Analysis of Testing Methods For PTM-Enabled Open Source Software Projects

Souradip Pal*
Purdue University
West Lafayette, IN, USA
pal43@purdue.edu

Mohit Pandiya*
Purdue University
West Lafayette, IN, USA
mpandiy@purdue.edu

Mohammed A.Y. Metwaly*
Purdue University
West Lafayette, IN, USA
mmetwaly@purdue.edu

ABSTRACT

Introduction. Although there have been several surveys and research done on testing Machine Learning (ML) programs that deal with testing dataset quality, training & evaluation pipelines of ML models, etc., very few provide a comprehensive understanding of the testing practices prevalent in software applications using **P**re-**T**rained **M**odel (PTM) features. Nowadays, most applications and software systems include one or more components that depend directly or indirectly on Pre-Trained models creating a novel challenge in testing those systems since their behavior is jointly determined by the code that implements them and the data that is used for training them. This is also true in the case of open-source software (OSS) domain as well.

State-of-the-Art Work. Previous state-of-the-art works include papers exploring the generic process of creating ML programs, encompassing data preparation to production deployment, and highlighting the main sources of faults or errors. Researchers describe recent testing techniques for detecting, debugging, and testing ML programs at both the conceptual (model) and implementation levels, filling gaps in the literature and suggesting future research directions. Others aimed to provide a consolidated reference document to empower the ML and software engineering communities with insights into existing software testing techniques and enhance the overall quality of ML programs.

Proposed Contribution. Our work provides a comprehensive study of the testing practices like Unit Testing, Integration testing, and End-to-End (E2E) testing involved in open-source software (OSS) projects that enable the use of Pre-Trained Model features in their application for better overall performance and versatility.

Proposed Method. The proposed approach is to mine open-source GitHub repositories from **PeaTMOSS** and **ML Products** dataset and extract information like the quantity and type of tests available inside those projects for different models and ML pipelines. Moreover, the purpose was to investigate multiple aspects of application testing concerning Pre-Trained models which are generally present in open-source projects using GitHub APIs or by doing manual checks to determine their popularity, relevancy, and ease of usage. Our method also shows how we can provide an extension to the PeaTMOSS dataset by automating the PTM testing process for the different repositories.

1 INTRODUCTION

Due to recent advancements in Machine Learning (ML) and Artificial Intelligence (AI), there has been a steady increase in the use of Pre-Trained ML models alongside traditional algorithms in complex and large-scale software systems. This is mainly due to the enhanced functionality and predictive capabilities that can be achieved by using such state-of-the-art ML models. As more and more ML models are being utilized in safety or business critical domains such as automotive, healthcare, and finance [[20], [18], [5]], it becomes essential to check for the reliability of software systems having PTM-enabled features i.e., to understand to what extent they can be trusted in response to previously unseen scenarios that might produce unpredictable results [[7], [16]]. To this aim, software testing techniques are being used to check the functionality of such systems and ensure their dependability [12].

Even, open-source projects often leverage these Pre-Trained models for experimentation so that software developers can avoid the sheer cost of creating, training, and validating those models on their own. As such, those open-source projects would require a different set of testing practices than traditional open-source software due to the challenges involved in testing the behavior of such Pre-Trained models. Previous works have been done to consolidate open-source projects using such Pre-Trained model features as in [8]. Also, some systematic testing procedures have been studied to test ML programs in production software systems in general as shown in [9].

However, there has been no comprehensive study to determine what kind of testing techniques are generally involved in open-source projects involving PTM-enabled features and which of those testing practices are widely used in the source code to remove any implementation or execution issues present in those software applications. Analysis of those testing methods will also help us recognize any common bugs or code smells that should be rigorously tested before running and using the applications.

Our approach was to filter out popular open-source GitHub repositories and analyze them to identify testing patterns in those repositories. We looked for testing patterns as per the testing pyramid levels (unit testing, integration testing, and system-level or end-to-end testing). Then we qualitatively gave an estimate of which of those testing practices are more common and generally used in OSS projects and whether they are useful and can be adopted in general.

2 BACKGROUND AND RELATED WORK

This paper focuses on testing practices prevalent in software applications using **P**re-**T**rained **M**odels (PTM). Hence, this related work section explores recent published work on the generic process of testing ML programs, encompassing data preparation to production deployment, and highlighting the main sources of faults or errors. We first analyzed the current work done in developing

*All authors contributed equally to this research.

testing methods for PTM. Second, this research focuses on testing practices used when incorporating a PTM into an ML product using OSS, thus we reviewed some integration testing methods that are available. Third, we showcased the current challenges that exist in testing ML products that use PTM.

Several related works have been done in developing testing methods for analyzing the behavior of applications involving pre-trained models. In addressing the challenges posed by the inductive nature of ML programs, this paper [3] explores the generic process of creating ML programs, encompassing data preparation (ensuring consistency and removing redundancy) to production deployment (creating data pipelines), and highlights the main sources of faults. The researchers describe recent testing techniques for detecting, debugging, and testing ML programs at both the conceptual (model) and implementation levels, filling gaps in the literature and suggesting future research directions. The aim was to provide a consolidated reference document to empower the ML and software engineering communities with insights into existing testing techniques and enhance the overall quality of ML programs. Also, the authors of the paper [6] investigated flaky tests, which are software tests exhibiting seemingly random outcomes despite unchanged code. Through surveys and analysis of 200 fixed flaky tests, the study revealed various causes of flakiness that are challenging to fix, emphasizing the significant impact of flakiness on developers' perceptions, resource allocation, scheduling, and test suite reliability.

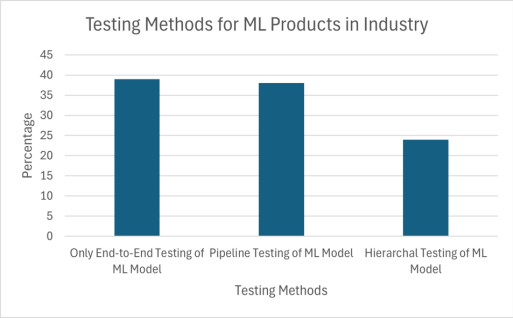


Figure 1: Percentage of different types of ML Product testing practices followed in the industry. The most common is the End-to-End testing of a single model starting from testing a model's input to its output. Hierarchical testing is more uncommon due to the difficulties in the testing method.

2.1 Enhancement of Testing Framework

In the realm of integration test generation, Abdessalem et al. [1] focused on ensuring the functional correctness of various machine learning models that interact within autonomous vehicles. Pezzè et al. [17] proposed a methodology that capitalizes on the utilization of pre-existing unit tests. Their approach unfolded through a three-phase process. Initially, the identification of class dependencies within the System Under Test (SUT) is carried out, represented through an object relation diagram (ORD). Subsequent to this, the computation of data flow information within the input test cases takes place, enabling the segmentation of test cases into coherent blocks. The final phase involves the creation of novel and intricate test cases, achieved by extracting code blocks from the existing unit tests using both the ORD and the data flow information obtained in the preceding steps. This approach exemplified an innovative strategy leveraging unit tests to enhance the systematic generation of integration tests. Grechanik et al. [10] introduced ASSIST, a tool designed to enhance the efficiency of integration testing in software applications by automatically deriving models that describe frequently interacting components. The approach involved utilizing an existing test suite to extract information from the System Under Test (SUT). Execution traces obtained through the execution of the test suite were then analyzed using a frequent pattern matcher, identifying recurrent patterns of method calls. Subsequently, a model was generated from these patterns, aiming to produce a more effective test suite for improved testing outcomes.

