Advanced Topics in Software Engineering (CS540)

Course Project Report
An Investigation into Open Source DevOps Jenkins Pipeline Repositories

Guillermo Rojas-Hernandez
Viren Mody

UNIVERSITY OF ILLINOIS AT CHICAGO
SPRING 2018

Table of Contents

1 Introduction **3**

2 Formulations of the Research Questions **3**

2.1 Research Question #1: Triggers and Stages 3

2.2 Research Question #2: Tools 3

2.3 Research Questions #3, 4, 5, 6: Archived Artifacts 4

3 Architecture and Description of the System **5**

3.1 Architecture of the System 5

3.2 Description of the System 5

4 Details of the Implementation **6**

4.1 Instructions/Setup 6

4.2 Language: Python 3.6.3 6

4.3 Technologies/Libraries 6

4.4 Algorithm/Implementation Design 6

5 Descriptions of the Experiments **7**

5.1 Research Question #1: Triggers and Stages 8

5.2 Research Question #2: Tools 9

5.3 Research Questions #3, 4, 5, 6: Archived Artifacts 9

6 Results and their Explanations. **12**

6.1 Research Question #1 12

6.2 Research Question #2 12

6.3 Research Questions #3, 4, 5, 6 14

7 Conclusion **18**

8 References **19**

1 Introduction

Open-source projects available through online code repository websites such as GitHub provide software engineers with a rich reference for potential projects. Oftentimes, the code repository will not only include the source code, but will also include the build and pipeline configuration so that developers can build the project or library. This project will be analyzing the build configuration files for the popular automation tool Jenkins, in order to get information on how software engineering projects are built today.

2 Formulations of the Research Questions

For this project, formulating questions and even choosing questions from the example list was not a simple task and required some investigation. In order to formulate these questions, we started by understanding what a Jenkinsfile is and then we had to sift through the all the details of the pipeline syntax in the Jenkins documentation in order to start formulating some questions. Next, we performed a variety of search queries on GitHub's web search interface for repositories containing actual Jenkinsfiles that we could examine to spark more wonderings about DevOps pipelines.

2.1 Research Question #1: Triggers and Stages

1: How does the number of triggers correlate with the number of stages in the pipeline?

The first question of the relationship between the number of triggers and the number of stages was simply an examples of the course project description and seemed to be a plausible start in exploring DevOps pipelines, more specifically Jenkinsfiles.

At first thought, the presence of triggers (i.e. cron or pollSCM) would leave one to believe that this pipeline must be run apparently regularly and often. I was inclined to then assume that then the pipeline must be well developed with many stages and steps. However, it is possible to have many triggers on a simple pipeline that just builds source code, or even to have a triggerless well developed pipeline with many stages such as building, testing and deploying.

Nonetheless, my assumption is that since DevOps pipelines are a strong indication of automation within a software company, that the presence of triggers or a higher number of triggers would indicate a well-developed pipeline with multiple stages.

2.2 Research Question #2: Tools

2: What tools are used by the Jenkins declarative pipeline?

The purpose of the second research question is to figure out what kinds of tools are used by the Jenkins declarative pipeline.

This is important because it can give a snapshot of what kinds of tools developers are using for their programs. It can give an idea of what build tools are on the rise, and which build tools are being used less and less.

For software architects, it can suggest that new projects should be built with the most popular tools, or that existing projects should be accomodated to use the newest and most popular tools. It also suggests that the Jenkins build server should be prepared to accommodate all of the different build tools.

For large corporations in particular, they might have different technology departments and teams that are experimenting with new technologies and different build tools. Depending on how the DevOps process is set up at the company, there might be several Jenkins build servers that need to be able to be configured to anticipate the needs of different teams. By investigating this research question, we can get a snapshot of popular tools, and be prepared for the needs of different teams.

2.3 Research Questions #3, 4, 5, 6: Archived Artifacts

3: In which pipeline sections/directives are artifacts most and least frequently archived?
4: Which artifact file extensions are most frequently and least frequently archived?
5: What percentage of archived artifacts are archived with a fingerprint?
6: Which archived artifact extensions are most and least frequently fingerprinted?

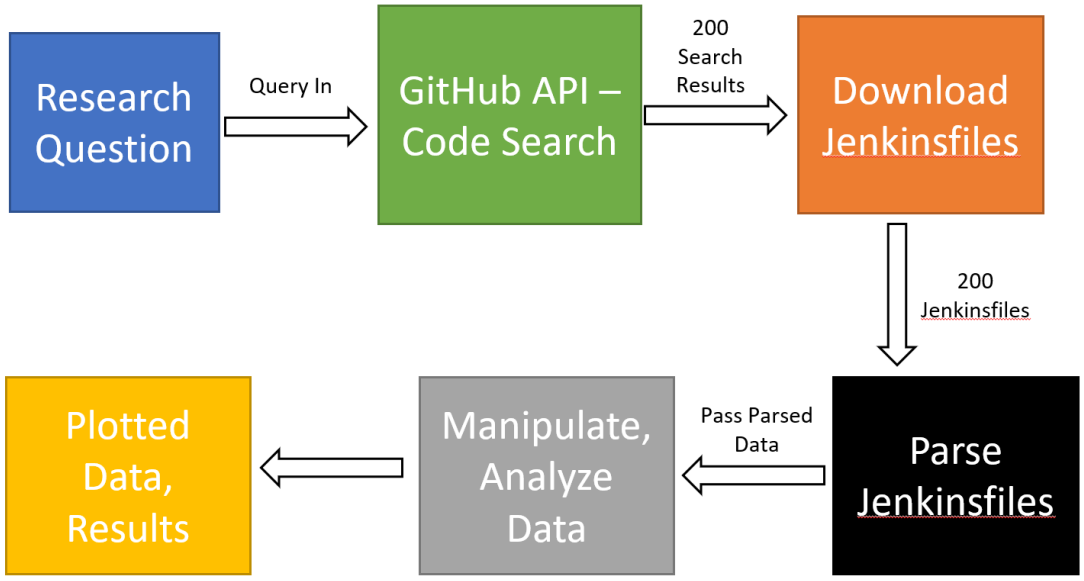
In exploration of Jenkinsfiles on GitHub for pipeline interaction with project artifacts, I found myself curious to learn more about the step: ‘archiveArtifact’. I originally was under the impression that artifacts were only archived in the post section, but I found that they were in variety of sections. I also noticed that there was a variety of different types of artifacts being archived, but in order to categorize artifacts, I thought the simplest way was to focus on file extensions (i.e. jar, war, log, etc.). One of the last things I discovered was the attribute: fingerprint. After a quick Google search (StackOverflow: jenkins, what does fingerprint artifacts means?), I learned that when the fingerprint attribute is set to true (by default, it is false), it records an MD5 checksum to keep track of which build the artifact came from and would be valuable to find problem builds, find functional builds, etc. I also imagined that only certain artifacts (i.e. builds) would be archived, but not all.

