Empirical Analysis of Jenkins Pipelines

Debojit Kaushik (dkaush4@uic.edu), **Sandeep Joshi** (sjoshi37@uic.edu)

CS540: Advanced Software Engineering - Course Project
Instructor: Prof. Mark Grechanik
University of Illinois at Chicago

Abstract

This report explains and demonstrates an empirical study performed on a mid-sized data set comprised of Jenkinsfiles to analyze and extract insights from existing DevOps pipelines. We have tried to analyze properties such as timeout periods, actions taken commonly in specific blocks. We observed a specific patterns emerge after we analyzed the results. We also used machine learning techniques to analyze repositories with similar properties.

1 Introduction

The primary aim of this project is to discover patterns and trends in various repositories and their continuous integration pipelines. We saw that the settings for different repositories and their intrinsic properties are related to their corresponding continuous integration pipeline settings. The scope of the project itself is huge due to the various types of co-routines and tasks available to be performed in automated testing processes. We analyzed groovy syntax and extracted stages, shell script commands, build invocations using different build tools used.

For the rest of the report, we shall explore the technical approaches we applied to the project, the architecture and methods, what specific queries we tackled, what kind of results we came up with, and our conclusion on the information.

2 Formulations of research questions

2.1 Research Questions

We formulated 8 empirical questions which we formed by reading and understanding the structure of a Jenkinsfile and how it controls the automated flow and calls. Here we discuss the questions and their insights at length.

2.1.1 What are the most used build tools?

As Jenkins is used for continuous integration, build tools are used extensively. We want to see what are the most popular build tools used in the Jenkinsfiles and how often they were used.

2.1.2 What are the statistics of timeouts?

From analyzing the Jenkinsfiles, we wanted to see what are the average timeout values, standard deviation and how many files had the timeout values.

2.1.3 Do files with parallel blocks have more stages?

Since parallel blocks are used to divide the work and make the pipeline execution faster, it was an intuitive question whether pipelines containing parallel blocks contain more stages(indicating more work being done) on average? We check that with getting the average number of stages in files with parallel blocks, without parallel blocks and all files.

2.1.4 What are the most and least frequent post condition blocks?

As we analyzed the Jenkinsfiles, most of them contained post condition blocks, we wanted to see most used blocks and their distribution across diff repositories.

2.1.5 Correlation of number triggers to average number of stages?

We investigate if having more triggers in the pipeline correlate with more/less number of stages in the pipeline.

2.1.6 Do different post blocks indicate more stages?

Since we had a lot of files with post condition blocks, we wanted to see if having certain post condition blocks correlate to having more number of stages. For example, does having more number of stages correlate to having failure blocks? If there are more number of stages, there might be more chances of having failures and the user might want to handle the failures. Also this helped to get the correlation between occurrences of different post blocks. For example, does success and failure blocks occur together?

2.1.7 What are the most popular things done in stages?

Since most of the operations in the pipelines are performed in stages, we wanted to see what are the most performed operations in the stages. Here we consider only the name of the stages.

2.1.8 What are the most popular things done inside stages?

Here we are trying to see what are the most popular operations done inside the stages. Since the scope to do operations in the pipeline is open ended, we visualize this in the form of a word cloud.

