

Building an ML Driven System  
for Real-Time Code-Performance Monitoring

by

Mikhail Delyusto

A thesis submitted to the

Graduate Committee of Ramapo College of New Jersey

in partial fulfillment of the requirements for the degree of

Master of Science in Data Science

Spring, 2023

Committee Members:

Sourav Dutta Ph.D., Advisor

Debbie Yuster Ph.D., Reader

Scott Frees Ph.D., Reader

## **COPYRIGHT**

© Mikhail Delyusto 2023



## Acknowledgments.

I would like to thank all the professors who helped me to fulfill my dreams and learn data science throughout the course of my master's studies at Ramapo College of New Jersey.

Special thanks to Professor Dutta for agreeing to help me finalize this course as my thesis advisor. Your knowledge, expertise and insight helped me to better understand my thesis and is greatly appreciated. Thank you for your time and effort!

# Table of Contents

Acknowledgements

Table of contents

List of tables

List of figures

Abstract 1

## CHAPTER

1. Introduction	3
2. Background	6
3. Methodology	10
- Dataset approach	13
- Moving average	25
- ARIMA model	28
- Random Forest	32
4. Conclusions	36
Future work	40

References 42

Appendices 44

## List of tables

Table 1. Data about real executions, parsed from log files.	14-18
Table 2. Results of a batch of 20 jobs execution (label 0 - good data)	20-21
Table 3. Results of a batch of 20 jobs execution (label 1 - bad data)	22
Table 4. List of features in the dataset.	22-23
Table 5. Results of rolling mean predictions for job 0.	25
Table 6. Classification report for moving average is the same for every job.	27
Table 7. Classification report for the ARIMA model for job 0.	29
Table 8. List of accuracies of label 1 predictions for the ARIMA model along with average accuracy and average F1-score.	31
Table 9. Classification report for Random Forest Classifier on full dataset (20 jobs).	33
Table 10. Classification report for Random Forest Classifier on job 0.	34
Table 11. A comparison of the models' results	37-38

## List of figures

Figure 1. Current state of testing versus future state.	12
Figure 2. Confusion matrix of the predictions for job 0.	26
Figure 3. Confusion matrix for the ARIMA model for job 0.	30
Figure 4. Confusion matrix for Random Forest Classifier on full dataset (20 jobs).	33
Figure 5. Confusion matrix for Random Forest Classifier on job 0.	34
Figure 6. Example of metrics collected from one log file.	40



## Abstract

This project is a part of a multidirectional attempt to increase quality of the software and data product that is being produced by Science and Engineering departments of Aetion Inc., the company that is transforming the healthcare industry by providing its partners (major healthcare industry players) with a real-world evidence generation platform, that helps to drive greater safety, effectiveness, and value of health treatments. Large datasets (up to 100Tb each) of healthcare market data (for example, insurance claims) get ingested into the platform and get transformed into Aetion's proprietary longitudinal format.

This attempt is being led by the Quality Engineering Team and is envisioned to move away from conventional testing techniques by decoupling different moving parts and isolating them in separate, maintainable and reliable tools.

A subject of this thesis is a particular branch of a large quality initiative that will be helping to continuously monitor a number of metrics that are involved in execution of the two most common types of jobs that run on Aetion's platform: cohorts and analyses. These jobs may take up to a few hours to generate depending on the size of a dataset and the complexity of an analysis.

Implemented, this monitoring system would be supplied with a feed of logs that contain certain data points, like timestamps. Enhanced with a built-in algorithm to set a threshold on the metrics and notify its users (stakeholders from Engineering and Science)

when said threshold is exceeded, would be a game-changing capability in Aetion's quality space. Currently, there is no way to say if any given job is taking more or, otherwise, significantly less time and most of the defects get identified in upper environments (including production).

The issues identified in upper environments are the costlier of all the types and, by different industry considerations, can cost \$5000 - \$10000 each.

As a result of implementing said system we would expect a steep decrease in a number of issues in upper environments, as well as an increase in release frequency, that the organization will greatly benefit from.

## Introduction

In a modern world of software development testing has become a very powerful discipline as it does allow us to iterate frequently when adding new features or updating existing ones.

It is being established as industry standard to build and maintain automation frameworks that run against the new features that usually come as code or configuration changes that are built and deployed into certain environments.

A simple example of how these testing automation frameworks can be integrated into the Software Development Life Cycle (SDLC):

1. A pull request (PR) containing a code or configuration change for a repository is open at one of the code version control systems like GitHub or Bitbucket;
2. The repository gets built and deployed into one of the lower environments (pre-production environments);
3. A test automation suite that contains tests that covers most (ideally 100%) functionality runs in that environment;
4. The result of the suite is used as a mark for making a programmatic go/no go decision of whether the code is okay to be added.

The way these approaches are designed is that the test automation suite always runs against the environments that have a master (main) branch with a certain allowed degree (most often, 0%) of failures. Commonly referred to as a “green” run, it is a state that becomes a baseline for a validation of an incoming code change. A rationale used in this

approach is relatively simple: if the tests pass for the code in the master branch and fail in a feature branch that means that that particular code is breaking existing functionality. Most often pull requests that contain these breaking changes get rejected and reworked.

The testing automation frameworks have become very popular and diverse in recent years. There are many different approaches and programming languages that are used to build. The most common tools among these are Java + Selenium based test automation tools that allow for running of up to tens of thousands of tests in parallel in standalone internet browser windows. [1]

Building, maintaining and running these tests, though proven to be efficient in locating bugs, are very expensive to run:

- Tests are hard to build and maintain:
  - architecture cost - need to hire a test automation architect as the test infrastructure can be very complex;
  - organizational effort cost:
    - every test should be properly written against acceptance criteria;
    - tests should be timely updated if any of the acceptance criteria change;
    - need to maintain a team of quality assurance engineers;
- As these test suites may run against many of the feature branches deployed in many environments in many iterations (until all the defects get fixed), the compute cost can become exorbitant as the number of tests grows and the velocity of the releases increases.

Many organizations struggle with proper set up in this field, resulting in:

- Lack of engineering skills and leadership lead to building testing frameworks that are:
  - over-inflated and therefore requiring extensive maintenance;
  - consist of “flaky” test scenarios that intermittently fail for no apparent reason and require additional effort to evaluate;
- The rate of spilling defects into the production increases.

A study by the National Institute of Standards and Technology found that it is five times (5x) more expensive to fix a defect during the application development/coding phase, ten times (10x) more expensive during integration testing, fifteen times (15x) more expensive during customer beta testing, and an astounding thirty times (30x) more expensive to fix post release than if it is addressed in the requirements development and analysis phase. [2]

The higher velocity an organization is getting at with the software releases, the more overhead cost it incurs while utilizing the traditional approach in creating testing infrastructure for end to end testing in user interface (UI).

## Background

A scope of this work will be around a methodology for designing a tool that would help to significantly reduce the number and duration of the test automation suite that is being used by the engineering organization at Aetion, Inc., the company that is transforming the healthcare industry by providing its partners (major healthcare industry players) with a real-world evidence generation platform, that helps to drive greater safety, effectiveness, and value of health treatments.

Aetion's existing flagship product, **Substantiate** (formerly/also known as "AEP" (Aetion Evidence Platform)) is a complex tool used by epidemiologists to conduct research and analysis based on real world evidence (RWE). [3] **Substantiate** in all its complexity is a powerful tool, capable of generating regulatory-grade analyses.