I feel the answers to these questions can be used to help develop best practices in writing Jenkinsfiles and developing pipelines for a given software company such as which artifacts to archive and which ones not to, at one point in the pipeline should artifacts be archived, which ones require tracking and which ones don’t, etc. Moreover, at very large software companies, this data used along with machine learning, may help automate Jenkinsfile development or

possibly used to verify proper Jenkinsfile commits or even correcting improper Jenkinsfile commits.

3 Architecture and Description of the System

3.1 Architecture of the System



3.2 Description of the System

Architecture and Description of the System:

- Authenticate GitHub credentials
    - Setting GitHub credentials and authenticating them was a necessary step because we utilized the code search GitHub API Call, which limits requests unless the user is authenticated.
- Search GitHub repositories for Jenkinsfiles that match the keywords specific to each of our questions
    - It was necessary to tailor each search by keyword in order to avoid Jenkinsfiles that did not have tools, or Jenkinsfiles that were not empty
    - One of the drawbacks of using GitHub for analysis is that a lot of developers upload “toy” repositories to their GitHub accounts—repositories that are oftentimes used for learning purposes or to try Jenkinsfiles for the first time. While testing different ways of searching GitHub repositories and downloading the contents of Jenkinsfiles, we ran into a lot of empty Jenkinsfiles, oftentimes it would be hundreds of results later when we’d find a useful Jenkinsfile for the question that we were answering. We considered using all of the repositories returned during a general query for Jenkinsfiles, and downloading each repository even if it contained a Jenkinsfile that was incomplete or empty. However, hitting the GitHub usage limits was a concern, and creating potentially thousands of repository directories for each research question was also a concern.
    - So, to find quality Jenkinsfiles for the questions we were trying to answer, we decided to include a search query. We found that this was effective at retrieving usable Jenkinsfiles.

- Download each Jenkinsfile
    - To download each Jenkinsfile, the algorithm search_by_code executes the search for matching Jenkinsfiles on GitHub. Once a set of matching Jenkinsfiles is found, a GitHub SearchCode object is returned for each jenkinsfile. This SearchCode object contains information on the Jenkinsfile—including the repository owner, repository name, and commit hash for that version of the Jenkinsfile. The raw version of each file on GitHub, without any website code, can be found at the root address raw.githubusercontent.com. With the information returned by the SearchCode object, and the GitHub raw file root address, the algorithm generates the raw file URL by concatenating the information together.
    - The contents from the raw file URL are then retrieved by the python requests library.
    - Using the pathlib library, the algorithm creates a directory for that repository's jenkinsfile, and then writes the results from the call to requests to the Jenkinsfile.
    - Along the way, hash maps (dictionaries) are created for bookkeeping purposes, and passed to different functions, so that information is stored on how to access the Jenkinsfiles that are created. These dictionaries are then passed to the analyze and parse functions

4 Details of the Implementation

4.1 Instructions/Setup

Please see ReadMe file on BitBucket.

4.2 Language: Python 3.6.3

Python is an interpreted, object-oriented programming language. The entire algorithm was implemented in Python 3.6.3. We opted to use Python as our language for implementation because it is well supported through tools and libraries for data collection, processing, analysis, and presentation as well as in its online communities.

4.3 Technologies/Libraries

- Github3.py: Python Wrapper for GitHub's REST API
- Pandas: Python Data Structure and Analysis Library
- Numpy: Python Scientific Computing Library
- Matplotlib: Python 2D Plotting Library
- Pycharm: Python IDE

4.4 Algorithm/Implementation Design

Please see sections **3 Architecture and Description of the System** and **5 Descriptions of the Experiments** for details.

5 Descriptions of the Experiments

The experiments performed for each research question were all similar at a general level, but differed when it came down to details.

Queries
All our research questions were limited to analyzing open source Jenkinsfiles DevOps declarative pipelines (and not scripted) found on GitHub. We opted to analyze Jenkinsfiles because a search produces thousands of results and declarative pipelines because they are seemingly more predictable than scripted pipelines for parsing. More specifically, all queries in our GitHub searches included ‘filename:jenkinsfile’ for Jenkinsfiles and ‘q:pipeline’ because the keyword pipeline indicates the start of a declarative pipeline. Queries vary for each experiment/research question because we wanted to ensure that each experiment had concrete relevant data to work with.

Architecture
All our research question experiments are also generally similar in their architecture as noted in section **3 Architecture and Description of the System**, but vary slightly in addition to search queries. For each question/experiment, we parsed Jenkinsfiles, processed the data and presented the data differently to cater to the question at hand.