3 System Architecture

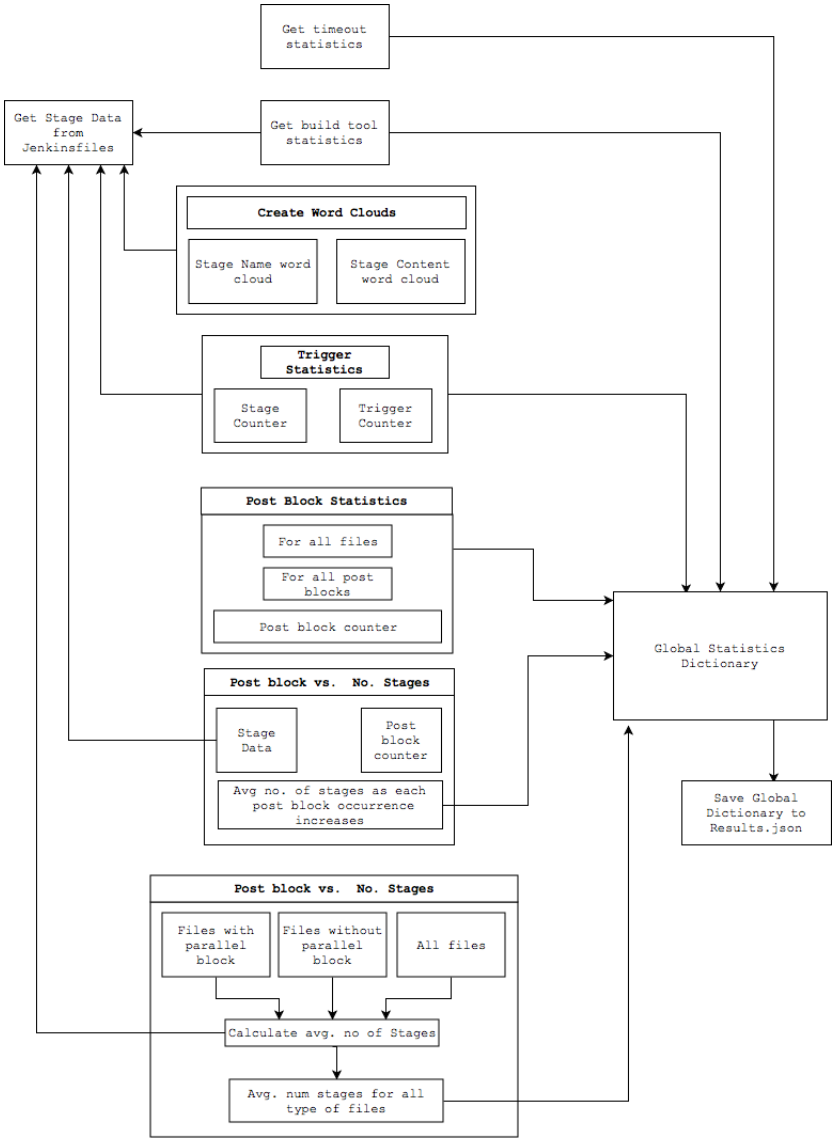

Figure 1: System Architecture

4 Details of Implementation

4.1 Most Used Build Tools

Our approach to gather insights about the build tools was to parse the complete data set and recognize the build tool commands. If any build tools command or expression is present in the file we recorded the occurrence and kept track of how many times it occurred in the directory. We ignored multiple occurrences of the same build tool in the same file. This gave us a frequency metric for the Jenkins data set.

4.2 Timeout Statistics

We also observed, there were timeouts specified for many files. We parsed through the files and fetched the time out periods, if there were any, from the files and extracted the average

time out period, standard deviation about the mean and variance. The timeout periods were mentioned in the form Timeout(time: number). We used 'shlex', a groovy parser, output to parse through the immediate tokens after the timeout token to get the time period.

4.3 Files with parallel blocks

To analyze whether files with parallel blocks had more stages, we selected the files with parallel blocks and then obtained the number of stages for those files. Now to compare this result, we extracted the number of stages for files without parallel blocks and all the files. This way, we can compare number of stages with parallel blocks, without parallel blocks and overall.

4.4 Frequent post condition blocks

To get the units in post condition blocks, we first extracted the post blocks in the files and then using the content in the post blocks, we looked for the always, success, failure, unstable, changed and aborted blocks. We then checked how frequently they occur.

4.5 Triggers and number of stages

We extracted the number of triggers present in files and then extracted the number of stages present for each of those files. Further we calculated the average number of stages for number of triggers in a file.

4.6 Different post condition blocks and stages

We checked as number of stages increase in a pipeline, how does post condition blocks vary. We extracted number of different post condition blocks and number of stages for each file which had post condition blocks. Then we averaged the number of post condition blocks for each number of stages. This also helped to visualize the correlation of number of post condition blocks among themselves.

