Apache Flink: Dependency Analysis Using Import Statements and GPT3.5 Turbo FlinkForce

Rafael Dolores[1] **Hashir Jamil**[1] **Alex Arnold**[1] **Zachary Ross**[1]
Nabaa Gaziy[1] **James Le**[1] **Walid AlDari**[1] **Maaz Siddiqui**[1]
[1]Department of Electrical Engineering & Computer Science, York University
{rafd47,hashirj,alex290,zachross,Nabaagz,jamesqml,walidald,msiddiqi}@my.yorku.ca

Abstract

This report explores the evaluation of three architectural dependency extraction techniques—Understand, Import Checking, and LLM Extraction (GPT3.5 Turbo). The document details the implementations, description about each method, qualitative and quantitative analysis. The quantitative analysis involves statistics on total dependencies and commonalities, accompanied by precision and recall metrics. The qualitative examination employs statistical inference techniques to understand differences comprehensively. The report concludes with insights into the pros and cons of each technique, aiming to inform software development decisions.

1 Introduction

In large scale open-source software projects, architectural recovery and subsequent dependency analysis are key steps to provide updated descriptions of the software project. It is not uncommon for well established projects to evolve faster than their documentation; furthermore, poor project awareness by developers can also lead to documentation discrepancies [1]. One key area of interest in software architecture is the mapping and management of dependencies between components. Many efforts are directed towards modelling these dependencies such as the dependency structure matrix as proper management of dependencies is key to proper management of large scale open-source software [2].

In this project show three techniques for architectural recovery for the distributed data stream framework Apache Flink [3]. We do this by building upon previous investigations into the architecture of Apache Flink [4–6]. The data from the three techniques is used to generate dependency lists. We take stratified samples of these lists then verify their statistical significance. The purpose is to provide recommendations and propose future research concerning architectural recovery. Before moving further into the study it is necessary to provide some brief background.

1.1 Background

Apache Flink is a widely used distributed stream processing framework used to process large volumes of data in custom applications [3]. It is managed by Confluent and the Apache Flink Project Management Committee (PMC) [4–7]. This framework has several modules and is written with multiple languages such as Java, Scala, Python and Typescript [8]. Apache Flink's key promise is stateful checkpoints that allow for robust disaster recover procedures. In the previous work the focus was on showing the underpinnings of the overall architecture but with a focus on the checkpoint subsystem. Moreover, this previous work used the source code analysis tool called Understand for the extraction and recovery of the concrete architecture [9]. This was done with a focus on mapping dependencies between subsystems/directories/Java files. The results in these works were conclusive and satisfying, however, it is important to use other techniques to verify these results. This is for two reasons. First, it is to ensure that the results are truly conclusive. Second, Understand is a closed source project and the exact methodology of its extraction is basically a black box. Therefore, we created two novel techniques to verify the architectural recovery.

2 Methodology

There are 4 major methodologies we used to conduct this study. They will be described in detail here. As well, the source code is available on the GitHub Repository for this project [10]. The first three methodologies consist of the three different software extraction techniques. The forth methodology is the process we followed to compare the different extraction techniques.

2.1 Understand Technique

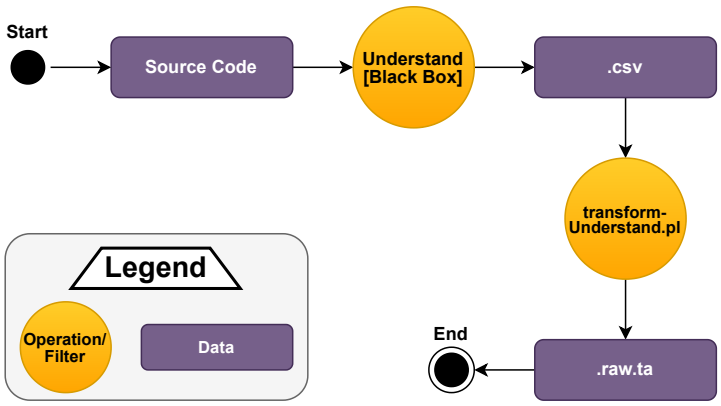

Fig. 1: Process flow for Understand architectural extraction

Understand conducts dependency analysis by scrutinizing the connections among various elements within the codebase. Dependency extraction is vital for comprehending the diverse components of the system and their interactions. While the specific dependency extraction methods employed by Understand are not explicitly documented, there exists a general understanding of its operational approach.

The initial phase of dependency extraction involves parsing the code, enabling Understand to identify distinct elements and classes within the provided source code. Subsequently, Understand generates a representation of the code structure in the form of an Abstract Syntax Tree (AST), allowing for the interpretation of hierarchy and relationships among different entities.

Following thorough analysis and comprehension of dependencies, Understand categorizes various relationships, distinguishing between direct and transitive dependencies. This capability is crucial for developers as it facilitates an understanding of the impact of changes and aids in identifying potential issues.

Moreover, the outcomes of the dependency analysis are visually presented through graphical representations, expediting developers' understanding of dependencies and providing a tangible depiction of the code structure, thereby enhancing recall. Leveraging such a tool assists developers in gauging code complexity, overall quality, and maintainability [9].

The procedure for employing this technique unfolded in the following manner: initially, we input the Flink source code into the Understand program, generating a CSV file containing dependencies in a challenging format to interpret. Subsequently, we utilized a Perl script to process the CSV file, converting it into a `.raw.ta` format that was both legible and practical for analysis. The described process is visually represented below Figure 1. The .raw.ta file outlined all the dependencies that the Understand program has extracted from the Aache Flink source code.

Importantly, developers can actively engage with the analyzed results, exploring the codebase, searching for specific dependencies, and navigating through different software segments. This interactive functionality is pivotal for developers to gain a practical understanding of code structures.

In conclusion, the Understand tool empowers developers to make informed decisions regarding code modifications, refactoring efforts, and overall software maintenance.

2.2 Import Technique

A python script was developed to meticulously extract and map these relationships in order to gain a deeper understanding of the intricate web of dependencies that is present in the Apache Flink project's source code. The algorithm, which ensures a comprehensive and intelligible extraction process, is depicted in Figure 2. The pseudocode for this process is show in Algorithm 1.

Launching the procedure begins with a scan of the source code repository. This procedure retrieves "import" statements (bearings to dependency links) recursively from each file. Next, in a critical mapping, the extracted package names are converted into corresponding file paths, creating a physical link between the package names and their locations within the project's directory structure.

The script makes a choice after locating the import statements. The operation continues if imports are discovered; if not, it terminates. When imports are present, a dependency list is generated at the end that highlights the inter-dependencies throughout the codebase and lists the relationships between files. This dependency list, which is an essential process artifact, is then saved in a file called `raw.ta`.

The formalization of this process enhances our understanding of the code's structure and makes traversing and managing the source more efficient. By automating the dependency extraction, the approach lowers the potential for human error and provides a dependable and repeatable procedure.