Search Results
For all research questions, based on multiple trials, we noticed that requesting 200 search results (Jenkinsfiles) from GitHub produced the best results. When working with less than 100 Jenkinsfiles, there wasn’t enough data to confidently support answers to questions. When working with 300 or more results, we found that we were receiving Jenkinsfiles with unique formats, attributes, properties that seemed futile to try to process in our parse step and pulled away from what we discovered when maintaining our results at 200.

Parsing
Furthermore, the parsing methodology for all our Jenkinsfiles was the same. Since Jenkinsfiles are structured to have only statement per line, we parsed them one line at a time searching for specific keywords relevant to the question.

Data
All parsed data was stored to the Pandas DataFrame data structure.

5.1 Research Question #1: Triggers and Stages

1: How does the number of triggers correlate with the number of stages in the pipeline?

Query

Originally, this question was focused on the presences of triggers as opposed to the number of triggers. For that reason, we chose to not to include the keyword triggers in our search as it would skew the results. This was problematic because we found our results included many empty Jenkinsfiles. After changing the question from the presence of triggers to the number of triggers, we changed our query to

query=“filename:jenkinsfile q=pipeline triggers stage tools”.

We included the keyword “tools” because it produced much better results and weeded out empty or meager Jenkinsfiles.

Parsing

As mentioned earlier, Jenkinsfiles for declarative pipelines relatively have a high level of predictability, so we searched for specific keywords relevant to triggers and stages. For triggers, there are 3 standard ones: pollSCM, cron, and upstream, followed by an argument or value for that trigger. We extracted the trigger type and the value to ensure that it was a valid trigger and kept track of how many triggers we captured. For stages, we had to look for “stage” and not “stages” as “stages” is the parent section for “stage”. We also extracted the stage name as a way to validate that it was a real stage. Trigger argument/value and stage names were extracted using the regular expression library. All triggers and stages that didn’t have an argument/value or a name respectively were discarded with exception and error handling.

High Level Pseudocode:

```
triggers_data =[]
stages_data = []
num_triggers = 0
num_stages = 0
For each Jenkinsfile
      For each line in Jenkinsfile
            If line contains a standard trigger type (pollSCM, cron, or upstream)
                  Extract trigger argument/value
                  Store trigger type and trigger argument/value in triggers_data
                  Increment num_triggers
            If line contains the word “stage” and not “stages”
                  Extract stage name
                  Store the stage name in stages_data
                  Increment num_stages
Return triggers_data, num_triggers, stages_data, num_stages
```

Data

To calculate the Pearson correlation coefficient for the number of triggers to the number of stages, it simply required just the number of occurrences of each. We used the numpy library to easily calculate the Pearson correlation coefficient of the two.

5.2 Research Question #2: Tools

2: What tools are used by the Jenkins declarative pipeline?

Query

The following query provided the most accurate results:

query="filename:jenkinsfile q=pipeline tools".

Parsing

For parsing, we built off of the code for research question number 1, as "tools" was a relatively straightforward keyword. The regex phrase used had to be changed to capture the tools that were within quotation marks. For the tools parsing, we also optimized parsing by setting a tools capture on and capture off boolean that stopped parsing lines for tools once an open bracket was detected. This caused the parsing to be faster for this tool than for question 1, but the parsing was more clear cut in this case--all tool configurations started with open brackets and ended with closed brackets.

Data

Once the type of tools used in each repository was captured, we calculated the occurrences of each tool, the average number of tools used in each repository, and then the percentages of each tool found.

5.3 Research Questions #3, 4, 5, 6: Archived Artifacts

3: In which pipeline sections/directives are artifacts most and least frequently archived?
4: Which artifact file extensions are most frequently and least frequently archived?
5: What percentage of archived artifacts are archived with a fingerprint?
6: Which archived artifact extensions are most and least frequently fingerprinted?

Query

For this question, the main focus is the step `'archiveArtifact'`, but to enhance the quality of the results, other keywords like "tools", "triggers", "stage", "post", etc. were used, but we found the best results in using a simple query to search for 200 Jenkinsfiles:

query="filename:jenkinsfile q=pipeline archiveartifacts".

Based on our results, it appears that the keyword "archiveArtifacts" warrants a relatively more substantial Jenkins pipeline given all the repositories available on GitHub.

Parsing

Parsing for archive artifacts was more complex than parsing for triggers and stages because there was less predictability in what we could expect in the Jenkinsfile. For this reason, we relied a little more on regular expressions to extract data.

For question #3, we needed to extract the section that `'archiveArtifact'` was used in which either was a “stage” or one of the many post section conditions (i.e. always, success, failure, unstable, etc.). This was a unique challenge since we were parsing line by line, we needed a way to access previous lines to extract the section the artifact(s) were archived in. For this I implemented a list as a stack (First In Last Out), and any time we found an artifact, we would pop from the stack or iterate the list in reverse until we reached the section it was in.

For question #4, we needed to extract file extensions. We used the regex library to extract the artifact, and then used the Python pathlib library to extract extensions from the artifacts.

For question #5, we needed to extract the fingerprint attribute boolean. This was handled simply with regex, but when testing it on a couple hundred results, we came across a couple unique formats, where the fingerprint attribute was on the following line, and it wasn’t a boolean, but the actual artifact. In those cases, if there was no fingerprint attribute, we would just check the next line. If an artifact was the fingerprint value, then it was safe to store the boolean as true.

For question #6, we already had the data we needed from the previous questions.