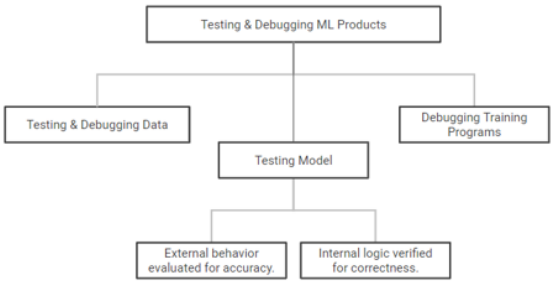


Figure 2: Stages of testing and debugging an ML product. Testing and debugging data in conjunction with the model inference is necessary in the case of ML products using Pre-Trained models whereas debugging training programs is not that essential.

2.2 Challenges while Testing

Other work has been published showcasing the challenges developers face when testing their ML models and products. In the paper [9] evaluating ML models, a group of 87 experienced ML program developers participated in a survey regarding testing practices and techniques used in the industry as well as concerns regarding them. 98% of the respondents found it challenging to select metrics for properties like correctness, robustness, and fairness, with no consensus on robustness metrics. Teams resorted to ad hoc approaches, facing unexplored challenges in designing metrics for hierarchical ML systems. The work explained how the testing process for ML problems in the industry was done in three ways. First, a single model where there is only one ML model in the system, and this model solves the problem in an end-to-end manner. Second, a pipeline where an ML problem is decomposed into several subproblems, each of which is solved by a distinct model and is tested in the pipeline. Lastly, hierarchal testing is where an ML problem is also decomposed into several sub-problems, but multiple ML models cooperate to solve a sub-problem. As evident from Fig 1, [9] observed that only 38% use the pipeline method and only 24% of the industry

use the hierarchical method of testing showing how testing between models is very minimal. The paper concludes by stating that an open challenge arose due to the lack of consensus on metrics, especially for the non-functional properties of ML models. Moreover, in their study, [13] differentiated between a machine learning (ML) project and an ML product. They provided a specific definition for an ML product and identified 262 open-source repositories on GitHub that aligned with this definition. The authors focused on addressing six research questions, with our particular interest lying in the query: "How are open-source ML products and their components tested?". The authors found out that in open-source products there were very scarce projects that included model evaluation, and a few projects that contained model evaluation scripts approach ML model testing as unit testing. They also showed how testing and debugging ML products are split into three areas: testing and debugging data, testing the model, and debugging training programs as in Fig. 2. In these open-source projects, there was no option to evolve or retain the model showing that product teams had a static view of the model and the ML team struggled to keep models dynamic. The research question pertaining to these testing practices can now be answered using the data set compiled by [8].

In the realm of integrating ML and non-ML components, challenges can arise due to the lack of generalized system-wide testing methods. Nahar et al. [14] demonstrated that a recurrent issue was the ambiguity surrounding the testing of the entire product after the integration of ML and non-ML elements. Teams frequently grappled with unclear demarcations, particularly when model teams explicitly disclaimed responsibility for overall product quality, including integration testing and testing in production. Notably, model teams often asserted that their obligations conclude with delivering a model evaluated solely for accuracy. Compounding this issue, certain product teams also exhibited a lack of planning for comprehensive system testing with the incorporated models. In some instances, system testing was conducted in an ad-hoc manner at best. These claims were supported by both [2] and [19].

3 MOTIVATION

Here, we aim to understand the usage of certain testing patterns present in common open-source software repositories involving Pre-Trained models. Recent works in ML testing provide several software testing approaches to adequately identify errors and potential issues in applications with PTM-enabled features to improve the reliability of their ML-based systems. Most of the literature focused on testing the ML algorithms and other components of ML training, like issues in data, models, pipelines, etc., and does not touch upon how these methods extend to applications involving only Pre-Trained models as can be seen in [3], [11], [15]. Hence, we aim to identify whether any testing patterns are common in PTM-enabled applications.

The idea is to find the source code of the tests available in open-source repositories and categorize those tests based on the testing patterns they belong to. This will also help answer some of the long-standing research questions currently prevalent in this domain.

- **(RQ1)** *What sort of testing practices are prevalent in open-source software projects involving Pre-Trained models?*
- **(RQ2)** *Do those projects follow the testing approaches provided by the recent literature?*
- **(RQ3)** *How many open-source software projects involving Pre-Trained models have tests covering all the testing pyramid (vmodel) test levels?*

We hope to successfully answer some of these questions and formulate a general testing practice covering all the areas of the testing pyramid including unit testing, integration testing as well as end-to-end (E2E) testing that will provide significant test coverage for the entire source code of any PTM-enabled OSS project. We aim to provide a partial automation framework that can help in easily identifying test methods or files corresponding to a particular Pre-Trained model utilized for a specific application and determine if these files or methods were covered via. unit testing.

4 RESEARCH METHODOLOGY

As part of our research initiative, we conducted a comprehensive examination of the testing practices employed in open-source repositories. Our focus was on two datasets. The first dataset, referred to as the **ML Products**[A.1] dataset, was readily accessible and encompassed 262 repositories that met specific criteria designating them as products rather than projects. The second dataset is known as the **PeaTMOSS** which is also open-source and a sample of it is available on GitHub[A.1], where a link to the full dataset with instructions on how to access it are also available. The PeaTMOSS dataset comprises 281,638 PTM packages, 27,270 GitHub projects utilizing PTMs as dependencies, and 44,337 links connecting these GitHub repositories to the PTMs on which they rely. The ML products and PeaTMOSS datasets can be accessed through the respective GitHub links.

