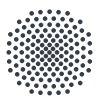Software Lab
Institute of Software Engineering
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart

Master Thesis

DyPyBench: A Benchmark of Executable Python Software

Piyush Krishan Bajaj

Course of study:	INFOTECH
Examiner:	Prof. Dr. Michael Pradel
Supervisor:	Islem Bouzenia
Started:	November 7, 2022
Completed:	May 7, 2023

University of Stuttgart
Germany

SOLA
SoftwareLab

Abstract

Python has evolved from a language to write small scripts to one of the most popular programming languages. Besides being popular, Python also is a very dynamic language that does not require the programmer to provide type annotations and allows object structures to be modified during execution. These two properties make Python a prime candidate for dynamic program analyses to detect programming errors, security vulnerabilities, and performance issues. To test and evaluate dynamic analyses and runtime systems, researchers and practitioners commonly rely on benchmark suites. Unfortunately, there currently is no comprehensive benchmark of executable Python projects, which hampers the development of dynamic analyses.

In this work, we present DyPyBench, a dynamic benchmark of executable Python software. The benchmark is large-scale, diverse, versatile, long-term, ready-to-analyze and ready-to-run and is useful for a wide range of applications. The 50 projects in DyPyBench provide test suites that help to perform dynamic analyses. We provide this benchmark as a Docker image which makes it easier to use for researchers and practitioners. The benchmark also includes three tools for code analysis tasks, namely, LExecutor, DynaPyt and PyCG.

We evaluate the usage of the benchmark with the three tools integrated in DyPyBench. Our benchmark provides 41451 successful test cases for dynamic analyses. The benchmark is used to generate 547830 data points for neural model training in LExecutor, which provide an accuracy in the range of 71.86% and 93.67%. Furthermore, we use the benchmark to generate static and dynamic call graphs using the tools PyCG and DynaPyt respectively. The comparison of these call graphs show us a match of 3147 caller functions and 2938 callees despite the limitations of DynaPyt and PyCG.

Contents

1 Introduction

Python is a popular and versatile programming language that is distinguished by its interpreted nature, high-level syntax, and support for a variety of programming paradigms. It makes use of dynamic typing and efficient memory management techniques, and its modular design makes it easy to extend and integrate programmable interfaces into pre-existing applications [27]. Furthermore, Python provides a variety of dynamic features, such as object creation, attribute access, module loading, and function definition, making it appealing to developers of all skill levels.

Recent studies have demonstrated Python's widespread usage in various applications, including analysis, bug detection, performance assessment, and vulnerability testing [37, 28, 11]. This can be attributed to its accessible syntax, extensive community support, and comprehensive libraries. Python's robust support for data analysis, visualization, statistical analysis, and cross-platform compatibility renders it an exceptional choice for the development of cross-platform software and tools. Furthermore, Python is equipped with an assortment of testing and debugging tools, ranging from built-in frameworks like unittest and pytest to third-party tools such as PyCharm [25] and PyDev [26] for debugging and code refactoring. These tools aid developers in identifying and resolving bugs while optimizing code performance.

In the realm of high-level programming languages, Python, along with Java and C++, are frequently employed for their potency, object-oriented structure, and dynamic features, rendering them suitable for intricate software development applications. To assess the dynamic features of these languages, dynamic benchmarks can be employed, which involve gauging the performance of a system or application under real-world conditions. Contrasting with static benchmarks that evaluate systems or applications under controlled circumstances, dynamic benchmarks endeavor to emulate actual usage patterns of users and applications to attain more accurate performance measurements. Java and C++ boast numerous dynamic benchmarks, such as Java Microbenchmark Harness (JMH) [16], The DaCapo Benchmark [8], The Computer Language Benchmarks Game [33], and Boost.Benchmarks [9]. Although Python possesses some dynamic benchmarks, such as PyPerformance [34] and Apache Bench [1], they are not general-purpose benchmarks like those offered for Java and C++. These python benchmarks evaluate program performance for a single dynamic feature in a specific domain.

While the existing python benchmarks evaluate single dynamic feature from a specific domain, a general-purpose benchmark would allow us to analyze the different dynamic features in the real world conditions. Such a benchmark would allow the evaluation of program performance from a wide range of application domains such as machine learning, web development, server development and virtual environments, brought to the table due to the popularity and wide spread usage of

1

python. With the general-purpose python dynamic benchmark one could generate run-time analysis using tools such as systrace [32] and DynaPyt [14]. A general-purpose benchmark can provide training data for machine learning (ML) and deep learning (DL) tasks of software engineering such as bug detection [24], code completion [19], quality analysis [2, 4], code refactoring [3], and testing [17, 23, 38] which improve the productivity, efficiency and quality of code.

In this work, we provide such a general-purpose dynamic benchmark of executable python software. This benchmark is named DyPyBench and consists of 50 python projects including frameworks and libraries. The entire benchmark, encapsulated inside a docker container, can be setup using the docker image containing the projects, alongside the analysis frameworks such as DynaPyt [14] and PyCG [29]. Additionally, the benchmark also integrates LExecutor [31], which is a learning-guided approach for executing arbitrary code snippets in an underconstrained way. All of the python projects are open-sourced and available on GitHub [15] including the analysis frameworks and LExecutor. The chosen projects are popular and belong to diverse application domains such as audio e.g., pydub, command-line tools e.g., thefuck, and computer vision e.g., scikit-learn. The benchmark provided in this work provides a single command line access interface to run test suites and analysis. The access interface provides the necessary functionality to execute dynamic analysis through DynaPyt, produce trace files via LExecutor, and generate static call graphs using PyCG. This work aims to achieve the following properties for the dynamic benchmark:

- Large-scale

- Diverse

- Ready-to-run

- Ready-to-analyze

- Compositional

- Long-term

- Extensibility

Developing a benchmark that meets the aforementioned properties is a challenging endeavor that presents various obstacles. The following paragraphs discusses some of the challenges encountered during the development process and outlines the strategies implemented to overcome them.

One major challenge for DyPyBench is to incorporate a large and diverse set of projects from various application domains. Although numerous Python projects satisfying the criteria are available on GitHub, the benchmark can only accommodate a limited number of them. To address this limitation, we apply rigorous selection criteria to choose projects from a predefined and classified list. Additionally, we consider GitHub stars as a metric to filter out non-reputable projects and ensure the inclusion of high-quality projects.

DyPyBench faces another hurdle in executing projects effectively, which is essential for its role as a benchmarking tool that relies on dynamic analysis. The successful execution of projects is achieved through the execution of test suites for each project. While not all projects include

test suites that function flawlessly, it is crucial for test suites to run without errors to ensure the accurate operation of dynamic analysis tools. DyPyBench overcomes this challenge by either skipping problematic test suites or fixing them to ensure proper execution.

Ensuring the accessibility and longevity of DyPyBench for users presents another challenge. By packaging and exporting DyPyBench as a docker container, its accessibility is enhanced, as it eliminates operating system dependencies and provides a consistent, readily usable environment to all users. This also overcomes the longevity challenge, as it facilitates the preservation of the projects and their respective environments.

The fluid nature of open-source projects often leads to regular modifications to their source code. Such changes can affect the execution environment of a project, consequently impacting the effectiveness of DyPyBench. To overcome this challenge, DyPyBench clones a stable and fixed version of the source code to ensure consistency in the execution environment.

In order to evaluate the dynamic nature of the benchmark, we run test suites and find that overall 91.83% of test cases out of 45,086 execute successfully in 1362.56 seconds. For evaluation of the usefulness of the benchmark, we use LExecutor and the analysis frameworks for two distinct applications. First, we provide a large amount of training data for the neural model of LExecutor in an attempt to improve the accuracy of the model. DyPyBench is able to provide nearly 2.5 times the data compared to original training data used by the authors. With this new training data, the neural model is able to predict the true value with an accuracy between 71.86% and 93.67% in different experimental setups. Second, we provide some statistical analysis of call graphs generated using static and dynamic code analysis for Python. PyCG generates static call graphs, whereas DynaPyt generates run-time call graphs in DyPyBench. We find that static and dynamic call graphs have 3147 and 2938 common caller and callee entries respectively. DynaPyt has 66.41% of callers present in PyCG, whereas PyCG has 51.93% of callers present in DynaPyt. For callees, DynaPyt has 28.85% present in PyCG, wheras PyCG has 47.58% present in DynaPyt.

In summary, this work contributes the following:

- A python benchmark containing 50 projects from varied application domains.

- A benchmark with preinstalled frameworks for code analysis of python projects.

- A new and large dataset for training a neural model of LExecutor.

- A statistical analysis of static and run-time call graphs for Python.

2 Background

In this chapter, we provide an overview of the topics needed to understand the work done in this thesis.

2.1 Benchmarks

Benchmarks are tools used to measure the performance and efficiency of computer systems or components. They evaluate the speed and quality of hardware, software, and applications to optimize performance, identify bottlenecks, and make informed decisions about system configurations. There are many types of benchmarks that measure specific aspects of a computer system, such as processing power, memory speed, disk access time, or application performance. Benchmarks simulate real-world workloads and produce scores that can be compared across different systems or components. When it comes to evaluating the performance of a software system, there are two primary types of benchmarks: static benchmarks and dynamic benchmarks. Static benchmarks involve running a predetermined set of tests on a given system or application, with fixed input parameters and test conditions. Dynamic benchmarks involve running a set of tests that are designed to adapt and respond to the behavior of the system being tested. These benchmarks are typically more complex than static benchmarks. Overall, benchmarks are an essential tool for evaluating computer systems and components, enabling users to optimize performance and make informed decisions about hardware and software configurations.[6, 7]

2.2 Code Analysis Frameworks

Code analysis frameworks are tools used in software development to analyze source code and identify potential issues, vulnerabilities, and errors. They help developers to maintain code quality, improve security, and ensure compliance with coding standards and best practices. Popular code analysis frameworks include SonarQube, Checkstyle, ESLint, and Fortify, among others. [18] These frameworks are valuable tools for software developers as they provide detailed reporting and analysis features, enforce coding standards, identify potential errors, and help to remediate security vulnerabilities. Ultimately, code analysis frameworks help developers to write better, more secure, and higher-quality software code. [2, 4, 30] In our benchmark, we use two code analysis frameworks. We use DynaPyt to perform dynamic analysis and PyCG to generate call graphs using static analysis of code.

2.2.1   DynaPyt

DynaPyt is a tool for dynamic analysis of Python code. It aims to provide insights into the behavior of Python programs, such as performance and memory usage, by analyzing the runtime behavior of code. DynaPyt provides several features, such as the ability to collect data on function calls, memory allocation, and object creation, and to visualize the results in a user-friendly interface. The tool is designed to be easy to use and to work with existing Python code, making it accessible to a wide range of users, from researchers to developers. The tool makes it possible to get a detailed understanding of the behavior of Python programs. This information can be used to identify performance bottlenecks and optimize the program for better performance. With its ease of use, flexibility, and range of features, DynaPyt is a valuable tool for anyone working with Python code. [14]

2.2.2   PyCG

Call graphs are useful in various contexts, including profiling and vulnerability analysis. Despite Python's popularity, there are very few tools available for generating call graphs, and these tools suffer from effectiveness issues that limit their practicality. The article proposes a static approach for call graph generation in Python, which involves computing all assignment relations between program identifiers of functions, variables, classes, and modules through an inter-procedural analysis. The resulting call graph is produced by resolving all calls to potentially invoked functions. The proposed approach is designed to be efficient and scalable, handling several Python features such as modules, generators, function closures, and multiple inheritance. [29]

3 Approach

In this chapter, we describe the approach which we have adopted to create the dynamic benchmark of executable python software. We also explain the evaluation metric we use to measure the effectiveness of DyPyBench. The overall workflow of the approach is shown in the Figure 3.1.
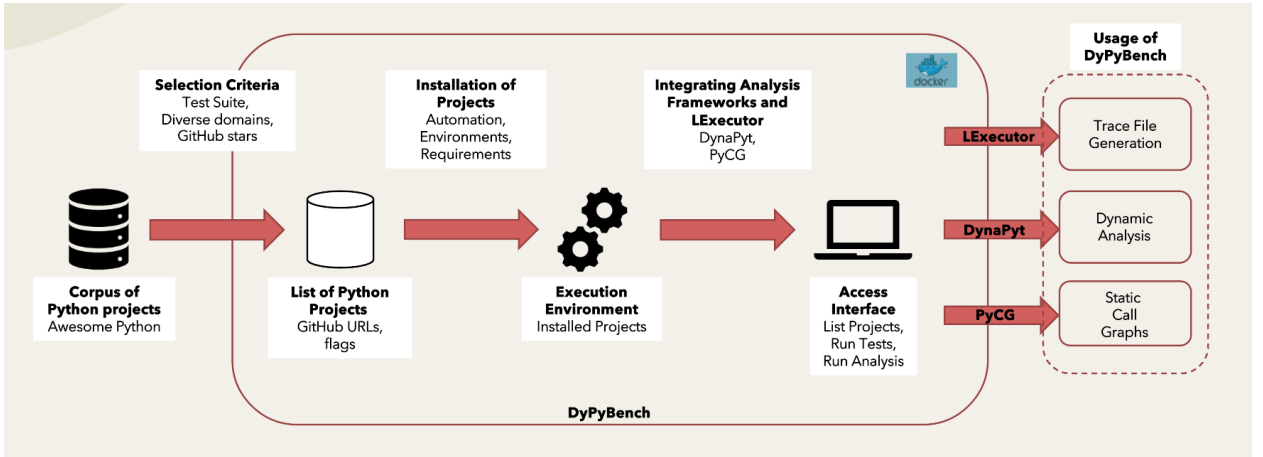

Figure 3.1: Overall Approach of DyPyBench

As can be seen from the Figure 3.1, we first apply the selection criteria to the corpus of python projects to select a number of projects (Section 3.1) to generate a list. The list of projects is then used to prepare the benchmark by installing the projects and their dependencies using automation (Section 3.2). We proceed by integrating the two analysis frameworks, DynaPyt and PyCG (Section 3.3). We also integrate LExecutor into the benchmark (Section 3.4). Finally, we package and export the benchmark with an access interface for use by researchers and developers (Section 3.5).