### Key Concepts:

- Environment: refers to the various collections of AWS resources provisioned for specific purposes:
  - Lower environments (CI, QA);
  - Upper environments (CQA, Production).
- Instance: an instance is a client/purpose-specific instance of an application within an environment;
- Jobs: refers to the operations performed against the patient datasets that are ingested and made available within a customer's instance. There are two main types of jobs:

- Cohort Jobs: These jobs generate a cohort from a cohort definition, essentially filtering all the patients within a dataset to create a subset of patients that matches a set of specified criteria (e.g., male patients who were diagnosed with rheumatoid arthritis during a given timeframe).
- Analysis Jobs: These jobs execute an analysis based on an analysis plan, against a cohort which has already been generated. Analysis jobs are complex; there are different subtypes of analysis with different configurations to perform a variety of different calculations and comparisons. A simple example is an analysis to show the average cost of treatment for the patients in the specified cohort for the period of 5 years following their RA diagnosis, grouped by patient age and geographic region.
- Populations and Cohorts: analyses are executed against cohorts, groups of patients which match a set of criteria specified in a cohort definition.
- Outcomes and Analysis Plans:
  - an Analysis Plan represents the definition of a job that needs to be executed against a generated cohort;
  - an Outcome is technically part of an Analysis Plan – it answers the question “what are you trying to measure?”.
- Results: once an Analysis Job completes, results are made available for display, exporting, etc.

A regular (industry standard) automated end to end test scenario for the purposes of validating proper functionality of the platform from the user interface perspective would include:

1. Logging into the platform;
2. Creating measures (essentially, filters for data);
3. Defining a cohort and triggering its generation;
4. Waiting on the web page for cohort generation;
5. Defining parameters for analysis and triggering its generation;
6. Waiting on the web page for analysis generation;
7. Navigating to the results section and validating those against some predefined baseline.

While this will be an exact replication of the actual workflow and this is what and how most of the users of the platform would be using the platform for, there are many disadvantages of replicating it to run in an automated fashion:

- Steps 4 and 6 can take any amount of time, completely depending on the size of the dataset and number of measures, outcomes, etc. The more robust the scenario is in terms of the input parameters for cohorts and analyses, the more it resembles real tasks that the users would be stressing the system with, but respectively the longer it will take to generate.
- Most of the valid cases of the end to end scenarios have long execution times and can run for up to a few hours.
- Staying on the same page in the internet browser spun up by UI test automation tool for up to a few hours polling every second, waiting for the generation status to display as “Generated” is the most brittle and expensive solution. [4]



As a part of Quality Assurance practices we still need to have a continuous confirmation of the platform's ability to receive a user's input as well as generate and display a desired outcome.

In the scope of this project we would like to elaborate an approach that would significantly reduce time and cost of the UI piece by effectively decoupling parts of the process where generation of jobs (cohorts and analyses) takes longer than 10-15 minutes into a separate testing tool.

## Methodology

In our proposed solution we would like to split long running tests suites into two different categories of testing capabilities:

1. On one hand we still would like to run end to end scenarios to make sure that all the components of the system as well as UI respond well to a user input. To achieve scalability of these suites in terms of time and cost we will select the input that would result in extra small cohorts, limiting the total time of test suite execution to 10-15 minutes. This will allow us to:
  - a. incorporate this lightweight suite into any CI process;
  - b. produce test results for faster feedback;
  - c. make go/no go decisions on the pull requests;
  - d. significantly decrease compute cost and maintenance.
2. We would still like to execute valid end to end scenarios (long running cohorts and analyses), but orthogonal to the regular (lightweight) test automation suite. The test suite will bear the following distinct characteristics:
  - a. It will consist of two dependent structures:
    - i. First will be a UI automation test suite that works as a cron job, effectively just starting a set of distinct jobs (cohorts and analyses with predefined input); this automation just triggers execution, it does not wait for it to finish nor does it perform any validations;

- ii. A python script that will start as soon as the UI automation triggers a set of the predefined jobs, that predicts the execution times for each and every job based on the historical data.
- b. This way the system will acquire the predictions and will use those to:
- i. Stop executions as soon as the systems learns that those are not going to be executed within a predicted time frame;
  - ii. Send a signal to the maintainers of the repositories that contain the code for cohorts and analyses' generation to check the latest commits for possible errors.

In the scope of this project we will concentrate on expanding on the methodology around point 2.a.ii. (predicting execution time frames for a set of the predefined jobs).

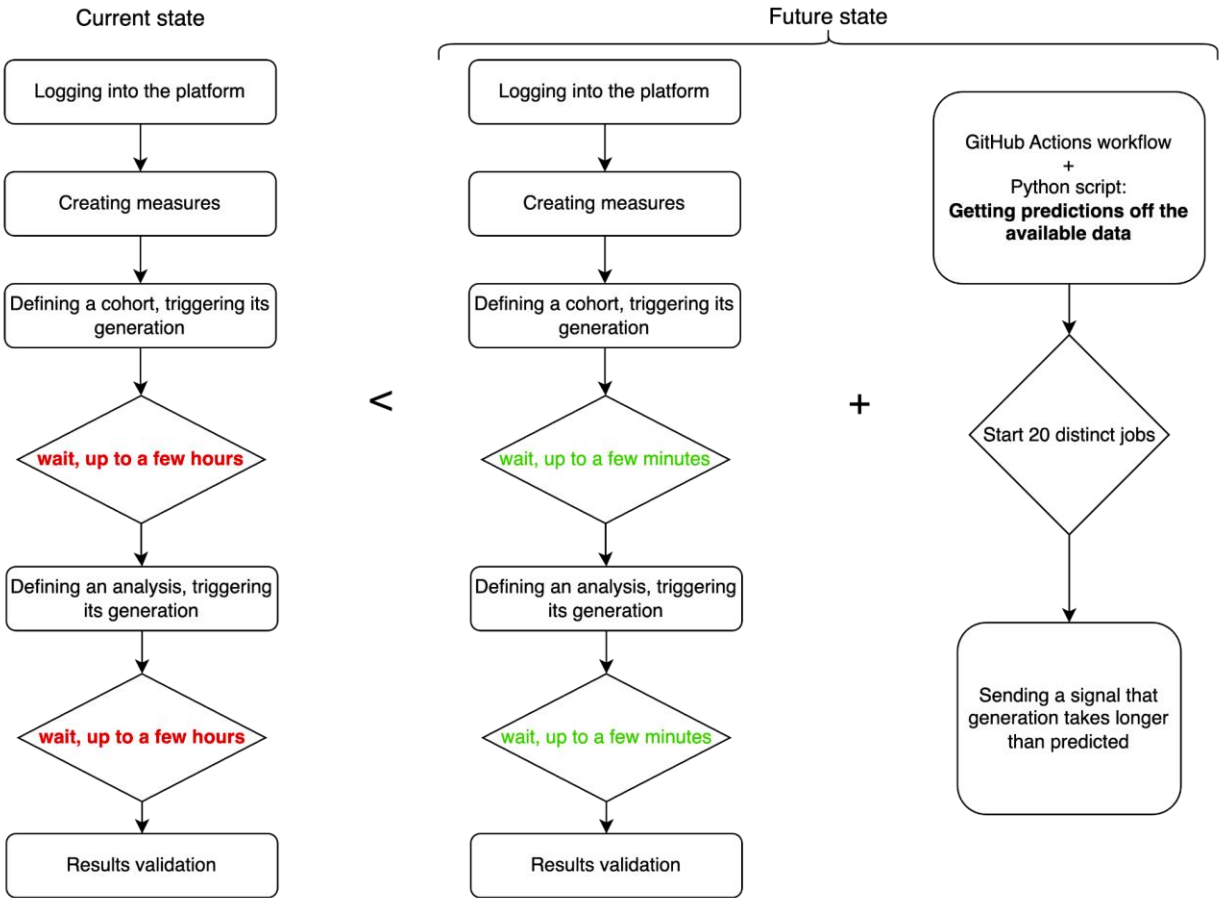


Figure 1. Current state of testing versus future state.

