Advanced Software Engineering: Course Project Report

Architecture and Description

Implementation

The general flow for each of the research questions is as follows:

i. Fire query to GitHub API to search for repositories having Jenkinsfile containing the section to be analysed for the corresponding research question. Since there are huge number of results, we use pagination to process them in batches.
ii. Using the API of the *jenkinsci/pipeline-model-definition-plugin*, we parse the Jenkinsfile (which is in Groovy) of each repo into a JSON structure. This is a plugin for Jenkins, which means to parse a Jenkinsfile, we need to first run it in Jenkins, where we make the required API calls for converting to JSON structure, which is then stored for analysis
iii. We now check if the JSON structure contains the section corresponding to the research question in the desired format. This is because the JSON structure may contain errors due to improper syntaxes or a keyword not supported by the plugin. In such a case, we would not be able to use this file for our study, and hence must be neglected.
iv. From the files that contain the desired section(s), we store the results i.e. counting of various fields and actions observed in the file, into a common JSON file, which would be later retrieved for analysis.
v. While the above portion has been implemented using NodeJS, we perform the actual analysis, inference and visualization using Python scripts, one for each research question. Based on the results stored in a global JSON structure as mentioned above, we plot various graphs to help understand the data distribution, and in some cases store analysis results in the common JSON.

Research Questions, Experiments and Results

Question 1

Implementation

i. While retrieving query results from GitHub API, we check if the Jenkinsfile has POST section, without which we have nothing to analyse in this file for this question.
ii. With reference to pagination (as mentioned above in the general implementation), we take 35 repositories per page, and 3 such pages in total.
iii. Each repository has a Jenkinsfile. We store frequency of various post-condition blocks as they appear in the files.

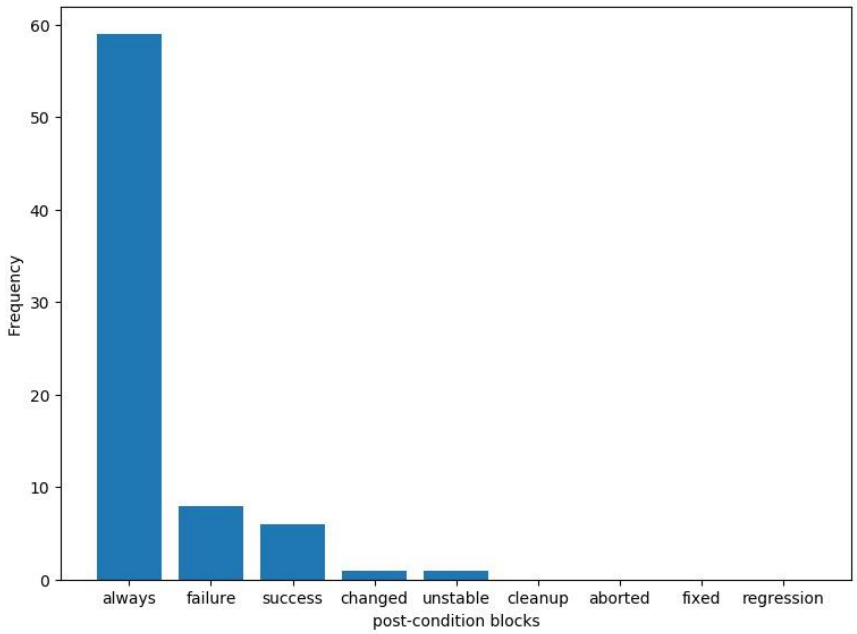
a. What are the most frequent post-condition blocks in the post section within Jenkins pipelines?