Our initial focus was on the ML products dataset due to its smaller size and the presence of well-maintained repositories. Our examination of the dataset has revealed that, on average, these repositories have approximately 28 contributors, with the number of contributors spanning from 1 to 734. Given the substantial number of contributors in a repository, it was expected to have a test suite in place. This consideration was why we decided to do our preliminary study on this dataset. The preliminary examination involved a manual study of 33 repositories, a selection made through considerations such as language, technology, contributors, stargazers, and workload. Given the team's proficiency in Python, we refined our focus from the initial 262 ML-products repositories to 134 that predominantly utilize Python. These repositories encompass a combination of web and desktop applications. Among these, the top 33 repositories had more than 20 contributors each. With a team consisting of three members, we effectively aimed to analyze 11 repositories per team member. In our manual examination of the repository, we intended to assess various types of tests which would evidently provide an answer to our question **(RQ1)** on the type of testing practices prevalent in current open-source projects. Typically, these repositories were found to include a test directory housing files for unit tests. Directly identifying the usage of Pre-Trained models within the files in such test directories will provide a rough idea of whether the OSS repositories followed the testing approaches mentioned in the recent literature (**RQ2**). Given the predominant use of Python in these repositories, `pytest`

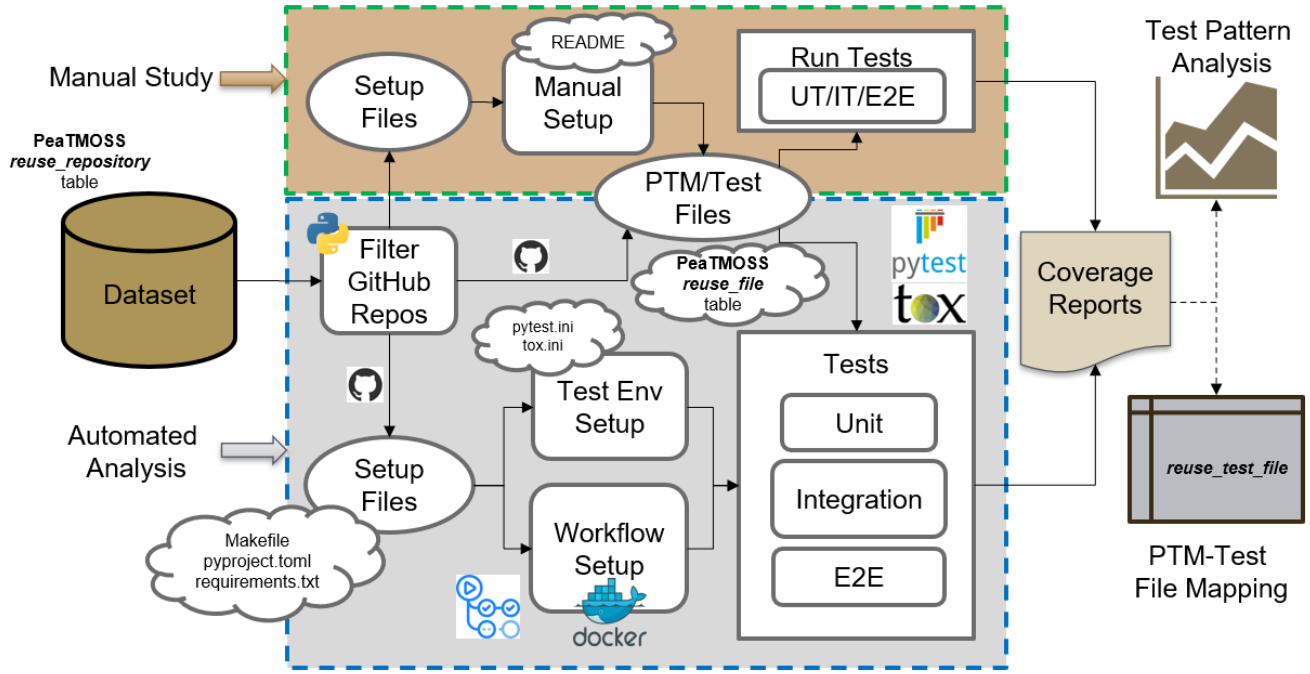


Figure 3: An overview of the Manual and Automated Analysis Workflow showing the two-step approach to analyzing PTM-enabled OSS repositories and the associated tools used. Here, the PeaTMOSS dataset provides two tables *reuse_repository* and *reuse_file* for easier test pattern analysis and test file mapping.

was found to be a common framework for unit testing, employing assert statements for value verification. Other prominent Python test frameworks we looked for were Unittest, Behave, Lettuce, and Robot. Our approach involved first comprehending the repository's overarching functionality. This helped us understand exactly where the machine-learning features were located in the code base. Then we checked if the methods having machine learning features were being tested or not. We specifically looked at a few criteria for ML infrastructure testing mentioned in [4] and qualitatively assessed the OSS projects.

In addition, we manually scrutinized some of the repositories for integration tests. GitHub employs continuous integration through a feature known as GitHub Actions, utilizing different pipelines presented as YAML scripts that sequentially execute various CI/CD workflow instructions with pre-commit or post-commit stages. We analyzed these YAML scripts to determine if different functions were being executed together. Our manual investigation extended to examining the Issues, Pull Requests, and Projects within the GitHub repository, providing insights into end-to-end or system testing. Conversations within the comments of Issues, PRs, and project tasks offered additional context on unit and integration testing. Next, we conducted thorough research on methodologies for automating the unit as well as the integration testing process across repositories which will give us the ability to analyze repositories on a larger scale, thus answering the question (**RQ3**) on quantifying the practices at various levels of the testing pyramid.

The success of our work can be gauged based on accomplishing the following tasks:

- **Quantitative Analysis** - Find out useful testing patterns (Unit, Integration, or E2E) in the repositories and correlate them with the characteristics of the project be it a research or a tutorial, or a product-based project, and the type of pre-trained models used like text, vision or multimodal models.
- **Partial Automation** - Partially automate fetching test method information from a repository and identify and run the test files or methods covering the Pre-trained model features to generate coverage reports.
- **Extension to PeaTMOSS dataset** – Map the files or methods involving PTM features or model usage to their corresponding test files or methods, thus serving as an extension to the PeaTMOSS dataset bearing additional PTM test information.

5 EXPERIMENTS

5.1 Manual Data Collection

Following the manual examination of 14 out of the 33 repositories from the ML products dataset and 50 repositories from the PeaTMOSS dataset, we conducted a statistical analysis of the results, enabling comparisons with a broader set of repositories. The manual study aims to uncover patterns in test writing, including aspects such as test file names, directory structure, and utilized packages. Leveraging these insights, we developed scripts for automation, facilitating rule-based checks across a large repository corpus, a workflow of which is depicted in Figure 3. Our target dataset for

automation is the PeaTMOSS dataset, providing information about the models used in specific repositories. This information aids in pinpointing the code sections where models are implemented, subsequently locating associated tests.

Since the PeaTMOSS dataset contains only metadata, obtaining file structures and specific files (particularly test files) requires exploring potential API solutions. If this proves challenging, we may consider cloning repositories and running our scripts locally. The PeaTMOSS dataset does not have information on GitHub Actions, so we plan to use GitHub API that could help with automatically fetching GitHub Actions to identify integration and E2E tests.

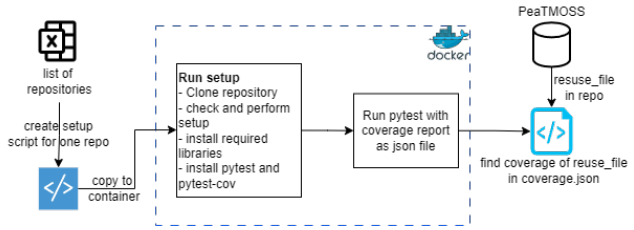


Figure 4: An overview of the Automated Setup & Testing workflow using Docker container to automate the generation of coverage reports.