High Level Pseudocode:

```
section_stack = [] // stack to store section names
artifacs_data = []
num_artifacts = 0
For each Jenkinsfile
        For each line in Jenkinsfile
                If the line contains an open bracket
                        Push that line (which is a section name) on to section_stack (stack)
                If the line contains a closing bracket
                        Pop the top line (which is a section name) off section_stack (stack)
                If the line contains ‘archiveArtifact’
                        Extract the artifact and store in artifacts_data
                        If the line contains ‘fingerprint’
                                Extract the fingerprint boolean and store in artifacts_data
                        Else
                                If the next line contains ‘fingerprint’
                                        Extract the fingerprint value and store in artifacts_data
                                Else
                                        Store the fingerprint value as ‘false’ (the default value)
                        If the line contains ‘onlyIfSuccessful’
                                Extract the onlyIfSuccessful boolean and store in artifacts_data
                        While section_stack not empty // Extract the section name archive artifact occurs
                                popped = pop top element from section_stack
                                If popped contains ‘stage’
                                        Store section as ‘stage’ and stage name as section name in artifacts_data
                                If popped contains any of the post conditions (i.e. always, success, etc.)
                                        Store section as ‘post’ and condition as section name in artifacts_data
                        Increment num_artifacts
        If section_stack is not empty (at the end of the Jenkinsfile) // imbalanced brackets
                Discard data stored and skip Jenkinsfile // because the brackets are not balanced
Return artifacts_data, num_artifacts
```

Data

Calculating the data for the archived artifacts research topic was more involved because there were 4 questions, but also involved performing operations and manipulations on the data to retrieve data to answer the questions.

For question #3, we filtered out all data except for sections and occurrences. Then we grouped the data by section, performed a count of occurrences of each section (i.e. stage, post-always, post-success, etc), added a column for percentage and calculated it, sorted the dataframe by occurrences descending, and reset the index. We wrote the data to a csv file and created a bar chart to represent the Number of Archived Artifacts Per Section and a pie chart to represent the Percentage of Archived Artifacts Per Section.

For question #4, we filtered out all the data except for extensions and occurrences.Then we grouped the data by extension, performed a count of occurrences of each extension (i.e. jar, war, log, etc.), added a column for percentage and calculated it, sorted the dataframe by occurrences descending, and reset the index. Now, our data showed a large number of extensions, many of which that occurred less than 3% of the time. We felt like this was taking away from what the data was trying to tell us, so any file extension occurring less than 3% of the time was categorized as “Other” and then grouped together and their occurrences averaged as if it was a single extension. We decided to do this data manipulation to really show the weight of the occurrences of the more meaningful extensions. We wrote the data to a csv file and created a bar chart to represent the Number of Archived Artifacts Per Extension and a pie chart to represent the Percentage of Archived Artifacts Per Extension

For question #5, we filtered out all the data except for fingerprint and occurrences. Then we grouped the data by fingerprint, performed a count of occurrences of each fingerprint (i.e. true/false), added a column for percentage and calculated it, sorted the dataframe by occurrences descending, and reset the index. We wrote the data to a csv file and created a bar chart to represent the Number of Archived Artifacts Fingerprinted (vs not Fingerprinted) and a pie chart to represent the Percentage of Archived Artifacts Fingerprinted (vs not Fingerprinted).

For question #6, we filtered out all the data except for fingerprint, extension, and occurrences. This was slightly more complex because we were working with two data columns instead of one in extensions and fingerprint. We grouped the data by fingerprint and extension, performed a count of occurrences of each fingerprint (i.e. true/false) per extension (i.e. jar, war, log, etc.), added a column for percentage and calculated it, sorted the dataframe by occurrences descending, and reset the index. This time around, we only cared about artifacts that were fingerprinted, so we filtered out all rows of data containing fingerprints that were false and recalculated the percentages since about half the data was removed. Now at this point, we had the same problem as the previous questions with too many extensions that occur less frequently. In order to get more out of the data, we filtered out all extensions that were fingerprinted but occurred less than 2% of the time. We then calculated the percentage again since some data was removed. We wrote the data to a csv file and created a bar chart to represent the Number of Archived Artifacts Fingerprinted Per Extension and a pie chart to represent the Percentage of Archived Artifacts Fingerprinted per Extension.

6 Results and their Explanations.

6.1 Research Question #1

1: How does the number of triggers correlate with the number of stages in the pipeline?

The correlation coefficient for the number of triggers to the number of stages in the pipeline is **0.13571**. Correlation coefficients range between -1.0 and +1.0, where -1.0 represent a negative correlation, 1.0 represents a positive correlation, and 0 represents no correlation. 0.13571 shows a very low correlation between the number of triggers and the number of stages which proves my assumption wrong. However, since it is possible to have a pipeline with many stages and no triggers and vice versa, it is very plausible to find a low correlation value between the two. I personally do not see any value coming from this research question. I assume that this correlation value is really low because we are extracting projects from GitHub and there's no guarantee that we are getting industry level or calibur Jenkinsfiles. On that note, I assume that if we were to process software companies' Jenkinsfiles, the correlation value would be much higher assuming that there would be multiple triggers and stages.

