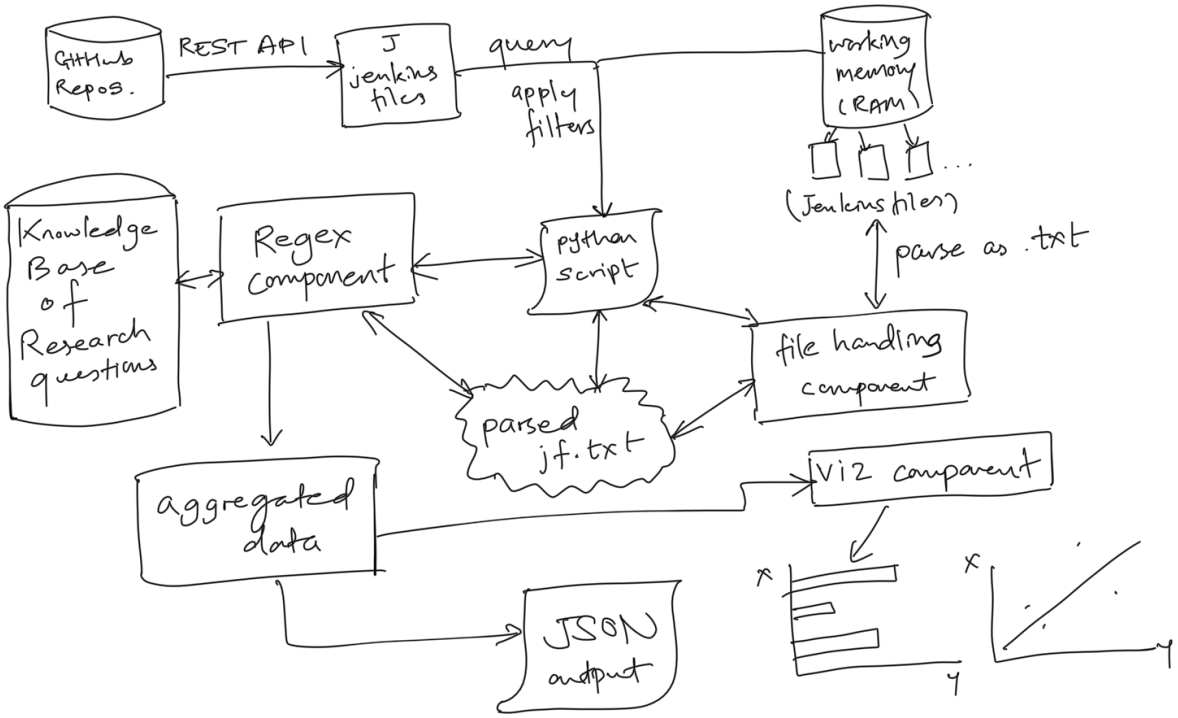
CS 540 Course Project - Jenkins Pipeline Analyzer

Pratik Kshirsagar and Sai Phaltankar

We have implemented a Jenkins pipeline analyzer in Python. We aim to address particular aspects of a Jenkins pipeline and we explored a lot of possibilities. A detailed description of our actions is given throughout this document. Below, we start with an abstract of our system.