## Dataset approach

For the purposes of creating data to train our models and predict on, we used the historical data of the past executions in the lower environment CI.

The baseline for data comes as a result of parsing ~5000 log files, filtering those by the following criteria:

- log file should contain **{"metrics": ...}** JSON, effective filtering out the log files pertaining to the jobs ran before 10/2022, when those metrics about the executions started to be collected;
- log file should contain **[jobSucceeded]**, effectively filtering out jobs that did not run successfully;
- **cohortid** or **analysisid** are present in {"metrics": ...} JSON (Appendix A and B)

While parsing the log files, we collected the following metrics:

1. Execution time, as a difference in time elapsed between the timestamp in the first line of the log JSON and the timestamp in the last line of the log JSON:

$$\textit{End execution time} - \textit{Start execution time}$$

(Appendix C)

2. Number of occurrences of {"metrics": ...} JSON in one file.
3. Metrics' JSON, containing job and system info, example:

```
{"metrics":[{"size":56623104000,"committed":1.2847222222222223E-4,"init":4.513888888888889E-
```

```
5,"max":0.0021551649305555556,"used":1.2815574363425925E-
4},{ "count":115,"peak":115,"started":115},{ "sysavg":0.83,"processors":16,"cpuavg":0.051
875}], "jobid":129726,"jobtype":"GENERATE_COHORT","cohortid":82097,"anchor":"_JM
T_"}

```

We identified the unique jobs using pairs of the corresponding values of objects **jobtype** and **cohortid** (or **analysisid**). Example of a unique job id: **GENERATE\_COHORT\_82097**.

Parsing all the available log files, we collected the following existing historical data about past executions:

Job ID	Date	Time	Execution time (s)
GENERATE_COHORT_82097	2022-11-14	4:20:52	54.23
GENERATE_COHORT_82097	2022-11-14	4:04:00	57.71
GENERATE_COHORT_82097	2022-11-14	4:33:32	57.90
DESCRIPTIVE_ANALYSIS_166884	2022-11-09	17:52:15	44.80
DESCRIPTIVE_ANALYSIS_166884	2022-11-08	14:14:56	50.57
DESCRIPTIVE_ANALYSIS_166884	2022-11-07	10:01:31	59.85
DESCRIPTIVE_ANALYSIS_166884	2022-11-04	19:42:58	41.50
GENERATE_COHORT_82198	2022-11-14	20:55:30	59.27
GENERATE_COHORT_82198	2022-11-14	17:46:38	59.39
GENERATE_COHORT_82198	2022-11-14	21:01:07	58.91
GENERATE_COHORT_82198	2022-11-14	17:47:26	58.92
ML_SUBGROUPS_ANALYSIS_143642	2022-12-08	21:23:12	45.17

ML_SUBGROUPS_ANALYSIS_143642	2022-12-08	21:40:48	44.33
ML_SUBGROUPS_ANALYSIS_143642	2022-11-18	19:32:17	44.25
ML_SUBGROUPS_ANALYSIS_143642	2022-12-06	19:50:48	37.00
ML_SUBGROUPS_ANALYSIS_143642	2022-12-06	19:25:57	39.50
ML_SUBGROUPS_ANALYSIS_143642	2022-12-06	19:33:33	30.00
ML_SUBGROUPS_ANALYSIS_143642	2022-12-07	22:19:42	46.71
ML_SUBGROUPS_ANALYSIS_143642	2022-12-06	20:01:44	50.17
ML_SUBGROUPS_ANALYSIS_143642	2022-12-07	22:35:33	43.67
ML_SUBGROUPS_ANALYSIS_143642	2022-12-07	22:10:08	42.80
ML_SUBGROUPS_ANALYSIS_143642	2022-12-08	18:43:18	44.83
ML_SUBGROUPS_ANALYSIS_143642	2022-12-07	21:09:10	44.50
GENERATE_COHORT_86451	2022-12-07	22:13:00	56.67
GENERATE_COHORT_86451	2022-12-08	22:06:05	56.67
GENERATE_COHORT_86451	2022-12-08	22:08:26	56.75
GENERATE_COHORT_86451	2022-12-08	22:12:27	55.26
GENERATE_COHORT_86451	2022-12-08	22:48:19	55.95
GENERATE_COHORT_86451	2022-12-12	19:53:05	57.75
GENERATE_COHORT_86451	2022-12-07	22:11:28	57.84
GENERATE_COHORT_86451	2022-12-09	8:51:39	56.52
DESCRIPTIVE_ANALYSIS_143642	2022-12-08	21:22:19	52.25
DESCRIPTIVE_ANALYSIS_143642	2022-12-07	22:18:28	54.22
DESCRIPTIVE_ANALYSIS_143642	2022-12-07	22:35:03	52.25
DESCRIPTIVE_ANALYSIS_143642	2022-11-14	21:10:01	45.33

DESCRIPTIVE_ANALYSIS_143642	2022-12-06	19:22:24	50.12
DESCRIPTIVE_ANALYSIS_143642	2022-12-06	19:20:04	53.12
DESCRIPTIVE_ANALYSIS_143642	2022-12-07	22:08:44	45.00
DESCRIPTIVE_ANALYSIS_143642	2022-12-08	21:40:17	50.62
DESCRIPTIVE_ANALYSIS_143642	2022-12-07	21:06:36	53.12
DESCRIPTIVE_ANALYSIS_143642	2022-12-08	18:40:46	50.88
AUTO_FEAT_GEN_167736	2022-11-19	11:16:08	59.57
AUTO_FEAT_GEN_167736	2022-11-19	11:13:28	59.54
AUTO_FEAT_GEN_167736	2022-11-19	11:20:01	59.08
AUTO_FEAT_GEN_167736	2022-11-18	17:56:35	47.00
GENERATE_COHORT_86680	2022-12-14	1:59:06	42.67
GENERATE_COHORT_86680	2022-12-14	1:10:49	43.00
GENERATE_COHORT_86680	2022-12-08	22:47:09	42.00
GENERATE_COHORT_86680	2022-12-08	21:34:03	42.33
GENERATE_COHORT_86680	2022-12-14	1:25:57	41.00
GENERATE_COHORT_81958	2022-11-11	22:33:54	57.56
GENERATE_COHORT_81958	2022-11-11	22:08:04	57.11
GENERATE_COHORT_81958	2022-11-11	22:25:40	55.08
GENERATE_COHORT_87608	2022-12-15	18:26:55	32.00
GENERATE_COHORT_87608	2022-12-15	18:28:36	32.67
GENERATE_COHORT_87608	2022-12-15	18:25:46	43.00
GENERATE_COHORT_87608	2022-12-15	18:25:35	46.67
DESCRIPTIVE_ANALYSIS_170796	2022-12-08	22:23:11	54.00



DESCRIPTIVE_ANALYSIS_170796	2022-12-09	19:15:25	54.29
DESCRIPTIVE_ANALYSIS_170796	2022-12-12	19:53:46	56.67
DESCRIPTIVE_ANALYSIS_170796	2022-12-09	21:25:32	56.00
DESCRIPTIVE_ANALYSIS_170796	2022-12-09	20:14:46	55.50
GENERATE_COHORT_80780	2022-11-07	16:26:54	55.90
GENERATE_COHORT_80780	2022-11-07	16:26:31	55.56
GENERATE_COHORT_80780	2022-11-07	16:36:03	59.79
GENERATE_COHORT_80364	2022-11-02	21:50:12	55.62
GENERATE_COHORT_80364	2022-11-02	20:57:02	55.76
GENERATE_COHORT_80364	2022-11-02	19:16:21	59.68
GENERATE_COHORT_80364	2022-11-02	23:12:40	56.42
GENERATE_COHORT_80364	2022-11-02	23:11:57	57.10
GENERATE_COHORT_80674	2022-11-07	1:11:50	54.91
GENERATE_COHORT_80674	2022-11-07	1:11:18	57.94
GENERATE_COHORT_80674	2022-11-07	1:12:20	57.15
GENERATE_COHORT_80608	2022-11-05	4:18:14	55.15
GENERATE_COHORT_80608	2022-11-05	4:18:16	57.47
GENERATE_COHORT_80608	2022-11-05	4:18:45	57.70
DESCRIPTIVE_ANALYSIS_172248	2022-12-16	21:17:48	38.25
DESCRIPTIVE_ANALYSIS_172248	2022-12-16	21:30:02	47.50
DESCRIPTIVE_ANALYSIS_172248	2022-12-16	21:29:33	55.40
DESCRIPTIVE_ANALYSIS_166783	2022-11-02	22:09:30	44.25
DESCRIPTIVE_ANALYSIS_166783	2022-11-02	21:48:22	43.60