5.2 Brute-force Text based PTM Test Searching

In our research, we sought to assess the extent to which test functions developed for model integration were utilized across various files within a project, beyond their initial file of creation. These test functions were identified using the PeaTMOSS dataset, specifically through the *reuse_file* table, which specifies the file where a particular pretrained model (e.g., 'gpt2') is employed. Our methodology involved traversing the repository to locate the designated file and identifying any function in which the model was invoked.

Focusing specifically on the integration of the Pre-trained Model, we examined the semantic purpose of each function where the model was referenced, determining whether the function was intended for "testing" or "checking" aspects related to the model's integration. Instances where the model's use did not align with these testing or integration purposes were excluded from further analysis. Subsequently, we expanded our examination to include a cross-referencing of the identified function names against all other files within the repository. This allowed us to tabulate the frequency of each test function's usage outside its original file. Moreover, we quantitatively analyzed the number of functions within each repository dedicated to either testing the PTM or integrating it into the project.

5.3 Automatic Coverage using Docker

Coverage reports are crucial as they reveal which files have been unit tested and to what extent. Manually performing tests using `pytest` framework and generating a coverage report is relatively straightforward when following the setup instructions found in the *Readme.md* file of a well-documented repository. However, automating this process for multiple repositories presents challenges.

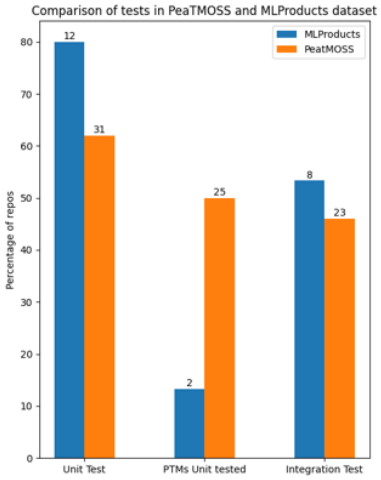
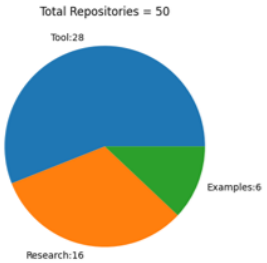


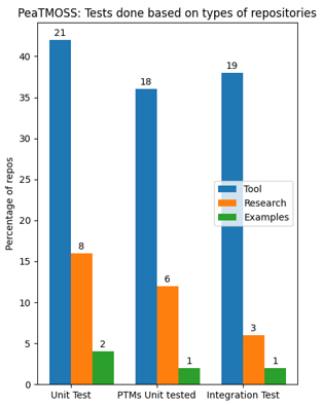
Figure 5: A comparison of ML Products vs PeaTMOSS dataset showing the percentage of repositories analyzed in each dataset that contained unit or integration tests for PTM features.

To address this, we've opted to use Docker containers. By utilizing containers, we can create isolated environments for each repository, allowing us to efficiently set up and execute tests.

Figure 4 outlines the process for generating automated test coverage for a repository. First, a list of repositories is sourced from the PeaTMOSS database. We then construct a single Docker image on the *python3.8-slim* base image, which includes the installation of `pytest` and `pytest-cov`. For each repository, we generate a setup script that clones the repository. The script then checks whether the repository includes a setup.py, Makefile, or *pyproject.toml*, and executes the appropriate setup commands based on this configuration. Subsequently, `pytest` is run to generate the coverage reports in JSON format.



(a) Repository Categories



(b) Test Patterns based on repository categories

Figure 6: Analysis of PeaTMOSS dataset

Repositories	PTM Used	# PTM Test Methods	# Files with PTM Test Methods
Beomi/exbert-transformers	gpt2	10	12
microsoft/LoRA	gpt2	9	7
rizwan09/REDCODER	gpt2	5	5
alexgaskell10/nlp_summarization	gpt2	7	6
CuongNN218/zalo_ltr_2021	gpt2	2	18

Table 1: A list of sample repositories showing the number of test methods containing the PTM usage & the corresponding number of files with such test methods found as a result of brute-force text searching.

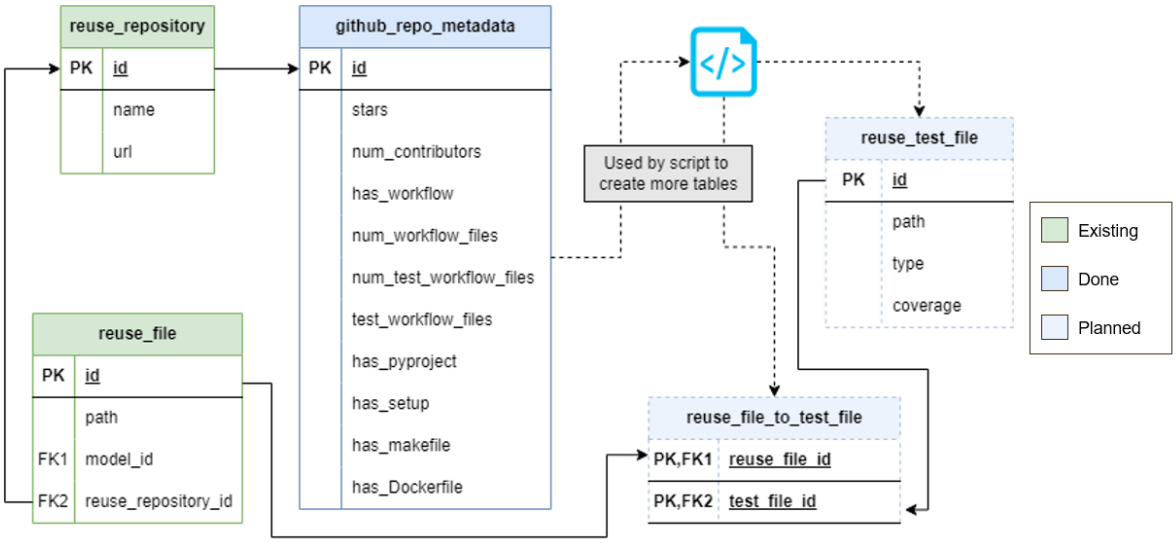


Figure 7: Extension to PeaTMOSS dataset where each file containing PTM features in *resuse_file* table are correlated to test files in the table *reuse_test_file* using intermediate metadata table *github_repo_metadata* and table *resuse_file_to_test_file*.