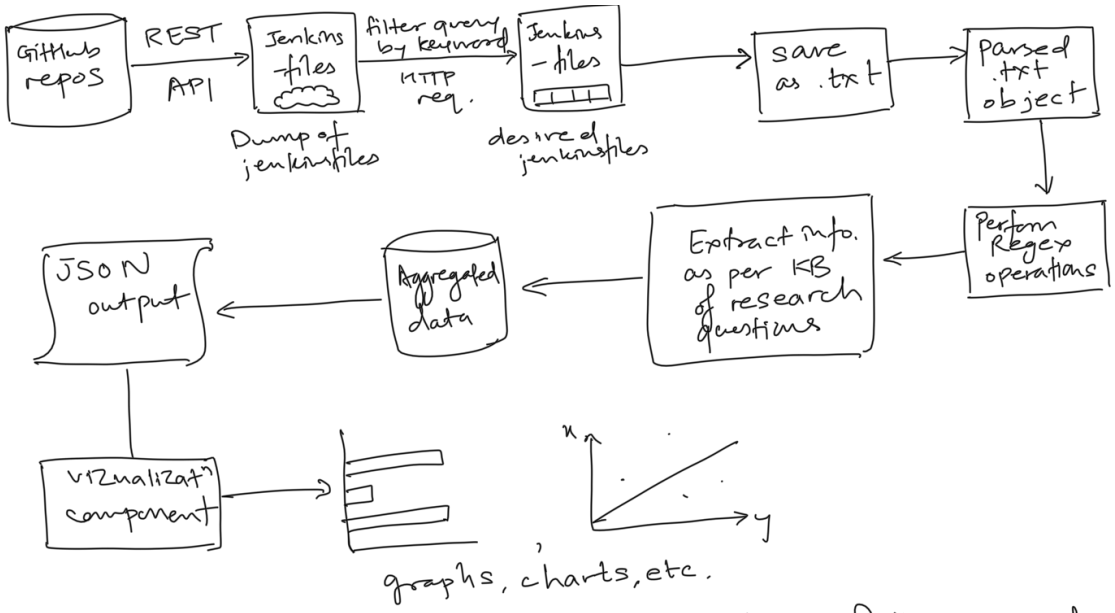


Architectural Diagram of Our System

As you can see, at the heart of our system lies a Python script, which governs the behaviour of all the other components. We tried to make our code as modular as possible so as to enable maximum reusability, easy debugging and high performance and scalability. We use GitHub as our source of Jenkinsfiles of various open source repositories and we have all of our programming modules in Python. We rely on converting our research requirements as a knowledge base of questions and make use of Regular Expressions to aggregate the apt data. Using JSON, we output our results

onto a file and we also use some visualisation components to plot certain important measurements.

Architectural pattern: Pipe and Filter.



Pipe and filter : Architecture Pattern used by our system.

Our system follows the Pipe and Filter pattern and the general workflow of our system is as follows:

1. Obtain Jenkinsfiles from GitHub repositories through REST API.
2. Filter out the Jenkinsfile which are of importance, using certain keywords in the query along the HTTP request. We are firing unique queries which are appropriate to each of the research questions.
3. We thus obtain the raw contents of a Jenkinsfile and we store it in a .txt file which can be easily parsed.
4. We parse the .txt file and use Regular Expressions to extract vital information that can be used in the empirical analysis, as we will explain in depth in further sections.

5. The vital information that is extracted, is mapped with a Knowledge Base of the Research Questions to aggregate data that is essential to the empirical analysis.
6. The output of the empirical analysis is processed in a JSON file and some apt visualisation is performed with the results obtained.

Formulation of Research Questions:

>> “How many Jenkins pipelines have some form of **Exception Handling** incorporated in them?”

We were particularly interested in this aspect of a DevOps pipeline as pipeline being a fairly complex process, automating one would bound to have some forms of Exceptions arise as there is plenty of margin for error due to the high number of interdependent components. We were interested in finding out how many of such pipelines **do** actually take care of handling **runtime exceptions**.

>> “What is the most frequent **post-condition** blocks in the post section within jenkins pipelines?”

We were interested in finding out what were the most/least popular post actions in a Jenkinsfile. We try to see which one or more additional **steps that are run upon the completion of a Pipeline’s or stage’s run**.

>> “What is the most frequently used **agent** in a Jenkins pipeline?”

Agents specify **where the entire Pipeline, or a specific stage, will execute** in the Jenkins environment. We knew that there are certain kinds of agents that are available to the pipeline, so we wanted to know which ones of them are popular among applications and which ones are not so much.

>> “What fraction of Jenkinsfile pipelines have no **global agents**?”

We know that lack of a global agent implies that each stage section will need to contain its own agent section, this taking away a certain degree

of centrality. We wanted to find out what fraction of Jenkinsfile follow such a pattern.

>> “What percent of Jenkins pipelines involve **Docker** at some stage of execution?”

Docker, a container technology is prevalent in DevOps due to its **performance and scalability** in addition to its **platform independence**. We wanted to find out how many Jenkins pipelines have adopted this technology as part of their routine.

>> “What is the **correlation** between **triggers** in a pipeline and the number of **stages**?”

We wanted to find out if there exists a correlation amongst the two and if it does, what is the significance of it. It seemed interesting to investigate co dependence of the two phenomena.

>> “What are the most **frequent** Jenkins pipeline **operations**?”

We discovered through observing various Jenkinsfiles that there exist many kinds of operations in a Jenkins pipeline so we try to discover which ones of them are the most frequent.

>> “What is the **average number of stages** in a Jenkins pipeline?”

A Jenkins pipeline can have variable number of stages, so naturally we wanted to find the average number of stages, ie, how many stages is a Jenkins pipeline would have, on average.

Details of the Implementation:

The system was implemented using Python 3.6.5 and PyCharm IDE. The logistic details of the implementation have been mentioned in the previous sections of this document. Here are some Python modules we used to implement the code:

- Requests - We used this package to deal with HTTP requests in order to interact with GitHub API to retrieve Jenkinsfiles.

