize the purely higher processing power offered by the service. Each program in Vertex deployment has been executed by *tensorfow* Anaconda virtual machine, by default provided with Vertex AI instance.

**Average % of 0 ticks during CPU measure:** *data\_grab()*: 45%, *evaluate()*: 81%

**Specifications:** Intel(R) Xeon(R) 2.20GHz 4 cores, RAM: 16GB

**Boost:** AMD EPYC 7B12 2.25GHz 8 cores, RAM: 32GB

# RESULTS

The study's main conclusions are presented in this section, with an emphasis on the performance indicators that were gathered throughout the iterations. The findings are arranged to draw attention to the most significant outcomes. The particular outcomes for each component of the study are described in depth in the ensuing subsections.

## Data pull – data\_grab()

Firstly, let’s focus on procedure responsible for gathering the data. There are few important things worth to remember in this case: each deployment utilizes slightly different approach of getting the data from the data source. Local has a hosted object storage server, imitating Cloud Storage. Cloud and Containerized both uses the exact same code to download the data from Cloud Storage bucket, and Vertex has this bucket mounted onto the instance.

At this point, it is also important to introduce another layer to the analysis – Local deployment’s runs can be further classified into two sub-types. First 50 runs (from 0 to 49), were executed with time breaks in-between each run, and runs 50-99 were executed in ‘batch runs’ with 10 or more consecutive runs within a loop. This approach was engineered to allow the only on-premises solution to ‘rest’ between the runs (in the first 50), and potentially observe the effect of components heating in the 2nd half. Similar approach was applied for other deployments and occurs for Vertex around 30th iteration and for Cloud for 60th. Containerized solution utilizes a Cloud Run queue where user can schedule number of runs for the job and was run mostly in consecutive queues consisting of 3 scheduled runs. Because these 3 solutions are deployed in the cloud and processing units would be applied dynamically there is no guarantee that two succeeding runs are being run on the exact same machine.

### Execution time

A graph showing a graph of data

Description automatically generated with medium confidence

Figure 1. Execution time [seconds] of data\_grab() function across different deployments.

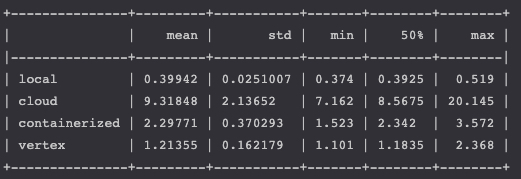


Table 1. Mean, Standard Deviation, Minimum Value, Median, Maxium Value of data\_grab() execution time [seconds] per each deployment.

As shown in the presented plot and table all solutions, except for Cloud, had a quite stable, consistent execution time. There is a significant gap between **Cloud** and other deployments, which can be characterized as the slowest solution with the highest variance. Even when looking at the medians, which could be treated as a ‘baseline’ for each solution, the Cloud run would finish in about 8.5 seconds which is 266% slower than the slowest deployment from the other 3 – **Containerized**, which has the 2nd highest standard deviation. However, the profiles of variance in these two deployments (Cloud, Containerized) are significantly different.

In **Cloud**, we can spot several rapid upward spikes, marking notably slower runs, whereas when looking at Containerized’s plot shape we can observe the two-directional (upwards, downwards) fluctuation. Cloud solution is executed entirely within the Cloud Shell, which definitely isn’t a dedicated service for computational heavy operations, and as stated in Google’s documentation, Cloud Shell is a tool to *‘manage your projects and resources from your web browser’*. Moreover, Cloud Shell already sacrifices part of its resources to manage the current user session, synchronize files, or process any information related to the browser interface. In conjunction with these facts, some of slower runs could be caused by the use of Cloud Shell Editor (GCP’s shell IDE) which drains additional resources.

On the other hand, **Containerized** deployment has a dedicated line of resources designed and available, specifically for the processing task on the heavier side. This is probably the main reason why we see such an improvement over the Cloud solution. To determine what could cause the fluctuation, it’s worth looking at characteristics of a ‘serverless’ execution of a cloud container. Even if resources are dedicated for the task, these are still dynamically allocated, which together with ‘cold start’ (which occurs after container idle for some time) could cause observed variability. Another factor to consider is the latency caused by network connection, GCP’s resource sharing (multi-tenancy) when main resources can be shared across multiple users (heavy users called ‘noisy neighbors’) or resource isolation initialized by the container itself (isolating parts of resources for specified sets of processes, defined by ‘cgroups’ kernel feature).