5.4 Locally running GitHub CI/CD workflows

Some of the OSS repositories explored contained CI/CD workflows which included commands to set up the repository and run appropriate unit, integration, and end-to-end tests. We experimented with *act* [A.2], an open-source tool for executing GitHub workflows locally (inherently using a Docker container), to understand its potential in our use case. The workflow files were available in YAML format which was easy to read and provided the exact steps to set up the repository and run the tests on a local machine. Similar to our previous approach with Docker, we intended to run the *act* tool to automatically generate coverage reports. However, these workflows don't usually generate coverage reports which are required in our case but rather check whether the tests are passing or not. Also, there are additional static-analysis checks like linting or code formatting done using Python frameworks like *black* or *flake8* that are also included as part of the CI/CD pipelines. Since these can be ignored in our use case, we simplified the workflow by extracting only the most relevant steps directly from the YAML files and step by step executed those commands automatically via. a script. Since some repositories contained workflow scripts for publishing packages or performing other checks, those workflows were excluded from our automation and only YAML files with signatures like {'test', 'integration', 'ci', 'e2e', 'dev', 'cov', 'unit', 'main', 'build', 'integration'} were included. Additional arguments were appended to the testing command to generate the coverage reports as well. For example, if a run command like `$[python -m pytest .]` is being called in the testing step for a repository, then we change the command to `$[python -m pytest . –cov-report json –cov {module_name}]`. The change will generate a coverage report in JSON format for each line of the source code present in the entire module. From the coverage reports, it becomes possible to correlate the files using PTM features to their corresponding unit tests and help us infer to what extent the Pre-Trained model features have been unit tested. Some CI/CD pipelines especially UI or web-based applications using PTM features containing integration or E2E tests are even more complicated to run as it is difficult to spin up the additional servers or browsers to run UI app testing in a local machine under significant memory and CPU constraints.

6 RESULTS AND ANALYSIS

6.1 Manual Investigation

An initial manual analysis of 14 repositories from the ML Products dataset resulted in the following observations. Out of them, 7 repositories contained pre-trained models and others contained usage of ML/DL libraries. From the repositories containing pre-trained models, we observed that only 2 repositories had unit tests and integration tests related to pre-trained models. Most repositories contained other application-specific unit tests and integration tests, which are generally present in the CI/CD workflow scripts under GitHub Actions. It was also noticed that documentation was also provided on how to run those tests and set up the environments required for testing.

Upon analyzing 50 repositories from the PeaTMOSS dataset, including those we couldn't set up. Our examination yielded valuable insights. Among these repositories, 30 had some form of unit testing implemented, while 22 even incorporated integration testing and 4 contained E2E tests. Moreover, out of the 30 repositories containing unit tests, 25 of them had some sort of test on testing the pre-trained models and their associated features. Additionally, as part of our analysis, we calculated the coverage of the test files. On average, among the 30 repositories with unit tests, we ran the unit tests of 11 repositories and got the code coverage to be 69.8%. This coverage assessment has provided us with substantial insights into the structure of the repositories, which facilitated the automation of certain processes. We also found it beneficial to categorize the repositories broadly into example or tutorial, research, and tool or product categories. Thus it was evident that most open source repositories included unit tests for testing parts of the PTM enabled features but not that many contained integration and end-to-end tests which answers our question **RQ1**. This may be due to the fact that these open-source projects mainly deal with Python based software packages and not production level applications or pilpelines which require integration and E2E tests. Enterprise applications of popular organizations are mostly closed source and are out of scope of our analysis.

As far as the previous ML Products dataset is concerned, it contained a notably small fraction of ML products that utilize pre-trained models, based on our current understanding as shown in Fig. 5. A portion of those repositories consists of ML models intended for training in conjunction with other models. A significant portion lacks documented unit testing, integration testing, and end-to-end testing protocols. The absence of testing documentation in the repositories suggests that the unit test coverage for the dataset will likely be lower compared to the PeaTMOSS dataset. In comparison to the PeaTMOSS dataset, the utilization of pre-trained models is exceedingly low in the ML Products dataset.

6.2 Extending PeaTMOSS Dataset

To do analysis on a huge scale we needed to automate the setup and testing of repositories. Turns out that even setting up a repository manually is such a tedious task, automating it would even be more difficult. Hence, we decided to focus on repositories that have a streamlined setup using GitHub CI/CD workflows, Makefiles, setup files, Dockerfiles, etc. A script was created to extract detailed metadata information about repositories (i.e. whether a project contains setup.py or *pyproject.toml*, etc.) within the PeaTMOSS dataset via the GitHub APIs. We have created a table called *github_repo_metadata* as shown in Fig. 7. Using this table we can decide what kind of setup we need to do and then test the repositories automatically. This will later allow us to create a new table called *reuse_file_to_test_file* which will map files with PTM to their corresponding test file.

6.3 Unit Testing Setup & Automation

6.3.1 Direct Text Searching. Direct text searching of certain code signatures or methods proved to be quite useful. From Table 1, we observe a sample of repositories, which includes the number of test functions employed for the pre-trained model and the respective files in which these functions were utilized. This data illustrates the variability in how developers approach testing the same pre-trained model. While some developers consistently apply a common set of functions, typically the same 10 functions, to test the model 'gpt2', others diverge by either not testing the model or by using fewer functions. This variance suggests differing levels of testing rigor. Moreover, by analyzing the number of files that incorporate these test functions, we gain insights into the integration of the pre-trained models within the broader project context, revealing how comprehensively each model is tested and integrated across different parts of the project.

In Fig. 8, we present the distribution of test functions, test files invoking these functions, and test files directly calling pre-trained models (without a function) across 50 repositories. The analysis covers five pre-trained models: 'gpt2', 'albert-base-v2', 'bert-base-uncased', 'roberta-base', and 'clip-vit-large-patch14'.

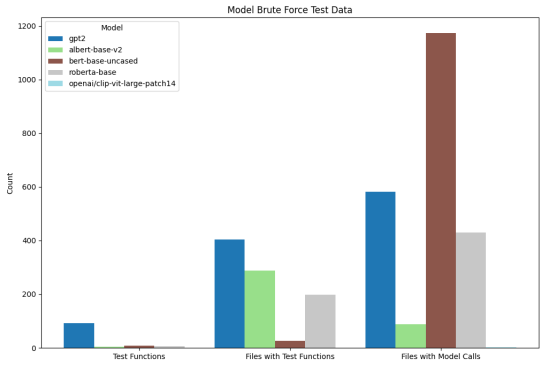


Figure 8: Distribution of model usage and tests obtained by brute force text searching of model signatures in test methods and files.

The figure highlights a tendency among developers to directly call the models without encapsulating them within specific test functions, as evident by a significantly higher frequency of model invocations outside of dedicated test functions. In general, the tests were single model tests covering a specific flow of the source code

involving PTM feature usage but there are multiple such tests checking separate flows of the same model. We observed very less implementation of pipeline or hierarchical testing patterns even in integration or E2E test files which makes it adamant that open-source projects mostly resort to unit testing single pre-trained model feature rather than the overall pipeline.

Hence, this approach not only highlighted the reuse of specific test functions within multiple project files but also provided insights into the broader integration practices of PTMs within the software development life-cycle thus giving an answer to our question (**RQ2**).

6.3.2 Using Pytest in Docker. After running our automated pipeline with Docker containers on 50 repositories, we managed to successfully set up 34 of them using our standard procedure. The ones that failed did so mainly because of module mismatches, with errors like "[tool.poetry] section not found in /app/project/pyproject.toml" and issues building the wheel for project packages. Fixing some of these issues required manual intervention and extensive debugging, so we excluded them from our study since we couldn't ensure these fixes could be integrated into our pipeline.