4.7 Code Extraction from the pipelines

One of the tasks is to get the unit of specified code from the pipeline for a given specification. For example, to extract all the post condition blocks in a pipeline, we first get all the contents in the post block of the pipeline and then find all the blocks in those contents. Same goes for the stages. We wrote a simple parser which extracts the code given a pattern like that of a stage or post blocks. This made the task easy to get data for analysis for a given question.

4.8 popular operations for 'Stages'

Due to the nature of the data and uncertainty of the occurrences of different forms of operations, we extracted the overall word corpus of the data which described what the stage was for. Using this we created a word cloud to get a thorough overview of what the stages were responsible for most commonly. We used the the output of our 'StageExtractor' class to get the arguments used while defining a stage class. This provided some interesting results.

4.9 popular operations inside 'Stages'

Due to the nature of the data and uncertainty of the occurrences of different forms of operations within a stage block as well, we extracted the overall word corpus of the data which described what operations were popular within stage definitions. Using this we created a word cloud to get a thorough overview of what the stages were contained. We used the the output of our 'StageExtractor' class to get the arguments used while defining a stage class. This provided some further interesting results.

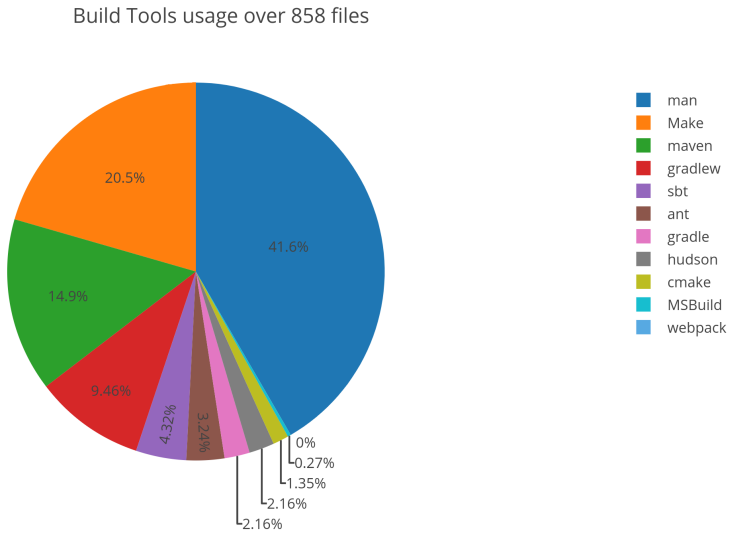

Figure 2: % chart representing build tool usage

5 Results and Explanations

5.1 Build Tool statistics

After parsing through 858 Jenkins files, we gathered the following results of build tools occurrences.

Tool	Occurrence
webpack	0
cmake	5
gradle	8
gradlew	35
mvn	154
ant	12
maven	55
msbuild	1
make	76
sbt	16
hudson	8

We see here the most popular build tools used with Jenkins from our data is Maven followed by tools like make and Gradle. Webpack surprisingly even being popular has no occurrence in the data set. We have see it is generally used in combination with Travis CI pipelines commonly.

5.2 timeout statistics

After analyzing timeout periods the following are the results. we observed some outliers in the data which we cleaned for convenience. If not they increased the mean and deviation about the mean but it was not representative of the true data insights.

Quantity	Value
Standard Deviation	20.66507955030691
Average Timeout	16.675
% of files with timeout periods	4.667444574095683