Algorithm 1 Import Checking

```
1:  procedure EXTRACTDEPENDENCIES
2:      sourceDir ← source code directory of the Apache Flink project
3:      dependencyFile ← empty list for dependencies
4:      for each file in sourceDir do
5:          if file is a Java source file then
6:              for each line in file do
7:                  if line contains an import statement then
8:                      packageName ← extracted package name
9:                      filePath ← map package name to file path
10:                     Add (filePath, packageName) to dependency-
    File
11:                     convert packageName ∈ (filePath, package-
    Name) to packageFilePath
12:                 end if
13:             end for
14:         end if
15:     end for
16:     Write dependencyFile to raw.ta
17: end procedure
```

2.3 LLM Model GPT3.5 Turbo Technique

For the LLM technique, we began by selecting an appropriate LLM model to use. CodeLlama is a great option however the VRAM requirements were higher than what was available to us at the time, and is slightly better at code related tasks than GPT3.5. GPT4 would also have likely yielded the best results of all 3 models, however the additional cost made it prohibitively expensive. In initial cost estimates we estimated $10 for GPT3.5 and more than $600 for GPT4. As such we opted to go with GPT3.5 Turbo 1106 (the latest release). This release has added functionality for parsing JSON and batching requests however we chose to do that part ourselves so that we could reduce API calls via feedback prompting.

2.3.1 Loading Documents

We began by going through the source code file by file, ignoring anything that was not a Java file. One thing we could have done here was remove this limitation and look at all files, however the team has decided to reduce overall cost and run-time. Additionally, refraining from analyzing additional data would simplify the comparison with other techniques. For each file we also chose to remove comments as we believe. Another measure we took to cut cost was truncating the source files when they reached over a certain token threshold. The degree to which we truncated the files, and the threshold at which we did so, was determined by initial tests with only 20 files in the directory. We used these tests to quickly determine the best parameters such that we could avoid losing any dependencies from the vast majority of files.

We chose to truncate to 800 characters when the threshold of 1000 tokens was reached. Given more resources, we possibly could have improved the results by not truncating anything and scanning the whole code-base. Working in java however, the imports will always be found at the top of the file, and likely main methods and dependency injections as well, so we would not expect to lose a lot of dependencies by doing this. The python functions to accomplish

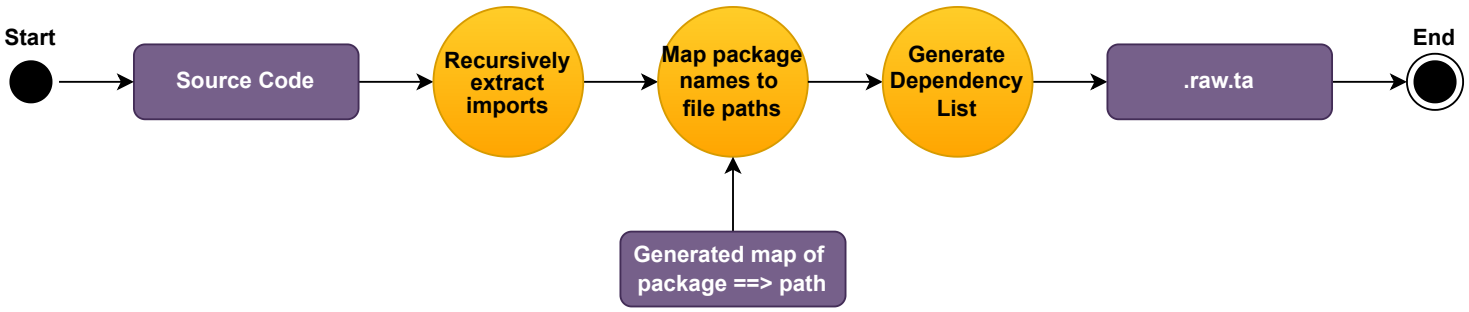

Fig. 2: Process flow of custom script for recursive Import Checking architectural extraction

this pre-processing is shown below.

```
import tiktoken
import os
import re
path = ".\\flink-1.17.1"

def remove_comments(code):
    # Removing /* ... */ comments
    code = re.sub(r'/\*.*?\*/', '', code, flags=re.DOTALL)

    # Removing // comments
    code = re.sub(r'//.*', '', code)
    return code

def read_file(file_path):
  with open(file_path, "r", encoding="utf-8") as f:
    return f.read()

def load_documents():
  documents = []
  encoding = tiktoken.encoding_for_model("gpt-3.5-turbo-instruct")

  for dir_path, _, file_names in os.walk(path):
    for file_name in file_names:
      if file_name.endswith(".java"):
        content = read_file(dir_path + "\\" + file_name)
        content = remove_comments(content)

        if len(encoding.encode(content)) > 800:
          content = content[:1000]

        documents.append({"file_path": dir_path + "\\" + file_name, "source": content})
  return documents
```

2.3.2 Prompts

Next we defined our prompts. When parsing JSON from LLM output, we chose to use our own method of retry prompting wherein we give feedback to the model on its previous incorrect response to further improve the second try. The first prompt we defined was the initial lookup, wherein we ask the model to extract any valid internal dependencies that the given file may depend on, and then we provide the file name and source. We also reiterate several times to reinforce the output being in valid JSON format. Our second prompt is for the retry; when the first prompt produces invalid JSON or JSON which does not consist of valid dependency strings. For the retry prompt we simply feed back the previous incorrect response, and then reiterate the initial prompt. The prompt structure is shown below.

```
from langchain.prompts import PromptTemplate

initial_prompt = PromptTemplate(
    template="""
    Instruction: You will only return valid JSON. Given the following code, extract any
      internal dependencies.
    Output must be a valid JSON array of strings. For the given code you must
    determine what external files or packages it depends on, and return them.
    File Path: {file_path}
    Code: {source}
    Answer in valid JSON: \n\n###\n\n""",
    input_variables=["file_path", "source"]
)

retry_prompt = PromptTemplate(
    template="""
    The previous response, "{response}" was not valid JSON. Please try again.
    Instruction: You will only return valid JSON. Given the following code, extract any
      internal dependencies.
    Output must be a valid JSON array of strings. For the given code you must
    determine what external files or packages it depends on, and return them.
    File Path: {file_path}
    Code: {source}
    Answer in valid, unformatted JSON: \n\n###\n\n""",
    input_variables=["response", "file_path", "source"]
)
```

2.3.3 Batching

Given the large size of Flink, there are several issues with simply sending all of the requests concurrently. First, there is a rate limit on tokens per minute, per day, as well as requests per minute and per day. With the API key we used, our rate limits would cut us off approximately 3/4 of the way through processing Flink if we tried to do it all in one day. Similarly if we try to send more than 5-10 requests at once, we will also get rate limited by tokens per minute. Hence, we chose to just send the requests sequentially in batches.

Overall the process was done in about 3 hours, nevertheless, introducing even a minimal level of concurrent requesting could have significantly accelerated this process. To get around the tokens per day limit, we saved each incremental state of the dependency list as we added to it, making it easy to reload the previous state whenever we get rate limited. That way, whenever an API failure occurs, we will not lose all of the data from that run, we can simply load the last save point and start over again. This was also a measure to prevent having any large failures that would incur further cost.