Out of the 34 repositories that were successfully set up, we ran tests and compiled their coverage reports. However, not all had the necessary files or packages to run `pytest`, so ultimately only 28 repositories generated coverage reports. These repositories are listed in A.3. The "PTM Percent %" column in 2 indicates the percentage of test coverage for files that use a pre-trained model. This list has been sourced from the *reuse_file* table in the PeaTMOSS dataset. The data shows that 13 repositories have "NaN" in this column, suggesting that files with pre-trained models weren't tested. The other 15 repositories tested these files, but we haven't yet found an efficient way to confirm if the functionality of the pre-trained models was actually tested.


```
...
"megvii-research/Sparsebit/sparsebit/quantization/modules/conv.py":{
            "executed_lines":[
                1,2,3,4,7,8,9,23,37,45,46,47,62
            ],
            "summary":{
                "covered_lines":12,
                "num_statements":34,
                "percent_covered":35.294117647058826,
                "percent_covered_display":"35",
                "missing_lines":22,
                "excluded_lines":0
            },
            "missing_lines":[
                24,25,26,27,33,34,35,39,40,41,42,48,49,50,51,58,59,60,64,65,66,67
            ],
            "excluded_lines":[
            ]
        }
...
```


Figure 9: Sample Coverage Report in JSON format

Moreover, the coverage percentage for these files is low compared to the overall coverage, indicating that not much emphasis has been placed on testing files that use pre-trained models. Along with the coverage reports, we examined the `pytest` command logs for additional insights, revealing that many tests failed with *ModuleNotFoundError* despite seemingly proper setup and ensuring that the correct development dependencies were installed.

6.3.3 Executing CI/CD commands. Using the *act* tool, we faced similar challenges in our efforts to automate testing across repositories through GitHub workflows or actions. This task turned out to be more complex due to the diverse configurations and technical requirements each repository presents. Our attempts to use this tool for executing GitHub Actions locally for running tests in containers have not been totally successful, revealing a steep learning curve and notable integration hurdles. For instance, while running GitHub workflows locally, we noticed that some repositories require external API keys like OpenAI, Datadog, etc. which hinders the automated running of those workflows locally. Even environment variables and secrets associated with these workflows make automation cumbersome forcing us to add complex command filtering rules in the automation scripts. Moreover, there were version issues with some of the repositories which essentially required a specific platform as well as a specific version of a package for running the tests. This combination of obstacles affected our analysis, emphasizing the need for us to adopt more flexible strategies.

One such strategy was to extract the contextually relevant CI/CD commands and run those directly without the use of the *act* tool. After stripping out the irrelevant parts of the CI/CD workflows and directly running the remaining steps, we generated coverage reports for some of the repositories, making the automation a bit more flexible. Fig 9 shows parts of a sample coverage file obtained with all the necessary test information which can then be utilized in correlating the files or lines in source code using PTM features and the amount of source code covered in those files. Consequently, this correlation provided a sense of the testing practices prevalent in OSS repositories using Pre-trained model features. Similar to our approach with the Docker container, the automated script for running the useful CI/CD commands was executed for 50 repositories. Out of these repositories, the execution of the script was successfully completed for XX repositories, and their corresponding coverage reports were generated. The results are shown in A.3. Some repositories required additional manual debugging to remove any errors during a second pass through the repositories. The data also shows that YY repositories have "NaN" suggesting that files with pre-trained models weren't tested.

These strategies gave a rough approximation of the quantity of unit tests that open source projects had on average and whether they covered the PTM enabled features which gave some initial ideas on answering our question *RQ3*. These numbers should not be treated as perfect due to the complexity and challenges involved in generating accurate coverage results but rather serve as an estimation on what type of testing patterns these repositories lean towards and how hard and cumbersome the repository setups were.

7 CONCLUSION

In conclusion, the manual investigation of the open-source repositories revealed crucial information about the testing patterns based on the characteristics and type of the repository. As expected, projects developing ML tools mostly contain some form of unit, integration, or E2E tests whereas tutorial repositories lacked proper tests. Also, in the case of popular repositories, we find mostly the usage of Pre-trained Transformer models and their variants like 'bert' and 'gpt2' which are mainly used for text tokenization as part of certain

downstream applications for NLP-based repositories and Stable-Diffusion models like 'sdxl' and 'openai/clip-vit-large-patch14' for vision based task. Tests were associated with these PTMs mostly in the case of tool-based projects covering different parts of the application, especially validating outputs obtained as a result of using Pre-Trained text tokenizers. Moreover, the study gave an overview of how the projects are structured, which helped automate the testing process enabling us to statistically analyze repositories on a larger scale. Though the automation methods are not entirely perfect and are not devoid of technical challenges, these approaches provide a stepping stone to make the analysis pipeline faster and much more streamlined in future. Our methodology will enable developers and the software community to take advantage of the GitHub metadata resources related to model testing and extend this study even further so that a general testing standard can be formulated for any open-source software projects or production-level enterprise applications utilizing Pre-Trained models.

REFERENCES

[1] Raja Ben Abdessalem, Annibale Panichella, Shiva Nejati, Lionel C. Briand, and Thomas Stifter. 2018. Testing autonomous cars for feature interaction failures using many-objective search. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE '18)*. Association for Computing Machinery, New York, NY, USA, 143–154. https://doi.org/10.1145/3238147.3238192

[2] Denis Baylor, Levent Koc, Chiu Koo, Lukasz Lew, Clemens Mewald, Akshay Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Whang, Martin Wicke, Eric Breck, Jarek Wilkiewicz, Xin Zhang, Martin Zinkevich, Heng-Tze Cheng, Noah Fiedel, Chuan Foo, Zakaria Haque, and Vihan Jain. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. 1387–1395. https://doi.org/10.1145/3097983.3098021

[3] Houssem Ben Braiek and Foutse Khomh. 2020. On testing machine learning programs. *Journal of Systems and Software* 164 (2020), 110542. https://doi.org/10.1016/j.jss.2020.110542

[4] Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, and D Sculley. 2016. What's your ML test score? A rubric for ML production systems. (2016).

[5] Janne Cadamuro. 2021. Rise of the Machines: The Inevitable Evolution of Medicine and Medical Laboratories Intertwining with Artificial Intelligence—A Narrative Review. *Diagnostics* 11, 8 (2021). https://www.mdpi.com/2075-4418/11/8/1399

[6] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding flaky tests: the developer's perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19)*. ACM. https://doi.org/10.1145/3338906.3338945

[7] Vahid Garousi, Austen Rainer, Per Lauvås, and Andrea Arcuri. 2020. Software-testing education: A systematic literature mapping. *Journal of Systems and Software* 165 (2020), 110570. https://doi.org/10.1016/j.jss.2020.110570

[8] Wenxin Jiang, Jason Jones, Jerin Yasmin, Nicholas Synovic, Rajeev Sashti, Sophie Chen, George K. Thiruvathukal, Yuan Tian, and James C. Davis. 2023. PeaTMOSS: Mining Pre-Trained Models in Open-Source Software. arXiv:2310.03620 [cs.SE]