- RE - This Regular Expressions library was used to a great extent as part of our Regex component. In the next section, we'll show you how we used Regex to search/match patterns in the Jenkinsfile to find word strings that are pertinent in the empirical analysis.

- Math - We use this library in order to implement formulae used to find Carl Pearson's Correlation coefficient.

- JSON - We use this library to produce our output in a pretty JSON format.

- Logging - We use this package for logging purposes, ie, creating and maintaining log files.

- Plotly - We use this package for generating visualisations such as charts and plots.

Our source code is well commented and self explanatory to a great extent.
We have made our code as modular as possible, and used functions to maximise reusability and robustness. **We've made the following functions:**

> **docker()** - returns the fraction of Jenkins pipelines that incorporate Docker in some way.
> **exception_handlin()** - returns fraction of Jenkins pipelines that incorporate Exception Handling in some way.
> **popular_agent()** - returns a list of popular agents with their frequency.
> **get_post_cond_stats()** - returns a list of frequent post-condition operations with their frequency.
> **get_pipeline_stage_stats()** - returns the average number of stages found in the Jenkins pipelines.

> **get_trigger_stages_correlation()** - returns the Pearson Correlation Coefficient between triggers and stages, along with related details.

We have also implemented functions for retrieving Jenkinsfiles and their contents from GitHub API as well as converting into .txt files and parsing them:
> **jenkinsfile_query** - Iterate through all GitHub repositories for retrieving Jenkinsfiles.
> **contents_query** - Grab the raw content of the Jenkinsfile.
> **readyFile** - Create a .txt from the raw content and return parsed contents for further processing.

Descriptions of Experiments:
So as we mentioned earlier, we used Regex to get the vital information and we performed necessary mathematical operations on the enumerated data to generate results for the research questions. Here are the details of how we approached each of those aspects. Please refer to functions in code for a comprehensive understanding.

Exception Handling > We observed that a 'try' block in groovy DSL would signify exception handling in a Jenkins pipeline**.** So we used Regex to count the occurences of such 'try' blocks. Dividing the number of Jenkinsfiles that had at least 1 occurrence of the word 'try { ' (exact match) by total number of Jenkinsfiles we obtained gave us the answer.
Regex expression = '\btry\b\s*\{'

Post-Condition block > We calculated the occurrences of all the possible post actions (always, fixed, changed, regression, aborted, failure, success, unstable, cleanup) similar to above. We then sorted the results.
Regex expression = '\balways\b\s*\{' [replace 'always' with others.]

Popular Agents > Similar to above, for all the agents(any, none, label, docker, node, dockerfile.)

Regex expression = '\bagent\b\s*\bnone\b' [replace 'none' with others.]

No Global Agent > Subset of above question. Return the occurrence of agent of type 'none'. Execute similar to previous.

Docker > We observed that a 'docker { ' or 'dockerfile { ' block in groovy DSL would signify use of Docker in a Jenkins pipeline**.** So we used Regex to count the occurences of such blocks. Dividing the number of Jenkinsfiles that had at least 1 occurrence of these words (exact match) by total number of Jenkinsfiles we obtained gave us the answer.
Regex expression = '\btry\b\s*\{'

Correlation(Triggers, Stages) > Treat number of triggers as 'x' and number of stages as 'y'. Then find sum(xy), sum(x^2) and sum(y^2) and other related quantities. Apply Pearson's Correlation Coefficient to find Correlation(x,y). Using Regex in a similar way, count the aforementioned quantities. Please refer to code for better understanding.

Popular Operations > We calculated the occurrences of all the possible operations (operations are equivalent to stages) similar to finding post-condition blocks. We then sorted the results.
Regex expression = '\bstage\b\s*\((["'])(?:(?=(\\?))\2.)*?\1\'

Average Stages > Subset of above problem. Find occurrence of 'stage' over all of the Jenkinsfiles. Returns a fraction.

Results and Explanations:
We generated the results of the analysis and output it in JSON. We also generated visualisations of the results in the form of Bar Charts, Pie Charts, and Box Plots. The charts and plots are very easy to understand and self explanatory. The JSON that we've generated contains keys and values. The key represents a particular research question and the contents within represent the answer to the question along with extra details when

applicable (such as frequency, sorted order, etc.) Sample outputs have been uploaded to the repository.

Conclusions:

We were able to answer the questions we formulated to a pretty good extent. JSON and Plotly allowed us to give a neat and comprehensible display of the results. We used Regex to perform our core functionality yet Regex is known to be a heavy process that doesn't give sublinear time complexity. Overall, our system has time complexity of **$O(n^3)$** where n represents number of input Jenkinsfile available. In the future we can try using different approaches such tokenizers from Keras or Spacy to give us better time performance.