At the end of all batches, we were left with 126,518 dependencies in the list, 311 JSON parse failures, and 0 retry failures. The below code listing shows the steps and GPT3.5 Turbo API calls performed in each batch.

```
import json
from langchain.llms import OpenAI

output = []
parse_fails = retry_fails = 0

llm = OpenAI(temperature=0, model="gpt-3.5-turbo-instruct", frequency_penalty=0,
      presence_penalty=0, top_p=0.9, max_tokens=1000)

def invoke_model(document, prev_response=None):
```

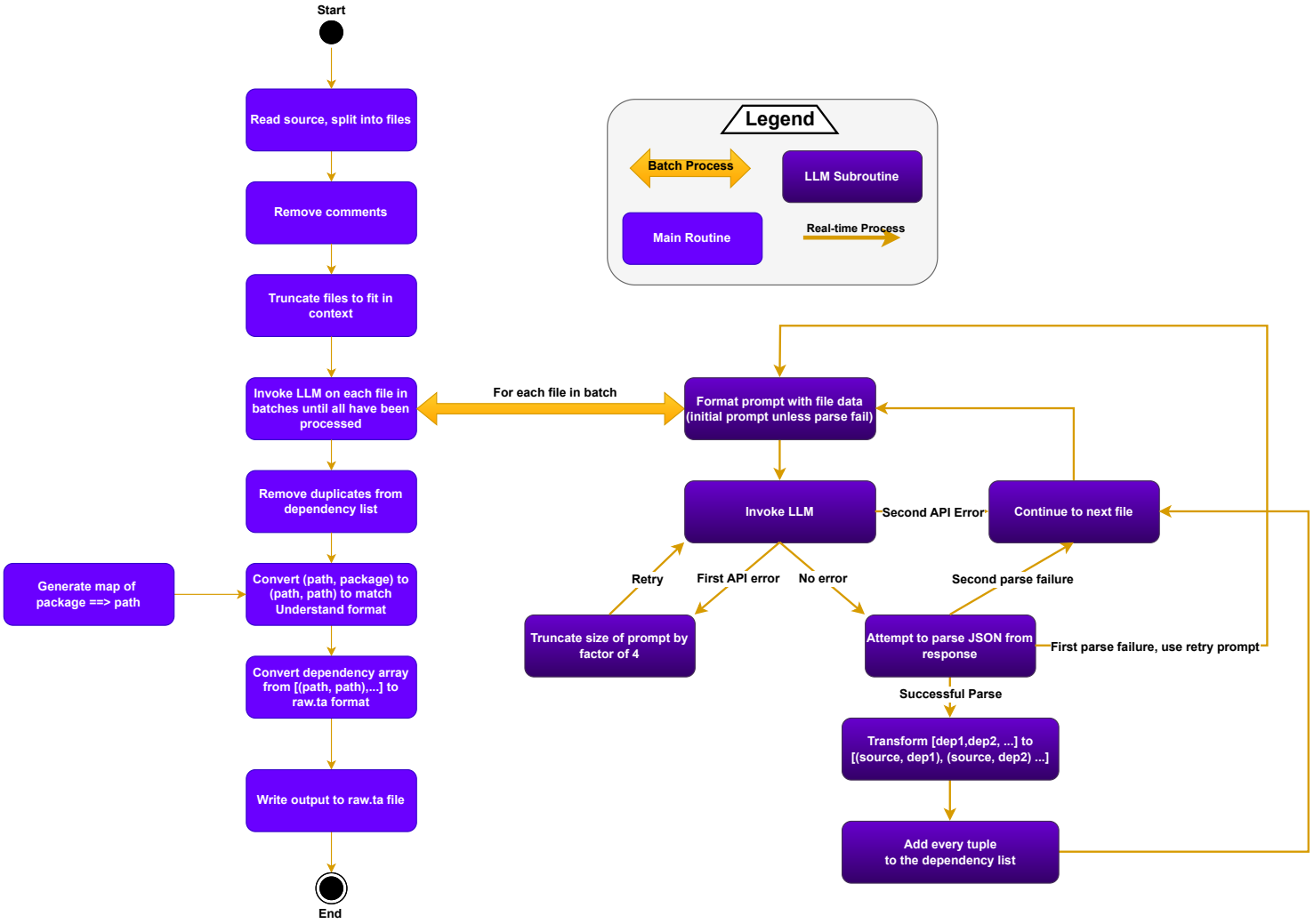

Fig. 3: Data pipeline for LLM architectural extraction Technique using GPT3.5 Turbo

```
    file_path = document["file_path"]
    source = document["source"]
    is_retry = prev_response is not None

    if is_retry:
        formatted = retry_prompt.format(file_path=file_path, source=source, response=
      prev_response)
    else:
        formatted = initial_prompt.format(file_path=file_path, source=source)

    response = llm.invoke(formatted)
    response = response.replace("\n", "").replace(" ", "")

    success = handle_response(file_path, response, is_retry)
    if success or (not success and is_retry):
        return
    else:
        invoke_model(document, prev_response=response)

```

2.3.4 Generating the TA file

After the batch processing was done, we went through and removed duplicates in case anything was counted twice. We actually removed about 6000 duplicates meaning there were many times the LLM was counting things twice in one file. The LLM produces mappings from file path to the package name, however we want the mappings to be from one file to another. To do this, we went through source and mapped the files within each package to their appropriate package qualified name. Using this we were able to easily convert the LLM dependency list into path to path mappings. Then using the same process as in the previous technique, we printed all the unique calling files, and then all of the dependency links which they have. The source code for this process is shown below.

```
# clean duplicates
output = sorted(set(output))
print(f" {len(output)} uniques")

# generate map of package -> path where it is defined
package_to_path_map = {}

for dir_path, _, file_names in os.walk(path):
    for file_name in file_names:
        if file_name.endswith(".java"):
            package = dir_path.split("\\java\\")[-1].replace("\\", ".") + "." + file_name.
      replace(".java", "")
            package_to_path_map[package] = dir_path + "\\" + file_name

print(f"Size: {len(package_to_path_map)}")
for k,v in list(package_to_path_map.items())[:10]:
    print(k + " -> " + v)

# apply mapping to depedency list, generate final output (to use for raw ta)
final_output = []
```

```
20
21  for source, depdendency in output:
22      dep_path = package_to_path_map.get(depdendency)
23      if dep_path is not None:
24          final_output.append((source[2:].replace("\\", "/"), dep_path[2:].replace("\\", "/")))
25
26  print(f"Final output length: {len(final_output)}")
27  for x,y in final_output[:10]:
28      print(x,y)
29
30  # write to TA
31  raw_ta_output = "./source_raw_ta/llm_dependencies.raw.ta"
32
33  with open(raw_ta_output, "w+") as f:
34    f.write("FACT TUPLE : \n")
35
36    unique_file_paths = set(file_path for file_path, _ in final_output)
37
38    # first generate all the concrete instances
39    for file_path in unique_file_paths:
40      f.write(f"$INSTANCE {file_path} cFile\n")
41
42    # now add in all the dependencies
43    for file_path, dependency in final_output:
44      f.write(f"cLinks {file_path} {dependency}\n")
```

2.4 Quantitative Comparison

The quantitative comparison process is an integral part of this technique comparison study, and this comparison is built on 2 statistical identifiers: precision and recall.

Precision answers the question of what percentage of positive identifications was actually correct? [11], while recall answers the question of what proportion of positive identifications was actually correct? [11]

The quantitative comparison process aims to derive these 2 identifiers for every technique, to be used in the qualitative comparison.

An important assumption to recognize in the comparison process, is that a specific dependency that two or more techniques extract is considered a correct dependency. In other words, if two or more techniques overlap on specific dependencies, these dependencies are considered correct.

In order to determine the identifiers mentioned, we aim to calculate the numbers for the comparison between the technique's numerical values and its oracle, where the oracle pertains to the true values (the overlapping between technique outputs) and the latter pertains to the noise.

In other words, we are focused on retrieving the false positives of the technique and the false negatives of the oracle to determine the overlap between the two being the true positives, in order to calculate the precision and recall of every technique.

We begin with determining the number of overlapping and non-overlapping dependencies between each technique. We then proceed to manually verify a sample of these non-overlapping dependencies to determine the accuracy of every technique. And finally we utilize these accuracies to compare each of the techniques to its oracles, which will lead us to derive the precision and recall.