[9] Shuyue Li, Jiaqi Guo, Jian-Guang Lou, Ming Fan, Ting Liu, and Dongmei Zhang. 2022. Testing machine learning systems in industry: an empirical study. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice* (Pittsburgh, Pennsylvania) *(ICSE-SEIP '22)*. Association for Computing Machinery, New York, NY, USA, 263–272. https://doi.org/10.1145/3510457.3513036

[10] Yu Liu, Pengyu Nie, Owolabi Legunsen, and Milos Gligoric. 2023. Inline Tests. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (, Rochester, MI, USA,) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 57, 13 pages. https://doi.org/10.1145/3551349.3556952

[11] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*. ACM. https://doi.org/10.1145/3238147.3238202

[12] Silverio Martínez-Fernández, Justus Bogner, Xavier Franch, Marc Oriol, Julien Siebert, Adam Trendowicz, Anna Maria Vollmer, and Stefan Wagner. 2022. Software Engineering for AI-Based Systems: A Survey. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 37e (apr 2022), 59 pages. https://doi.org/10.1145/3487043

[13] Nadia Nahar, Haoran Zhang, Grace Lewis, Shurui Zhou, and Christian Kästner. 2023. A Dataset and Analysis of Open-Source Machine Learning Products. arXiv:2308.04328 [cs.SE]

[14] Nadia Nahar, Shurui Zhou, Grace Lewis, and Christian Kästner. 2022. Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. arXiv:2110.10234 [cs.SE]

[15] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM. https://doi.org/10.1145/3132747.3132785

[16] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. tsDetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1650–1654. https://doi.org/10.1145/3368089.3417921

[17] Mauro Pezzè, Konstantin Rubinov, and Jochen Wuttke. 2013. Generating Effective Integration Test Cases from Unit Ones. *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, 11–20. https://doi.org/10.1109/ICST.2013.37

[18] Jia Wang, Tong Sun, Benyuan Liu, Yu Cao, and Degang Wang. 2018. Financial Markets Prediction with Deep Learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 97–104. https://doi.org/10.1109/ICMLA.2018.00022

[19] Amy X. Zhang, Michael Muller, and Dakuo Wang. 2020. How do Data Science Workers Collaborate? Roles, Workflows, and Tools. arXiv:2001.06684 [cs.HC]

[20] Chuheng Zhang, Yuanqi Li, Xi Chen, Yifei Jin, Pingzhong Tang, and Jian Li. 2020. DoubleEnsemble: A New Ensemble Method Based on Sample Reweighting and Feature Selection for Financial Data Analysis. In *2020 IEEE International Conference on Data Mining (ICDM)*.

A APPENDIX A

A.1 Datasets

The following datasets have been used in this research study

- ML Products - https://osf.io/hx4m7
- PeaTMOSS - https://github.com/PurdueDualityLab/PeaTMOSS-Demos

A.2 Tools and Frameworks

The tools that have been used for our research are as follows:

- Docker - https://www.docker.com/
- GitHub API - https://docs.github.com/en/rest
- nektos/act - https://github.com/nektos/act
- tox - https://tox.wiki/en/4.14.1/
- pytest - https://docs.pytest.org/en/8.0.x/
- pytest-cov - https://pytest-cov.readthedocs.io/en/latest/
- Coverage - https://coverage.readthedocs.io/en/7.4.3
- AST - https://docs.python.org/3/library/ast.html

A.3 Repositories Analyzed

Table 2 shows the repositories analyzed through automated coverage report generation and shows a comparison between the results obtained using both Docker as well GitHub CI/CD workflow commands.

B APPENDIX B

B.1 Ethical Assessment

B.1.1 ***Souradip Pal.*** From an ethical standpoint, the main factor contributing to the potential risk and other concerns involved is the data collected from GitHub repositories which should be used ethically and responsibly, respecting users' privacy rights and adhering to relevant data protection regulations. Mining this data

Repositories	Pytest + Docker		CI/CD Workflows	
	PTM Coverage %	Overall Coverage %	PTM Coverage %	Overall Coverage %
github.com/CarperAI/trlx	2.89	25.12	3.02	13.25
github.com/EleutherAI/lm-evaluation-harness	0.00	33.98	NaN	NaN
github.com/EleutherAI/gpt-neox	NaN	NaN	10.46	2.79
github.com/PaddlePaddle/PaddleNLP	2.18	17.55	NaN	NaN
github.com/Sygil-Dev/Sygil-WebUI	0.00	13.33	NaN	NaN
github.com/bmaltais/kohya_ss	NaN	2.27	NaN	NaN
github.com/carson-katri/dream-textures	NaN	0.0	NaN	NaN
github.com/facebookincubator/AITemplate	NaN	5.26	NaN	NaN
github.com/facebookresearch/ParlAI	1.17	35.34	NaN	NaN
github.com/hpcaitech/ColossalAI	0.00	27.72	48.64	13.253
github.com/huggingface/accelerate	1.56	32.92	NaN	NaN
github.com/huggingface/datasets	19.15	34.31	NaN	NaN
github.com/huggingface/diffusers	7.73	30.14	0.0	10.80
github.com/huggingface/text-generation-inference	NaN	8.38	0.0	43.74
github.com/huggingface/tokenizers	NaN	24.13	NaN	NaN
github.com/hwchase17/langchain	NaN	45.66	NaN	NaN
github.com/jerryjliu/llama_index	NaN	42.1	NaN	NaN
github.com/kingoflolz/mesh-transformer-jax	0.00	4.17	NaN	NaN
github.com/kohya-ss/sd-scripts	NaN	2.44	NaN	NaN
github.com/ludwig-ai/ludwig	24.70	50.97	NaN	NaN
github.com/lvwerra/trl	1.98	25.27	NaN	NaN
github.com/microsoft/DeepSpeedExamples	73.08	67.46	NaN	NaN
github.com/microsoft/guidance	NaN	NaN	0.0	23.82
github.com/opencv/cvat	NaN	11.19	NaN	NaN
github.com/pytorch/tutorials	NaN	18.84	NaN	NaN
github.com/ray-project/ray	NaN	10.62	NaN	NaN
github.com/salesforce/LAVIS	0.00	5.83	NaN	NaN
github.com/skypilot-org/skypilot	NaN	12.14	NaN	NaN
github.com/tensorflow/models	NaN	24.02	NaN	NaN
github.com/zilliztech/gptcache	11.54	15.19	NaN	NaN
github.com/gradio-app/gradio	NaN	NaN	0.0	23.82
github.com/NVIDIA/NeMo	NaN	NaN	0.0	6.78

Table 2: Coverage results from automated setup and testing of popular repositories by running `pytest` in Docker and running GitHub Workflow CI/CD commands including the percentage of files containing the Pre-Trained models that are unit tested. PTM coverage represents coverage of file containing PTM features. 'NaN' represents that the coverage was not generated due to setup issues or unavailability of tests. This shows our approach to run `pytest` in Docker proved to be more successful than running GitHub CI/CD workflow commands locally.

without explicit consent may raise concerns about privacy and data ownership. Developers may not anticipate their code being used for any other research, leading to potential breaches of trust. GitHub projects may be subject to various licenses and intellectual property rights, and hence using these projects for research purposes could potentially infringe upon these rights, especially if the analysis is used for commercial purposes without proper attribution or compliance with licensing terms. However, we are only dealing with open-source projects that are publicly available and are not involved in any sort of modification or distribution of the code and data associated with any of the repositories. We are following the necessary guidelines to obtain the appropriate permissions for data access while citing any sources from where the data was obtained e.g. the PeaTMOSS and ML Products dataset. Care was also taken to handle sensitive information responsibly and to mitigate any potential security risks associated with accessing and analyzing code from public repositories.