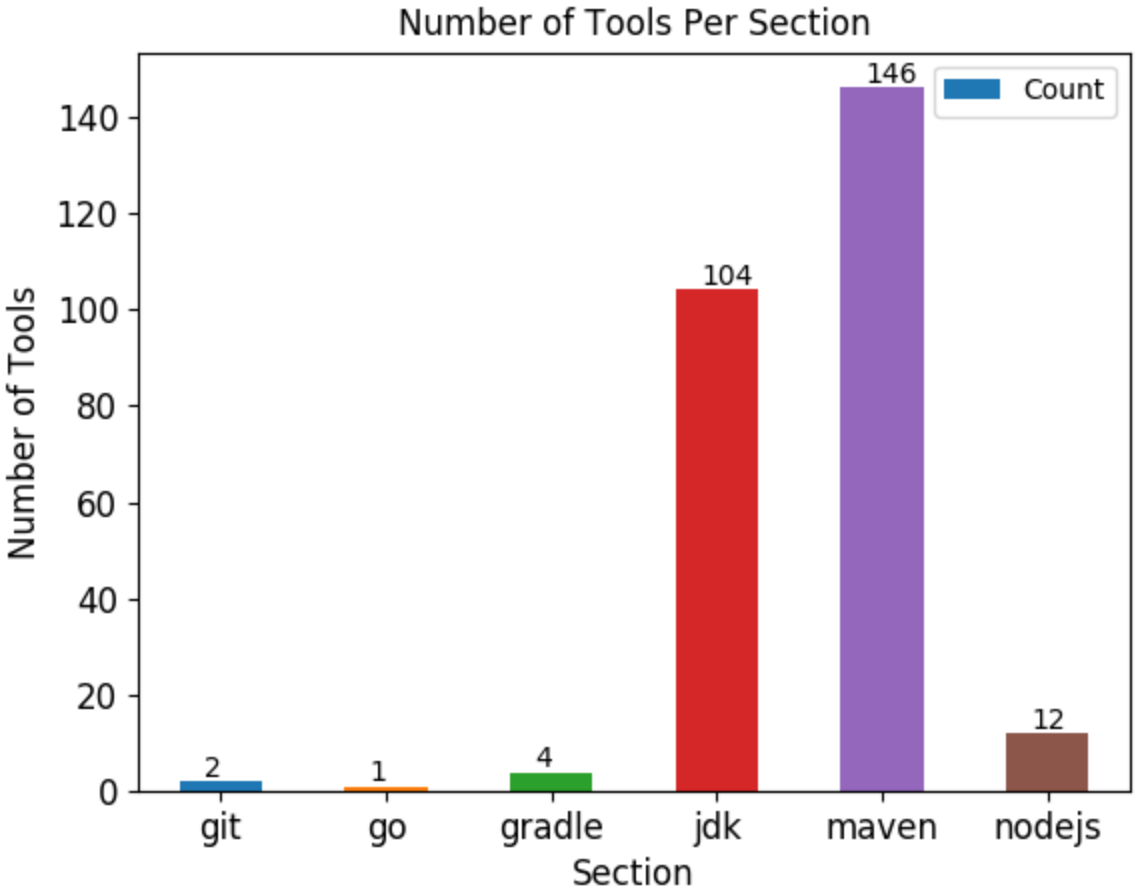
For question #1, all data was written to **research_question_1_stages_triggers.csv** which can be found in the /src folder.

6.2 Research Question #2

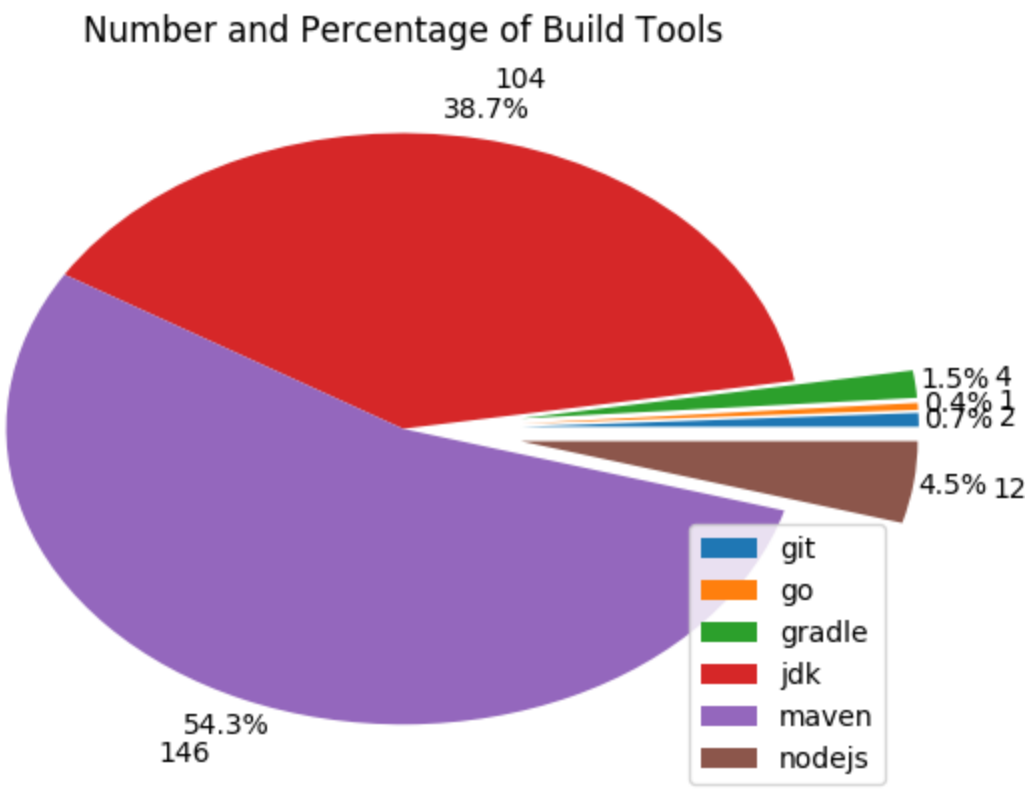
2: What tools are used by the Jenkins declarative pipeline?

The most frequently used tool is Maven, followed by Java's JDK. While we did expect Maven to be a very frequently used build tool, it was unexpected that Java's JDK would be so popular. However, this makes sense, as projects that are built using a Jenkinsfile pipeline are more likely to be Java projects with complex build procedures, as opposed to GitHub repositories using Python or web-development technologies that do not require a build tool.

For question #1, all data was written to **research_question_2_tools.csv** which can be found in the /src folder.



After processing 200 Jenkinsfiles, we were able to parse 180 files successfully. We had 6 different build tools. The average number of build tools used per repository was 1.4944.