2.4.1 Comparing Dependency Overlap Process

To compare the different extraction techniques, using a python script we followed the process shown in Figure 4. First, the `.raw.ta` file for each of the three techniques is parsed for their dependencies. Using a key value pair dictionary, each dependency is mapped to the techniques they were extracted from. The key is the dependency strings extracted from the `.raw.ta` file and the value is an array of numbers that represent what technique the dependency was extracted from. Each technique was given a number value (1: Understand, 2: Imports, 3: LLM). We call this dictionary our "technique dependency map". The format of this dictionary can be seen below.

```
{
    "dependency_1": [1,2,3],
    "dependency_2": [1,2],
    "dependency_3": [2]
}
```

Now using this technique dependency map we count the amount of non-overlapping, partial overlapping, and total overlapping dependencies. Then we display these amounts in a visual format like a Venn diagram as seen in Figure 6. The pseudocode for this process is show in Algorithm 2.

2.4.2 Stratified Sampling Process

In order to determine the accuracy of each of the dependency techniques utilized, one must evaluate the dependencies generated by every technique. However, this poses a great issue of time. There are thousands of dependencies generated by every technique, and it is extremely inefficient to dig through each of them to determine their correctness. Thus we have used a stratified sampling process to quickly determine the accuracy of the techniques.

The stratified sampling process works by deriving the amount of dependencies that do not overlap between the techniques. Plugging that into the sampling calculator provided by the Professor. And setting the confidence level to 95%, the confidence interval to 5 and finally setting the population to equate to the amount of non-overlapping dependencies between the techniques.

After deriving the required sample amount. We determine which of those samples correspond to which dependency's techniques based on how much they make up in the total non-overlapping dependencies. For instance, if the amount of non-overlapping dependencies is 100, and technique 1 makes up 50% of those dependencies, then it has to make up 50% of the sample size.

2.4.2.1 Manual Dependency Verification

Having established the sample size and its breakdown for each technique, we can now commence the process of assessing the accuracy of each technique based on the provided sample.

2.4.2.2 Procedure for Manually Verifying Selected Dependencies

We select the required amount of dependencies/samples for each technique randomly. To randomly select these samples, using a python script we followed the process shown in Figure 5. The pseudocode for this process is show in Algorithm 3.

2.4.2.3 Manual Sample Verification Criteria

One important question to be posed is, how do we determine if the given dependency is correct or not? If assumed that the existence of a dependency between two files; file A and file B. Let the dependency claim that file A utilizes file B, and so the correctness

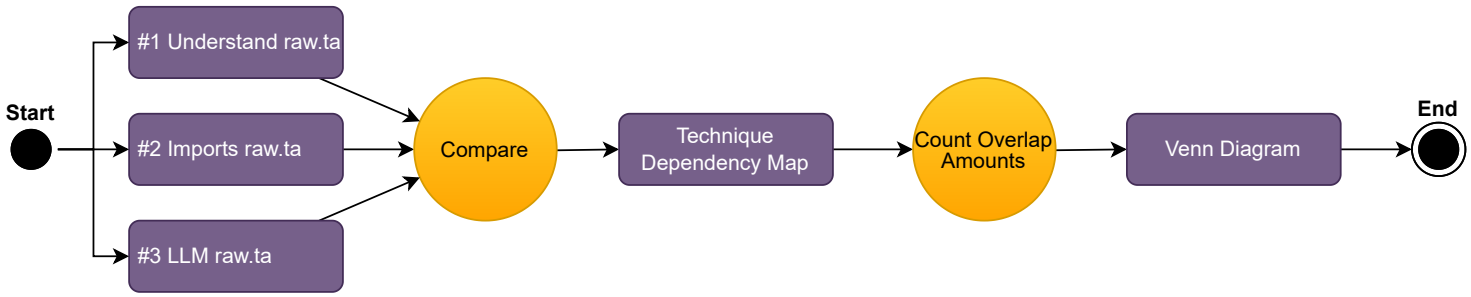

Fig. 4: Dependency comparison process to generate Venn diagram of dependency overlap by technique

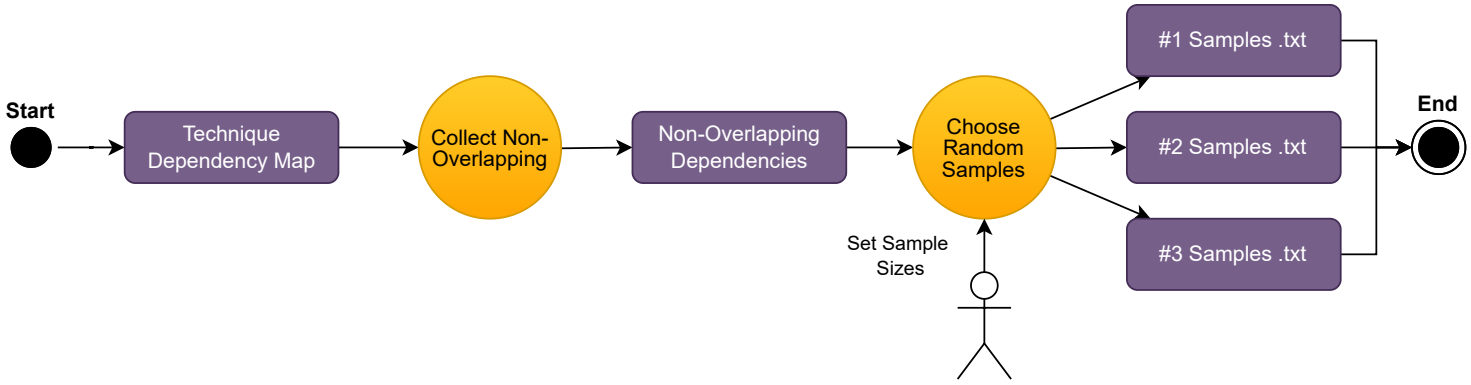

Fig. 5: Process flow to generate random stratified samples of dependencies across the three techniques for manual verification

criteria used is as follows: Does file A import file B? If so, does file A use the imported functionality from file B? If that is the case, then the dependency is set as correct. However if file A imports file B without using the imported functionality, then the dependency is labelled as incorrect.

Let's assume another scenario, what if file A utilizes functionality from file B without an import? This is an example of an internal dependency, which is labelled correct.

2.4.2.4 Technique Accuracy Calculation

After the manual verification of the samples. We can begin calculating the accuracy of each technique. To calculate the accuracy of each technique, we divide the number of verified correct dependencies by the total number of dependencies verified; and that gives us the accuracy. For instance, let us assume technique i had 10 samples that need manual verification. After manually verifying the 10 samples, we found 2 to be correct, therefore the accuracy of technique 1 is 2/10 which is 20%.

2.4.3 Comparison Statistical Technique Overview

Now that we have derived the number of overlapping and non-overlapping dependencies, as well as the accuracy of each technique based off our manual verification process. We can begin to derive the values for the oracle, the technique, and the intersection between the two.