In a Jenkinsfile, the post section contains steps that run upon the completion of the pipeline or a stage, depending on the location of the post section within the pipeline. It may contain one or more of the following blocks:

-always	-changed	-fixed
-regression	-aborted	-failure
-success	-unstable	-cleanup

Experiments & Results:

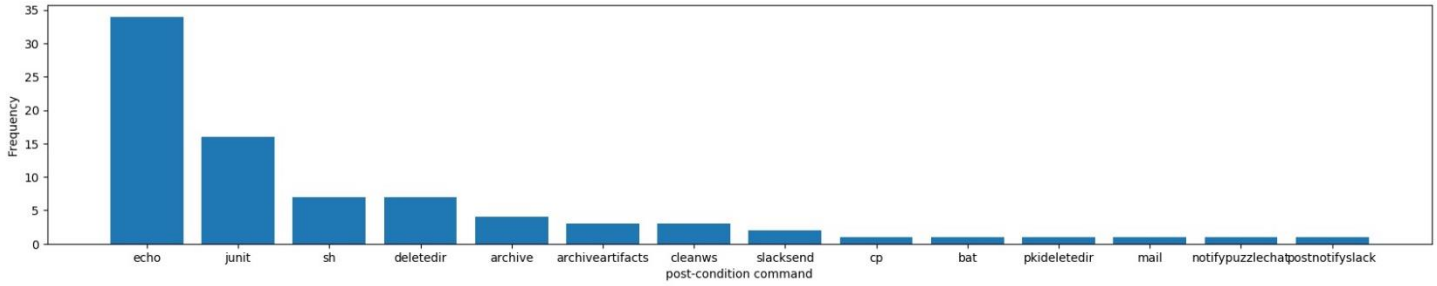
Of the 75 relevant files found on GitHub, the most frequent post-condition blocks are (in non-increasing order of frequency):



Post-condition block	Frequency
always	59
success	8
failure	6

b. What are the most frequent activities in the post section conditional blocks within Jenkins pipelines?

In each of the post-condition blocks, there are commands executed. Overall, this would be a huge list of commands, since there is a large variety according to the project's logic requirement. Following is a visualization of the most frequent commands used in the post-section:



Here, in each file, we check for post-condition blocks and commands used within them and store a count of all such commands. As can be inferred from above graph, following are most frequent activities(commands):

Commands in post-condition block	Frequency
echo	34
junit	16
sh	7
deletedir	7

Question 2

How is the presence of triggers in a pipeline correlates with the number of stages in the pipeline? What are the common types of triggers used in pipelines?

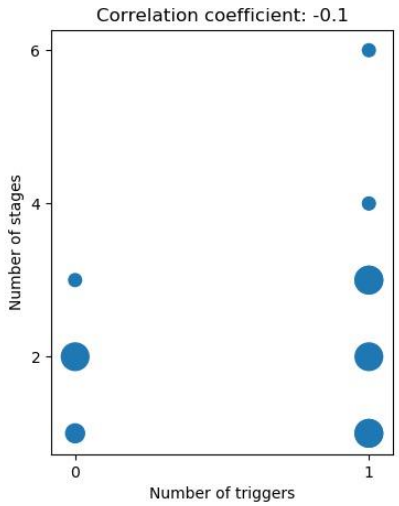
Triggers in Jenkinsfile pipeline define when the pipeline should be re-triggered. Although there are 3 supported trigger types *cron*, *pollSCM*, and *upstream*, our study found an additional trigger method: *githubpush.*

Implementation:

i. To answer the question, we parse all Jenkinsfiles and store count of triggers as well as count of the number of stages in each pipeline. This gives us 2 lists, one for frequency of triggers and other for that of stages
ii. We then calculate Pearson correlation coefficient between these lists.

Experiment & Results:

Although we tried some variations with the search query, the number of files accepted and few other aspects such as filtering by keywords and size of the file, yet the correlation coefficient was consistently in the range (-0.05 to -0.2). A correlation of this range indicates very low correlation between the number of triggers and number of stages. Following is a plot to visualize the data distribution (Bigger the bubble, more is the frequency):



Question 3

a. What are the most and the least frequent commands in pipeline stages?