With the provided approach, we aim to achieve the properties as listed below.

- **Large-scale** The benchmark should comprise tens of real-world open-source projects, allowing users to evaluate their analyses on a wide range of code.

- **Diverse** The benchmark should contain projects from a diverse range of application domains that reflect the state of today's Python ecosystem.

- **Ready-to-run** There should be a single interface to run each project in the benchmark, making it easy for users to set up and execute the entire benchmark.

- **Ready-to-analyze** To enable dynamic analyses, the executions of all projects in the benchmark should be set up to be analyzed.

- **Compositional** To help users understand the behavior of specific projects or even individual test cases, it should be easy to run subsets of full benchmark.

- **Long-term** The benchmark should be built using commonly used tools and formats, e.g., pip and Docker, to ensure its longevity.

- **Extensibility** The benchmark should be easily extensible, allowing users to add, remove or update the tools, frameworks and projects.

3.1 Projects Selection

In this section, first we discuss about the available python projects and why we use awesome-python as a corpus for our benchmark (Section 3.1.1). Then we discuss about the three selection criteria which we use to select projects from the corpus to make our benchmark diverse and large-scale (Section 3.1.2).

3.1.1 Corpus of Python Projects

Python's ease of use and popularity, as well as the usage in different domains has led to a large number of projects being developed and made available to the community as open-source projects. GitHub alone contains a lot of open-source repositories that have python as their primary programming language. Furthermore, there are some GitHub repositories which provide a collection of python projects such as Awesome Python [1], Awesome Python Applications [2] and Python Projects [3]. These collections of projects also provide us with a classification for the projects based on the application domains. In this work, we use the awesome-python project which contains a curated list of some of the awesome open-source python projects including libraries, frameworks and software. The awesome-python project is the corpus of projects for our benchmark as it contains a collection of 679 python projects. Awesome-python further classifies these projects into 92 main categories, out of which some of the categories are further divided into sub-categories. These sub-categories contain the subset of the main category based on technology, tool and their intended usage. For example, the sub-categories for Distributed Computing are Batch Processing and Stream Processing which are the two intended ways of processing data. The top 10 categories and the number of projects in those are listed in table 3.1. Appendix A provides more details on the various categories and number of projects in each category provided by awesome-python project.

3.1.2 Selection Criteria

To select projects for our benchmark, we use three criteria namely diverse domain, test suite execution with pytest and GitHub stars. To ensure the diversity property presented before, we sample the projects from across multiple categories covering as many application domains as possible provided by the awesome-python corpus. Since, in this work we are creating a dynamic benchmark

---

[1] https://github.com/vinta/awesome-python/
[2] https://github.com/mahmoud/awesome-python-applications
[3] https://github.com/practical-tutorials/project-based-learning

Category	Number of Projects
Testing	30
Text Processing	22
Science	21
Code analysis	18
Debugging Tools	18
Specific Formats Processing	18
Command Line Tools	17
GUI development	16
Database Drivers	15
Image Processing	15

Table 3.1: Top 10 Project Categories (Awesome Python)

we focus on the projects for which we can perform execution. Test suites provide us with files that can easily execute the code using testing libraries such as pytest and unittest. Thus we select projects with test suites which can run using the popular pytest library that is also compatible with unittest. Lastly, to ensure the criterion of GitHub stars, we select projects which have at least 500 stars on its GitHub repository. The number of stars a project has on GitHub is an indication of how popular and well-regarded it is in the community. More stars generally mean more people have found the project useful and it has a larger user base and community of contributors [10]. We find 500 stars to be sufficient since it allows us to have the required number of well-regarded projects.

3.2 Benchmark Preparation

In this section, we start with a discussion about how we use flags in a list of projects to automate the installation process (Section 3.2.1). We then discuss about how we use these flags in our work to install 50 projects (Section 3.2.2). Finally, we discuss about the working environment which is created by the installation of the above 50 projects (Section 3.2.3).

3.2.1 List of Python Projects

Applying the selection criteria as described before, we select 50 open-source python projects. Although, the installation and setup of these projects follow similar steps, there are certain differences which vary from project to project. For example, the path and the name of requirements file to install dependencies varies for each project. Some of these differences can be attributed to the directory structure of source code present in the GitHub repository. In order to automate the process of installation and setup of projects, we need a way to work around these differences for each project. In this work, we use a list with specific flags inside a text file to handle the aforementioned differences. This structure of a single entry which represents one project in this list is **project_url requirement_flag *requirement_file test_suite**. In the structure, *project_url* corresponds to Git URL for the project, whereas *requirement_flag* represents the information pertaining to the presence or absence of requirement file. The *requirement_flag* can have 2 possible

values, *rt* and *t*. The next flag in the entry is the *requirement_file*, which is the path of require-ment file. This flag is optional and is present only when the *requirement_flag* is *rt*. The final flag in the entry is *test_suite* that contains the path of the test directory which is needed to execute tests. Furthermore, it is easy to add, modify or delete an entry from the text file which makes the benchmark extensible. Table 3.2 shows two example entries in the list. The first entry is a project with a requirement file present at src/requirements.txt and test suite in the directory tests, while the second entry does not contain a requirement file and its test suite is present in the directory src/tests.

https://github.com/test/test-project.git	rt	src/requirement.txt	tests
https://github.com/tester/test-project.git	t	src/tests	

Table 3.2: List of Projects (Collection Text File Structure)

3.2.2 Installation of Projects

The above list of projects with the specified flags in a text file is used to automatically install the projects with a single command. With the URL to the GitHub Repository and optional requirement file, we perform a series of steps to install each project with its required dependencies. Firstly, we clone the repository from the URL provided in the text file to get the source code. We then create a virtual environment to avoid dependency conflicts between different projects. Then we install the project to the virtual environment using the source code cloned before. Finally, if there is a requirement file we install the dependencies specified in this file to the virtual environment. The mundane and repetitive task of performing these steps for each of the 50 projects is automated using scripts. Scripts can handle repetition through the use of loops such as for and while. They can also work with exceptional cases using conditionals such as if else statements.

3.2.3 Execution Environment

The automated installation of projects provide us with a collection of 50 python projects. This collection forms the pool of projects which our benchmark provides to perform code analysis or to generate training data for neural models. Each project installed using the automation is present inside a sub-folder in the collection. This makes it easier to work on individual projects when needed. When a particular project from the collection needs to used, a duplicate copy of the same is created. This ensures the long-term usage of the projects in the benchmark by preserving the projects and their virtual environments. The dynamic nature of python and open-source ecosystems results in frequent changes to the projects. Another advantage provided by duplication is avoiding the re-installation of projects which saves significant time and effort.

3.3 Integrating Analysis Frameworks

To satisfy the property of ready-to-analyze, we need to integrate some code analysis frameworks into our benchmark. In this work, we inncorporate two such frameworks, namely, DynaPyt, which

a general purpose dynamic analysis framework and PyCG, which is a static analysis framework to generate call graphs. The following sections explain the integration of these two frameworks.

3.3.1   DynaPyt

In this work, we use DynaPyt to perform dynamic analysis for generating call graphs. But it is possible to perform any dynamic analysis using DynaPyt. We first develop the Call Graph Analysis using the function pre_call hook provided by DynaPyt. This is then used to instrument the source code of the project. Finally, the test suite of the instrumented project is executed which generates a run-time call graph. The steps of instrumentation and execution of tests is integrated into the access interface.

3.3.2   PyCG

In this work, we use PyCG to generate call graphs for the collection of projects. The test files are provided as the entry point for generating the the call graphs. The required arguments and command to generate the call graphs based on static code analysis is integrated into the access interface.

3.4   Integrating LExecutor

While code analysis provides a usage scenario of our benchmark, we add another usage scenario of collecting data for fine-tuning a neural model. This is done using LExecutor, which provides a command to fine-tune the model and improve its accuracy using the training data. In this work, we first instrument the project which we need to add into the training data. We then run the test suite of the instrumented project, which generates one or more trace files. These trace files are then fed to the LExecutor which provides a command to use the trace files and generate training data. The instrumentation and test suite execution are integrated into the access interface provided by the benchmark.

3.5   Artifact Packaging and Interface

In this section, we first provide the details regarding the access interface we design to provide make the benchmark easily accessible (Section 3.5.1). Then we talk about why we use Docker to package and export our benchmark and how it makes the benchmark extensible (Section 3.5.2).

3.5.1   Access Interface

To provide an easy access to the projects and the analysis frameworks in the benchmark, we create a single command line access interface. Along with easy access, we also aim to make the benchmark ready-to-run and ready-to-analyze with this interface. The access interface provides various alternatives to the end user specified by the command line options. The available options are provided in the table 3.3, where the first column specifies the option and the second column provides

description of its usage. With the options *dynapyt_instrument*, *dynapyt_run*, *lex_instrument*, *lex_test* and *pycg*, one can specify a variable number of projects which makes the benchmark highly versatile.

Option	Description
list	List the project number, name and GitHub URL
test	Run test suite of specified projects
save	Save standard output and error to file specified
timeout	Timeout in seconds for commands
update_dynapyt_source	Clone or update source code of DynaPyt
update_lex_source	Clone or update source code of LExecutor
dynapyt_instrument	Instrument files for DynaPyt
dynapyt_file	Specify file to use for instrumentation of DynaPyt
dynapyt_analysis	Specify the analysis to perform for DynaPyt
dynapyt_run	Execute DynaPyt analysis for specified projects
lex_instrument	Instrument files for LExecutor
lex_file	Specify file to use for instrumentation of LExecutor
lex_test	Execute LExecutor for specified projects
pycg	Execute PyCG for specified projects

Table 3.3: Access Interface (Command Line Options)

3.5.2   Packaging and Export

In order to use the benchmark readily, it should contain all of its constituent parts and must be easy to setup. In this work, we package the benchmark containing the collection of projects, the analysis frameworks, LExecutor and the access interface into a Docker [12] image. This Docker image can be used to create containers to access the benchmark. Since, Docker provides a loosely isolated environment for its containers we install all the operating system related dependencies into the image itself. The Docker image is then exported on the Docker Hub [13] repository. This approach of using Docker image also makes the benchmark long lasting, since the image is available long after the release. Another advantage of using Docker is the simplicity of modifying the benchmark as per the need of the user. For example, one can add another analysis tool or change some projects in the benchmark or add new test cases for some projects. All of these can be done by importing the Docker image and making the changes as per the needs of the developer or a user. This makes our benchmark easily extensible.

3.6   Evaluation Metric

In order to evaluate the benchmark created using our approach, we use the following metrics. Firstly, we evaluate the dynamic nature of the benchmark. To do so, we perform some statistical analysis on the integrated test suites of projects in the benchmark. These include running the test suites to find the running time, number of passed and failed test cases for each project and their combination, where running time refers to executing the entire test suite including failed, passed, skipped, xfailed, xpassed and error cases. Secondly, the efficiency for generating data for neural models is evaluated. This is done by comparing the number of data points collected and

the range of validation accuracy for the model. Data points in the case of LExecutor refers to the unique use-value events which are used by the neural model to train and validate. The accuracy for the model is how correctly the model predicts the missing input values using LExecutor for the validation data. We train the model using the data points we generate and compare the validation accuracy to the original data points. Lastly, we evaluate the effectiveness to provide data for comparison of static and dynamic analysis for call graphs. This is done with the help of some statistical comparison of the call graphs generated using the projects in the benchmark. These comparisons include the number of matching callers and callees in DynaPyt and PyCG. The match refers to the matching string values of callers as keys and callees as values in DynaPyt and PyCG. Furthermore, the percentage match with respect to DynaPyt and PyCG is also calculated.

Overall, these evaluation metrics help us to understand the dynamic nature of the benchmark and its usefulness in usage for generating data to answer some research questions on code analysis.