2.4.3.1 Statistical Methods

We can now begin discussing the equations utilized to calculate the precision and recall of every technique. The following are the components of the recall and precision equation that need to be derived.

$$a = T_i(1 - T_{i\%}) \tag{1}$$

$$b = T_i(T_{i\%}) + T_{iz} + T_{iy} + T_{izy} \tag{2}$$

$$c = T_i(T_{i\%}) + T_z(T_{z\%}) + T_y(T_{y\%}) + T_{iz} + T_{iy} + T_{izy} \tag{3}$$

Let T_i be the number of non-overlapping dependencies of technique i.
Let T_{iz} be the number of overlapping dependencies between technique i and technique z.
Let T_{izy} be the number of overlapping dependencies between technique i,technique z, and technique y.
Let $T_{i\%}$ correspond to the accuracy of technique i, which was the value derived through the manual verification process.
We can now tackle the precision of recall and precision using the variables above. These are the equations utilized in our calculations, the below apply to technique i.

$$precision = \frac{b}{a+b} \tag{4}$$

$$recall = \frac{b}{b+c} \tag{5}$$

Algorithm 2 Comparing

```
1: techniqueOnePath ← path to technique one's .raw.ta file
2: techniqueTowPath ← path to technique two's .raw.ta file
3: techniqueThreePath ← path to technique three's .raw.ta file
4: dependencies ← empty key value pair dictionary
5: procedure ADDDEPENDENCY(dependency, techniqueNumber)
6:     if dependency in dependencies keys then
7:         Append techniqueNumber to dependencies[dependency]
8:     else
9:         Add new dependencies[dependency] with techniqueNumber
10:    end if
11: end procedure
12: for each line in techniqueOnePath do
13:    if "cLinks" in line then
14:        AddDependency(line, 1)
15:    end if
16: end for
17: for each line in techniqueTwoPath do
18:    if "cLinks" in line then
19:        AddDependency(line, 2)
20:    end if
21: end for
22: for each line in techniqueThreePath do
23:    if "cLinks" in line then
24:        AddDependency(line, 3)
25:    end if
26: end for
27: techniqueOneCount ← 0
28: techniqueTwoCount ← 0
29: techniqueThreeCount ← 0
30: techniqueOne&TwoCount ← 0
31: techniqueOne&ThreeCount ← 0
32: techniqueTwo&ThreeCount ← 0
33: techniqueOne&Two&ThreeCount ← 0
34: for each dependencyTechniques array in dependencies values
    do
35:    if 1 and 2 and 3 in dependencyTechniques then
36:        techniqueOne&Two&ThreeCount += 1
37:    else if 1 and 2 in dependencyTechniques then
38:        techniqueOne&TwoCount += 1
39:    else if 1 and 3 in dependencyTechniques then
40:        techniqueTwo&ThreeCount += 1
41:    else if 2 and 3 in dependencyTechniques then
42:        techniqueTwo&ThreeCount += 1
43:    else if 1 in dependencyTechniques then
44:        techniqueOneCount += 1
45:    else if 2 in dependencyTechniques then
46:        techniqueTwoCount += 1
47:    else if 3 in dependencyTechniques then
48:        techniqueThreeCount += 1
49:    end if
50: end for
51: Print counts
```

Algorithm 3 Random Sample Selection

```
1: dependencies ← key value pair dictionary created in Algorithm 2
2: amountOne ← size of sample for technique one
3: amountTwo ← size of sample for technique two
4: amountThree ← size of sample for technique three
5: justInOne ← empty array
6: justInTwo ← empty array
7: justInThree ← empty array
8: for each dependency in dependencies keys do
9:     if dependencies[dependency] array length equal to 1 then
10:        if 1 in dependencies[dependency] value then
11:            Append dependency to justInOne
12:        else if 2 in dependencyValues then
13:            Append dependency to justInTwo
14:        else if 3 in dependencyValues then
15:            Append dependency to justInThree
16:        end if
17:    end if
18: end for
19: sampleOne ← random sample of size amountOne from justInOne
20: sampleTwo ← random sample of size amountTwo from justInTwo
21: sampleThree ← random sample of size amountThree from justInThree
22: Write sampleOne to .txt
23: Write sampleTwo to .txt
24: Write sampleThree to .txt
```

2.4.3.2 Tool used to calculate

Due to the repetitive nature of the calculations, we have chosen to utilize Microsoft Excel. An Excel sheet has been prepared, requiring the straightforward input of numbers to generate the results presented in this report.

2.5 Qualitative Comparison

2.5.1 Basic Tactic

The initial tactic at hand is to evaluate and pick solely off the statistical identifiers of precision and recall only without considering any other factor.

Essentially, the process involves ranking each technique according to precision and recall individually. If the top-ranking positions differ for precision and recall, the evaluation considers the combination of precision and recall to determine the best overall performance.

2.5.2 Recommended Tactic

The recommended tactic is to utilize not only the statistical identifiers but also take into account how each technique functions. For instance, if technique A has a slightly higher precision value than technique B, but technique B functions in an optimized manner; for example it finds deeper dependencies, then technique B's value is considerably higher than if we were to have only looked at the numbers.

Details into how each technique functions and how each will influence our decision to pick a technique are all considered and mentioned in section 4.2.1.

3 Results

3.1 Quantitative Results

3.1.1 Technique Comparison Output

Listed in table Table 1 are the overlapping and non overlapping dependency amounts of each technique. These values are vital in starting the quantitative comparison process.

Venn Diagram Region	Number of Dependencies
All Dependencies	202 178
Understand Total	42 465
Import Checking Total	85 213
GPT3.5 Turbo Total	74 500
Understand Only	10 408
Import Checking Only	11 122
GPT3.5 Turbo Only	3 877
Understand ∩ Import Checking	5 177
Understand ∩ GPT3.5 Turbo	1 709
Import Checking ∩ GPT3.5 Turbo	43 743
Understand ∩ Import Checking ∩ GPT3.5 Turbo	25 171

Table 1: Unique dependencies broken down and categorized by overlap among techniques

The values in the table shown correspond to the T_i variable(s) mentioned in section 2.4.3.1. More specifically T_1 refers to the Understand technique, T_2 refers to the Import Checking technique, and T_3 refers to the LLM as specified below in equations (6), (7), and (8).

$$T_1 = 10408 \tag{6}$$
$$T_2 = 11122 \tag{7}$$
$$T_3 = 3877 \tag{8}$$

3.1.1.1 Venn Diagram Technique Output Explained

The T_i values discussed in section 3.1.1 are illustrated more clearly in Figure 6. This Venn diagram outlines the overlapping sections between each technique which corresponds to the T_{iz} values mentioned in section 2.4.3.1, and specified below more precisely in equations (9), (10), and (11).

$$T_{12} = 5177 \tag{9}$$
$$T_{23} = 43743 \tag{10}$$
$$T_{13} = 1709 \tag{11}$$

3.1.2 Manual Dependency Verification Instances

Recall that the Manual Dependency Verification step involves manual verification of dependencies that are randomly selected for each sample. This step was a vital step towards the calculation of precision and recall as it helps in determining the accuracy of the techniques.