DESCRIPTIVE_ANALYSIS_166783	2022-11-02	23:11:57	44.25
DESCRIPTIVE_ANALYSIS_166783	2022-11-02	21:46:00	43.40
DESCRIPTIVE_ANALYSIS_168097	2022-12-01	0:59:09	26.50
DESCRIPTIVE_ANALYSIS_168097	2022-12-01	1:14:53	25.00
DESCRIPTIVE_ANALYSIS_168097	2022-12-01	1:06:30	27.50
GENERATE_COHORT_84908	2022-12-07	22:25:10	51.67
GENERATE_COHORT_84908	2022-12-07	22:25:01	54.67
GENERATE_COHORT_84908	2022-12-07	22:56:48	48.67
GENERATE_COHORT_86416	2022-12-07	19:13:28	17.00
GENERATE_COHORT_86416	2022-12-07	18:53:15	17.00
GENERATE_COHORT_86416	2022-12-07	18:43:22	17.50

Table 1. Data about real executions, parsed from log files.

As we can see, there are 20 distinct jobs that have run intermittently. Unfortunately, there was no automation in place that would run the same jobs every day, which would be ideal in our case. The number of execution times varies but is not enough for us to make any predictions on.

For the purposes of this project we generated synthetic data off the real data, using its mean and standard deviation.

We assumed that when implemented, this tool would consume data of daily executions of these jobs and should be able to have enough data points to predict a baseline for the future executions.

For the purposes of finding the best model of predicting that baseline, we generated 365 synthetic data points for every job.

To test our algorithm on whether it is capable of properly predicting the outcome, we also ingested noise following a Gaussian distribution using the following approach:

1. Calculated the means and standard deviation of the distributions for every distinct job. Example:

GENERATE\_COHORT\_82097:

Job number=0,  
min observation=54.23,  
max observation=57.90,  
mean=56.613333,  
standard deviation=1.687055

2. Used the mean and standard deviation to generate 54 noised data points (15%) of the total dataset;
3. In order to emulate noised data properly to better resemble the outliers, we exclude noised data that falls inside the range of:  
  
[one standard deviation - min of the regular data ... max of the regular data + one standard deviation].
4. Randomly injected noised data points into the dataset the way it would resemble a logical flow:
  - a. 20 different jobs get run every day by the system that use the same code;

- b. if there has been a change in the codebase that affects performance, it should affect the performance of all 20 jobs;
  - c. therefore we inject the noised data points using the blocks of 20 noised data for the same date.
5. Assigned labels 0 and 1 to every data point depending on the type (regular data, noised data respectively):

Job number	Date (integer)	Execution time	Label
0	20220323	56.98	0
1	20220323	51.97	0
2	20220323	59.35	0
3	20220323	40.52	0
4	20220323	56.21	0
5	20220323	53.41	0
6	20220323	59.20	0
7	20220323	42.26	0
8	20220323	55.56	0
9	20220323	35.60	0
10	20220323	55.83	0
11	20220323	55.91	0
12	20220323	58.25	0
13	20220323	55.92	0
14	20220323	57.61	0

15	20220323	53.39	0
16	20220323	43.68	0
17	20220323	26.94	0
18	20220323	54.15	0
19	20220323	17.27	0

Table 2. Results of a batch of 20 jobs execution (label 0 - good data)

Job number	Date (integer)	Execution time	Label
0	20220324	52.25	1
1	20220324	67.92	1
2	20220324	58.65	1
3	20220324	55.90	1
4	20220324	53.91	1
5	20220324	38.83	1
6	20220324	71.84	1
7	20220324	43.73	1
8	20220324	59.08	1
9	20220324	52.22	1
10	20220324	58.67	1
11	20220324	52.82	1
12	20220324	52.92	1
13	20220324	58.75	1

14	20220324	59.44	1
15	20220324	28.77	1
16	20220324	45.12	1
17	20220324	28.64	1
18	20220324	46.58	1
19	20220324	17.70	1

Table 3. Results of a batch of 20 jobs execution (label 1 - bad data)

As a result of these transformations, we prepared a dataset that contains simulated data of one year of the observations for the execution times of 20 distinct jobs.

Total number of observations = 7300.

Features:

	Type	Range
job_id	Numeric	0...19
date	Numeric, continuous	20220314...20230313
execution_time	Numeric	-

is_bad	Binary	0 or 1
--------	--------	--------

Table 4. List of features in the dataset.

The dataset is created this way, the farthest the execution times are from the mean in both directions, the higher chance there is that there was an invasive change in the codebase that affected the execution in some way. Lesser execution time does not always mean better as it can mean that the code is now skipping some parts of the data that can lead to an incorrectly produced cohort or analysis and therefore an incorrect real world evidence.

In this regard, “good” will be referred to as a data point that does not deviate from the mean drastically, and “bad” will be referred to as the data point that does.

This dataset works as an input into our following models that are utilized in the scope of this project to predict whether data of new executions is good or bad:

- Moving average;
- ARIMA model;
- Random Forest.

## Moving average

The first and most intuitive approach to predict the next value of the execution time will be to calculate a rolling mean of previous periods in a time series.

For a sequence of values, we calculate the simple moving average at time period  $t$  as follows: [5]

$$SMA_t = \frac{x_t + x_{t-1} + x_{t-2} + \dots + x_{M-(t-1)}}{M}$$

To calculate the rolling mean of our dataset we used the following syntax:

```
job_features['volume_moving'] =  
job_features['execution_time'].rolling(window=7).mean(),
```

where **job\_features** is a subset of the total data that contains data points only pertaining to one particular job.

Applying rolling mean methodology:

- applying this rolling mean on all data;
- split results into training and testing part (75% train, 25% test);
- applying a function to assign labels (0 or 1) to test data;
- matching the predicted labels with the original labels:
  - “good”: predicted data point is within a range of “good” source data;



- “bad”: predicted data point is:
  - less than (minimum of good data minus one standard deviation of good data);
  - greater than (maximum of good data plus one standard deviation of good data).
- calculating the number of properly predicted labels.

Results:

min and max of good source data	54.25	57.9
min and max of bad source data	50.75	61.81
standard deviation of good data	0.98701	
number of matched predictions of bad data (1)	0	
number of all matches (1)	77	
accuracy	84%	

Table 5. Results of rolling mean predictions for job 0.