**Vertex** deployment showcases how beneficial it can be to use Google’s features, like bucket-instance mount. This solution achieved very consistent execution times (with only few exceptions), while being 2nd fastest to download the data.

The fastest solution, **Local** – proves that locally hosted storage server with a minimal latency, run on a competent machine, will allows for faster data acquisition than any other ‘online’ solution.

### Memory used

Next, the memory used by each deployment during the process of downloading the data from its designated data source.

A graph showing a memory used

Description automatically generated

Figure 2. Memory used [megabytes] of data\_grab() function across different deployments.

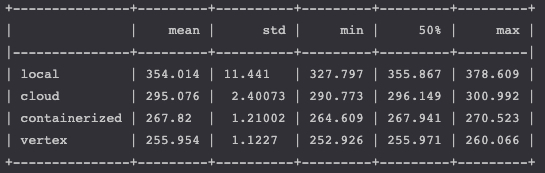


Table 2. Mean, Standard Deviation, Minimum Value, Median, Maxium Value of data\_grab() memory used [megabytes] per each deployment.

Figure 2 clearly shows how well-tuned GCP’s (or really any cloud-based) memory management is. When compared to the **Local**, it can be immediately observed that on-premises solution performed the worst. This series of runs shows how inconsistent, in terms of memory-management, the locally hosted solution can be.

There is an interesting change in **Cloud** performance around 40th iteration (x-axis). For approximately 3 iterations the average memory used rises (almost linearly) from the state of 292.5MB to 295.9MB (1.16% increase – not much, but interesting due to the previous consistency and low variation). It is hard to tell what the direct reason behind such a behavior was. One potential hypothesis would be that the entire project has been ‘hard allocated’ to a different group of resources available, after detecting an increased computational traffic (simply ‘flagged’ by Google). However, this hypothesis isn’t confirmed so further work would be needed to unearth the underlying reason.

**Containerized** and **Vertex** solutions performed very well, providing a memory stable infrastructures, with the latter being the best optimized deployment in this comparison.

### Average CPU usage

A graph showing a graph of data

Description automatically generated with medium confidence

Figure 3. Average CPU usage [percent] of data\_grab() function across different deployments.

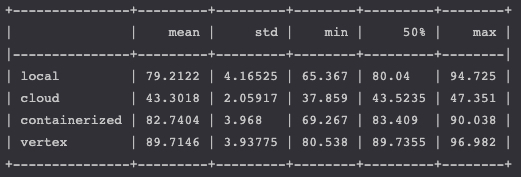


Table 3. Mean, Standard Deviation, Minimum Value, Median, Maxium Value of data\_grab() average CPU usage [percent] per each deployment.

At first, by looking at CPU measurements, it can be noticed that **Cloud** deployment, on average, used the least % of its available processing unit’s resources. However, it doesn’t necessarily mean that it performed the best – this fact, combined with the slowest run, reveals that low CPU consumption in this case might be related to the limited availability of these resources. This theory directly interlinks with the ‘flagged’ hypothesis, as Google’s states in Cloud Shell documentation ‘*there are also limits on Cloud Shell resources’*, hinting that if the terminal gets flagged when trying to run computationally heavy task, its resources get blocked and won’t be fully available.

The remaining 3 deployments tend to oscillate between 80 and 90 (%) values. Box plot captures these differences more efficiently than a line plot.

A graph showing a graph

Description automatically generated with medium confidence

Figure 4. Average CPU usage [percent] of data\_grab() function shown as box plot.

Interesting outcome from this plot is the variance / outliers of **Local** deployment. These can be also seen in the line plot above, presenting a significant change of CPU usage from around 50th iteration. From that point executions have been run within a loop, one after another, not letting the machine to cool down and ‘rest’. There is a visible increase in standard deviation from the mean after that mark, which can indicate that constant use of CPU affected the performance of the machine. However, the observed differences are mainly sudden drops in the average CPU usage, which could mean that processor is hard locking clock cycles (GHz) to prevent overheating. The two halves of Local plot would need to be further investigated if there is a statistically significant difference between runs that happened in a quick succession (in a loop) and runs when the machine had a chance to cool down.