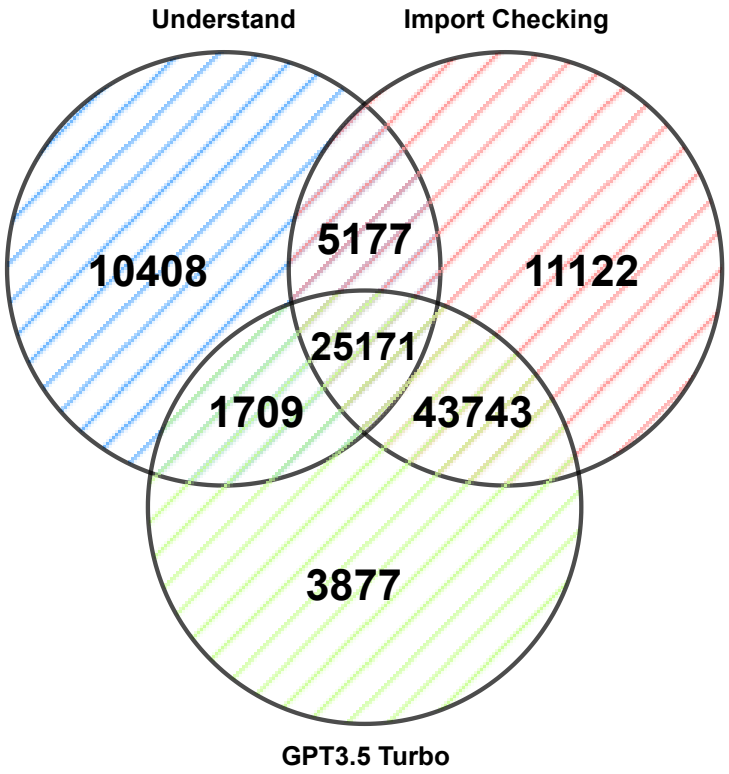

Fig. 6: Partitions of unique dependencies by technique. Overlapping regions are intersections common to multiple techniques.

In Table 2, the accuracy for each technique is listed under the "Accuracy" column. The accuracies listed in these tables correspond to the accuracies mentioned in section 2.4.3.1. Where $T_{i\%}$ is used to the accuracy of technique i. Meanwhile in this case, here are the following variables we can derive listed below in equations (12),(13), and (14) from Table 2.

$$T_{1\%} = 77.92\% \tag{12}$$
$$T_{2\%} = 94.58\% \tag{13}$$
$$T_{3\%} = 87.93\% \tag{14}$$

Technique	Samples to Verify	Accuracy
Understand	155	77.92%
Import Checking	165	94.58%
GPT3.5 Turbo	58	87.93%
Total	378	n/a

Table 2: Stratified sampling breakdown by technique.

The following three sections being titled "False Dependency Instance" are going to mention one false or incorrect sample from each of the techniques' dependencies, that was encountered during our manual verification process.

Technique	Strata Proportion	Strata Size
Understand	40.97%	155
Import Checking	43.78%	165
GPT3.5 Turbo	15.26%	58
Total	100.0%	378

Table 3: Stratified sampling breakdown by technique.

3.1.2.1 Sample 1 False Dependency Instance, with code listing

```
1  package org.apache.flink.test.streaming.runtime;
2
3  import org.apache.flink.api.common.RuntimeExecutionMode;
4  import org.apache.flink.api.common.eventtime.WatermarkStrategy;
5  import org.apache.flink.api.common.functions.FlatMapFunction;
6  import org.apache.flink.api.common.serialization.SimpleStringEncoder;
7  import org.apache.flink.api.java.tuple.Tuple2;
8  import org.apache.flink.configuration.BatchExecutionOptions;
9  import org.apache.flink.configuration.Configuration;
10 import org.apache.flink.connector.file.sink.FileSink;
11 import org.apache.flink.connector.file.src.FileSource;
12 import org.apache.flink.connector.file.src.reader.TextLineInputFormat;
13 import org.apache.flink.runtime.jobgraph.IntermediateDataSetID;
14 import org.apache.flink.runtime.minicluster.RpcServiceSharing;
15 import org.apache.flink.runtime.scheduler.ClusterDatasetCorruptedException;
16 import org.apache.flink.runtime.testutils.MiniClusterResourceConfiguration;
17 import org.apache.flink.streaming.api.datastream.CachedDataStream;
18 import org.apache.flink.streaming.api.datastream.DataStream;
19 import org.apache.flink.streaming.api.datastream.DataStreamSource;
20 import org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator;
21 import org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
22 import org.apache.flink.streaming.api.functions.ProcessFunction;
23 import org.apache.flink.streaming.api.transformations.CacheTransformation;
24 import org.apache.flink.streaming.util.TestStreamEnvironment;
25 import org.apache.flink.test.util.AbstractTestBase;
26 import org.apache.flink.test.util.MiniClusterWithClientResource;
27 import org.apache.flink.util.AbstractID;
28 import org.apache.flink.util.Collector;
29 import org.apache.flink.util.OutputTag;
30
31 import org.apache.commons.io.FileUtils;
32 import org.junit.jupiter.api.AfterEach;
33 import org.junit.jupiter.api.BeforeEach;
34 import org.junit.jupiter.api.Test;
35 import org.junit.jupiter.api.io.TempDir;
36
37 import java.io.File;
38 import java.io.FileWriter;
39 import java.io.IOException;
40 import java.nio.file.Path;
41 import java.util.ArrayList;
42 import java.util.Arrays;
43 import java.util.Collection;
44 import java.util.Collections;
45 import java.util.List;
46 import java.util.UUID;
47
48 import static org.assertj.core.api.Assertions.assertThat;
49
50 ...
```

Listing 1: File "CacheITCase.java" -> File "SideOutputDataStream.java" dependency

Listing 1 is a sample that was manually verified to be false. The sample refers to the dependency between File "CacheITCase.java" and File "SideOutputStream.java", and the above code is an excerpt of "CacheITCase.java"'s full code. In this instance, "SideOutputStream.java" was never imported, nor was it ever used in the code that's not visible above. Therefore this dependency from the samples of the Understand technique has been marked as incorrect.

3.1.2.2 Sample 2 False Dependency Instance, with code listing

```
1  package org.apache.flink.runtime.operators;
2
3  ...
4
5  import org.apache.flink.runtime.operators.shipping.OutputCollector;
6  import org.apache.flink.runtime.operators.shipping.OutputEmitter;
7  import org.apache.flink.runtime.operators.shipping.ShipStrategyType;
8  import org.apache.flink.runtime.operators.sort.ExternalSorter;
9  import org.apache.flink.runtime.operators.sort.Sorter;
10 import org.apache.flink.runtime.operators.util.CloseableInputProvider;
11 import org.apache.flink.runtime.operators.util.DistributedRuntimeUDFContext;
12 import org.apache.flink.runtime.operators.util.LocalStrategy;
13 import org.apache.flink.runtime.operators.util.ReaderIterator;
14 import org.apache.flink.runtime.operators.util.TaskConfig;
15 import org.apache.flink.runtime.plugable.DeserializationDelegate;
16 import org.apache.flink.runtime.plugable.SerializationDelegate;
17 import org.apache.flink.runtime.taskmanager.TaskManagerRuntimeInfo;
18 import org.apache.flink.util.Collector;
19 import org.apache.flink.util.InstantiationUtil;
20 import org.apache.flink.util.MutableObjectIterator;
21 import org.apache.flink.util.Preconditions;
22 import org.apache.flink.util.UserCodeClassLoader;
23
24 import org.slf4j.Logger;
25 import org.slf4j.LoggerFactory;
26
27 import java.io.IOException;
28 import java.util.ArrayList;
29 import java.util.List;
30 import java.util.Map;
31
32 import static java.util.Collections.emptyList;
33
34 ...
```

Listing 2: File "BatchTask.java" -> File "OutputCollector.java" dependency

Listing 2 showcases a snippet from the file "BatchTask.java" to emphasize its dependency on the file "OutputCollector.java." Since this specific instance was identified by the import technique's dependency analysis, it implies detection through import statements. Hence, Listing 2 includes only a portion of the import statements related to this dependency.