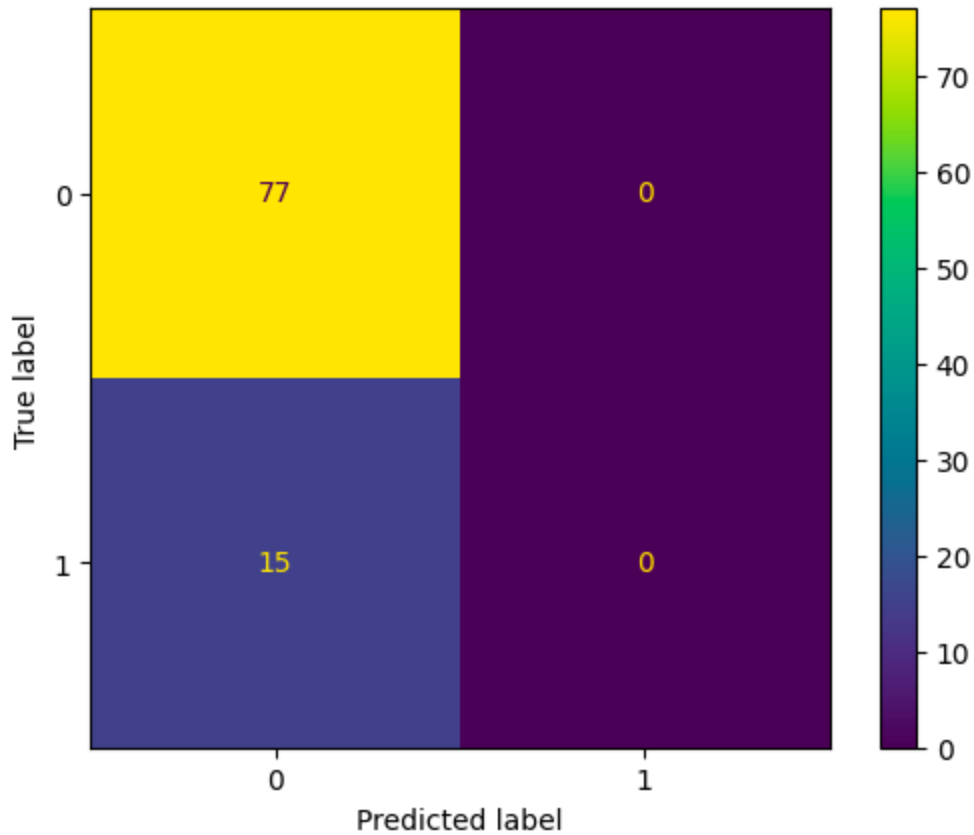


Figure 2. Confusion matrix of the predictions for job 0.

Running the algorithm on 20 jobs provides us with average accuracy of 83.7%, which correlates to:

- training dataset size =  $365 * 25\% = 92$  data points;
- in all cases the rolling mean was able to predict 0-labeled data, which is 77 data points;
- $77 / 92 = 0.836956 \%$

label	precision	recall	f1-score	support
0	0.84	1.00	0.91	77
1	0.00	0.00	0.00	15
accuracy			0.84	92
macro avg	0.42	0.50	0.46	92
weighted avg	0.70	0.84	0.76	92

Table 6. Classification report for moving average is the same for every job.

Most notable inferences on the results of this model are:

- All good data is predicted properly;
- This model does not predict any outliers.

## ARIMA model

ARIMA stands for Autoregressive integrated moving average, and is used to comprehend the data or to forecast upcoming series points when fitted to time series data.

We utilized the **ARIMA(1,1,0)** model suggested for non-seasonal data in “How to Create an ARIMA Model for Time Series Forecasting in Python”. [6]

“ARIMA(1,1,0) = differenced first-order autoregressive model: If the errors of a random walk model are autocorrelated, perhaps the problem can be fixed by adding one lag of the dependent variable to the prediction equation--i.e., by regressing the first difference of  $Y$  on itself lagged by one period. This would yield the following prediction equation:

$$\hat{Y}_t - Y_{t-1} = \mu + \phi_1(Y_{t-1} - Y_{t-2})$$

$$\hat{Y}_t - Y_{t-1} = \mu$$

which can be rearranged to

$$\hat{Y}_t = \mu + Y_{t-1} + \phi_1(Y_{t-1} - Y_{t-2})$$

This is a first-order autoregressive model with one order of nonseasonal differencing and a constant term--i.e., an ARIMA(1,1,0) model.” [7]

We used the following approach while utilizing ARIMA model to make predictions, for every distinct job we:

- split the data into training and testing part (75% train, 25% test);
- use walk-forward validation:
  - looping over test data, starting from data point with the index 273;

- predicting next data point on the train data (i.e. **history** dataframe);
- adding data point with index that we predicted to **history** dataframe;
- predicting next data point on a **history + 1**;
- this way the prediction of test[x] is made using ARIMA model that is trained on train[273 + x];
- collect all predictions into a list (size = 92, 25% of 365);
- convert the predictions into a (0, 1) form using minimum, maximum and standard deviation of the good data as a rule to assign labels:
  - if a predicted value is less than (minimum of good data - one standard deviation) and is greater than (maximum of good data + one standard deviation), label 1 gets assigned, 0 otherwise.

Results:

label	precision	recall	f1-score	support
0	0.83	0.91	0.87	77
1	0.12	0.07	0.09	15
accuracy			0.77	92
macro avg	0.48	0.49	0.48	92
weighted avg	0.72	0.77	0.74	92

Table 7. Classification report for the ARIMA model for job 0.

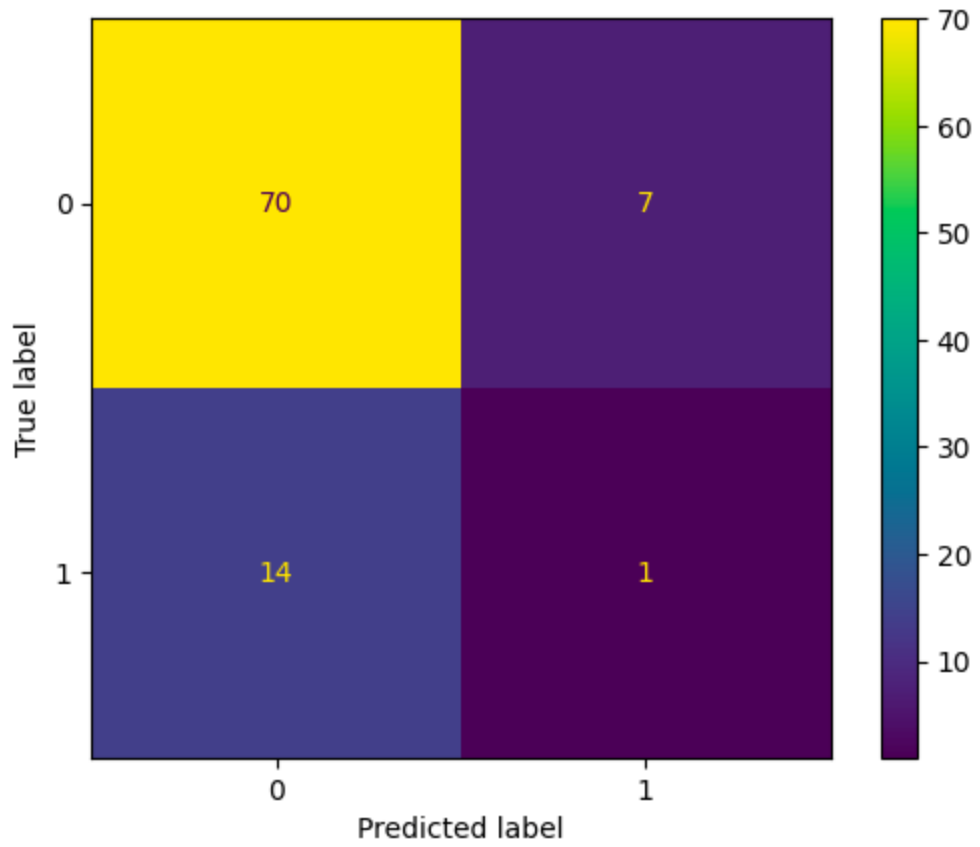


Figure 3. Confusion matrix for the ARIMA model for job 0.

accuracies for label 1, per job, %			
0	9.0	10	3.0
1	2.0	11	3.0
2	2.0	12	7.0
3	2.0	13	0.0
4	2.0	14	1.0
5	5.0	15	4.0
6	5.0	16	2.0
7	1.0	17	7.0
8	2.0	18	3.0
9	4.0	19	2.0
average accuracy:			3.42%
average F1-score:			1.966

Table 8. List of accuracies of label 1 predictions for the ARIMA model along with average accuracy and average F1-score.