4 Implementation

In this chapter, we describe the implementation details to create the dynamic benchmark DyPy-Bench based on our approach. We use various tools and libraries to accomplish the goals of ready-to-run, ready-to-analyze, versatile, extensible, diverse and large-scale for our benchmark. In the following sections, we describe the steps we perform and the tools and libraries we use. The source code of this implementation is present on GitHub at `https://github.com/sola-st/master-thesis-piyush-bajaj/tree/automation`.
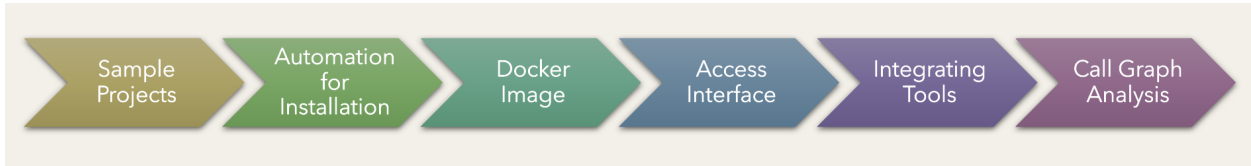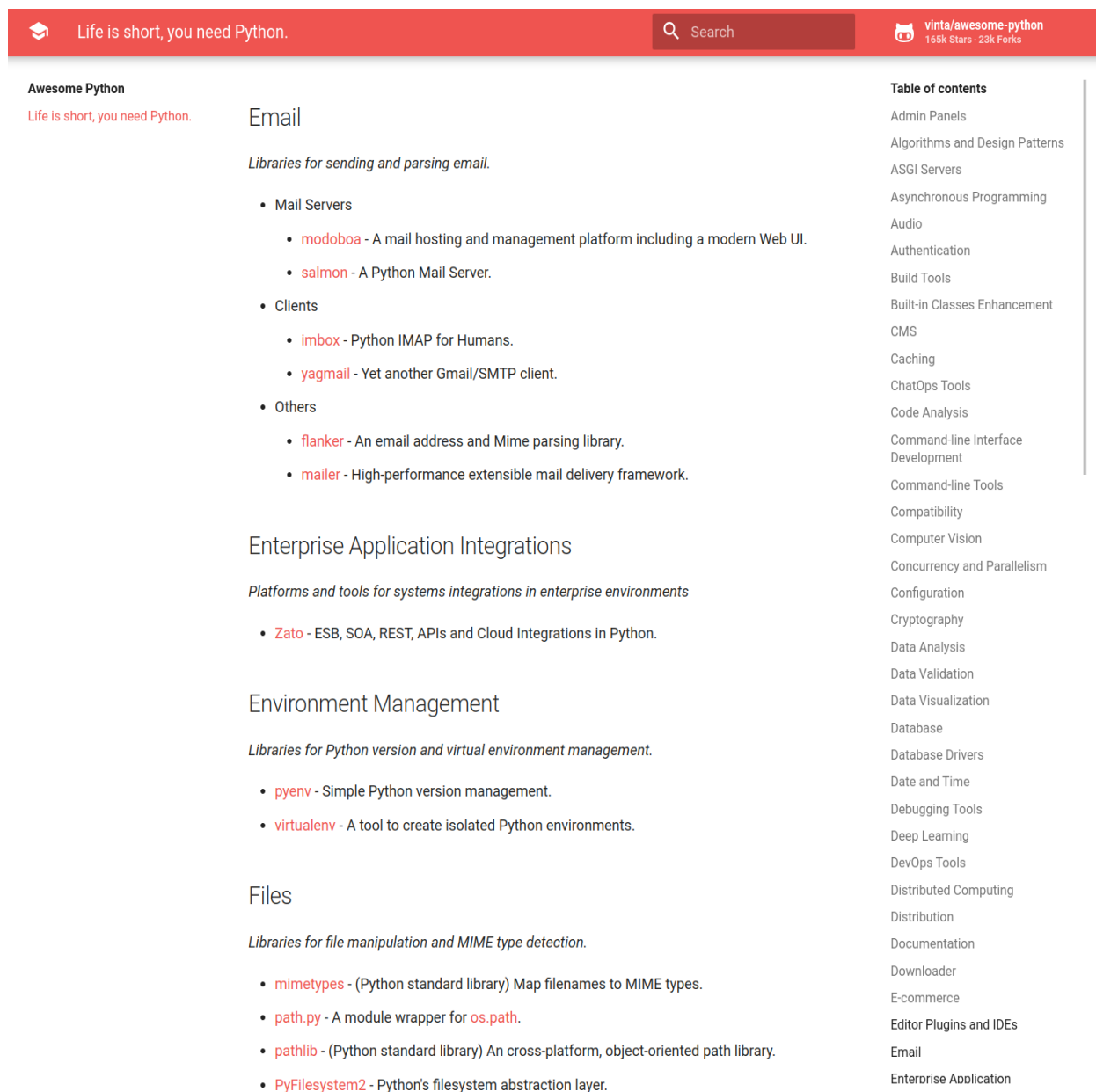

Figure 4.1: Implementation Flow

Figure 4.1 shows the flow of implementation steps for the creation of DyPyBench. We start with installation and setup of sample projects from the awesome-python corpus (Section 4.1). With these sample projects, we create an automation script to gather a list of 50 python projects for our benchmark (Section 4.2). We then create a Docker image for our benchmark, and use the list of projects to install and setup the execution environment automatically (Section 4.3). With the Docker environment ready with the projects, we create a command-line access interface to the benchmark (Section 4.4). We integrate LExecutor, PyCG and DynaPyt to the benchmark and provide options for their usage through the access interface (Section 4.5). Finally, we implement Call Graph Analysis in DynaPyt to generate run-time call graphs (Section 4.6).

4.1 Sample Projects

As mentioned in the approach, we select the projects belonging to the different application domains from a predefined set of 679 projects from the the awesome-python project. We use the table of contents as the application domain for selection. Figure 4.2 shows the awesome-python website with the table of contents and some projects. Source code of these projects is fetched from GitHub repository which is provided by awesome-python or project website linked by awesome-python.

We decide to select 5 sample projects to start our implementation. We start by manually picking a random project from one of the categories in the awesome-python corpus. Then the number of stars for the project repository on GitHub is checked. If the project has less than 500 stars we

Figure 4.2: Awesome Python Website

choose another project randomly, otherwise we proceed with the picked project. This helps us
to ensure the GitHub stars selection criteria mentioned in section 3.1.2. We proceed by cloning
the source code of the project and installing it in a python virtual environment. If the project is
successfully installed, we add the pytest library to the virtual environment. Finally, the test suite
of the project is run using pytest. The project is chosen only if the execution is successful ensuring
the selection criteria of presence and execution of test suite.

All of the above mentioned steps, from selecting a random project to running the test suite
is performed a number of times till we get 5 different projects. Each time a random project is
selected, we ensure that it does not belong to a category which has been chosen before. This helps
us fulfill the selection criteria of diverse domain mentioned in section 3.1.2. Table 4.1 shows a list
the projects which were chosen for installation till 5 of them satisfied all the selection criteria. The
5 projects with **Yes** in the Criteria column were chosen as the sample projects for further tasks for
our benchmark. We installed some operating system libraries and module dependencies for some
of the projects listed in Table 4.1 for them to work properly.

Category	Sub-category	Name	GitHub Stars	Criteria
Web Crawling	-	grab	2.2k	Yes
DevOps Tools	SSH-style Deployment	fabric	13.7k	No
Robotics	-	PythonRobotics	16.8k	Yes
RESTful API	Flask	flask-api	1.3k	Yes
Deep Learning	-	mxnet	20.2k	No
Machine Learning	-	gym	29k	No
Job Scheduler	-	schedule	10.2k	Yes
Image processing	-	pagan	278	No
Image Processing	-	pillow	10.3k	Yes

Table 4.1: First 5 Projects for DyPyBench (Criteria : Test Suite Execution)

4.2   Automation for Installation

The installation and setup of the 5 sample projects helped us in understanding the general steps
required for installing 50 steps. Although, it is possible to install all the 50 projects manually,
however, we need to ensure our selection criteria are met. To do so, we might need to perform
the installation of much more projects than 50. In order to make it easier and quicker to get the
list of 50 projects which satisfy our needs, we automate the process of installation and execution
of projects. We use bash scripts for this since they support loops such as while for repetition and
conditional statements such as if then else for exceptions.

In order to create the bash script, we use the steps performed in sample projects. The Figure
4.3 shows the installation steps for two sample projects we installed before. The blue colored text
indicates similarities, whereas the red colored text indicates the differences. As can be seen from
the Figure 4.3, the two commands of git clone and pytest have different arguments depending on
the project. Furthermore, the project Python Robotics executes an extra command of pip install,
since this project provides us with a requirements.txt file. A bash script can accept command-line

Python Robotics

- git clone
  https://github.com/AtsushiSakai/PythonRobotics.git
- virtualenv vm
- source vm/bin/activate
- pip install .
- pip install -r requirements/requirements.txt
- pip install pytest
- pytest tests/

Flask-Api

- git clone https://github.com/flask-api/flask-api.git
- virtualenv vm
- source vm/bin/activate
- pip install .
- pip install pytest
- pytest flask_api/tests

Figure 4.3: Execution of Projects (Similarities and Differences)

Listing 4.1: Sample Entry github-url.txt

```
1 https://github.com/AtsushiSakai/PythonRobotics.git rt requirements/requirements.
    txt tests
2 https://github.com/flask-api/flask-api.git t flask_api/tests
```

arguments and use these to execute different commands or same command with different arguments. In this work, instead of the command-line arguments we use a text file to provide these arguments to the bash script. We create a github-url.txt file, which contains the arguments that the bash script needs. Each line in github-url.txt contains the arguments for one project. The Listing 4.1 shows the entry of two projects shown in the Figure 4.3 in github-url.txt file.

As seen in the Listing 4.1, each line contains a collection of space separated strings. The first line contains 4 strings in total whereas the second line contains only 3. The first string on both lines is the Git URL of the repository to clone and caters to the difference in the git clone command. The last string on both lines, i.e., 4th on line 1 and 3rd on line 2, is the test directory path from the project root and caters to the pytest command. The second string on both lines represents a flag wot indicate the presence or absence of the requirements file. The possible values for the flag are *rt* or *t*, where the first specifies the presence and second specifies the absence. If the requirement file is present then we specify the path to the requirement file from the project root, as done in line 1 as a third string in the collection. This string caters the difference of the extra **pip install** command described above.

The automation script we create, starts by creating a Project directory if it does not exists. Then we read the github-url.txt file we created before line by line and perform the following steps for each line. First, we split the space separated strings to get the arguments as described before. We then create a numbered sub-folder for the project to be installed and clone the source code from the Git URL into the sub-folder. Inside the sub-folder, we create a python virtual environment *.vm* and activate this environment. We then install the project using the source code. Next, we check if the flag provided is *rt*, if yes, then we install the dependencies using the requirements file. We then check, if the the project needs some extra dependencies to execute test suite and install those if needed. This check is based on the Git URL of the project. We proceed by installing the pytest

library to the virtual environment. Finally, we execute the test suite using pytest. The automation
bash script we create for installation and execution to gather a list 50 projects is provided in Listing
B.3.

With the sample projects, we get 5 entries in github-url.txt. The next step is to include more
projects in this list. For this, we start by picking random projects from awesome-python and check
for GitHub stars selection criteria. If the criteria is met, we add an entry to github-url.txt file as
per the format. We add 10 such entries at a time and run the automation script we created before.
If the installation and execution was successful, we then select the project for our list. We try to
fix issues related to module dependency, if they occur and rerun the test suite for some projects.
If needed, we modify the automation script to install these dependencies during installation. We
repeat the steps of adding 10 projects until we have 50 projects which install and execute properly.
The list of the 50 projects that we use in DyPyBench is provided in A.2.

4.3   Docker Image

With the list of projects and the automation script, we can install and setup our projects for use.
Since we package our benchmark as a Docker image, we create one and install our projects in it.
We use Ubuntu as the base image, since we performed the previous steps in our implementation on
a Ubuntu machine and have the list of system dependencies we would need. Also, we created bash
files which can only run on Linux based systems. We first create a simple image for the benchmark
with the required system dependencies and the source code files of DyPyBench. This image is
created using the Dockerfile provided in Listing 4.2.

Listing 4.2: Dockerfile

```
1  FROM ubuntu:latest

2

3  WORKDIR /DyPyBench

4

5  RUN apt-get update

6

7  RUN apt-get install python3 -yq

8

9  RUN apt install python3-pip -yq

10

11 RUN apt install python3-virtualenv -yq

12

13 RUN apt install git -yq

14

15 RUN apt install nano -yq

16

17 RUN apt install libjpeg8-dev -yq

18
```

```
19 RUN apt-get install ffmpeg libavcodec-extra -yq
20
21 RUN pip install --upgrade pip setuptools wheel
22
23 COPY . .
24
25 RUN chmod -R 777 ./scripts
```

We can see from the Listing 4.2, we install git, nano, python3, python3-pip and python3-virtualenv to our image. We also install libjpeg8-dev, ffmpeg and libavcodec-extra as these are needed by Pillow and pydub projects. After copying the source files into the working directory of */DyPyBench* on line 23, we modify the permissions of the scripts folder to allow execution of bash scripts.