**Vertex** solution, although having the highest mean, turns out to be the least scattered among all 4, having no outliers outside its confidence intervals.

## ML – evaluate()

Once the data was downloaded and pre-processed, the script could execute the main part of the problem – machine learning task. Following measurements have been collected when monitoring runs of such a task, for each deployment. The script, including inputs and outputs remains the same for each solution.

### Execution time

A graph showing a graph of time elapsed

Description automatically generated

Figure 5. Execution time [seconds] of evaluate() function across different deployments.

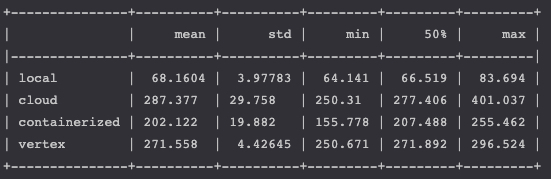


Table 4. Mean, Standard Deviation, Minimum Value, Median, Maxium Value of evaluate() execution time [percent] per each deployment.

**Local** deployment has again, the fastest and most consistent time across all solutions.

**Cloud** performs the worst, surprisingly, almost on par with **Vertex**.Additionally, odd ‘artifacts’ can be spotted when looking at the Cloud performance, starting (again) around 40th iteration mark. There is no confirmation in data to why this is happening, but this large change in the execution time, especially when several previous runs had a similar, much shorter times, can be caused by resource sharing. Even if there were no additional users working in this GCP project, the resource contention can still occur due to the other workloads running on the same physical machine. Also, potentially, as mentioned before, project’s Cloud Shell could be flagged as one with increased traffic, and potentially had its resources limited/blocked by Google.

**Containerized** has some, now more extreme, fluctuation to its execution times. However, it still has a solid performance (despite using low-end instance type with weak configuration), when compared to the other 2 cloud solutions.

**Vertex** deployment’s poor results can be caused by several reasons, one of them could be the fact that Vertex AI, despite being a service specifically designed for ML tasks, mainly supports TensorFlow and Keras utilities, hence running machine learning with Scikit-Learn tools won’t fully benefit from the platform offerings. On the other side, it’s the most stable and reliable cloud deployment with relatively little variance.

### Memory used

A graph showing a memory used

Description automatically generated

Figure 6. Memory used [megabytes] of evaluate() function across different deployments.

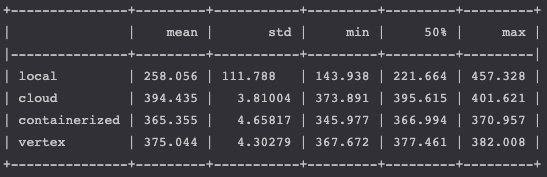


Table 5. Mean, Standard Deviation, Minimum Value, Median, Maxium Value of evaluate() memory used [megabytes] per each deployment.

By looking at this plot, first thing that needs to be addressed is the severe magnitude of how inconsistent, in terms of memory management, the **Local** deployment is. Around 150MB seems to be a min value boundary of memory used during model training (actual min value is 143.94MB). Even with such large outliers, the mean and median are the smallest across all deployments. This, however, isn’t easily interpretable with such a high variance. Changing to running the script in a loop, with no breaks in-between runs, only makes the spikes in the plot more frequent. For first 50 iterations we can observe only 5 runs using more than 350MB of RAM, whereas 2nd half contains 18 (260% more) of them.

Looking at **Cloud** memory usage, although not easily visible, but still shows that around 40th iteration the performance decreased. Before this point the mean was 392.6MB, and after it has risen to 396MB, staying consistent around that point and not coming back to the previous baseline.

**Vertex** ended up using on average more memory than **Containerized** but with lower standard deviation.

### Average CPU usage

A graph showing a graph of different colored lines

Description automatically generated

Figure 7. Average CPU usage [percent] of evaluate() function across different deployments.

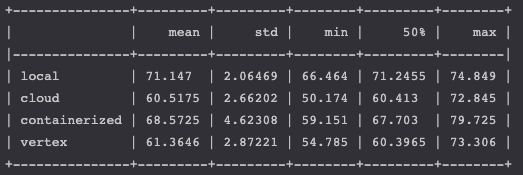


Table 6. Mean, Standard Deviation, Minimum Value, Median, Maxium Value of evaluate() average CPU usage [percent] per each deployment.