We also maintain transparency in the project's objectives, methodologies, and findings to ensure accountability and clearly communicate which portions of the project involve manual intervention and portions driven by data, and how the project's outcomes may impact various parties, including developers and the software community. Mining GitHub repositories introduces several complexities and opacity into the analysis process. Moreover, it could perpetuate biases present in the coding practices of the contributors, leading to a biased analysis. Depending on how the analysis is used in the future, it could influence coding practices, testing methodologies, and tooling preferences. It may be challenging to understand how

can individuals or organizations be held accountable just based on the outcomes of the analysis. Thus we maintain transparency about the methods used and the limitations of our work which are essential to maintain trust and accountability. Any analysis or study conducted in this regard was guided by ethical principles, with careful attention paid to mitigating risks and ensuring fairness, transparency, and respecting the rights of developers and contributors.

B.1.2 ***Mohit Pandiya.*** Data collection from a range of sources can present considerable ethical dilemmas, especially regarding the concept of consent. In our project, we use data from primarily 3 sources. These are ML Products [13] dataset, PeaTMOSS [8] dataset, and from GitHub API. Both the ML Products and PeaTMOSS datasets are openly available online and instructions to access them were mentioned in their respective papers. Currently, we use a sample of the PeaTMOSS dataset which is included in the GitHub repository, we will get access to the complete PeaTMOSS dataset located in Globulus if required. The repositories mentioned in this dataset are fetched automatically and when data is gathered through automated systems, obtaining clear and meaningful consent becomes particularly challenging. The risk here is that consent may become a mere formality rather than a genuine agreement. In the context of our project, we collect data automatically from GitHub, a widely used platform where developers can share code to collaboratively create open-source software projects. When a developer creates a repository on GitHub, they have the option to make it public or private. This choice gives the repository owner control over who can access the content—whether it's just a select group or the entire GitHub community. In our project, we only collect data from public repositories, respecting the boundary set by developers who opt to keep their projects private.

Another aspect is the usage of these repositories, all the repositories we looked at had some license, these licenses are in place to control the reuse and re-distribution of the project but since we are just analyzing the projects and how are these tested there was no question of actually using or reusing them.

Regarding the security aspects of our project's artifacts, we don't handle any personally identifiable information, so there's no need for encryption. The only details we track are the GitHub usernames and the repository names associated with those accounts. From the repository perspective, we don't use or process any of its internal data. Although we don't specifically check if the repository contains personal information, we clone and analyze it within a Docker container and then delete the container before moving on to the next repository. The only data we save from the repositories is their metadata, which includes the presence or absence of some key setup files and their test results.

B.1.3 ***Mohammed Metwaly.*** In our study, we respect intellectual property by strictly adhering to the licenses of the software and data used in our analysis. For example, when referencing research papers in our analysis, we follow best practices for citation and attribution. We provide a proper citation for each paper used, including the author(s), title, publication venue, and publication date. This not only acknowledges the original work but also allows readers to easily locate and verify the referenced papers. Additionally, in our analysis, we strictly adhere to ethical standards regarding the reuse of products and research papers. We ensure that any products or artifacts accessed during the study are appropriately cited, respecting their respective licenses. Furthermore, we emphasize the importance of proper citation for research papers, ensuring that all relevant contributions are acknowledged and attributed to their original sources.

During the brute force analysis, it's crucial to note that we do not reuse or redistribute any products or contents from the repositories under study. The analysis solely involves scanning the contents for keywords and patterns, without extracting or reusing any actual content. This approach maintains the integrity of the original work and prevents any unintended use or distribution of the repository contents. Furthermore, the results of the analysis do not share any private information about the repositories that is not already found in the PeaTMOSS dataset. The analysis strictly shows frequencies and the PTMs used within the repositories.

C APPENDIX C

C.1 Reflection - Project Postmortem

This work presented several challenges as well as unique opportunities to learn and grow in the area of software engineering at times when approaches to solving software engineering problems using pre-training machine learning or deep learning models are becoming more and more common. Here we provide a brief reflection on the process and outcomes of the project.

C.1.1 What Went Well? At the beginning of the project, we managed to conduct manual analyses of both the PeaTMOSS and ML Products datasets, which aligned with our planned objectives. Despite challenges, we were able to find unit and integration tests in 60% of the repositories analyzed. Additionally, we encountered repositories with end-to-end (E2E) tests even though it seemed uncommon since these are not production-level applications. Shifting our focus to the PeaTMOSS dataset proved fruitful, making it easier to locate tests manually, especially those covering pre-trained model (PTM) features. Through manual analysis, we identified prevalent usage of PTMs associated with Natural Language Processing (NLP), such as 'gpt2' and 'bert'. This insight enhanced our understanding of the datasets which propelled us to actively explore automation strategies and tools, such as developing scripts using the GitHub APIs and experimenting with *act* and Docker SDK and other essential steps towards automating repository testing. Deepening the understanding of the PeaTMOSS dataset's database schema, utilizing GitHub APIs to fetch repository metrics like stargazers and contributors, and filtering relevant repositories for further investigation improved analysis efficiency.

C.1.2 What Went Poorly? Identifying PTM or Deep Learning (DL) or Machine Learning (ML) library usage in large repositories with hundreds of files proved challenging, requiring a manual search through each repository. Setting up compatible environments to run unit and integration tests took more time than anticipated, particularly for repositories compatible only with Linux-based machines or utilizing particular setup approaches. Automating testing across repositories proved difficult due to diverse configurations, technical requirements, version issues, and external dependencies

like API keys. Streamlining the repository setup processes and possibly transitioning to Linux-based cloud servers like AWS will avoid package version mismatches and help us tackle the money and CPU limitations of notebook interfaces like Google Colab.

C.1.3 Issues & their Root Causes. - A critical challenge still persisting in our proposed approach is the use of external API keys in GitHub CI/CD workflows. Some projects uses OpenAI APIs to leverage better tokenization and other NLP features of their GPT-based models in their applications. Also, the API key property name varies a lot among the repositories. This is valid even for other environment variables and secret keys and hence a unified approach to setting such variables during execution is unattainable. Another issue is the Python version that each repository requires to run. There is either no information about the Python version that the project runs on or they exist in parts of the source code like *README.md* or *pyproject.toml* file from which extracting them is difficult. Thus these issues restrict us in achieving full automation and force us to be content with partial automation. By addressing these issues, we hope to overcome the aforementioned challenges and optimize the workflow for more effective and efficient repository analysis and testing in the future.