The Docker image created so far contains the source files for DyPyBench, automation scripts and the list of projects in the github-url.txt. We create a container from this image and run a bash script to automatically install and setup the projects from the list. This script performs the same steps as provided by the automation script in Listing B.3 except for two steps. First is the non execution of the test suite and second is the overwriting of test files. Test suite execution is not needed since we in this step we are providing the selected projects for the benchmark and test execution was a selection criteria. We overwrite test files to skip the failed test cases for analysis frameworks to work efficiently. After installing the projects from the list, we create the image from this container and export it on Docker Hub where the image name is dypybench/dypybench and the tag is v1.0. Figure 4.4 shows the created image on the Docker Hub website [1], available as a public repository.
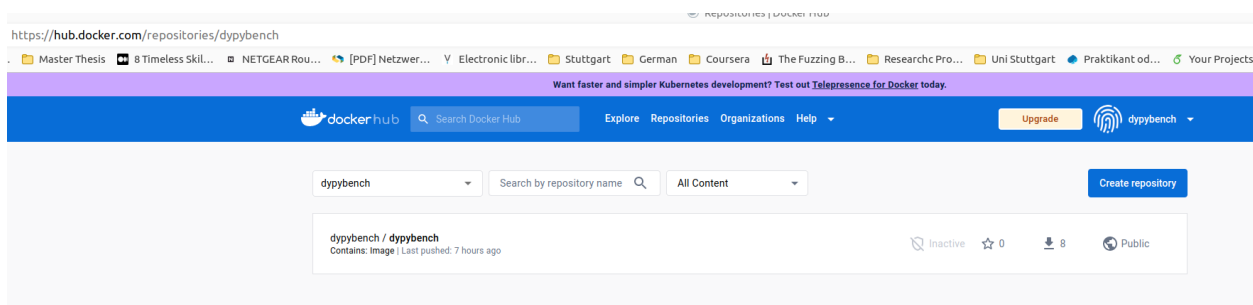
Figure 4.4: Docker Hub

4.4   Access Interface

In order to allow easy access to the collection of projects in the Docker image, we create a single command-line interface with python script. Python scripts are user-friendly and offer extensive support for both standard and third-party libraries that can be utilized for automating tasks, data processing, and analysis. They are also easily extensible, allowing for the addition of new features

---

[1]https://hub.docker.com/

without affecting the previous functionality. These scripts can be triggered from the command-line and also accept command line options.

In this work, we create a python script, dypybench.py, which acts as a single point of entry to trigger the execution of various tasks within the benchmark. The script is executed from the command-line using the command as shown below.

```
python3 dypybench.py <args>
```

The command-line arguments to the script are parsed with the standard Python library of argparse. The script first reads the github-url.txt file and stores some information related to the projects. The information stored is the mapping from project number to name and the collection of strings arguments as described earlier in section 4.2. The initial command-line arguments added to the script are *--list*, *--save*, *--test*, *--test_original* and *--timeout*. The argument *--list*, prints the number, name and Git URL of all the projects in the benchmark. The argument *--save*, copies the output to the file specified. The argument *--test*, executes the test suite of the project number specified to this argument. Essentially, the python script triggers a new process using the subprocess module to execute the bash script provided to run the test suite with the arguments provided in github-url.txt. The argument *--test_original* can be combined with *--test* and specifies the tests to be performed on the original copy of the code. Lastly, the argument *--timeout*, kills the subprocess and any command from the bash script that is still running after the seconds specified to this argument. More arguments are added to this interface when we add the analysis frameworks and LExecutor. Each of those arguments executes a new process to execute a specific bash script. These arguments are explained in their respective sections later. Multiple projects can be specified by space separated numbers in the respective arguments, whereas the project number 0 specifies the task to be run on all the 50 projects. In order to avoid manipulation to the source code, we create a duplicate copy of the project specified and perform the task on this duplicate copy. The code for the script is provided in the Listing B.2. The usage of the script and the command line options are shown in the Figure 4.5.

```
usage: dypybench.py [-h] [--list] [--timeout TIMEOUT] [--test TEST [TEST ...]] [--test_original] [--save SAVE] [--update_dynapyt_source] [--update_lex_source]
                    [--dynapyt_instrument DYNAPYT_INSTRUMENT [DYNAPYT_INSTRUMENT ...]] [--dynapyt_file DYNAPYT_FILE] [--dynapyt_analysis DYNAPYT_ANALYSIS]
                    [--dynapyt_run DYNAPYT_RUN [DYNAPYT_RUN ...]] [--lex_instrument LEX_INSTRUMENT [LEX_INSTRUMENT ...]] [--lex_file LEX_FILE]
                    [--lex_test LEX_TEST [LEX_TEST ...]] [--pycg PYCG [PYCG ...]]

options:
  -h, --help            show this help message and exit
  --list, -l            List all the projects DyPyBench supports
  --timeout TIMEOUT     Specify timeout to be used in seconds for execution of subprocesses
  --test TEST [TEST ...], -t TEST [TEST ...]
                        Specify the project number to run the test suite
  --test_original, -to  Run tests on original code
  --save SAVE, -s SAVE  Specify file name to save the stdout and stderr combined
  --update_dynapyt_source
                        get dynapyt source code
  --update_lex_source   get LExecutor source code
  --dynapyt_instrument DYNAPYT_INSTRUMENT [DYNAPYT_INSTRUMENT ...], -di DYNAPYT_INSTRUMENT [DYNAPYT_INSTRUMENT ...]
                        Specify the project no. to run DynaPyt instrumentation
  --dynapyt_file DYNAPYT_FILE, -df DYNAPYT_FILE
                        Specify the path to file containing the includes.txt file to run the instrumentation
  --dynapyt_analysis DYNAPYT_ANALYSIS, -da DYNAPYT_ANALYSIS
                        Specify DynaPyt analysis to run
  --dynapyt_run DYNAPYT_RUN [DYNAPYT_RUN ...], -dr DYNAPYT_RUN [DYNAPYT_RUN ...]
                        Specify the project no. to run DynaPyt Analysis
  --lex_instrument LEX_INSTRUMENT [LEX_INSTRUMENT ...], -li LEX_INSTRUMENT [LEX_INSTRUMENT ...]
                        Specify the project no. to run LExecutor instrumentation
  --lex_file LEX_FILE, -lf LEX_FILE
                        Specify the path to file containing the includes.txt file to run the instrumentation
  --lex_test LEX_TEST [LEX_TEST ...], -lt LEX_TEST [LEX_TEST ...]
                        Specify the project no. to run test suite for LExecutor
  --pycg PYCG [PYCG ...], -scg PYCG [PYCG ...]
                        Specify the project no. to run PyCG for static call graph generation
```

Figure 4.5: Command Line Options

4.5   Integrating Code Analysis and Neural Network Tools

With the access interface to the benchmark, we integrate the neural network tool LExecutor and
the code analysis tools PyCG and DynaPyt. These tools need projects which do not have failed
test cases in order to run efficiently. However, the some of the projects that we installed in the
benchmark have test cases that fail. To handle this challenge, we skip the test cases which fail
using a pytest marker. In some cases we need to skip entire files, since all the test cases in the
file fails. The two types of pytest markers are as shown in the Listing 4.3. The marker shown on
line 1 skips the entire file from execution, whereas line 3 shows the marker that skips only the test
function.

Listing 4.3: Skip Test Case using Pytest Marker

```
1 pytestmark = pytest.mark.skip("skip for dypybench")
2
3 @pytest.mark.skip("skip for dypybench")
```

We create a copy of the files for which we add pytest marks and replace the files in the original
project directory using a python script provided in the utility. We also add this step of replacing
files to the automation script of installation for future installations.

We use the source code of LExecutor and DynaPyt in our benchmark. To access interface
provides the arguments of *--udapte_lex_source* and *--update_dynapyt_source* to clone or update the
source code of LExecutor and DynaPyt respectively from Git. The following sections give more im-
plementation details on the integration of LExecutor, PyCG and DynaPyt and the access interface
arguments for them.

4.5.1   LExecutor

We clone the source code and create a python virtual environment with the dependencies from the
bash script when the argument *--update_lex_source* is run from the access interface. Since LExecutor
works in two steps to generate traces files, we provide two bash scripts one for each step. The first
script instruments the source code of the project and the second script generates trace files by
executing the project. The instrumentation is not necessarily done on all the source files, but
rather the files that run during execution. In this work, we provide the files to be instrumented
inside a text file as a list. The text file contains a list of two space separated values, the first
specifies the name of the project and the second specifies the path of the file to be instrumented.
We also provide a python script to generate this text file, by parsing all the projects and filtering
out the source files. Certain files are excluded from this text file to avoid issues with running test
suite. However, this text file can be created by any means in the required format. The Listing
4.4 shows sample entries from the above mentioned text file. This text file is specified to *--lex_file*
argument from the access interface.

Listing 4.4: lex_instrument_all.txt

```
1 grab ./temp/project1/tests/test_grab_pickle.py
2 grab ./temp/project1/tests/test_spider_queue.py
3 grab ./temp/project1/tests/test_ext_rex.py
4 grab ./temp/project1/tests/test_grab_cookies.py
5 errbot ./temp/project18/errbot/core_plugins/chatRoom.py
6 errbot ./temp/project18/errbot/core_plugins/backup.py
7 errbot ./temp/project18/errbot/core_plugins/acls.py
```

The argument *--lex_instrument* triggers the first bash script to instrument the files specified in the above text file for the specified project. This script executes the command shown in Listing 4.5. The parameter ${@:4} is replaced by the space separated file paths.

Listing 4.5: LExecutor Instrumentation

```
1 python -m lexecutor.Instrument --files ${@:4} --iids /DyPyBench/iids.json --
    validate
```

The second bash script mentioned above is triggered by the option *--lex_test*, which simply runs the test suite of the specified project using command shown in Listing 4.6. The parameter $3 is replaced by the test suite directory specified in github-url.txt.

Listing 4.6: LExecutor Test Suite Execution

```
1 pytest $3
```

The instrumentation also generates a iids.json file which is used by the LExecutor to convert the traces files into training data for the neural model. We use the various utility functions provided in the benchmark for gathering trace files, inspecting the output of trace files and training logs of LExecutor.

4.5.2 PyCG

The argument *--pycg* from the access interface triggers the execution of a bash script which first installs the pycg package to the virtual environment of the specified project. Then the script triggers the PyCG commands as shown in the Listing 4.7. Only one of the two commands is executed based on the if condition shown on line 1, which checks if the test suite is a directory or a file. The command on line 6 specifies all the .py files in the test suite directory as the entry point for the call graphs.

Listing 4.7: PyCG Execution

```
1 if [[ $4 == "file" ]]
2 then
3     pycg --package ./project$2 project$2/$3 -o pycg_$2.json
4 elif [[ $4 == "folder" ]]
5 then
```

```
6    pycg --package ./project$2 $(find project$2/$3 -type f -name "*.py") -o
        pycg_$2.json
7 fi
```

As we can see from the Listing 4.7, the output file is generated for each project in the project folder. Utility functions provided in the benchmark are used to collect these JSON output files for further analysis.

4.5.3 DynaPyt

The argument *--update_dynapyt_source* from the access interface, executes a bash script to clone the source code of DynaPyt. Similar to LExecutor, DynaPyt works in two steps,i.e., instrumentation and execution. Each of these steps trigger separate bash scripts. DynaPyt uses different modules to instrument directory and files. We need to specify the files and folders which are to be instrumented. Similar to LExecutor, we provide a text file which contains the list of directories and files which we want to instrument. However, the format of the file is different compared to LExecutor. This text file, has 3 space separated values on each line. The first value is the project name, whereas the second value is the flag indicating file (f) or directory (d). The third value in the space separated entry is the path of directory or file, relative to the root of the project. Listing 4.8 shows some sample entries from this file. This text file is specified to *--dynapyt_file* argument from the access interface.

Listing 4.8: dynapyt_instrument_all.txt

```
1 flask-api d ./flask_api
2 schedule d ./schedule
3 schedule f ./test_schedule.py
4 Pillow d ./src
5 Pillow d ./Tests
6 supervisor d ./supervisor
7 streamparse d ./streamparse
```

In this work, we provide two such text files are provided for instrumentation, one which includes all the source files and the test files and the other which excludes test files. To instrument the code, the first bash script first copies the DynaPyt source code that was cloned to the specific project directory. Then, DynaPyt is installed with its dependencies into the virtual environment of the specific project. Finally, the bash script runs the command to instrument the files or folder based on the flag provided by the above mentioned text file. The instrumentation is based on the hooks provided by DynaPyt for the particular analysis. Therefore, we need to provide the name of the analysis to be performed. We provide this with the argument *--dynapyt_analysis* to the access interface. The two instrumentation commands are shown in the Listing 4.9.

Listing 4.9: DynaPyt Instrumentation

```
1 if [[ $5 == "d" ]]
```

```
2 then
3     #run instrumentation on the given directory
4     python -m dynapyt.run_instrumentation --directory $3 --analysis $4
5 elif [[ $5 == "f" ]]
6 then
7     #run instrumentation on the given file
8     python -m dynapyt.instrument.instrument --files $3 --analysis $4
9 fi
```

The execution of test suite to perform the dynamic analysis is done using another bash script. This script, first create an entry file to run all the tests using pytest module with the parameter import-mode set to importlib. The Listing 4.10 shows a sample of the entry file created above. We then trigger the execution of tests with *run_analysis* module provided by dynapyt, specifiying the above entry file and the name of the analysis. Listing 4.11 shows the creation of the entry file on line 1 and execution with run_analysis on line 3.

Listing 4.10: DynaPyt Execution Entry File

```
1 import pytest
2
3 pytest.main(['--import-mode=importlib','/home/piyush/Documents/MT/CallGraph/flask
      -api/flask_api/tests/'])
```

Listing 4.11: DynaPyt Test Suite Execution

```
1 printf "import␣pytest\n\npytest.main(['--import-mode=importlib',␣'$ROOT_DIR/temp/
      project$2/$4'])\n" > run_all_tests.py
2
3 python -m dynapyt.run_analysis --entry ./run_all_tests.py --analysis $3
```

4.6   Call Graph Analysis

In this work, we developed a new analysis in DynaPyt to generate call graphs during run-time. This analysis is named CallGraph and we use the output of this analysis for comparison with PyCG. We use the function pre_call hook provided by DynaPyt and create the call graph in the form of dictionary. A key in this dictionary is the fully qualified names of the caller method and the value for this key is a list of fully qualified names of called methods. At the end of the execution, i.e., after the all the test cases are run we output the dictionary to a JSON file. A python utility script provide by the benchmark collects all the JSON files from the projects that were dynamically analyzed. Listing 4.12 shows the code of the pre_call hook implemented for the analysis. The complete code for the analysis is shown in the Listing B.1.