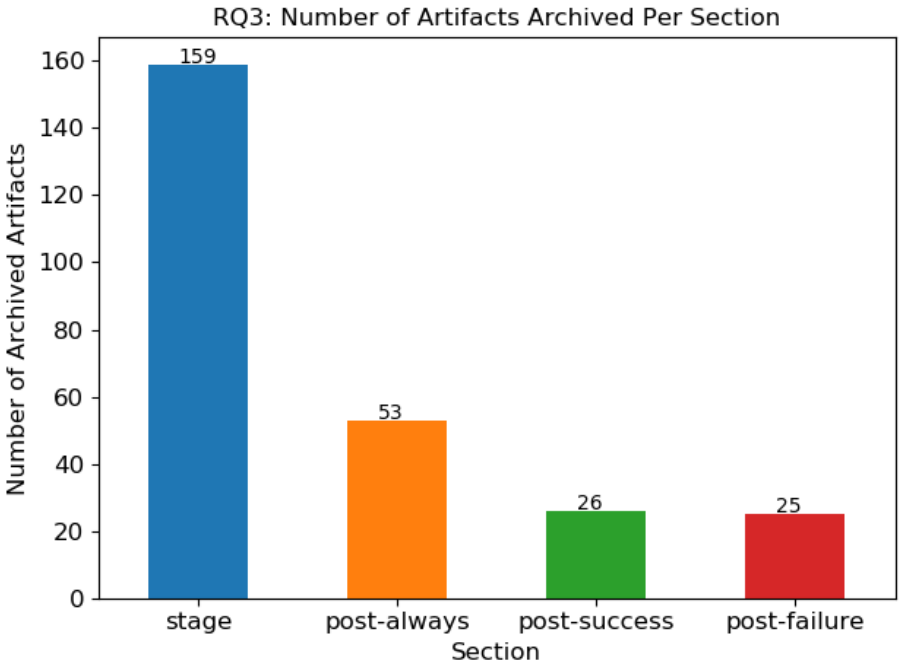
The most frequently used build tools were Maven, followed by Java's JDK. It was interesting to see unexpected tools represented, such as Go and NodeJS.

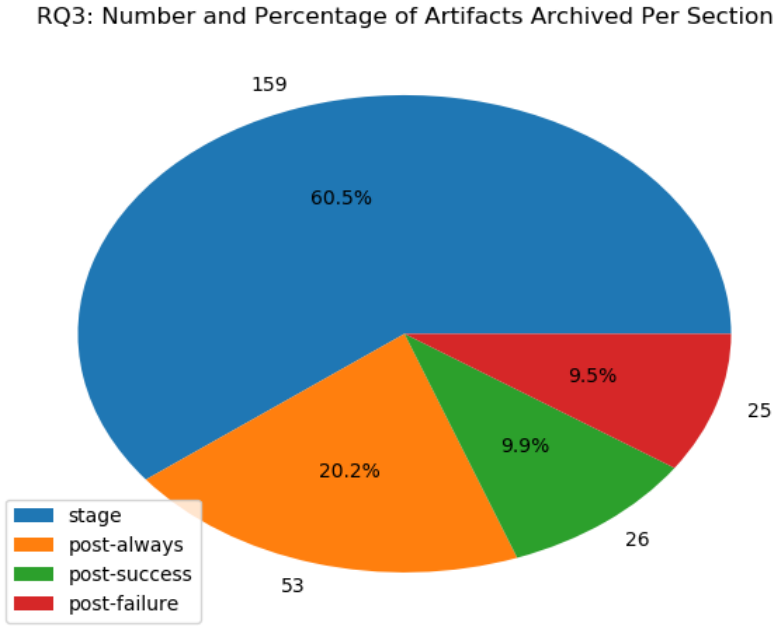
Future work may analyze what kinds of specific versions are the most popular. Looking through the output for the experiment, it was hard to devise a parsing pattern for the specific different tool versions used.

6.3 Research Questions #3, 4, 5, 6

For questions #3, 4, 5, 6, all data was written to the following csv files which can be found in the /src folder:
research_question_3456_artifacts_raw_data.csv
research_question_3_artifacts_sections.csv
research_question_4_artifacts_extensions.csv
research_question_5_fingerprint_tf.csv
research_question_6_fingerprint_extension.csv

3: In which pipeline sections/directives are artifacts most and least frequently archived?