## Random Forest.

Attempted application of Random Forest Classifier, which is a supervised machine learning algorithm that would make sense to apply in our case due to the nature of the input data and its simplicity and flexibility.

“Random Forest is a group of decision trees. However, there are some differences between the two. A decision tree tends to create rules, which it uses to make decisions. A random forest will randomly choose features and make observations, build a forest of decision trees, and then average out the results.

In a business, a random forest algorithm could be used in a scenario where there is a range of input data and a complex set of circumstances. For instance, identifying when a customer is going to leave a company. Customer churn is complex and usually involves a range of factors: cost of products, satisfaction with the end product, customer support efficiency, ease of payment, how long the contract is, extra features offered, as well as demographics like gender, age, and location. A random forest algorithm creates decision trees for all of these factors and can accurately predict which of the organization’s customers are at high risk of churn.” [8].

In our case we will predict labels 0 and 1.

We applied Random Forest Classifier [9] in two different ways:

1. Apply Random Forest on the whole data (unfiltered dataset containing data points for all 20 jobs):
  - splitting data into training and testing datasets (75% train, 25% test);
  - fitting a Random Forest Classifier, getting an average score of 0.87%;



- predicting on the testing dataset.

Results:

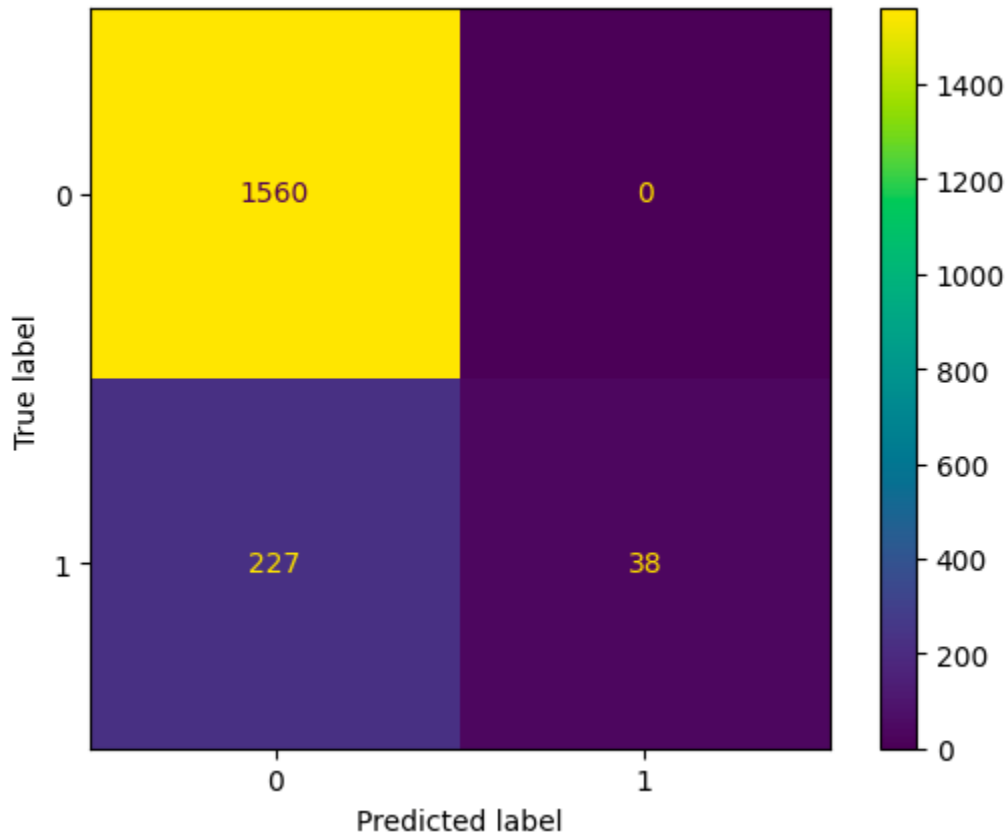


Figure 4. Confusion matrix for Random Forest Classifier on full dataset (20 jobs).

label	precision	recall	f1-score	support
0	0.87	1.00	0.93	1560
1	1.00	0.14	0.25	265
accuracy			0.88	1825
macro avg	0.94	0.57	0.59	1825
weighted avg	0.89	0.88	0.83	1825

Table 9. Classification report for Random Forest Classifier on full dataset (20 jobs).

- Apply Random Forest on the filtered data (running a classifier for every job in isolation).

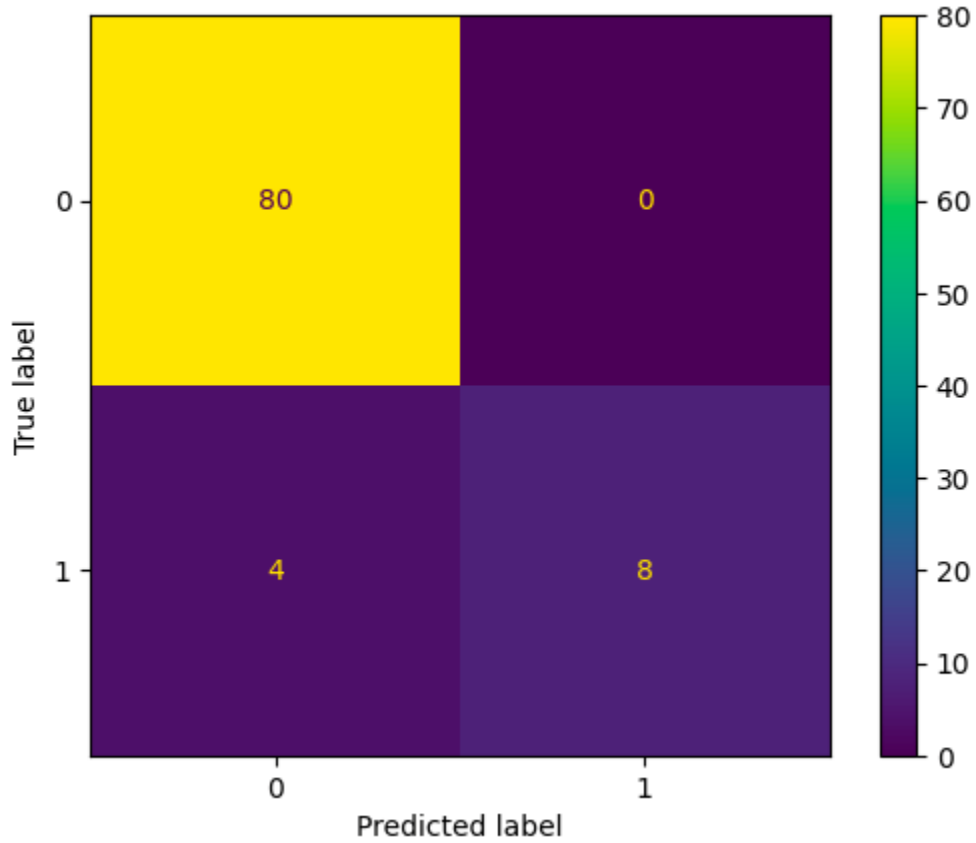


Figure 5. Confusion matrix for Random Forest Classifier on job 0.

label	precision	recall	f1-score	support
0	0.95	1.00	0.98	80
1	1.00	0.67	0.80	12
accuracy			0.96	92
macro avg	0.98	0.83	0.89	92
weighted avg	0.96	0.96	0.95	92

Table 10. Classification report for Random Forest Classifier on job 0.