Listing 4.12: Function Pre-Call Hook in Call Graph Analysis

```python
def pre_call(self, dyn_ast: str, iid: int, function: Callable, pos_args: Tuple
    , kw_args: Dict):
    ast, iids = self._get_ast(dyn_ast)
    module = getmodule(function)
    module = str(module).split('␣')[1] if module is not None else "''"
    # calling function
    caller = get_parent_by_type(ast, iids.iid_to_location[iid], m.FunctionDef
        ())
    # called function
    if hasattr(function, "__qualname__"):
        callee = module[1:-1] + '.' + function.__qualname__ if module != "''"
            else function.__qualname__
    else:
        temp = str(function)
        callee = temp

    #file name
    key = dyn_ast.replace('.py.orig', '').replace('/','.')

    if caller is None:
        f = key
    else:
        # if caller is a part of class, find the class name
        caller_parent = get_parent_by_type(ast, iids.iid_to_location[iid], m.
            ClassDef())
        if caller_parent is None:
            f = key + '.' + caller.name.value
        else:
            f = key + '.' + caller_parent.name.value + '.' + caller.name.value

    # if caller already added
    if f in self.graph.keys():
        temp = self.graph[f]
        # filter dupilcate callees
        if callee not in temp:
            temp.append(callee)
            self.graph[f] = temp
    else:
        self.graph[f] = [callee]
```

5 Experimental Setup

In this chapter, we explain in detail about the experimental setup of DyPyBench for the different usage scenarios.

DyPyBench consists of 50 open-source python projects from different application domains list in Table A.2. We download the source code of these projects along with their test suite into a Docker image which contains the neural model (LExecutor) and the static (PyCG) and dynamic (DynaPyt) analysis frameworks. We use the integrated test suites to execute the projects and collect various statistics and data. In the below sections, we explain the setup for LExecutor, PyCG and DynaPyt.

5.1 LExecutor

We use DyPyBench to generate the training data for the neural model of LExecutor. LExecutor instruments the files and then generates traces from these files during run-time. LExecutor provides its own instrumenter, and the accepts mandatory two arguments to perform instrumentation. The first is a JSON file to store the mapping of iids and the second is files to instrument. To generate the trace files, we execute the test suite consisting of the instrumented files. The trace files from the above step are used to generate training data in the form of .pt files for the neural model. For this task, LExecutor provides a module PrepareData which accepts the trace files and the JSON file created in the instrumentation step. To generate the .pt file, LExecutor works in two modes of abstraction, fine-grained and coarse-grained. This can be set in the Hyperparameters.py file. Fine-grained abstraction distinguishes between different values of a type, such as positive, negative and zero in numbers and empty and non-empty for lists, string etc. Coarse-grained abstraction on the other-hand maps all the values of a type to a single class such as int, string, list etc. [31] The Listing 5.1 shows the possible entries for the abstraction mode. In this work, we generate the data using the two abstraction modes shown on line 1 and line 2 in the Listing 5.1. The PrepareData module also generates a validation file in the .pt format. The trace entries are split into training and validation data based on the split provided in the Hyperparameters.py file as shown in Listing 5.1 on line 3.

Listing 5.1: Abstraction Modes in LExecutor

```
1    value_abstraction = "fine-grained"
2    value_abstraction = "coarse-grained-deterministic"
3    perc_train = 0.95
```

5.2   PyCG

PyCG generates call graph based on static analysis of code. The call graphs is created in the JSON format, where the key is caller and the value is a list of called functions. The Listing 5.2 provides a sample entry from the PyCG.

Listing 5.2: Key-Value Pair in PyCG

```
1  {
2      "tests.test_autoconfig.test_autoconfig_env": [
3          "decouple.AutoConfig",
4          "os.path.join",
5          "os.path.dirname",
6          "mock.patch.object"
7      ],
8      "decouple.AutoConfig": [],
9      "os.path.dirname": [],
10     "os.path.join": [],
11     "tests.test_autoconfig.test_autoconfig_ini": [
12         "decouple.AutoConfig",
13         "os.path.join",
14         "os.path.dirname",
15         "mock.patch.object"
16     ]
17 }
```

As we can see from the Listing 5.2, the structure of the key is *folder.file.class.method*. The folder structure for the file is present from the root of the project. While the other elements are filled in based on the structure of the caller method. For example, as shown on line 2 in Listing 5.2 the caller is a function *test_autoconfig_env* in the file *test_autoconfig* present in the folder *tests*. The structure of the values or callees is *module.function*. As seen in the Listing 5.2 on line 4, the called function is *join* of the module *os.path*. PyCG accepts 3 arguments, the package contains the source files, the entry point files and the output file save the JSON. In DyPyBench, we provide the package as source code cloned from Git and the test files as entry point files for each project. The execution of the PyCG module then provides us with the call graph in the mentioned JSON format.

5.3   DynaPyt

DynaPyt instruments the files and the execution of the instrumented code provides us with the dynamic analysis. DynaPyt, has its own modules for instrumentation and analysis. We first use the instrumentation module to instrument the source files. To run the analysis, we create a entry file to run all the test files using the main function provided by the pytest module. Additionally,

we provide the option of import-mode set to importlib to the pytest.main function. We add the
Call Graph analysis is DynaPyt with the function pre_call hook, and specify this analysis to the
analysis module of DynaPyt. The analysis generates the output in the JSON format, with the key
as caller and the value as a list of called functions. The JSON is stored in a file at the end of the
execution of the analysis as provided in the Listing B.1. The Listing 5.3 shows the sample JSON
generated from DynaPyt.

Listing 5.3: Key-Value Pair in DynaPyt

```
1  {
2      ".DyPyBench.temp.project22.decouple": [
3          "decouple.Undefined",
4          "collections.OrderedDict",
5          "decouple.AutoConfig",
6          "configparser.RawConfigParser.read_file"
7      ],
8      ".DyPyBench.temp.project22.tests.test_autoconfig.test_autoconfig_env": [
9          "decouple.AutoConfig",
10         "posixpath.dirname",
11         "posixpath.join",
12         "mock.mock._patch_object",
13         "<decouple.AutoConfig object at 0x7f9479012f20>"
14     ],
15     ".DyPyBench.temp.project22.tests.test_autoconfig.test_autoconfig_ini_in_subdir
        ": [
16         "decouple.AutoConfig",
17         "posixpath.dirname",
18         "posixpath.join",
19         "posix.chdir",
20         "mock.mock._patch_object",
21         "<decouple.AutoConfig object at 0x7f947859fa60>"
22     ]
23 }
```

As can be seen from the Listing 5.3, the structure of the key is *folder.file.class.method*. The
folder element represents the file path from the root of DyPyBench. While the other elements
are present based on the structure of the caller method. For example, as shown on line 8 in
Listing 5.3 the caller is a function *test_autoconfig_env* in the file *test_autoconfig* present in the folder
/DyPyBench/temp/project22/tests. The structure of the values or callees is *module.function*. As
seen in the Listing 5.2 on line 4, the called function is *OrderedDict* of the module *collections*.
Similarly, on line 11 the module is *posixpath* and the method is *join*. The callee on line 13 and
21 represents a case where the called function is *AutoConfig* of the module *decouple* with different
objects indicated by the hex number at the end. These are essentially the same callees as provided

on line 9 and 16, however, the dynamic nature of generation of the call graph provides such results.

6 Evaluation

In this chapter we discuss the experimental evaluation of DyPyBench. We do so by answering the following questions:

RQ1: How do the included test suites in Python projects within DyPyBench contribute to a dynamic benchmark?

RQ2: How efficient is our benchmark to generate (valid and good) training data for the neural models?

RQ3: How effective is DyPyBench in providing data to aid a comparison of static and dynamic call graph?

6.1 RQ1: Effectiveness of Test Suite

We find that DyPyBench provides 45086 test cases and the running time of test suites is between 0.02 seconds and 1362.86 seconds. Overall, 41451 test cases pass where as 429 fail. The successful execution of such a high number of test cases makes our benchmark dynamic in nature. The number of projects which have 0 failed test cases in DyPyBench is 31, whereas 13 others have failed tests between 1 and 10. The number of test cases in each project varies from 1 to 10552. Figure 6.1 shows the percentage of passed test cases in all projects of DyPyBench. We see that 41 projects have more than 90% passed test cases, while 46 have 80% or more passed test cases.
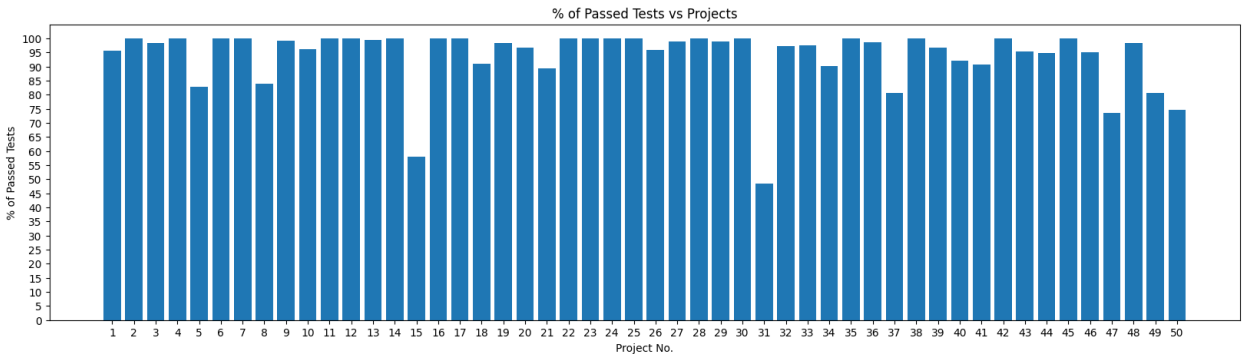
Figure 6.1: Percentage of Passed Tests in DyPyBench Projects

Since the execution of the project is important for dynamic analysis, the high pass percentage of 91.93% for all tests cases is an advantage for efficient analysis. The running time for each project is shown in Table 6.2. The total run time for all the projects in DyPyBench is 3568.86 seconds.

There are 39 projects which run the test suite within 50 seconds, whereas 4 projects have a run time greater than 200 seconds.

Project	1	2	3	4	5	6	7	8	9	10
No. Tests	181	104	63	38	3950	1375	57	50	353	598
Run Time (s)	242.8	365.45	0.46	0.31	41.4	11.52	0.77	0.37	101.06	1.76
Project	11	12	13	14	15	16	17	18	19	20
No. Tests	1	65	457	113	250	395	243	213	251	1949
Run Time (s)	0.02	0.2	54.84	22.47	0.6	1.92	18.71	62.24	36.32	7.47
Project	21	22	23	24	25	26	27	28	29	30
No. Tests	1142	63	1833	34	3	1971	2680	340	207	290
Run Time (s)	28.35	0.21	16.66	2.87	2.64	362.66	46.59	1.14	0.61	0.26
Project	31	32	33	34	35	36	37	38	39	40
No. Tests	796	76	593	10552	28	80	412	1182	622	1132
Run Time (s)	3.61	5.81	71.64	1362.86	0.12	29.81	1.32	3.26	179.45	29.71
Project	41	42	43	44	45	46	47	48	49	50
No. Tests	2824	842	3400	310	77	1086	345	185	1051	224
Run Time (s)	144.18	3.52	197.77	1.07	9.65	46.67	3.23	3.76	34.76	4.08

Figure 6.2: Test Suite Run Time in DyPyBench Projects

Overall, we find that the included test suites makes it easier to run the projects which is an important requirement to make a benchmark dynamic. The successful execution of a high number of test cases and the lower running time makes DyPyBench, a dynamic benchmark readily available for use.

6.2 RQ2: Effectiveness for Neural Model

DyPyBench contributes 547,830 new data points for LExecutor. This results in the model accuracy between 71.86% and 93.67% after training the model.