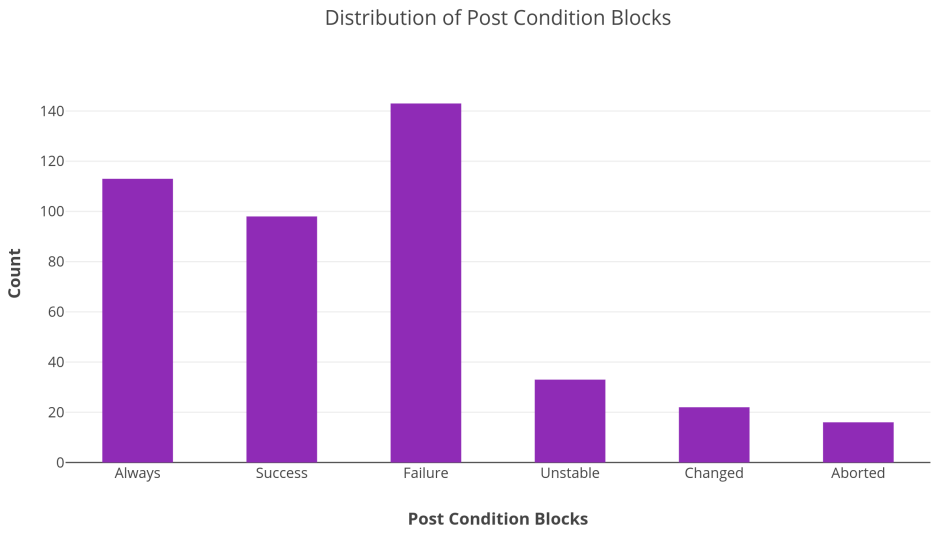

Figure 3: Distribution of Post condition blocks

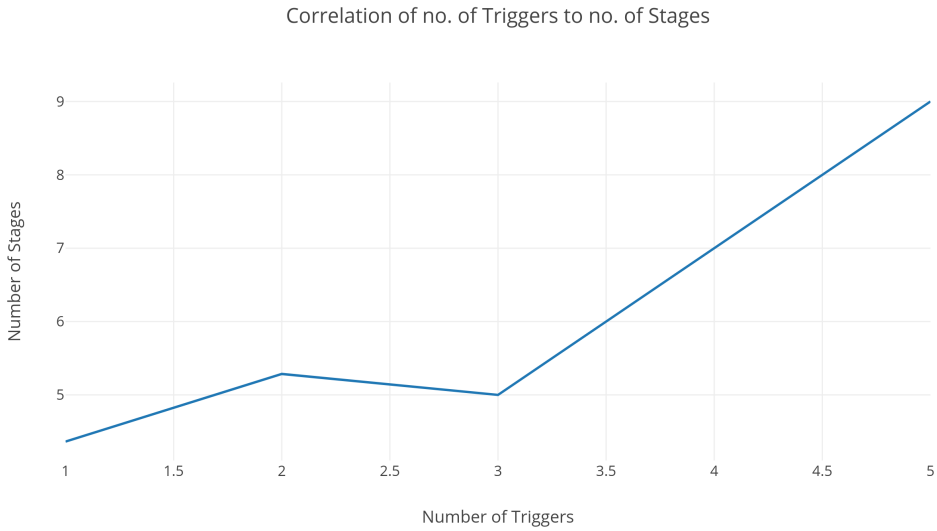

Figure 4: Number of triggers vs. Number of stages

We can see very less files use timeout periods. T

5.3 Distribution of Post condition blocks

As we can see in the Figure 2, most used post condition block is failure followed by always and success.

5.4 Number of triggers to Number of Stages

As Figure 3 shows, in general as number of triggers increase, number of stages also increase. We can infer that there is a correlation between number of triggers and number of stages.