A graph showing different colored squares

Description automatically generated

Figure 8. Average CPU usage [percent] of evaluate() function shown as box plot.

For this measurement it’s easier to look at the summary statistics in Table 6, and box plots in Figure 8. This comparison shows that **Containerized** deployment was the most inconsistent in CPU usage, with 2nd highest mean and median. The highest ones were observed for **Local** which now has the lowest spread of its values and stays fairly consistent within its range. **Vertex** and **Cloud** solutions both contain significant outliers. The first might have been affected by resource sharing, or increased latency due to lag.

## Boost versions

A graph showing different colored lines

Description automatically generated Figure 9. Execution time [seconds] of evaluate() function with added plots for ‘boost’ deployments.

A black and white screen with numbers and lines

Description automatically generated

Table 7. Mean, Standard Deviation, Minimum Value, Median, Maxium Value of evaluate() average CPU usage [percent] per each deployment, with boost version included.

In yellow and green there are plots of boosted versions for Containerized and Vertex deployments respectively. By upgrading the hardware configuration of these, the most noticeable difference can be seen in ML task execution time. Approximately 10 first runs of **Vertex\_boost** have been affected by some initial lag, potentially due to instance setting up and changing to the new infrastructure. After that, deployment stays very consistent just below 100 second mark (median 94.8s), making Local only ~30% faster which is a great improvement to the default Vertex version.

**Containerized\_boost** also benefits from the upgrade, having now runs faster on average by ~90s when compared to its default version. It also places 3rd in terms of shortest ML execution times overall. Another important result of new configuration is its lower standard variation, which went down from 19.82s to 11.9s, indicating that boost version of Containerized is much more reliable in terms of performance.

Similarly, data pull process also shows the significant increase in performance for Vertex\_boost (no plot included), making Vertex\_boost 2nd fastest solution. The improvement in Containerized\_boost data pull time wasn’t enough to notice any important changes.

Interesting outcomes can be seen in CPU usages:

A graph of a graph showing a number of colored boxes

Description automatically generated with medium confidence

Figure 10. Average CPU usage of data\_grab() shown as box plot with added plots for ‘boost’ deployments.

A graph of different colored boxes

Description automatically generated

Figure 11. Average CPU usage of evaluate() shown as box plot with added plots for ‘boost’ deployments.

In both cases new, upgraded versions of deployments have higher means of CPU usage during *data\_grab()* or *evaluate()*. This indicates that deployments with more cores and memory fully utilized them and, in a result, performed the task faster and more effectively (strikingly reduced ML execution time). With more CPU power available some operations could be done in parallel, which increased the performance, but also the average CPU usage. ML CPU usage box plots (Figure 11) and execution time line plot (Figure 9) shows how much superior Vertex\_boost is when compared to other 5 deployments. It stays very consistent, utilizes most of its CPU power available, and executes scripts the fastest, losing only vs Local.

The analysis shows that while Local and Cloud deployments are convenient and cost-effective, they fall short in handling resource-intensive tasks efficiently. In contrast, the Containerized solution, proves to be the effective for high-performance needs, delivering significant improvements in execution time and cost-efficiency. Addition of boost versions of Containerized and Vertex deployments, unravels the potential in high-end configured Vertex AI instance, but also highlights how reliable well-specified Containerized application can be.

## Correlations

A group of blue dots

Description automatically generated

Figure 12. Local deployment’s correlations, with added CPU usage.

When looking at correlations in Figure 12 and Figure 13 it is crucial to remember that all CPU measurements were made during separate script calls, so technically doesn’t directly depend on measured execution time and memory used. However, since each of CPU measure scripts has been activated right after time/memory runs, there might be underlying correlations to discover. Unfortunately, due to this fact, none of found relationships for CPU usage should be considered significant.

Apart from partially obvious conclusions like positive correlation between **data\_time** and **ml\_time** (although weak – Pearson coef. = 0.35), or potential relationship of **data\_time** and **data\_memory** (longer runs result in more memory used – but it isn’t consistent across different solutions, perhaps only can take effect on local solution), there weren’t any interesting outcomes found during this research. This topic hasn’t been explored extensively and further work would might uncover more relationships by defining and excluding outliers.

A screenshot of a graph

Description automatically generated

Figure 12. Containerized deployment’s correlations, with added CPU usage.