Running 20 iterations of Random Forest Classifier (one for every job, splitting the dataset into subsets containing only 365 values for one specific job each) returned the following averages:

- average precision score: 95.92%
- average F1-score: 80.15%

## Conclusions

In the scope of this project we attempted to expand on a possibility to find a new approach for creating new testing capabilities that would significantly benefit organizations in certain cases.

The traditional testing techniques become insufficient and an impediment to the release cycles in the cases where there is a significant delay between the events that get validated through the user interface.

We considered an example of testing at Aetion, Inc, where Aetion's proprietary real world evidence generation platform, capable of generating regulatory-grade analyses, suffers from being built off the traditional approaches. Long running unstable testing jobs, a need to incur heavy maintenance cost of the testing framework are among the factors that slow the delivery speed of the product and drive up the cost.

We postulated that this testing capability can be split into two orthogonal processes, where we still would be able to check if a user can create input for those long running jobs, as well as start their generation. Along with that, there will be another tool created that would be analyzing a historical data of the jobs (cohorts, analyses) that had already been generated in the past and suggesting a baseline on the execution times.

In the scope of this project we also attempted to elaborate an approach for predicting the execution times of the future generations of these jobs in order to have a foundation for the testing tool that we envision.

We proceeded with creating a synthetic dataset of a year of observations for 20 distinct jobs (cohorts and analyses) in order to be able to have enough data points to successfully train a few prediction models. In order to create that dataset, we used the population distribution characteristics of the data collected from ~5000 log files of available data of real executions.

In order to properly train our models to find outliers we randomly injected 15% of noised data that follows the population distribution characteristics of the real data, but artificially adjusted to not include any data points from a range of the real data  $\pm$  one standard deviation of the real data. This approach would ensure that we have included data points that would in real life resemble the cases of how the system would react to the potentially invasive changes that could not only lead to an increased execution time, but also to a decreased one (when, for example, some parts of data get skipped unintentionally).

We then proceeded to train and validate the results of the predictions in a few different models and their variations:

Moving average	ARIMA model	Random Forest
need to distinguish between jobs as those have different degrees of the execution times	need to distinguish between jobs as well	can be supplied with the whole dataset, in this case scores at 87% of precision and does not produce any false negatives

<p>does not predict any outliers</p>	<p>predicts very few outliers with accuracy around 3.4%</p>	<p>when running a classifier per a subset of data, per job:</p> <ul style="list-style-type: none"> <li>- scores almost at 95% of precision;</li> <li>- has an average F1-score of 80.15%;</li> <li>- still does not return any false positives</li> </ul>
<p>predicts “good” data in 100% cases</p>	<p>predicts “good” data relatively well (precision is around 83%)</p>	

Table 11. A comparison of the models’ results.

Further analyzing the results of the research, we can conclude that it would be applicable to use either a simple moving average to predict a baseline for “good” data or Random Forest Classifier to try to identify incoming data as outliers.

In practice, that would make sense to use both to:

1. Predict the baseline and send a signal to recommend to stop the executions as soon as the maximum threshold is passed.
2. Collect the execution times of the successfully generated jobs and use those to predict the outliers and, respectively, identify a date when the difference in the execution times became noticeable. This case is extremely

important as the code changes may lead to some parts of datasets skipped during the cohorts and analysis' generations which can lead to false real world evidence reports.

## Future work, challenges and opportunities.

1. There is a dependency on the size and number of the processors in the instance where cohorts and analyses run. In the lower environments we can set these parameters to some default values therefore providing consistency on the input data. In the upper environments, the size and number of the processors varies and therefore will require additional work on finding a way to normalize the input against these parameters.

---

```
file: jobid_129726.20221114.log
  time committed      used count  sysavg  cpuavg
0  04:04:23  0.000128  0.000128   115    0.83  0.051875
1  04:05:23  0.000262  0.000261   313    4.77  0.298125
2  04:06:23  0.000297  0.000297   214   17.24  1.077500
3  04:07:23  0.000303  0.000303   213   19.85  1.240625
4  04:08:23  0.000310  0.000309   214   21.48  1.342500
5  04:09:23  0.000314  0.000313   213   23.56  1.472500
6  04:10:23  0.000318  0.000317   212   20.63  1.289375
7  04:11:23  0.000321  0.000320   211   20.84  1.302500
8  04:12:23  0.000324  0.000323   212   21.42  1.338750
9  04:13:23  0.000329  0.000328   211   20.83  1.301875
10 04:14:23  0.000340  0.000340   211   12.11  0.756875
11 04:15:23  0.000363  0.000363   209    4.81  0.300625
12 04:16:23  0.000363  0.000363   208    2.28  0.142500
13 04:17:23  0.000363  0.000363   208    1.33  0.083125
14 04:18:23  0.000363  0.000363   208    0.79  0.049375
15 04:19:23  0.000370  0.000369   304    1.18  0.073750
16 04:20:21  0.000388  0.000387   309    1.58  0.098750
-----
```

Figure 6. Example of metrics collected from one log file.

A challenge to properly normalize the execution times against a number of the processors is that there is a sequential part and a parallel part of the execution



process run by the system and the number of the processors would be affecting the parallel part of it.

2. We also should be able to use this testing capability to test the performance changes.
3. Implement density scan to identify a phase change (when a potentially positive code change takes place and the algorithm we use should adjust to a new baseline).

## References.

[1] Selenium Automation Testing

<https://www.simplilearn.com/tutorials/selenium-tutorial/selenium-automation-testing>

[2] The Economic Impacts of Inadequate Infrastructure for Software Testing, National Institute of Standards and Technology (Table 5-1. Relative Cost to Repair Defects When Found at Different Stages of Software Development).

<https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf>

[3] Real-World Evidence.

<https://www.fda.gov/science-research/science-and-research-special-topics/real-world-evidence>

[4] The Truth About End-to-End Testing

<https://www.qualitylogic.com/knowledge-center/the-truth-about-end-to-end-testing/#:~:text=End%2Dto%2Dend%20testing%20can,script%20re%2Dwrites%20to%20match.>

[5] Moving averages in python

<https://towardsdatascience.com/moving-averages-in-python-16170e20f6c>

[6] How to Create an ARIMA Model for Time Series Forecasting in Python.

<https://www.analyticsvidhya.com/blog/2020/10/how-to-create-an-arima-model-for-time-series-forecasting-in-python/>

[7] ARIMA models for time series forecasting

<https://people.duke.edu/~rnau/411arim.htm#arima110>

[8] What is a Random Forest?

<https://www.tibco.com/reference-center/what-is-a-random-forest>

[9] Will Koehrsen, Towards Data Science

<https://towardsdatascience.com/random-forest-in-python-24d0893d51c0>

## Appendix A

Dataset: derived from log files from the Aetion's private S3 location.