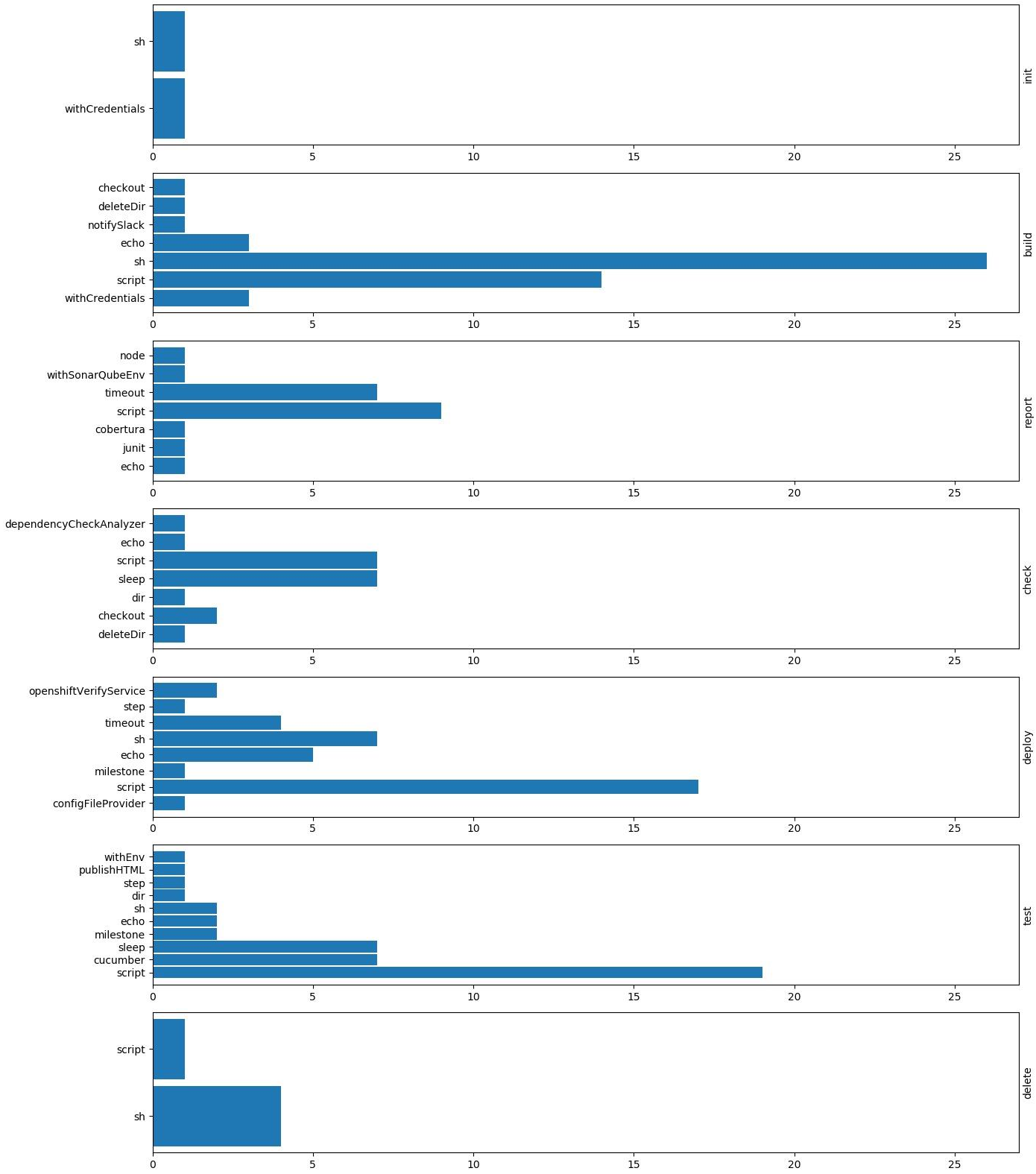
Similar to determining distribution of commands used in post-section, we also study the distribution of commands used in various stages of the pipeline, based on their frequency.

Implementation:

i. Every project has its own terminology for stages. For instance, stage “test” has a variety of names: “testing”, “perform tests”, “testing stage”, “run tests”, and so on for all stages. We generalize the stage names based on specific keywords present in the stage name., which helps us group more commands under same stage type.
ii. While storing frequency of each command, we store them grouped by the stage name.
iii. We filter out stages which appear rarely in the parsed Jenkinsfiles. Further, we also filter out commands which rarely appear in these frequent stages.