The new data points we found are generated from 37 projects in our benchmark and amount to nearly 2.5 times the data points used by LExecutor. The Figure 6.3 shows the data points from each project labelled with project number. As can be seen from the Figure 6.3, the distribution of these data points vary from project to project. This can be attributed to the number of files in each project that are instrumented.
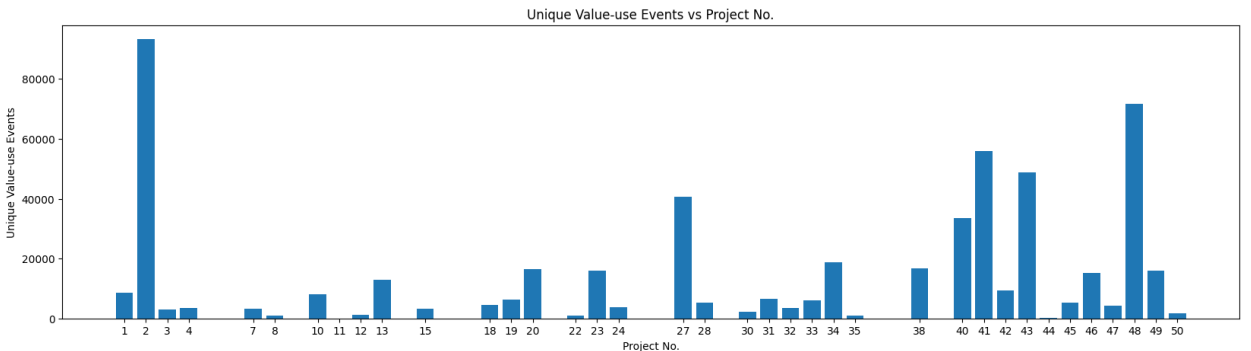
Figure 6.3: Unique Value-use Events in DyPyBench Projects

To evaluate the accuracy of the neural model based on the new data points, we perform a total of 6 experiments using different validation data and the abstraction modes. Overall, we use

3 different validation data-sets. First is a random split of 5% from the data points we found with 37 projects in DyPyBench, second is the data-set provided by the LExecutor. This data-set has data points which were not for training the model. Finally, the third data-set is the split from the new data points by projects. 6 projects were selected at random, which constitute nearly 5% of the data points and these projects were excluded from training data-set. For each of the validation data-set we perform 2 experiments, one with the abstraction mode as fine-grained while the other with coarse-grained as described before.

(a) Fine-grained   (b) Coarse-grained
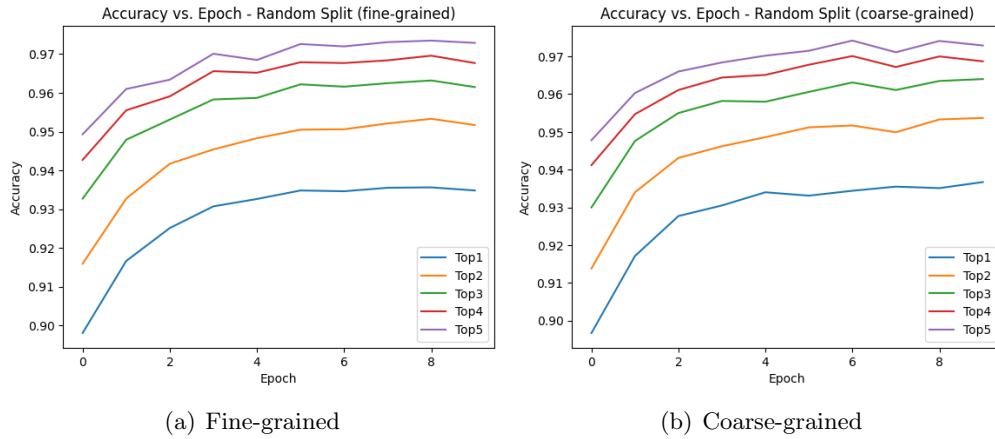
Figure 6.4: Accuracy vs Epoch (Random Split)

Figure 6.4 shows how the accuracy of the model varies with the number of epochs when the first validation data-set mentioned above is used. Each line in the Figure 6.4 shows the accuracy for top-n predictions for the same input variable. As can be seen, the accuracy for top-5 is the best followed by top-4 and so on. We mainly consider the accuracy of the first prediction, which starts at nearly 90% and improves up to 93% for both fine and coarse grained.

(a) Fine-grained   (b) Coarse-grained
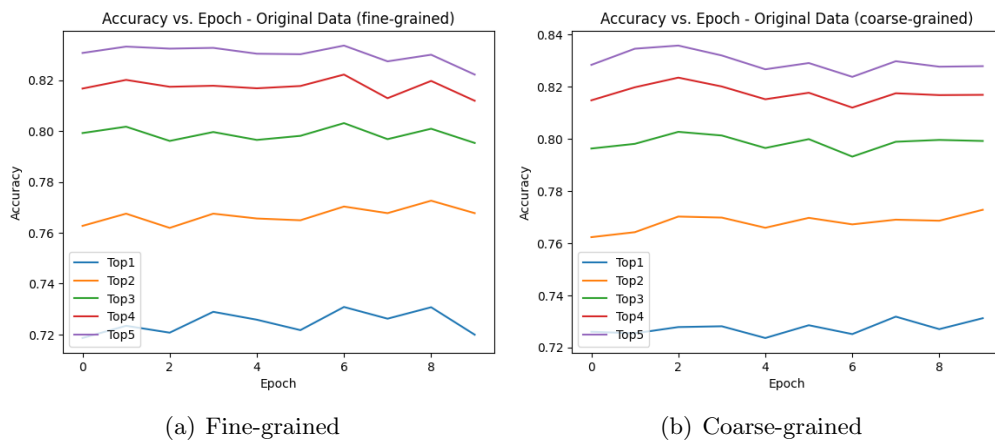
Figure 6.5: Accuracy vs Epoch (Original Data)

Similarly, Figure 6.5 shows the accuracy versus epoch for validation data-set used by the LExecutor. As we can see in the Figure 6.5, the accuracy is lower than the previous experiments plunging down to 73% in both cases of abstraction. Finally, the Figure 6.6 shows the accuracy

versus epoch comparison for the third validation data-set mentioned above. In this case, the accuracy ranges between the previous two experiments having the value of 85% for both levels of abstraction. For the first and second validation data-set, we get the best model in the 7th and 8th epoch. However, for the third data-set the best model is found at the 4th epoch itself. Since, the neural model we start with is a pre-trained model, the accuracy range we found is similar to the one specified by LExecutor.

(a) Fine-grained   (b) Coarse-grained
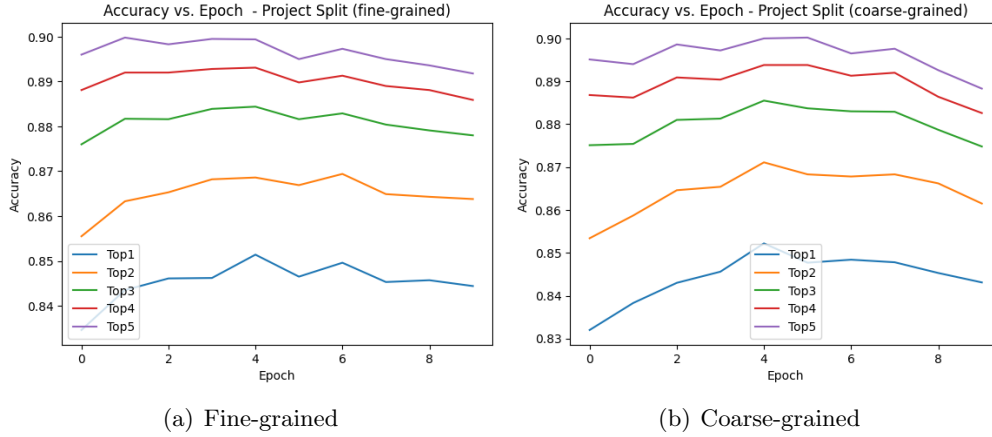
Figure 6.6: Accuracy vs Epoch (Project Split)

The quantity and quality of data is an important aspect for training neural network models. The higher quantity does not guarantee the quality of data. DyPyBench, provides projects which generates a large quantity of data and we evaluate the quality by calculating the accuracy. We find DyPyBench to be a good and valid data generator for LExecutor based on the amount of data generated and the accuracy maintained by the model.

6.3 RQ3: Effectiveness for Analysis Frameworks

We find that DyPyBench provides 10799 key data points from DynaPyt and PyCG combined for comparison of call graphs. These key data points refer to the callers and are generated from 20 projects in DyPyBench. From the above, 6060 data points belongs to PyCG, and 4739 are from PyCG. With a manual verification of these data points, we find that PyCG provides a key for every entry in the list of callees, resulting in a higher number of keys in PyCG output. However, when we compare the key data points in DynaPyt with PyCG there are no matches found. Manually checking the this provides us the insight that the keys in PyCG have starting point as the root of the project while those in DynaPyt start from the root of the benchmark. We fix this issue, by replacing the keys in DynaPyt with the appropriate values. After the above fix, we get 3147 key data points present in both. 66.41% of keys in DynaPyt match with PyCG, while 51.93% in PyCG are present in DynaPyt. Figure 6.7, shows the distribution of the keys amongst DynaPyt and PyCG.
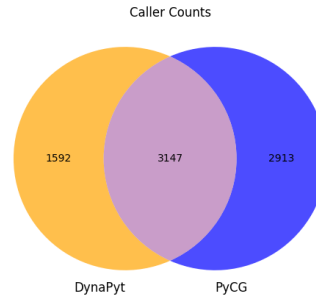
Figure 6.7: Distribution of Callers

Next, we compare the number of value data points, which represent the callees. We only compare the callees which are a part of the matching callers or keys. There are a total of 16493 value data points provided by DyPyBench for comparison, out of which 10318 belong to DynaPyt and 6175 to PyCG.The high number of values from DynaPyt is due to multiple callee entries of the same function, but from a different object. PyCG cannot detect callees based on objects since, it generates the call graphs from static analysis. Another factor that we saw from the manual inspection was the callee entries in DynaPyt for some private functions which PyCG is not able to provide. A direct string comparison of the value data points in DynaPyt with PyCG shows a match of 1691 values. This match contributes to 16.38% match from DynaPyt and 27.38% from PyCG. With further manual verification, we find that certain data points essentially representing the same values do not match with a direct string comparison. Figure 6.8 shows the differences of string in value data points from DynaPyt and PyCG.

Key: flask_api.tests.test_mediatypes.MediaTypeParsingTests.test_media_type_with_params

DynaPyt: ['flask_api.mediatypes.MediaType', 'builtins.str', 'unittest.case.TestCase.assertEqual', 'builtins.repr']

PyCG: ['flask_api.mediatypes.MediaType', 'unittest.TestCase.assertEqual', '<builtin>.str', '<builtin>.repr']

—-

Key: flask_api.tests.test_request.MediaTypeParsingTests.test_json_request

DynaPyt: ['_io.BytesIO', 'flask.app.Flask.test_request_context', 'unittest.case.TestCase.assertEqual']

PyCG: ['unittest.TestCase.assertEqual', 'io.BytesIO']

—

Key: flask_api.tests.test_app.custom_exception

DynaPyt: ['flask_api.tests.test_app.InvalidUsage']

PyCG: ['flask_api.tests.test_app.InvalidUsage.__init__']

—

Key: "tests.test_autoconfig.test_autoconfig_ini_in_subdir"

DynaPyt: ["decouple.AutoConfig", "posixpath.dirname", "posixpath.join", "posix.chdir", "mock.mock._patch_object", "<decouple.AutoConfig object at 0x7f947859fa60>"]

PyCG: ["os.path.dirname", "os.chdir", "decouple.AutoConfig", "os.path.join", "mock.patch.object"]

—

Figure 6.8: Unmatched Same Callees

Since the callees shown above are essentially the same, we include these in our comparison. The number of matches in the value data points is now increased to 2938, which significantly increases the percentage contribution of PyCG matches to 47.58% and DynaPyt to 28.47%. 10182 As mentioned before, DynaPyt contains multiple entries for callees with different objects, we remove duplicate entries of some of callees with similar patterns. Firstly, we replace the pattern `<decouple.AutoConfig object at 0x7f94786106a0>` by *decouple.AutoConfig* and pop the duplicate entries of `decouple.AutoConfig`. Second, we replace the pattern

`BoundMethodWeakref(<tests.test_saferef._Sample1 object at 0x7f73338edb10>.x)` by

`BoundMethodWeakref(<tests.test_saferef._Sample1>.x)` and pop its duplicate entries. The above modification of values data points leads to decrease in total data points to 16358 which is a decrease of 136 callees from DynaPyt. This results in a slight increase in the matching percentage of callees from DynaPyt to 28.85%. Figure 6.9 shows the distribution of the values amongst DynaPyt and PyCG for the matched keys.
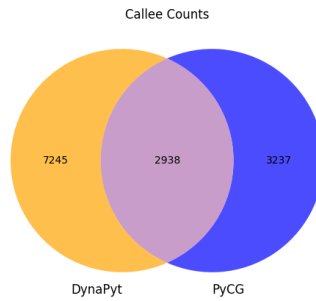
Figure 6.9: Distribution of Callees

Static and Dynamic Analysis are important tools for code analysis. A comparison of the two provides us an indication of how efficient is one over the other. DyPyBench helps with a comparison of call graphs generated using static and dynamic analysis. Although, DyPyBench provides comparable data from both approaches for call graphs, there are still certain errors present which would futher increase the percentage of matching callees and callers.

7 Related Work

In this chapter, we outline some of the work done on benchmarks. We also discuss some of the work done on dynamic analysis.

7.1 Benchmarks

Dynamic benchmarks, such as the DaCapo benchmark suite, provide a more comprehensive evaluation of software systems by taking into account the interactions between architecture, compiler, virtual machine, memory management, and application. These benchmarks evaluate the performance of software systems under more realistic conditions, allowing researchers and developers to better understand how their systems will perform in the real world. [8] The SPEC C++ benchmarks, including SPEC CPU2006 C++, SPEC OMP2012, and SPEC MPI2007 are designed to evaluate the performance of C++ compilers, libraries, and hardware platforms. These benchmarks are valuable for developers and researchers seeking to optimize the performance of C++ programs, as they provide insights into the performance characteristics of their code and can be used to compare the performance of different C++ compilers and libraries on different hardware platforms. [36, 21, 20] While, DaCapo and SPEC C++ Benchmark provide a dynamic benchmark for Java and C++, Python does not have one. DyPyBench, overcomes this ba providing a dynamic benchmark for Python.