Again, **data\_time** and **ml\_time** have a positive correlation with Pearson coef. = 0.43. In this example, a different observation can be formulated – based on comparison between **data\_cpu\_usage** and **ml\_cpu\_usage** (these two variables can be easily compared as come from the exact same runs), revealing a negative (Pearson coef. = -0.35) correlation between these two. This relationship is present in Local deployment, however, is much weaker (-15.6). deployments share the trend of having negative correlation in between those. This might potentially find a use when estimating the exact resource demands for both operations.

Remaining deployments have similar relationships as ones described above, but much weaker or not visible, blurred by outliers and high variation of one, or both variables.

## Other Factors

### Cost

In total, production of this project costed **£36.98**, after excluding Free Tier promotions and discounts. The full breakdown of how much costs each of the services generated can be found below:

A screenshot of a service

Description automatically generated

Starting from the bottom, **Pub/Sub** wasn’t used in this project, hence no costs.

Single **Cloud Storage** bucket, with train/test.csv files and results for Containerized solutions generated only £0.01 during the entire development of this project.

Only few tests with **Cloud Run Functions** were made, but the idea of using them was quickly dropped.

**Networking**, which according to GCP’s documentation is a fee taken for each data transfer happening within, in or out of the cloud project, generated £1. Cloud, and Containerized deployments both contributed to this value, while Vertex avoided generating any costs here by utilizing ‘mounted bucket’ feature.

**Artifact Registry** served as a Google’s proprietary hub for container images, later used in Cloud Run.

**Compute Engine** can be ignored, not used in this project (costs generated during initial tests).

**Cloud Run**, used to execute over 200 runs of Containerized (+boost version) deployment generated ~20% of all project expenses.

The most expensive service used in this study was definitely Vertex AI, hidden under **Notebooks** service name. Single instance over a course of a month made up to ~69% of all project costs. While this solution offered some of the most reliable and fast results (in boost version), the cost is significantly higher when compared to other cloud deployments.

Interestingly, Cloud Shell is completely free for GCP users, so there are no costs related to the Cloud deployment (except for Storage and Networking).

This reveals the discrepancy between Containerized and Vertex solutions in value-performance relation, and leads to the question, if it is worth paying for Vertex AI when not utilizing its full potential, by not using Tensorflow/Keras builds? To check this, the costs of running both boost versions can be compared (as these are more likely to be chosen for a real-life problem).

**Containerized\_boost**, with addition of **Artifact registry** and **Networking** fees generated total of £3.92 during over 100 runs of the application.

Running **Vertex\_boost** solution for the same number of runs costed £5.46, so was 40% more expensive than Containerized\_boost, while offering only ~10% faster runs (by comparing ML execution time means). This applies in this specific scenario, and shows that Vertex AI isn’t worth its price, when cheaper and with comparable performance solution is available. However, it can serve as a reference only when considering using Vertex AI without utilizing Tensorflow/Keras (otherwise Vertex would most likely perform much better and potentially justify its costs more efficiently).

### Ease of setup and use

The **Local** solution is fast but comes with significant drawbacks. It suffers from inconsistent memory management, leading to inefficient resource usage. Additionally, running intensive tasks locally causes the machine to overheat. As shown by splitting a way of executing script in this project (first half with breaks, second running consecutively), succeeding runs can be affected by this fact, and cloud providers’ offer is much more suitable if user wants to achieve a consistent and reliable performance. Automating the local solution can potentially further drain the machine's resources. Moreover, the solution is physically vulnerable to any damage or failures that might affect the machine itself. However, the local solution is the easiest one to set up. It also has the advantage of being free (cost of machine can be ignored as it’s needed for every deployment), which makes it an accessible option for those looking to avoid cloud costs.

**Cloud** solution is not intended for computationally heavy tasks. It is slow and low performing, which can be frustrating when dealing with more demanding workloads. However, it stands out for its ease of setup compared to other cloud deployments. Since everything runs within a Cloud Shell, setting up the environment is straightforward, and the difficulty of connecting to a Cloud Storage bucket is on par with setting up a MinIO bucket in the local solution (relatively easy). This might make the cloud option appealing for some users who prioritize ease of setup and don't require high performance. It’s also mainly free as there are no costs related to the computational operations.