Experiments & Results:

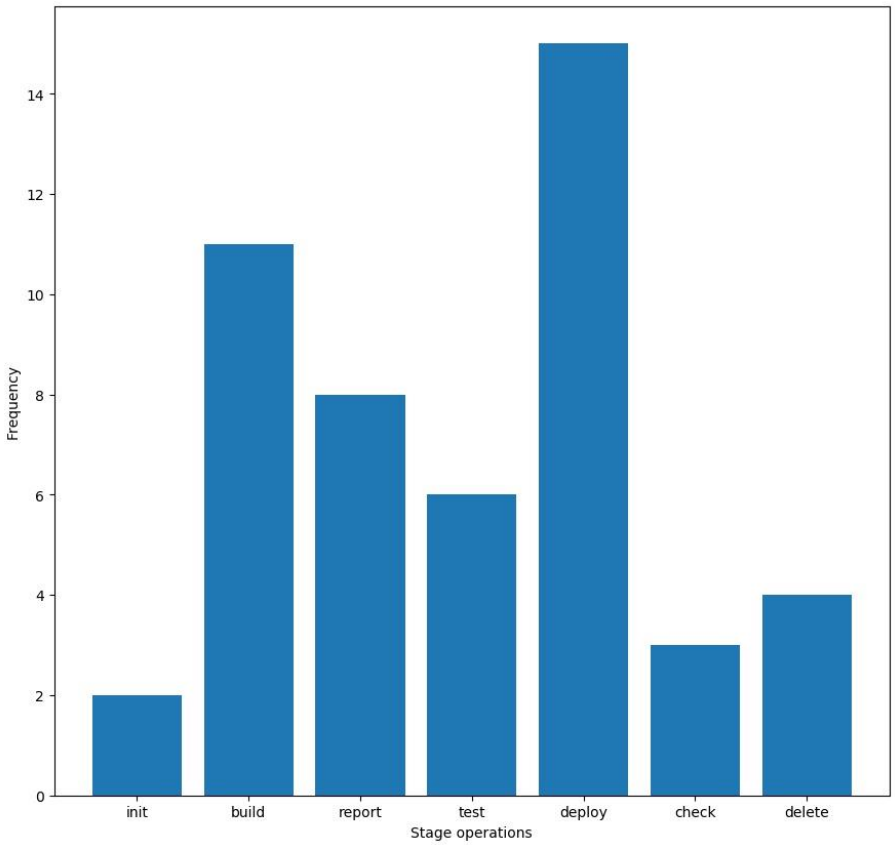
A huge variety of commands was found in different stages, and there are several commands that are found in only specific stages. For instance, “cobertura” command is found only in “report” stage, which makes sense. Following is a plot that shows frequency of various commands categorized by stages those are found in:



b. What are the most and the least frequent operations in pipeline stages?

In this question, we aim to analyze stages that are frequently found in various Jenkinsfiles retrieved. As explained above, different repositories have different names for the same stage operation. We generalize this variety by searching for keywords from a predefined list of popular stage operations already known, and then go on adding any new stage operation, if it is frequent enough.

Following is a visualization which helps us compare how frequently different stage operation are found in pipelines:



Stage operation	Frequency
Deploy	15
Build	11
Report	8
Test	6
Delete	4
Check	3
Initialization	2

Question 4

For stages in parallel, fail-fast feature is used for what type of operations(stage names)? When it is used/unused, what are the operations running in parallel?

There was an observation that “failFast” feature was used only when parallel staging was used in the pipeline. The fail-fast feature implies that out of the stage running in parallel, if one of the stages fails, then the other stages running in parallel would be aborted too.

We do an analysis of the correlation between the stage operations and presence of the fail-fast feature. Although the fail-fast feature was very rarely found, the following facts were observed:

- For majority instances, there are only 2 stages in parallel, which is the minimum number required for parallelizing stages.
- For most files, the operations of all parallel stages are the same. Eg. (“build” & “parallel build”), (“list files” & “list files2”)