7.2 Dynamic Analysis

The analysis of properties of a running program is known as dynamic analysis.[5] Dynamic analysis can be particularly useful in complex software systems, where issues may be difficult to identify or diagnose through manual testing alone. [35] TaintCheck proposes dynamic taint analysis for automatic detection and analysis of software vulnerabilities. The approach is to performs binary rewriting at run time and reliably detects most types of exploits without false positives.[22] Branch Coverage analysis counts program branch execution and outcome frequency with a single API hook. Results are stored in a dictionary mapping branch location and condition value to event occurrences. The hierarchy captures all branching events, enabling effective analysis with minimal implementation.[14] DyPyBench provides a call graph analysis to generate call graphs during runtime for Python programs.

8 Conclusion

This work highlights the importance of dynamic benchmarks and identifies the lack of such benchmarks in Python. To address this gap, we present DyPyBench, a dynamic benchmark comprising 50 diverse and readily usable projects. Our benchmark is ready-to-run, extensible, and ready-to-analyze for researchers and developers.

To evaluate the effectiveness of DyPyBench, we use various tools such as LExecutor, DynaPyt, and PyCG. Our benchmark provides 41,451 successful test cases, which generate 547,830 data points for LExecutor. The accuracy of the neural model for predicting input values to execute arbitrary code snippets ranges between 71.86% and 93.67%. Additionally, DyPyBench generates data points for comparing static and dynamic call graph analysis using DynaPyt and PyCG. We achieve matching percentages of 51.93% and 66.41% for callers and 47.58% and 28.58% for callees in PyCG and DynaPyt, respectively.

The key insight of our is that DyPyBench is a versatile benchmark framework that can be used with various code analysis tools. By providing a readily available Docker image with integrated analysis tools, we make it easier for researchers and developers to use DyPyBench.

We envision DyPyBench as a benchmark framework for dynamic analysis of Python projects that can generate valuable data to aid research in code analysis tasks.

A Tables

Table A.1: Awesome Python Categories

Category	Number of Projects
Admin Panels	9
Algorithms and Design Patterns	
Algorithms	4
Design Patterns	3
ASGI Servers	2
Asynchronous Programming	2
Audio	
Audio	9
Metadata	4
Authentication	
OAuth	7
JWT	3
Build Tools	5
Built-in Class Enhancement	5
CMS	8
Caching	7
ChatOps Tools	1
Code Analysis	
Code Analysis	5
Code Linters	4
Code Formatters	3
Static Type Checkers	3
Static Type Annotation Generators	3
Command Line Interface Development	
Command Line Application Development	6
Terminal Rendering	6

Table A.1: Awesome Python Categories (...continued)

Category	Number of Projects
Command Line Tools	
Productivity tools	10
CLI Enhancement	7
Compatibility	3
Computer Vision	7
Concurrency and Parallelism	6
Configuration	5
Cryptography	4
Data Analysis	6
Data Validation	7
Data Visualization	13
Database	3
Database Drivers	
MySQL	2
PostgreSQL	2
SQlite	2
Other Relational Databases	2
NoSQL Databases	6
Asynchronous Clients	1
Date and Time	10
Debugging Tools	
pdb-like Debugger	4
Tracing	4
Profiler	5
Others	5
Deep Learning	7
DevOps Tools	
Configuration Management	5
SSH Style Deployment	3
Process Management	2
Monitoring	1
Backup	1
Others	1
Distributed Computing	
Batch Processing	5
Stream Processing	2
Distribution	8

Table A.1: Awesome Python Categories (...continued)

Category	Number of Projects
Documentation	3
Downloader	5
E-Commerce	10
Editor Plugins and IDE	
Emacs	1
Sublime	2
Vim	3
Visual Studio	1
VS Code	1
IDE	2
Email	
Mail Servers	2
Clients	2
Others	2
Enterprise Application Integration	1
Environment Management	2
Files	7
Foreign Function Interface	4
Forms	6
Functional Programming	7
GUI Development	16
GraphQL	4
Game Development	9
Geolocation	5
HTML Manipulation	11
HTTP Clients	6
Hardware	7
Image Processing	15
Implementations	13
Interactive Interpreter	3
Internationalization	2
Job Scheduler	11
Logging	5
Machine Learning	9
Microsoft Windows	5
Miscellaneous	6
Natural Language Processing	

Table A.1: Awesome Python Categories (...continued)

Category	Number of Projects
General	9
Chinese	4
Network Virtualization	3
News Feed	2
ORM	
Relational Databases	8
NoSQL Databases	4
Package Management	3
Package Repositories	4
Penetration Testing	3
Permissions	2
Processes	3
Recommender Systems	8
Refactoring	3
RESTful API	
Django	2
Flask	3
Pyramid	1
Framework Agnostic	7
Robotics	2
RPC Servers	2
Science	21
Search	5
Serialization	4
Serverless Frameworks	2
Shell	1
Specific Formats Processing	
General	1
Office	9
PDF	3
Markdown	2
YAML	1
CSV	1
Archive	1
Static Site Generator	5
Tagging	1
Task Queues	5

Table A.1: Awesome Python Categories (...continued)

Category	Number of Projects
Template Engine	3
Testing	
Testing Frameworks	5
Test Runners	3
GUI / Web testing	6
Mock	8
Object Factories	3
Code Coverage	1
Fake data	4
Text Processing	
General	10
Slugify	3
Unique Identifiers	2
Parser	7
Third-party APIs	7
URL Manipulation	4
Video	3
Web Asset Management	7
Web Content Extracting	9
Web Crawling	8
Web Frameworks	
Synchronous	4
Asynchronous	1
Web Sockets	3
WSGI Servers	5

Table A.2: DyPyBench Available Python Projects

Project Number	Project Name	Project Category
1	grab	Web Crawling
2	PythonRobotics	Robotics
3	flask-api	RESTful API
4	schedule	Job Scheduler
5	Pillow	Image Processing
6	supervisor	DevOps Tools
7	streamparse	Distributed Computing
8	dh-virtualenv	Distribution

Table A.2: DyPyBench Available Python Projects (...continued)

Project Number	Project Name	Project Category
9	pdoc	Documentation
10	click	Command-line Interface Development
11	delegator.py	Processes
12	python-patterns	Algorithms and Design Patterns
13	uvicorn	ASGI Servers
14	pydub	Audio
15	pyjwt	Authentication
16	cerberus	Data Validation
17	python-diskcache	Caching
18	errbot	CahtOps Tools
19	black	Code Analysis
20	thefuck	Command-line Tools
21	python-future	Compatibility
22	python-decouple	Configuration
23	arrow	Date and Time
24	pudb	Debugging Tools
25	akshare	Downloader
26	seaborn	Data Visualization
27	pyfilesystem2	Files
28	wtforms	Forms
29	funcy	Functional Programmming
30	graphene	GraphQL
31	geopy	Geolocation
32	pyquery	HTML Manipulation
33	requests	HTTP Clients
34	scikit-learn	Machine Learning
35	blinker	Miscellaneous
36	zerorpc-python	RPC Servers
37	elasticsearch-dsl-py	Search
38	marshmallow	Serialization
39	pypdf	Specific Formats Processing
40	lektor	Static Site Generator
41	celery	Task Queues
42	jinja	Template Engine
43	pytest	Testing
44	python-ftfy	Text Processing
45	furl	URL Manipulation

Table A.2: DyPyBench Available Python Projects (...continued)

Project Number	Project Name	Project Category
46	moviepy	Video
47	webassets	Web Asset Management
48	html2text	Web Content Extracting
49	websockets	WebSockets
50	gunicorn	WSGI Servers

B Code Snippets

B.1 DynaPyt Analysis

Listing B.1: Call Graph Analysis in DynaPyt

```python
from typing import Callable, Tuple, Dict
import logging
import libcst as cst
import libcst.matchers as m
from .BaseAnalysis import BaseAnalysis
from ..utils.nodeLocator import get_parent_by_type
import json
from inspect import getmodule


class CallGraph(BaseAnalysis):
    def __init__(self):
        super(CallGraph, self).__init__()
        logging.basicConfig(filename="dynapyt.json", format='%(message)s', level=
            logging.INFO)
        self.graph = {}


    '''
    DynaPyt hook for pre function call
    '''
    def pre_call(self, dyn_ast: str, iid: int, function: Callable, pos_args: Tuple
        , kw_args: Dict):
        ast, iids = self._get_ast(dyn_ast)
        module = getmodule(function)
        module = str(module).split('␣')[1] if module is not None else "'''"
        # calling function
        caller = get_parent_by_type(ast, iids.iid_to_location[iid], m.FunctionDef
            ())
        # called function
        if hasattr(function, "__qualname__"):
```

```
27          callee = module[1:-1] + '.' + function.__qualname__ if module != "''"
                else function.__qualname__
28      else:
29          temp = str(function)
30          callee = temp

32      #file name
33      key = dyn_ast.replace('.py.orig', '').replace('/','.')

35      if caller is None:
36          f = key
37      else:
38          # if caller is a part of class, find the class name
39          caller_parent = get_parent_by_type(ast, iids.iid_to_location[iid], m.
                ClassDef())
40          if caller_parent is None:
41              f = key + '.' + caller.name.value
42          else:
43              f = key + '.' + caller_parent.name.value + '.' + caller.name.value

45      # if caller already added
46      if f in self.graph.keys():
47          temp = self.graph[f]
48          # filter dupilcate callees
49          if callee not in temp:
50              temp.append(callee)
51              self.graph[f] = temp
52      else:
53          self.graph[f] = [callee]

55  def end_execution(self):
56      try:
57          logging.info(json.dumps(self.graph))
58      except Exception:
59          logging.info("{")
60          for idx, key in enumerate(self.graph):
61              if not idx == (len(self.graph.keys()) - 1):
62                  logging.info("{} : {}, ".format(key, self.graph[key]))
63              else:
64                  logging.info("{} : {}".format(key, self.graph[key]))
65          logging.info("}")
```