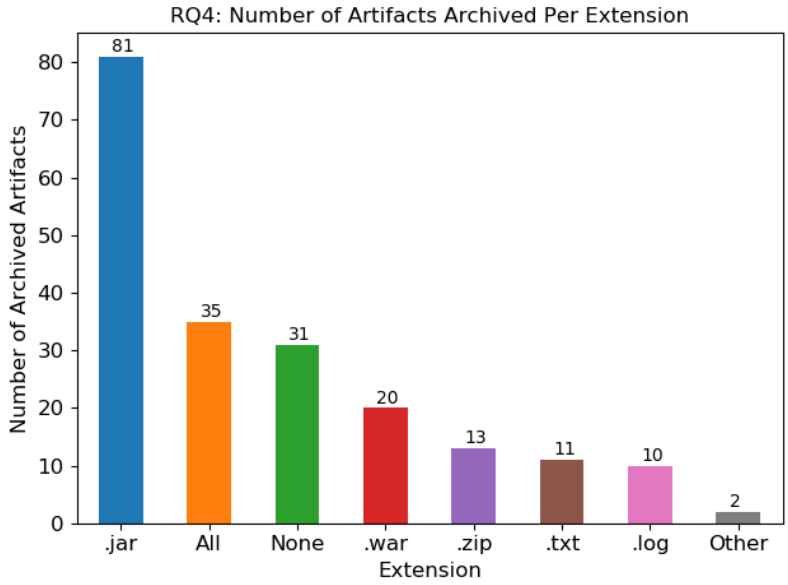


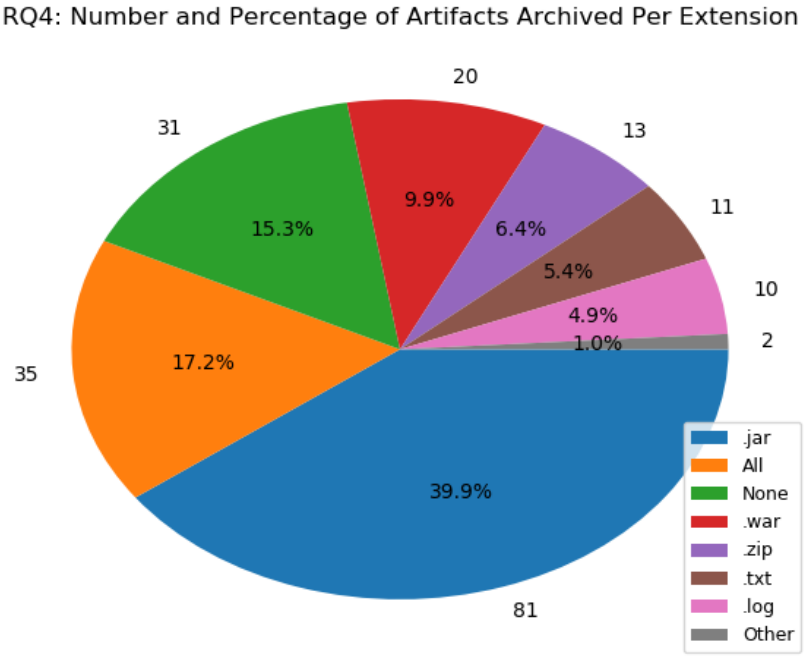


After processing approximately 200 Jenkinsfiles, artifacts were found to most frequently be archived in a stage at 60.5% of the time, and least frequently in the post section in the failure condition at 9.5% of the time. It makes sense to archive artifacts in the stage section especially if the stage is the “Build” stage because an artifact is archived immediately after being built. However, what if that build doesn’t pass the tests in the “Test” stage, then there’s no value to that artifact being archived? Although, post-failure is the lowest frequency for archiving artifacts, I wonder why are artifacts being archived if the pipeline failed and especially in the post condition failure? Which artifacts are being archived? I feel like this data is helpful because it can be used to understand the average developer’s practices in writing Jenkinsfiles. The stage section is where most artifacts are archived, but which artifacts are they?

Are they being archived when the “onlyIfSuccessful” attribute is set to true? Maybe it is a better practice to archive artifacts in the post condition:always section, rather than in stages. It’s also possible that not all developers know that this is in an option.

4: Which artifact file extensions are most frequently and least frequently archived?

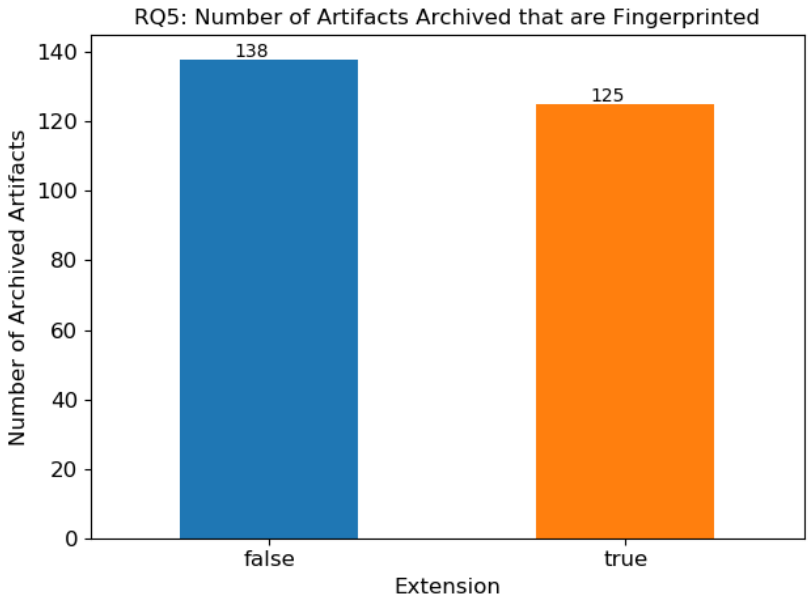


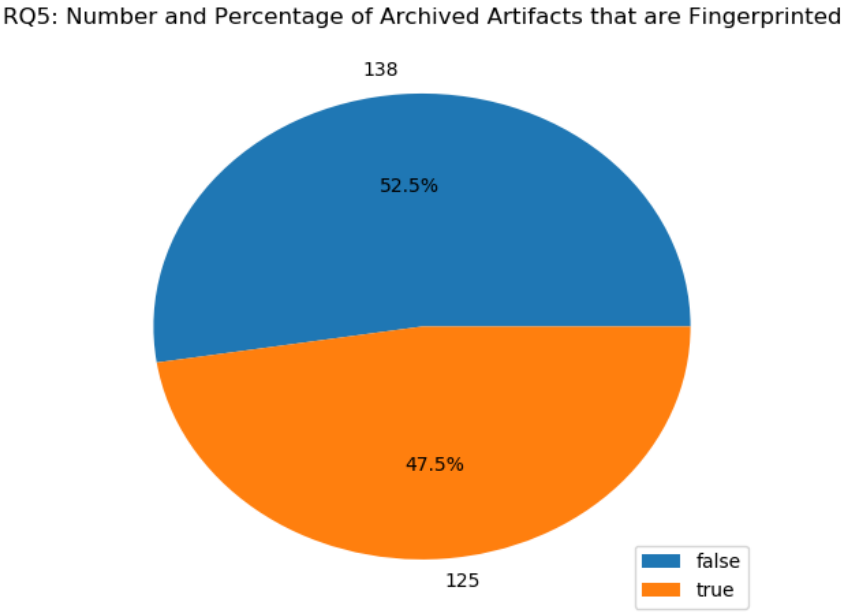


The most frequent extension archived are jar files at 39.9% and the least frequent after ‘Other’ are .log files. My guess is that if the most frequent are jar files, and 4th most frequent are war files, then we’re probably analyzing many Java projects. It makes sense to archive jar files in the case a developer needs to revert to an old build or jar file. Second most frequent are “All” extensions, which typically means archive all the artifacts of a given directory. I would have imagined that many more would have

archived log files associated to a build artifact because if it turns out to contain a bug, the log files would be beneficial to evaluate and find the bug.