The above code is marked as incorrect, because even though the dependency is imported, it is not used in the actual code when examined. Therefore it's a false positive.

3.1.2.3 Sample 3 False Dependency Instance, with Code Listing

There is no listing to be shown, in this scenario we are examining the dependency between File "MultipleComponentLeaderElectionDriverAdapter.java" and "LeaderElectionEventHandler.java". This dependency is marked as false because neither of these files actually exist, which is also why we do not have a code listing for this part.

This sample was found through the LLM technique, therefore this false dependency could possibly be a byproduct of hallucination [12]. Hallucinations and the role they play will be discussed in the Risks & Limitations section ahead.

3.1.2.4 Final Results: Precision and Recall

Now we have gathered all that is needed, the precision and recall can finally be calculated, and their values are listed in Table 4. The import technique takes the first place as the one with the best recall value of 46.81%, while the LLM technique is found to be the best in

Technique	Precision	Recall
Understand	94.59%	42.61%
Import Checking	99.29%	46.81%
GPT3.5 Turbo	99.37%	44.41%

Table 4: Precision and recall values by technique.

terms of precision with a value of 99.37%.
Below will be one instance of how the Understand technique's (Referred to as technique 1 in the equations) precision and recall was calculated following the methodology defined in section 2.4.3.1.

$$a = T_1(1 - T_{1\%}) \tag{15}$$
$$a = (10408)(1 - 0.7792) \tag{16}$$
$$b = T_1(T_{1\%}) + T_{12} + T_{13} + T_{123} \tag{17}$$
$$b = (10408)(0.7792) + 5177 + 1709 + 25171 \tag{18}$$
$$c = T_1(T_{1\%}) + T_2(T_{2\%}) + T_3(T_{3\%}) + T_{12} + T_{13} + T_{123} \tag{19}$$
$$c = (10408)(0.7792) + (11122)(0.9458) + (3877)(0.8793) + ... \tag{20}$$

$$precision = \frac{b}{a+b} \tag{21}$$
$$recall = \frac{b}{b+c} \tag{22}$$

3.2 Qualitative Results

3.2.1 Chosen Technique Based on Initial Tactic

If we were to utilize the initial tactic which is to only look at the precision and recall values, then based on looking at the results outlined in Table 4, Import checking would be the winner.

Import checking does not score the highest at Precision, however the LLM doesn't score the highest on recall either. But what we can observe is that the difference in precision is very minor, and can be overlooked or thought of as nearly the same. Meanwhile the recall value differs quite a bit relative to the difference of precision values, and in this case Import Checking comes out on top.

Therefore the final result for the initial tactic would be Import Checking technique.

3.2.2 Chosen Technique Based on Recommended Tactic

If we were to go the path of using the recommended tactic which involves considering how each technique functions as opposed to only considering the statistical identifiers, then LLM would be the suitable option.

Even though the identifiers prefer towards the Import Checking. The Import Checking methods' functionality outlines its gaps. When import statements are only evaluated, then internal dependencies are not considered, because a dependency involving files in the same packages do not require an import statement. Meanwhile, with an LLM, the code is evaluated, therefore dependencies not visible in the imports can still be caught by the LLM.

Therefore the final result using the recommended tactic would be the LLM. More details on the functionality of the techniques and how it impacted their considerations are discussed in section 2.4.3.1.

4 Discussion

There are several aspects concerning the analysis for the above results. In this section the quantitative results will be analyzed and then rationales for these results will be offered with links made to literature where possible. First, the similar and different aspects will be introduced then reasons will be given for these observations.

4.1 Quantitative Results Comparison

Table 4 shows that Understand was outperformed by both Import Checking and GPT3.5 Turbo in terms of precision and recall. When considering just these raw metrics as a black box, Import Checking is the clear winner. Moreover, the Import Checking and GPT3.5 Turbo techniques both found many more dependencies than the Understand technique. The suspected reasons for this will be proposed in the next section. Another interesting observation is that GPT3.5 Turbo had much overlap with the Import Checking technique. The intersection of both techniques was the biggest proportion of the Figure 6 Venn diagram. GPT3.5 Turbo also has the least amount of unique results, making the smallest of the Venn diagram partitions.

4.1.1 Rationale Behind Differences

There are several points of divergence in the results as seen by the highly variable number of dependencies reported by each technique. Moreover, these differences are the key to explaining the above discrepancies among the dependencies generated by the techniques.

It appears that the Understand technique goes beyond merely examining import statements, as it likely considers various tokens in the code. It is hypothesized that the process involves generating several potential dependencies before pruning the results, leading to a more refined list of dependencies. This reasoning may contribute to the technique reporting the smallest number of dependencies. Recall that the Import Checking technique had the highest number of generated dependencies. It is difficult to verify the exact reasons for this behaviour due to the closed source code of Understand.

Similar to the Understand technique, the GPT3.5 Turbo pipeline is suspected to engage in a pruning process during its analysis. This observation was confirmed by what was noticed whenever training was stopped early to inspect the generated data for confirmation of correctness and good progress. This inspection was not done rigorously as there is no heuristic that was used to investigate this. However, we can say with good certainty that this was a phenomenon that occurred.

Furthermore, it is proposed that the GPT3.5 Turbo technique might dynamically analyze the code, considering aspects such as tracing of the data flow, variable updates, and other dynamic elements. This dynamic approach could enhance the model's ability to comprehend the intricacies of code execution beyond static representations, potentially leading to more accurate and context-aware results. There is no real way to confirm this but not mentioning

this speculation reduces the rigor of this analysis. Machine learning and inadequate explainability of results is all too common of a phenomenon. One potential solution to improving the explainability is to use a wider prompt space and to use prompts that leave artifacts on dependencies indicating what makes it a dependency.

4.1.2 Rationale Behind Similarities

There are several points of overlap among the three techniques. Many of the dependencies are common to all three techniques or to two out of the three techniques. Very few of the dependencies generated are unique. There are 25 171 dependencies that are common to all three techniques. This is a sign that import statements are the primary markers for dependency extraction.

Understand, the Import Checking and the GPT3.5 Turbo technique all seem to follow convergent steps in their analyses. They all likely use import statements. The LLM likely initiates its examination by looking at import statements and subsequently extends its analysis to other tokens within the code. This holistic approach allows for a comprehensive understanding of the code base. The pruning process is common to both of these techniques to reduces the dependency list sizes as mentioned previously.

It is clear that the GPT3.5 technique ends up arriving at similar conclusions to Understand as they have almost 2000 dependencies common between them. However, it still seems that just doing raw import checks is the major contributor to dependencies.

4.2 Qualitative Results Comparison

4.2.1 Recommendations

All three techniques are recommended for use but with certain caveats. First, the Understand technique is a good choice if the architectural recovery is for a Java project and the Understand tool is available for free as it is fast and lightweight and specifically made for Java. Moreover, it provided fairly accurate results and showed decent results that clearly line up with the conceptual architecture [5, 6] and manual inspection. Import Checking is a great option to use and shows promising results but the fear is that the scope it focuses on is too narrow and is likely giving many false positives due to the very large returned list. Moreover, it may not work for every language even with heavy modifications. This will be elaborated on in the next section. The LLM Model GPT3.5 Turbo technique is the best technique to use (provided that funding for the API usage is available) as it can be tweaked and configured at multiple points to provide differential results. The aspects of the pipeline that can be altered include:

1. Differential pre-processing of the source code files (e.g. comments vs. no comments, file length reduction vs. full files used, etc.)
2. Choice of model(s)
3. Prompt engineering and exploration of a diverse "prompt space."