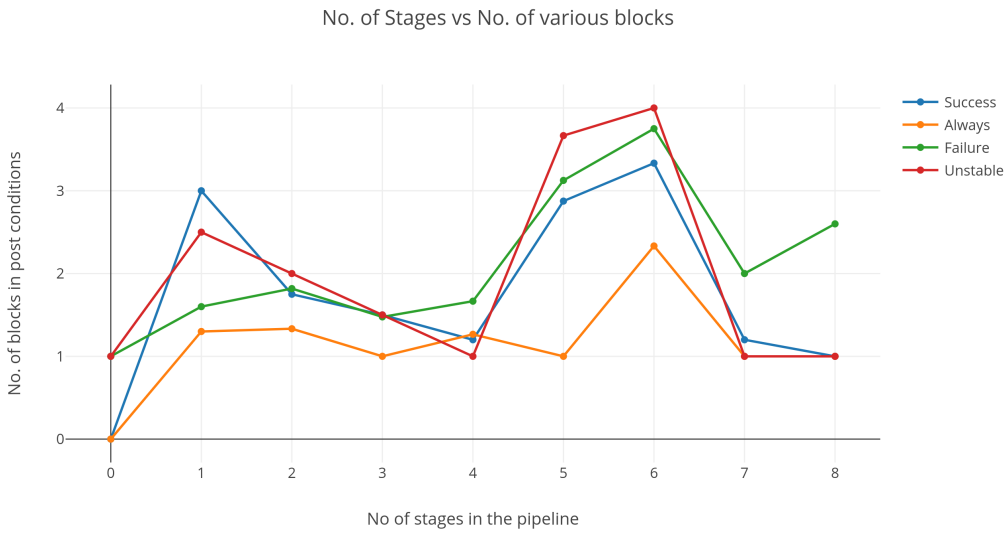

Figure 5: Number of stages vs. Various post condition blocks

5.5 Number of Stages vs. Various post condition blocks

From Figure 4, we can see there is a correlation between increase and decrease of various post condition blocks as number of stages increase.

5.6 Number of Parallel blocks vs. Number of stages

As we can see from the table below, having parallel blocks does indicate having more number of stages in the file on an average. Files without parallel block have considerably lower number of stages than the ones which do. We can infer that having parallel blocks indicate more work being done in the pipeline.

Type of files	Avg. number of stages
Files with parallel blocks	4.15789
Files without parallel blocks	2.90396
All files	3.0151

5.7 Stage blocks descriptors

We found word clouds to be very informative for different blocks in the groovy files. Following is the word cloud for all the stage descriptors and arguments.

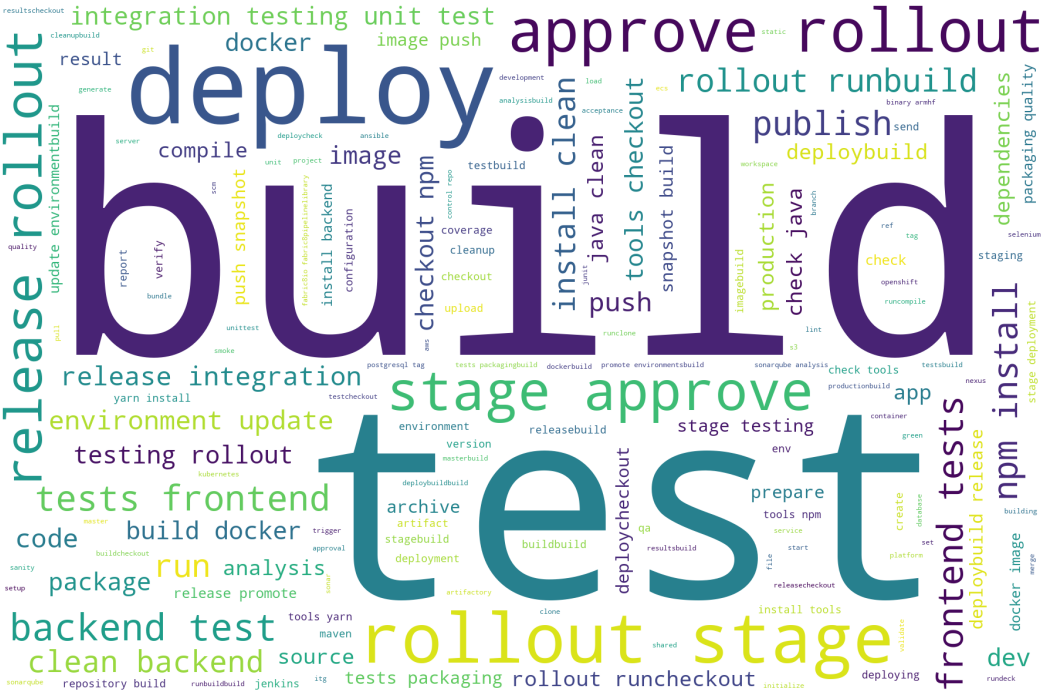

Figure 6: Overview of most used words and functions for a stage

5.8 Stage blocks content

We found word clouds to be very informative for different blocks in the groovy files. Following is the word cloud for all the stage level block contents.