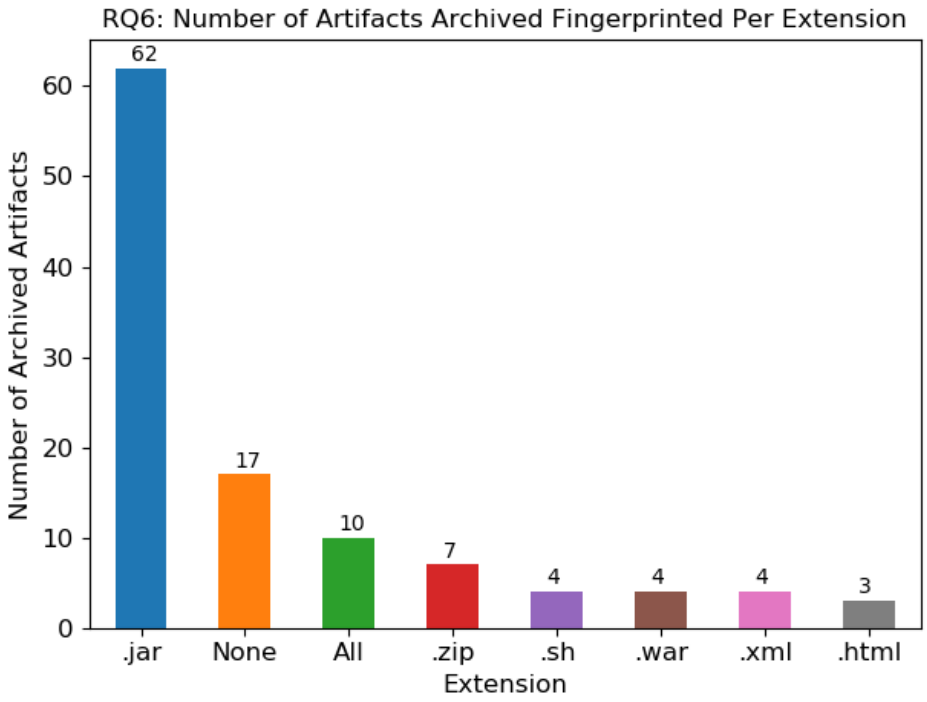
5: What percentage of archived artifacts are archived with a fingerprint?



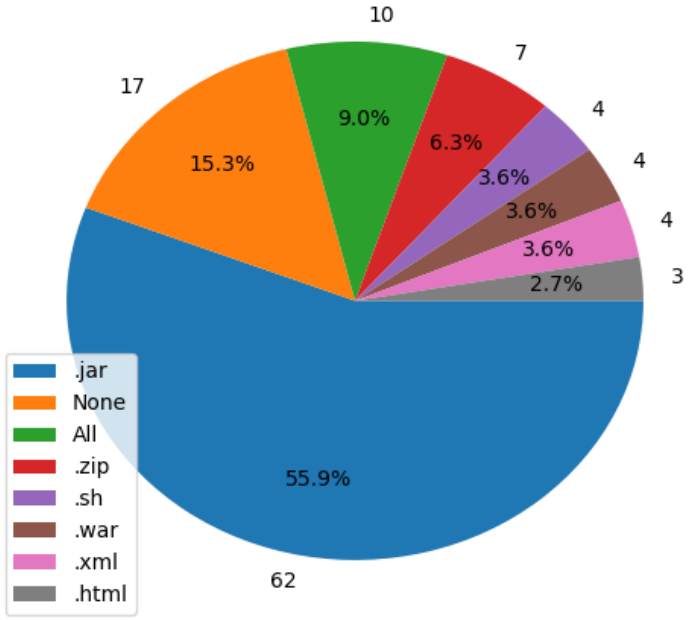


When artifacts are archived, almost 50% percent of the time (actually 47.5%), they are fingerprinted, which allows developers to track which version that build worked with. My understanding is that this is a crucial part of archiving artifacts because what is the use of a large storage of artifacts which cannot be reverted to without a fingerprint.

6: Which archived artifact extensions are most and least frequently fingerprinted?



RQ6: Number and Percent of Archived Artifacts Fingerprinted Per Extension



After going through questions #5 and #6, I started to wonder which extensions were most often fingerprinted. Again, jar files are the artifacts most frequently fingerprinted and the least frequent was a list of file extensions, some of which were filtered out because they appeared less than 3% of the time.

7 Conclusion

In conclusion, in this course project, we analyzed DevOps Jenkins declarative pipelines retrieved from GitHub open-source applications. We asked and answered 6 research questions which were grouped into 3 research topics: the relationship between triggers and stages, the tools commonly used, and the artifacts that are archived in the pipeline. For each research topic, up to 200 Jenkinsfiles were parsed into data which was analyzed and the results presented. These results helped us discover insights into Jenkins pipeline development practices as well as led us to new wonderings.

8 References

Jenkins Project Infrastructure Team. (First Published: 2017, February 2) (Last Updated: 2018, April 17). Pipeline Syntax. Retrieved from https://jenkins.io/doc/book/pipeline/syntax/