4.3 Risks & Limitations

There are several risks and limitations involved in using the techniques described in this report. There are some common elements among all three and certain aspects are applicable to specific techniques only. First, one major limitation is that Understand and GPT3.5 Turbo are not free and are both closed source. They do not have their code available and do not provide explainability for their results. Next, obtaining OpenAI keys at the time of the writing of this report is uncertain and not guaranteed. Therefore these results may be difficult to replicate unless a key is already possessed by the researcher(s). Another common limitation is that all three techniques in their current form are restricted to the Java programming language. Import Checking and the LLM technique can theoretically be modified to target other programming languages however they would require modification. Import checking would require the most modification as it is a script made specifically for the Apache Flink Java directories. Understand is not modifiable at all and is the most limited out of these approaches.

It is also risky to approach the architectural recovery by using OpenAI/GPT3.5 as we have. The cost was minimized by careful preprocessing to heavily reduce file sizes. Moreover, there may be heavy costs incurred if the pre-processing of the data is not approached carefully. The pricing structure for API calls can lead to unexpectedly high charges if not monitored, for this reason it is highly recommended to approach with caution and to build up towards higher levels of complexity in the prompting.

The Import Checking approach will also miss dependencies when the caller-callee pair of functions are within the same Java package or there are dependencies listed within the Maven/Gradle build files for the Java project. The Import Checking may also be limited even if modified to work in other programming languages as the syntax for imports and various styles of macros are more complicated in other languages. Java makes this part of the language fairly simple and invariant.

Finally, the last, and most interesting aspect is the phenomenon of hallucinations in LLMs. This process of neural network based hallucination is basically when the language model returns unintended, unrelated responses causing a reduction in transient performance [12]. There is a possibility that if prompts are not explored properly, in other words, a diverse "prompt space" is not explored, then results may not be adequate. To mitigate this it is recommended to try different language models in parallel and to ensure that stratified sampling is done to verify the results manually.

4.3.1 Few-shot Prompting

One additional thing we could have done to possibly improve results would have been to provide examples of code with deceiving dependencies (or ones that would be different at runtime) and what the true dependencies actually are so it could use those to more accurately infer dependencies from the source we provided. While this would have theoretically improved the quality of our results, we did not try it much as it would have increased cost significantly to include enough samples to be effective, and in the case of large files it would mean sacrificing more of the actual source code to fit the examples in context. Chain-of-thought and other prompting techniques also could have been useful however with limited resources we chose a quick and easy way to at least show that LLM can hold its own compared to the other techniques.

4.4 Lessons Learned

The correct technique to use for software architectural extraction is very important as it can change results drastically and vary in cost highly based on the specific use case. Proprietary extraction techniques like Understand are useful and reliable but they come at a cost. Moreover, the fast and lightweight performance with adequately reliable results make it a decent choice. However, the lack of explainability and use of the technique as a black box is undesirable. It is likely that Understand analyzed source code holistically at the static level and not just looking at import statements.

Next, the technique of Import Checking provides a wealth of information for all potential dependencies, however, it is too brute force of a technique that results in many likely false positives or repeat dependencies. Furthermore, it will miss all intrapackage dependencies and result in many false negatives that would be almost impossible to track for such a large scale project. Finally, it would require extensive testing and modification to be reliably used on projects in other programming languages.

Finally, the use of Large Language Models to analyze software architecture is a very promising approach due to its highly variable configurations. It is also the trickiest technique to use, advanced use cases may require past experience with prompt engineering. Moreover, a novice may rack up very high costs if the problem is not approached with caution. Its largest potential benefit is that it may be able to do a dynamic analysis of the source code simply by reading the code. One key recommendation is to use a cheap/free model to start and to keep the prompts/routines simple so that they can be examined early for correctness.

5 Conclusion

In conclusion there are many approaches one can talk to extract and recover the architecture of large scale open source software. In this project we showed the use of two novel techniques alongside a legacy technique to show that we can improve upon the legacy software's results. This research was the conclusion of a four part project involved in the extraction of the architecture of Apache Flink. The process involved first, studying the documentation for recovery of the conceptual architecture [4, 13]. This was followed by using the Understand technique with the LSEdit software to propose a model for the concrete architecture [5, 9, 14]. Third, the results from these two recovery process were reconciled and a revised architecture was proposed [6]. Here we have showed the merits of these past tasks by verifying the Understand techniques performance. It is clear that the technique gave great results and the two novel techniques also showed very good performance. In future works it is proposed that prompt engineering be explored for this task and a wide prompt space be used to fine tune the results.

References

[1] A. Baabad, H. B. Zulzalil, S. Hassan, and S. B. Baharom, "Software Architecture Degradation in Open Source Software: A Systematic Literature Review," *IEEE Access*, vol. 8, pp. 173 681–173 709, 2020, conference Name: IEEE Access. [Online]. Available: https://ieeexplore.ieee.org/document/9200327?denied=

[2] N. Sangal, E. Jordan, V. Sinha, and D. Jackson, "Using dependency models to manage complex software architecture," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, Oct. 2005, pp. 167–176. [Online]. Available: https://dl.acm.org/doi/10.1145/1094811.1094824

[3] P. Carbone, A. Katsifodimos, Kth, S. Sweden, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, Jan. 2015. [Online]. Available: https://www.researchgate.net/publication/308993790_Apache_Flink_Stream_and_Batch_Processing_in_a_Single_Engine

[4] R. Dolores, H. Jamil, A. Arnold, Z. Ross, N. Gaziy, J. Le, W. AlDari, and M. Siddiqui, "Apache flink conceptual architecture: Technical report," 2023. [Online]. Available: https://github.com/Anselmo21/EECS4314-FlinkForce/tree/main/Assignment_1

[5] R. Dolores, H. Jamil, A. Arnold, Z. Ross, N. Gaziy, J. Le, W. AlDari, and M. Siddiqui, "Apache flink concrete architecture: Checkpoint subsystem reflexion analysis," 2023. [Online]. Available: https://github.com/Anselmo21/EECS4314-FlinkForce/tree/main/Assignment_2

[6] R. Dolores, H. Jamil, A. Arnold, Z. Ross, N. Gaziy, J. Le, W. AlDari, and M. Siddiqui, "Apache flink: Checkpoint subsystem reflexion analysis," 2023. [Online]. Available: https://github.com/Anselmo21/EECS4314-FlinkForce/tree/main/Assignment_3

[7] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in Apache Flink®: consistent stateful distributed stream processing," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, Aug. 2017. [Online]. Available: https://dl.acm.org/doi/10.14778/3137765.3137777

[8] "Apache Flink," Oct. 2023, original-date: 2014-06-07T07:00:10Z. [Online]. Available: https://github.com/apache/flink

[9] "Understand: The Software Developer's Multi-Tool." [Online]. Available: https://scitools.com/

[10] 2023. [Online]. Available: https://github.com/Anselmo21/EECS4314-FlinkForce/tree/main/Assignment_4

[11] "Classification: Precision and recall," https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall, accessed: 2023-12-4.

[12] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Comput. Surv.*, vol. 55, no. 12, mar 2023. [Online]. Available: https://doi.org/10.1145/3571730

[13] "Overview," section: docs. [Online]. Available: //nightlies.apache.org/flink/flink-docs-release-1.17/docs/dev/table/sql-gateway/overview/

[14] "The Landscape Editor LSEdit." [Online]. Available: https://www.swag.uwaterloo.ca/lsedit/