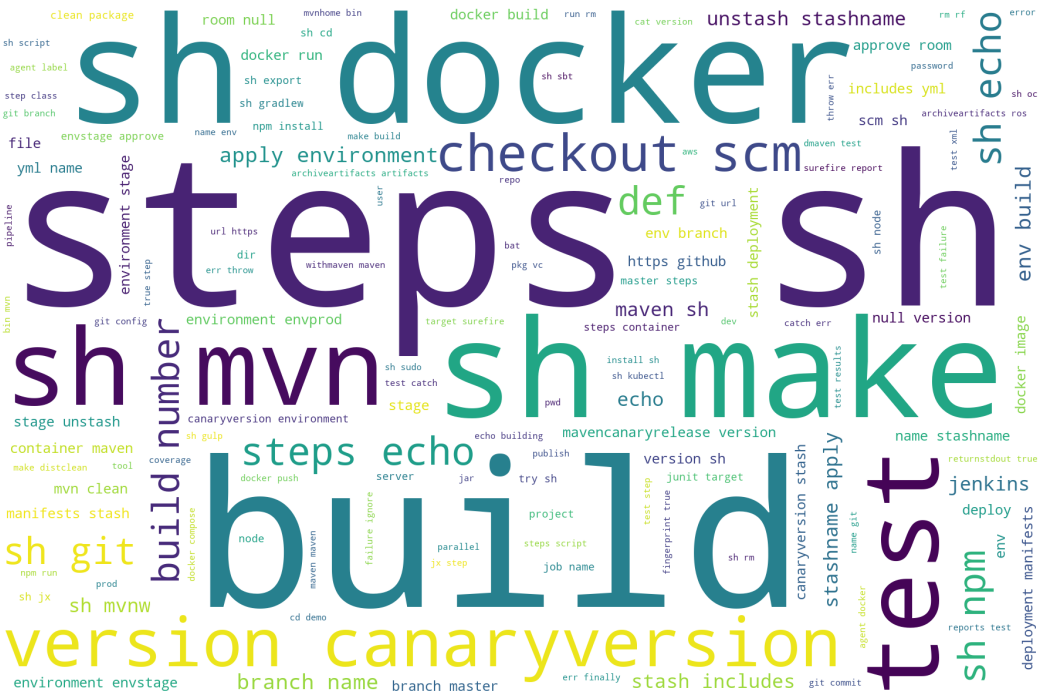

Figure 7: Overview of most used words and functions for a stage

6 Machine Learning: Clustering of Jenkins files

We used KMeans clustering to cluster all the Jenkins files. This allowed us to form relevant cluster of these files based on their intrinsic properties. The data set being very disparate posed some challenges with forming clusters but we saw better performance with 'k' clusters, k greater than 25.