The **Containerized** solution is by far the most challenging to set up. It requires knowledge of containerization techniques, software, and involves creating a custom container image for accessing GCP services, which requires a precise set of instructions to be passed into the build process. This solution also utilizes Google's proprietary container image storage—Artifact Registry—which, despite being easier to set up with GCP than Dockerhub (AR authorization is handled automatically), still adds complexity. Specifying the correct instructions for the container to execute each program in order was another hurdle, as any changes necessitate rebuilding and redeploying the container. However, once deployed, the containerized solution offers an immutable and stable environment. Managing the deployment through the GCP interface is very intuitive, and hardware upgrades are also straightforward, making it the easiest deployment to work with once the initial setup is complete.

The **Vertex AI** solution falls somewhere in the middle in terms of setup difficulty. It involves setting up a JupyterLab instance, which is a relatively straightforward process of following interface prompts. Once inside JupyterLab, the environment can be configured similarly to other deployments. A particularly useful feature is bucket mounting, which reduces latency and makes the storage easily accessible within the editor. Upgrading the machine in Vertex AI is also simple. However, despite these conveniences, the Vertex AI solution feels more like a sandbox environment than a fully deployed service. When compared to the premises of applications in a container, it lacks the one-click deployment capability that ensures a service runs seamlessly from start to finish.

# CONCLUSIONS

## General conclusions

Every analyzed solution offers a different balance of ease of use and performance, with trade-offs depending on the specific needs and constraints of the project.

After experimenting with each of these deployments, the recommendation drawn from the work described in this document would be to:

* Use containerized deployment when there is no heavy machine learning involved, or if it’s not utilizing Tensorflow/Keras libraries. With some experience in setting, it up, such a solution will provide a very stable, easy to work with environment that also performs very well.
* If the performance is our priority, we are willing to write the script to use Tensorflow/Keras and we are able to cover higher costs of Vertex AI solution, then it will serve as the best choice. Relative ease of setup, accessible editor and ‘mount bucket’ feature will surely speed up the work. Additionally, there is a GPU acceleration available so models like e.g. Convolutional Neural Networks and Recurrent Neural Networks can benefit from it.
* Possibly, the best would be to combine the best features of these two solutions and deploy a containerized application that handles ‘logistic’ part of the work on its side, then connects to the Vertex AI instance which hosts a prepared model to train it with data and send the results (or trained model) to the shared data source, making it available for app in the container. This way we could benefit from the high performance offered by Vertex service, while minimizing its costs by running only necessary ML related tasks on it and preserving the containerized immutable nature (and many more of its perks) of Containerized deployment.
* For problems that don’t require large-scale, scalable, fully managed, integrating solutions, simple Local deployment will suffice.

To summarize, solutions presented in this project caters to different needs. The Local and Cloud solutions are best for ease of setup and cost-effectiveness but fall short in performance. On-premises deployment suffers from memory and CPU management issues and isn’t suitable for long-running consecutive scripts. Cloud deployment doesn’t suffer that much from resource management issues, but is extremely slow, cannot fully utilize its memory/CPU and might be vulnerable to getting noticed by Google (due to heavy traffic) and have its resources limited. The containerized solution, though complex to set up, offers the most robust and reliable environment for demanding tasks, making it the top choice for production use. It performed fast enough and managed resources on a satisfactory level. In this study, tuning Containerized solution’s specification resulted in 45% faster ML execution time, while staying 40% cheaper than the Vertex AI. Vertex deployment provides a good middle ground for development and testing but to achieve reasonable results, it requires upgrading configuration and/or using Tensorflow/Keras builds. Therefore, the choice of deployment should be guided by the specific requirements of the project, user needs for performance, ease of use, and long-term reliability.

## Further work

**Improvement on data pull metric:** Develop more sophisticated data pipeline, involving operations that can generate enough complexity of the machine to better assess its performance under heavy workload.

**Refactoring Cloud deployment:** Although having an application running in a Cloud Shell is a novel solution, it is remarkably bad. To improve on this, it could be moved to a single-node Virtual Machine (GCP’s Compute Engine).

**Changing ML model:** Switch to TensorFlow library would allow Vertex AI to fully utilize its resources. However, this could significantly bias the outcome towards this single solution (in favor).

**Security Analysis:** Security and Privacy context could be added to the analysis and compared between deployments. This would create another way to assess how different these solutions can be.