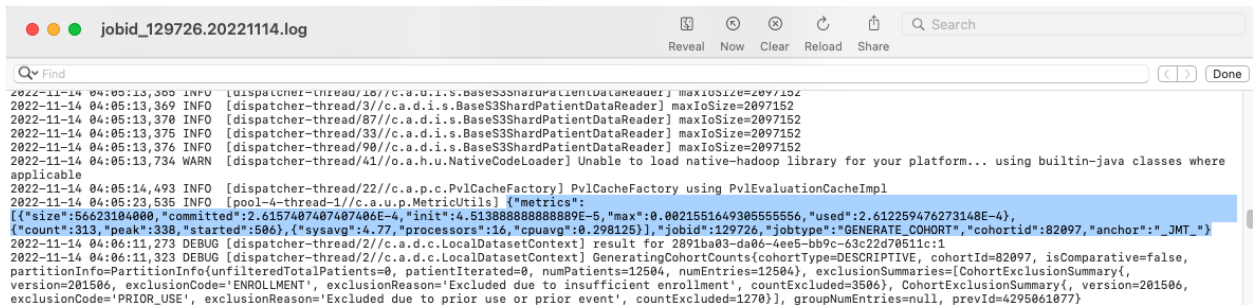
Start name: *jobid\_128001.20221102.log*, date: 11/02/2022

End name: *jobid\_132999.20221220.log*, date: 12/20/2022

Total number of files: 4990

File's content: logs of job executions.

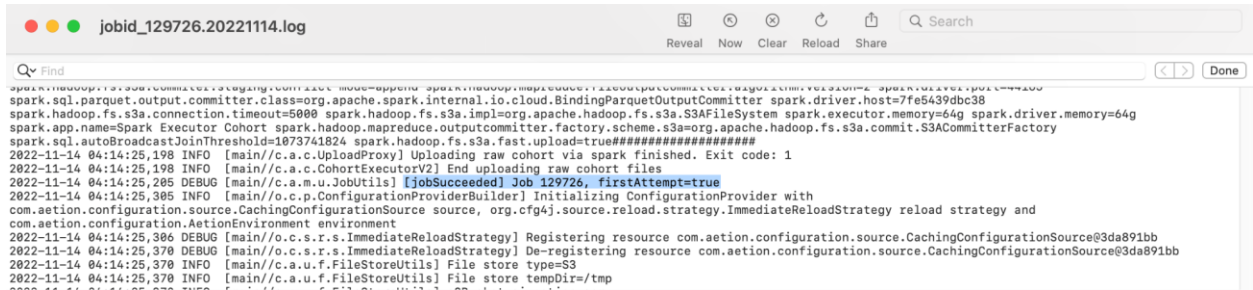
Example of input JSON log file (inner {"metrics": ...} JSON).



```
2022-11-14 04:05:13,305 INFO [dispatcher-thread/10//c.a.d.i.s.BaseS3ShardPatientDataReader] maxIoSize=2097152
2022-11-14 04:05:13,369 INFO [dispatcher-thread/3//c.a.d.i.s.BaseS3ShardPatientDataReader] maxIoSize=2097152
2022-11-14 04:05:13,370 INFO [dispatcher-thread/87//c.a.d.i.s.BaseS3ShardPatientDataReader] maxIoSize=2097152
2022-11-14 04:05:13,375 INFO [dispatcher-thread/33//c.a.d.i.s.BaseS3ShardPatientDataReader] maxIoSize=2097152
2022-11-14 04:05:13,376 INFO [dispatcher-thread/98//c.a.d.i.s.BaseS3ShardPatientDataReader] maxIoSize=2097152
2022-11-14 04:05:13,734 WARN [dispatcher-thread/41//o.a.h.u.NativeCodeLoader] Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
2022-11-14 04:05:14,493 INFO [dispatcher-thread/22//c.a.p.c.PvlCacheFactory] PvlCacheFactory using PvlEvaluationCacheImpl
2022-11-14 04:05:23,535 INFO [pool-4-thread-1//c.a.u.p.MetricUtils] {"metrics": [{"size":5662310400,"committed":2.6157407407407406E-4,"init":4.513888888888889E-5,"max":0.0021551649305555556,"used":2.612259476273148E-4},{count":1313,"peak":338,"started":506},{sysavg":4.77,"processors":16,"cpuavg":0.298126}], "jobid":129726, "jobtype":"GENERATE_COHORT", "cohortid":82097, "anchor": "_JMT_"}
2022-11-14 04:06:11,273 DEBUG [dispatcher-thread/2//c.a.d.c.LocalDatasetContext] result for 2891ba03-da06-4ee5-bb9c-63c22d70511c:1
2022-11-14 04:06:11,323 DEBUG [dispatcher-thread/2//c.a.d.c.LocalDatasetContext] GeneratingCohortCounts(cohortType=DESCRIPTIVE, cohortId=82097, isComparative=false, partitionInfo=PartitionInfo(unfilteredTotalPatients=0, patientIterated=0, numPatients=12504, numEntries=12504), exclusionSummaries=[CohortExclusionSummary{version=201506, exclusionCode='ENROLLMENT', exclusionReason='Excluded due to insufficient enrollment', countExcluded=3506}, CohortExclusionSummary{version=201506, exclusionCode='PRIOR_USE', exclusionReason='Excluded due to prior use or prior event', countExcluded=1270}], groupNumEntries=null, prevId=4295061077}
```

## Appendix B

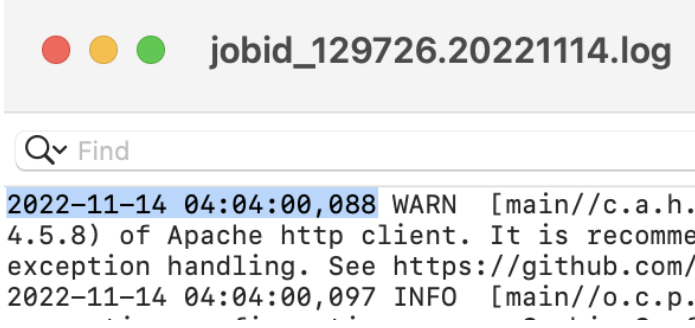
Example of input JSON log file ([jobSucceeded] mark).



```
spark.sql.parquet.output.committer.class=org.apache.spark.internal.io.cloud.binding.ParquetOutputCommitter spark.driver.host=7fe54390bc38
spark.hadoop.fs.s3a.connection.timeout=5000 spark.hadoop.fs.s3a.impl=org.apache.hadoop.fs.s3a.S3AFileSystem spark.executor.memory=64g spark.driver.memory=64g
spark.app.name=Spark Executor Cohort spark.hadoop.mapreduce.outputcommitter.factory.scheme=s3a=org.apache.hadoop.fs.s3a.commit.S3ACommitterFactory
spark.sql.autoBroadcastJoinThreshold=1073741824 spark.hadoop.fs.s3a.fast.upload=true#####
2022-11-14 04:14:25,198 INFO [main//c.a.c.UploadProxy] Uploading raw cohort via spark finished. Exit code: 1
2022-11-14 04:14:25,198 INFO [main//c.a.c.CohortExecutorV2] End uploading raw cohort files
2022-11-14 04:14:25,285 DEBUG [main//c.a.m.u.JobUtils] [jobSucceeded] Job 129726, firstAttempt=true
2022-11-14 04:14:25,385 INFO [main//c.p.ConfigurationProviderBuilder] Initializing ConfigurationProvider with
com.aetion.configuration.source.CachingConfigurationSource source, org.cfg4j.source.reload.strategy.ImmediateReloadStrategy reload strategy and
com.aetion.configuration.AetionEnvironment environment
2022-11-14 04:14:25,386 DEBUG [main//o.c.s.r.s.ImmediateReloadStrategy] Registering resource com.aetion.configuration.source.CachingConfigurationSource@3da891bb
2022-11-14 04:14:25,370 DEBUG [main//o.c.s.r.s.ImmediateReloadStrategy] De-registering resource com.aetion.configuration.source.CachingConfigurationSource@3da891bb
2022-11-14 04:14:25,370 INFO [main//c.a.u.f.FileStoreUtils] File store type=S3
2022-11-14 04:14:25,370 INFO [main//c.a.u.f.FileStoreUtils] File store tempDir=/tmp
```

## Appendix C.

Start and end of execution time frames at the log file (example).

a.	Start execution time:	
b.	End execution time:	