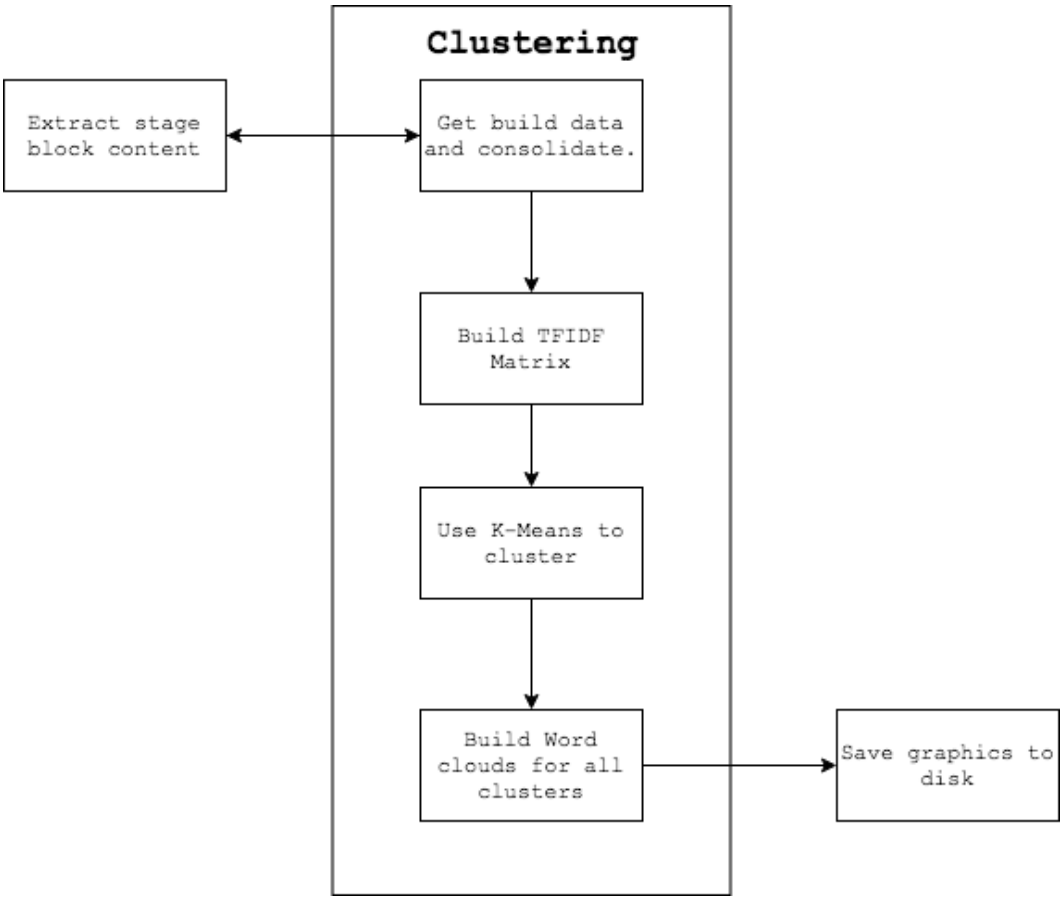

Figure 8: Clustering model

6.1 Challenges in clustering

- The data set and its preparation was challenging due to the fuzzy nature of the data. Cleaning of the data to prepare for clustering was also a challenge as the text meant groovy classes and not usual linguistic terms. But KMeans works on frequencies and its occurrences which guided our choice f the algorithm.
- The data is very disparate and doesn't cluster for less number or clusters. Increasing the cluster numbers risks our model being over-fitted. We found a sweet spot of the range 25-30 clusters which gave us considerable results.
- Now after the clusters were formed, we needed to check what the clusters signified and what was the pattern of these clusters with the repositories.

We again used **word clouds** to see what each cluster represented. Most of the clusters were very less populated but a few of them had considerable amount of members. In the following word clouds we can see what common patterns emerge in these clusters.