**Automation and CI/CD:** Integrate CI/CD pipelines to automate testing, building, and deployment processes, enhancing efficiency and consistency across deployments.

**Alternative Cloud Platforms:** Evaluate similar solutions on AWS or Azure to compare performance, ease of use, and cost, offering a broader view of platform choices.

# SELF-EVALUATION

Reflecting on the project, I recognize both the strengths and areas for growth that emerged during its execution. I demonstrated how much I have learned about cloud computing, starting with simple Cloud Shell scripting, through building and deploying a containerized application to monitoring costs of the project. The ability to set up and manage these environments, particularly the more complex ones like Containerized and Vertex AI solutions, gives me a great portion of satisfaction, as I find the challenge rewarding and take joy in overcoming the technical difficulties or simply learning about new services, frameworks and how to orchestrate them to work together. Throughout my analysis I made certain that the collected data is at the sufficient quality, so the obtained results will be interpretable. I believe that this has been achieved, and when not ideal, it constructs a solid foundation for potential further research in this domain.

Time management was difficult, particularly in the beginning, and more organised planning would have helped to deliver a more polished project. Even though I was able to improve the configurations afterwards, the project's results would have been refined if there had been an earlier emphasis on performance and cost effectiveness. Finally, comparing a wider range of cloud platforms early on, and including that analysis in the study, would add an additional dimension to the project. Moreover, it might have offered a more comprehensive understanding of the optimal deployment choices.

All things considered, the project was a worthwhile educational experience that both confirmed my technical implementation abilities and pointed out areas in which I can still improve for upcoming projects.

Marcin Majeran

# REFERENCES

1. Krivec, R. (2024) *Cloud computing statistics (how many companies use cloud computing?)*, *Colorlib*. Available at: https://colorlib.com/wp/cloud-computing-statistics/ (Accessed: 02 June 2024).
2. Fisher, C. (2018). Cloud versus On-Premise Computing. *American Journal of Industrial and Business Management*, *08*(09), 1991–2006. <https://doi.org/10.4236/ajibm.2018.89133>
3. B. Patel, P. H., & Kansara, P. N. (2021). Cloud Computing Deployment Models: A Comparative Study. *International Journal of Innovative Research in Computer Science & Technology*, *9*(2), 45–50. <https://doi.org/10.21276/ijircst.2021.9.2.8>
4. Bondenson, P. (2021). *Transitioning from on-premise computing to cloud computing*[Degree's project available at [https://uu.diva-portal.org/smash/get/diva2:1752929](https://uu.diva-portal.org/smash/get/diva2:1752929/FULLTEXT01.pdf)]. Uppsala University.
5. *Google Cloud Documentation. (n.d.). Google Cloud.*[*https://cloud.google.com/docs*](https://cloud.google.com/docs)(Accessed: 13 June 2024).
6. *Home*. (n.d.). Docker Documentation. <https://docs.docker.com> (Accessed: 10 July 2024).
7. Kumar, S. (2022, April 1). Basics of Container Isolation. *Medium*. <https://blog.devgenius.io/basics-of-container-isolation-5eabdb258409>
8. Fernandez, G. (2024, January 24). Docker in Mac M1/M2 - Colima! *Daemon | Award-Winning Technology Consultancy*. <https://www.dae.mn/blog/docker-in-mac-m1/m2-colima>
9. Saturn Cloud. (2023, June 13). How to Get Current CPU GPU and RAM Usage of a Particular Program in Python | Saturn Cloud Blog. *Saturn Cloud | #1 Rated ML Platform*. <https://saturncloud.io/blog/how-to-get-current-cpu-gpu-and-ram-usage-of-a-particular-program-in-python/>
10. *PyPI Docs. (n.d.). PyPI Docs.*[*https://docs.pypi.org*](https://docs.pypi.org)(Accessed: 10 June 2024).
11. Bala Priya, C. (2024, February 19). Introduction to Memory Profiling in Python - KDnuggets. *KDnuggets*. <https://www.kdnuggets.com/introduction-to-memory-profiling-in-python>
12. Olivieri, R. (2023, July 27). How to use a Python multiprocessing module | Red Hat Developer. *Red Hat Developer*. <https://developers.redhat.com/articles/2023/07/27/how-use-python-multiprocessing-module#error=login_required&amp;state=3d516c77-01ef-491e-8237-c4bf14a0f7eb>