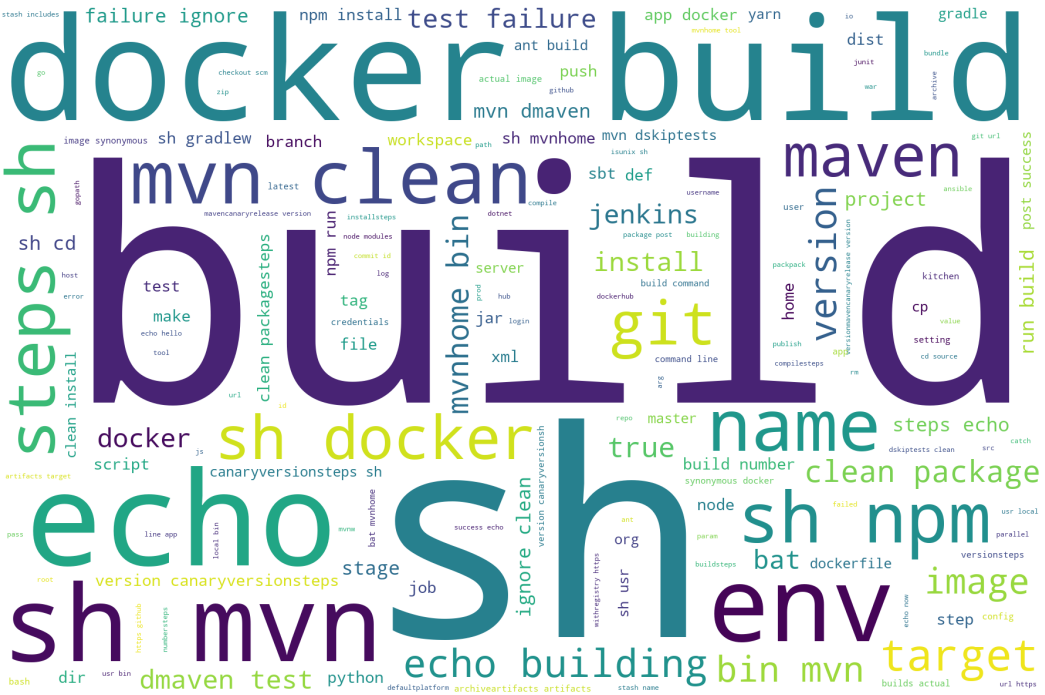

Figure 9: Overview of most used words and functions for most populated cluster

Figure 10: Overview of most used words and functions for second most populated cluster

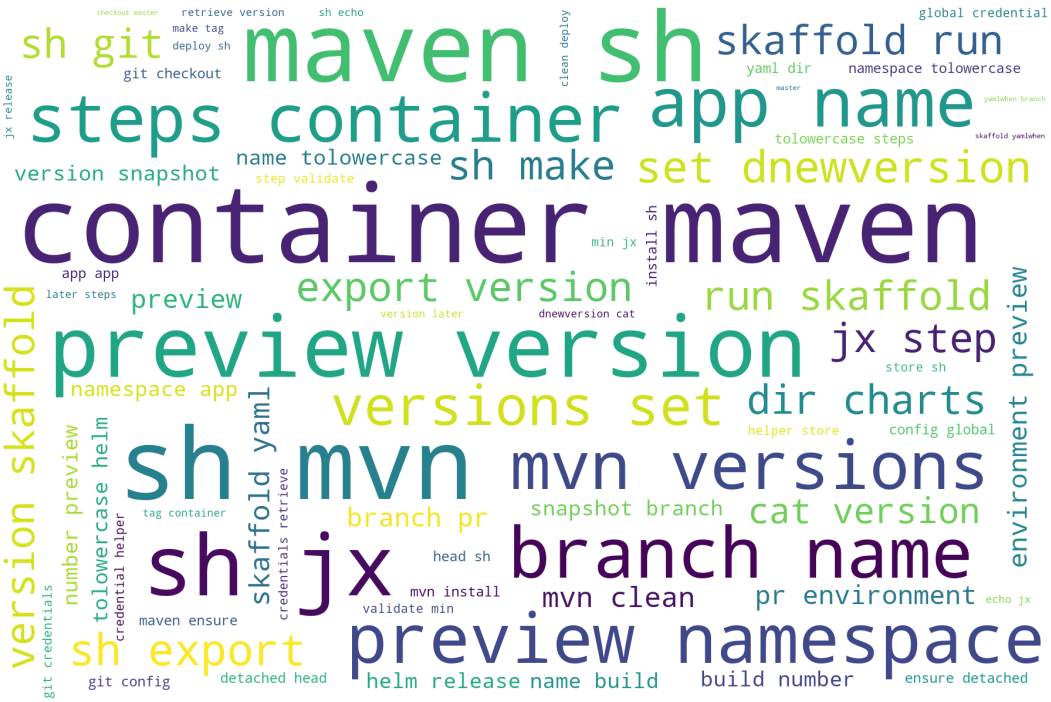

Figure 11: Overview of most used words and functions for third most populated cluster

7 Conclusion

We have found that most Jenkins files contain very basic settings rather than most advanced settings. The settings tend to contains shell scripts upon building and trigger builds. Testing and logging are a part of most pipelines. Developers use the popular build tools like Maven and Gradle. We also saw some docker settings being initiated or changed in many files and suggestive from the word clouds. We see timeouts being used very less, while they being important to have a robust, unfailing pipeline automation. The data gathering was also a big task due to the files having code rather than a corpus.

8 References

We used Python to build the whole system and our tools.We would like to mention the following in addition:

- PyGal
- Plotly
- Keras
- scikit-learn
- Jenkins Documentation[https://jenkins.io/doc/book/pipeline/syntax/]
- www.python.org
- wordcloud (Python Library)
- overleaf (Collaborative LaTex editor)
- www.draw.io