B.2   DyPyBench

Listing B.2: Access Interface of DyPyBench

```python
1  import argparse
2  import subprocess
3  import csv
4  import os

6  parser = argparse.ArgumentParser()
7  parser.add_argument(
8      "--list", "-l", action="store_true", help="List␣all␣the␣projects␣DyPyBench␣
           supports"
9  )

11 parser.add_argument(
12     "--timeout", type=int, help="Specify␣timeout␣to␣be␣used␣in␣seconds␣for␣
           execution␣of␣subprocesses", default=(60*10)
13 )

15 parser.add_argument(
16     "--test", "-t", type=int, nargs='+', help="Specify␣the␣project␣number␣to␣run␣
           the␣test␣suite"
17 )

19 parser.add_argument(
20     "--test_original", "-to", action="store_true", help="Run␣tests␣on␣original␣
           code"
21 )

23 parser.add_argument(
24     "--save", "-s", type=str, help="Specify␣file␣name␣to␣save␣the␣stdout␣and␣
           stderr␣combined"
25 )

27 parser.add_argument(
28     "--update_dynapyt_source", action="store_true", help="get␣dynapyt␣source␣code"
29 )

31 parser.add_argument(
32     "--update_lex_source", action="store_true", help="get␣LExecutor␣source␣code"
33 )
```

```
35 parser.add_argument(
36     "--dynapyt_instrument", "-di", type=int, nargs='+', help="Specify␣the␣project␣
           no.␣to␣run␣DynaPyt␣instrumentation"
37 )


39 parser.add_argument(
40     "--dynapyt_file", "-df", type=str, help="Specify␣the␣path␣to␣file␣containing␣
           the␣includes.txt␣file␣to␣run␣the␣instrumentation"
41 )


43 parser.add_argument(
44     "--dynapyt_analysis", "-da", help="Specify␣DynaPyt␣analysis␣to␣run"
45 )


47 parser.add_argument(
48     "--dynapyt_run", "-dr", type=int, nargs='+', help="Specify␣the␣project␣no.␣to␣
           run␣DynaPyt␣Analysis"
49 )


51 parser.add_argument(
52     "--lex_instrument", "-li", type=int, nargs='+', help="Specify␣the␣project␣no.␣
           to␣run␣LExecutor␣instrumentation"
53 )


55 parser.add_argument(
56     "--lex_file", "-lf", type=str, help="Specify␣the␣path␣to␣file␣containing␣the␣
           includes.txt␣file␣to␣run␣the␣instrumentation"
57 )


59 parser.add_argument(
60     "--lex_test", "-lt", type=int, nargs='+', help="Specify␣the␣project␣no.␣to␣run
           ␣test␣suite␣for␣LExecutor"
61 )


63 parser.add_argument(
64     "--pycg", "-scg", type=int, nargs='+', help="Specify␣the␣project␣no.␣to␣run␣
           PyCG␣for␣static␣call␣graph␣generation"
65 )


67 def printAllProjects():
```

```
68      print("{:<8}␣{:<20}␣{:<50}".format("Number", "Project␣Name", "Repository␣URL")
            )
69      print("{:<8}␣{:<20}␣{:<50}".format("-------", "--------------", "
            ------------------------------"))
70      for value in data:
71          no, name, url = value
72          print("{:<8}␣{:<20}␣{:<50}".format(no, name, url))


74  def setupProjects():
75      global data
76      global original_data
77      data = []
78      original_data = []
79      with open("/DyPyBench/text/github-url.txt", "r") as csv_file:
80          csvReader = csv.reader(csv_file, delimiter="␣")
81          for index, row in enumerate(csvReader):
82              temp=[]
83              temp.append(index + 1)
84              temp.append(row[0].split("/")[-1].split(".git")[0])
85              temp.append(row[0])
86              data.append(temp)
87              original_data.append(row)


89  def get_project_name(proj_no):
90      for value in data:
91          no, name, url = value
92          if(proj_no == no):
93              return name


95  def get_project_no(proj_name):
96      for value in data:
97          no, name, url = value
98          if(proj_name == name):
99              return str(no)


101 if __name__ == '__main__':
102     args = parser.parse_args()


104     setupProjects()


106     if args.list:
```

```
107          printAllProjects()

109      if args.update_dynapyt_source:
110          # print("Downloading the dynapyt source from git")
111          if args.save:
112              output = subprocess.run(["/DyPyBench/scripts/setup-DynaPyt-src.sh"
113              ], shell=True, stdout=open(args.save,'a+',1), stderr=subprocess.STDOUT
                   )
114          else:
115              output = subprocess.run(["/DyPyBench/scripts/setup-DynaPyt-src.sh"
116              ], shell=True, capture_output=True)
117              #if output needs to be printed on the console then comment above and
                   uncomment below
118              """output = subprocess.run(["/DyPyBench/scripts/setup-DynaPyt-src.sh"
119              ], shell=True, stderr=subprocess.STDOUT)"""

121      if args.update_lex_source:
122          # print("Downloading the LExecutor source from git")
123          if args.save:
124              output = subprocess.run(["/DyPyBench/scripts/setup-LExecutor-src.sh"
125              ], shell=True, stdout=open(args.save,'a+',1), stderr=subprocess.STDOUT
                   )
126          else:
127              output = subprocess.run(["/DyPyBench/scripts/setup-LExecutor-src.sh"
128              ], shell=True, capture_output=True)
129              #if output needs to be printed on the console then comment above and
                   uncomment below
130              """output = subprocess.run(["/DyPyBench/scripts/setup-LExecutor-src.sh
                   "
131              ], shell=True, stderr=subprocess.STDOUT)"""

133      if args.test:
134          projects = args.test
135          if 0 in projects:
136              projects = [x for x in range(1,51)]
137          for project in projects:
138              if(project < 0 or project > 50):
139                  print("Project number should be between 1 and 50")
140              else:
141                  proj_name = str(data[project - 1][1])
142                  proj_no = str(data[project - 1][0])
```

```
143              proj_flags = str(original_data[project - 1][1])
144              copy_folder = args.test_original
145              if(proj_flags == "rt"):
146                  proj_test_folder = str(original_data[project - 1][3])
147              elif(proj_flags == "t"):
148                  proj_test_folder = str(original_data[project - 1][2])
149              elif(proj_flags == "r"):
150                  proj_test_folder = ""

152              if args.save:
153                  output = subprocess.run(["/DyPyBench/scripts/run-test.sh %s %s
                         %s %s %s" %(proj_name, proj_no, proj_test_folder,
                         copy_folder, args.timeout)
154                  ], shell=True, stdout=open(args.save,'a+',1), stderr=subprocess
                         .STDOUT, timeout=args.timeout)
155              else:
156                  output = subprocess.run(["/DyPyBench/scripts/run-test.sh %s %s
                         %s %s %s" %(proj_name, proj_no, proj_test_folder,
                         copy_folder, args.timeout)
157                  ], shell=True, capture_output=True, timeout=args.timeout)
158                  #if output needs to be printed on the console then comment
                         above and uncomment below
159                  """output = subprocess.run(["/DyPyBench/scripts/run-test.sh %s
                         %s %s %s %s" %(proj_name, proj_no, proj_test_folder,
                         copy_folder, args.timeout)
160                  ], shell=True, stderr=subprocess.STDOUT, timeout=args.timeout)
                         """


162      if args.dynapyt_instrument:
163          projects = args.dynapyt_instrument
164          if 0 in projects:
165              projects = [x for x in range(1,51)]
166          for project in projects:
167              if(project < 0 or project > 50):
168                  print("Project number should be between 1 and 50")
169              else:
170                  proj_name = str(data[project - 1][1])
171                  proj_no = str(data[project - 1][0])
172                  instr_file = args.dynapyt_file
173                  analysis = args.dynapyt_analysis
```

```
175             with open(instr_file, 'r') as inst_file:
176                 csvReader = csv.reader(inst_file, delimiter="␣")
177                 instr_details = {}
178                 for row in csvReader:
179                     project_name, flag, path = row
180                     project_no = get_project_no(project_name)
181                     if project_no in instr_details.keys():
182                         temp = instr_details[project_no]
183                         temp.append((project_no, flag, path))
184                         instr_details[project_no] = temp
185                     else:
186                         instr_details[project_no] = [(project_no, flag, path)]

188             if args.save:
189                 output = subprocess.run(["/DyPyBench/scripts/clear-project.sh␣%
                        s␣%s" %(proj_name, proj_no)
190                         ], shell=True, stdout=open(args.save,'a+',1), stderr=
                            subprocess.STDOUT, timeout=args.timeout)
191             else:
192                 output = subprocess.run(["/DyPyBench/scripts/clear-project.sh␣%
                        s␣%s" %(proj_name, proj_no)
193                 ], shell=True, capture_output=True, timeout=args.timeout)
194                 #if output needs to be printed on the console then comment
                        above and uncomment below
195                 """output = subprocess.run(["/DyPyBench/scripts/clear-project.
                        sh %s %s" %(proj_name, proj_no)
196                 ], shell=True, stderr=subprocess.STDOUT, timeout=args.timeout)
                        """

198             for line in instr_details[proj_no]:
199                 project_no, flag, path = line
200                 if args.save:
201                     output = subprocess.run(["/DyPyBench/scripts/run-dynapyt-
                            instrumentation.sh␣%s␣%s␣%s␣%s␣%s" %(proj_name,
                            proj_no, path, analysis, flag, args.timeout)
202                     ], shell=True, stdout=open(args.save,'a+',1), stderr=
                            subprocess.STDOUT, timeout=args.timeout)
203                 else:
204                     output = subprocess.run(["/DyPyBench/scripts/run-dynapyt-
                            instrumentation.sh␣%s␣%s␣%s␣%s␣%s" %(proj_name,
                            proj_no, path, analysis, flag, args.timeout)
```

```python
205              ], shell=True, capture_output=True, timeout=args.timeout)
206              #if output needs to be printed on the console then comment
                     above and uncomment below
207              """output = subprocess.run(["/DyPyBench/scripts/run-dynapyt-
                     instrumentation.sh %s %s %s %s %s %s" %(proj_name,
                     proj_no, path, analysis, flag, args.timeout)
208              ], shell=True, stderr=subprocess.STDOUT, timeout=args.
                     timeout)"""

210     if args.dynapyt_run:
211         projects = args.dynapyt_run
212         if 0 in projects:
213             projects = [x for x in range(1,51)]
214         for project in projects:
215             if(project < 0 or project > 50):
216                 print("Project number should be between 1 and 50")
217             else:
218                 proj_name = str(data[project - 1][1])
219                 proj_no = str(data[project - 1][0])
220                 analysis = args.dynapyt_analysis
221                 proj_flags = str(original_data[project - 1][1])
222                 if(proj_flags == "rt"):
223                     proj_test_folder = str(original_data[project - 1][3])
224                 elif(proj_flags == "t"):
225                     proj_test_folder = str(original_data[project - 1][2])
226                 elif(proj_flags == "r"):
227                     proj_test_folder = ""

229                 if args.save:
230                     # os.system("/DyPyBench/scripts/run-dynapyt-analysis.sh %s %s %
                         s %s >> %s 2>&1" %(proj_name, proj_no, analysis,
                         proj_test_folder, args.save))
231                     output = subprocess.run(["/DyPyBench/scripts/run-dynapyt-
                         analysis.sh %s %s %s %s %s" %(proj_name, proj_no, analysis,
                         proj_test_folder, args.timeout)
232                     ], shell=True, stdout=open(args.save,'a+',1), stderr=subprocess
                         .STDOUT, timeout=args.timeout)
233                 else:
234                     # os.system("/DyPyBench/scripts/run-dynapyt-analysis.sh %s %s %
                         s %s" %(proj_name, proj_no, analysis, proj_test_folder))
```

```
235              output = subprocess.run(["/DyPyBench/scripts/run-dynapyt-
                     analysis.sh␣%s␣%s␣%s␣%s␣%s" %(proj_name, proj_no, analysis,
                     proj_test_folder, args.timeout)
236              ], shell=True, capture_output=True, timeout=args.timeout)
237              #if output needs to be printed on the console then comment
                     above and uncomment below
238              """output = subprocess.run(["/DyPyBench/scripts/run-dynapyt-
                     analysis.sh %s %s %s %s %s" %(proj_name, proj_no, analysis,
                     proj_test_folder, args.timeout)
239              ], shell=True, stderr=subprocess.STDOUT, timeout=args.timeout)
                     """


241  if args.lex_instrument:
242      projects = args.lex_instrument
243      if 0 in projects:
244          projects = [x for x in range(1,51)]
245      for project in projects:
246          if(project < 0 or project > 50):
247              print("Project␣number␣should␣be␣between␣1␣and␣50")
248          else:
249              proj_name = str(data[project - 1][1])
250              proj_no = str(data[project - 1][0])
251              instr_file = args.lex_file

253              with open(instr_file, 'r') as inst_file:
254                  csvReader = csv.reader(inst_file, delimiter="␣")
255                  instr_details = {}
256                  for row in csvReader:
257                      project_name, path = row
258                      project_no = get_project_no(project_name)
259                      if project_no in instr_details.keys():
260                          temp = instr_details[project_no]
261                          temp.append((project_no, path))
262                          instr_details[project_no] = temp
263                      else:
264                          instr_details[project_no] = [(project_no, path)]

266              if args.save:
267                  output = subprocess.run(["/DyPyBench/scripts/clear-project.sh␣%
                         s␣%s" %(proj_name, proj_no)
```

```
268                              ], shell=True, stdout=open(args.save,'a+',1), stderr=
                                     subprocess.STDOUT, timeout=args.timeout)
269                        else:
270                            output = subprocess.run(["/DyPyBench/scripts/clear-project.sh␣%
                                   s␣%s" %(proj_name, proj_no)
271                            ], shell=True, capture_output=True, timeout=args.timeout)
272                            #if output needs to be printed on the console then comment
                                   above and uncomment below
273                            """output = subprocess.run(["/DyPyBench/scripts/clear-project.
                                   sh %s %s" %(proj_name, proj_no)
274                            ], shell=True, stderr=subprocess.STDOUT, timeout=args.timeout)
                                   """

276                        files = []
277                        for line in instr_details[proj_no]:
278                            project_no, file_path = line
279                            files.append(file_path)

281                        path = '␣'.join([str(path) for path in files])

283                        if args.save:
284                            output = subprocess.run(["/DyPyBench/scripts/run-lex-
                                   instrumentation.sh␣%s␣%s␣%s␣%s" %(proj_name, proj_no, args.
                                   timeout, path)
285                            ], shell=True, stdout=open(args.save,'a+',1), stderr=subprocess
                                   .STDOUT, timeout=args.timeout)
286                        else:
287                            output = subprocess.run(["/DyPyBench/scripts/run-lex-
                                   instrumentation.sh␣%s␣%s␣%s␣%s" %(proj_name, proj_no, args.
                                   timeout, path)
288                            ], shell=True, capture_output=True, timeout=args.timeout)
289                            #if output needs to be printed on the console then comment
                                   above and uncomment below
290                            """output = subprocess.run(["/DyPyBench/scripts/run-lex-
                                   instrumentation.sh %s %s %s %s" %(proj_name, proj_no, args.
                                   timeout, path)
291                            ], shell=True, stderr=subprocess.STDOUT, timeout=args.timeout)
                                   """

293    if args.lex_test:
294        projects = args.lex_test
```

```python
295        if 0 in projects:
296            projects = [x for x in range(1,51)]
297        for project in projects:
298            if(project < 0 or project > 50):
299                print("Project number should be between 1 and 50")
300            else:
301                proj_name = str(data[project - 1][1])
302                proj_no = str(data[project - 1][0])
303                proj_flags = str(original_data[project - 1][1])
304                if(proj_flags == "rt"):
305                    proj_test_folder = str(original_data[project - 1][3])
306                elif(proj_flags == "t"):
307                    proj_test_folder = str(original_data[project - 1][2])
308                elif(proj_flags == "r"):
309                    proj_test_folder = ""

311                if args.save:
312                    # os.system("/DyPyBench/scripts/run-test-1executor.sh %s %s %s
                           %s >> %s 2>&1" %(proj_name, proj_no, proj_test_folder, args.
                           save))
313                    output = subprocess.run(["/DyPyBench/scripts/run-lex-test.sh %s
                           %s %s %s" %(proj_name, proj_no, proj_test_folder, args.
                           timeout)
314                    ], shell=True, stdout=open(args.save,'a+',1), stderr=subprocess
                           .STDOUT, timeout=args.timeout)
315                else:
316                    # os.system("/DyPyBench/scripts/run-test-1executor.sh %s %s %s
                           %s" %(proj_name, proj_no, proj_test_folder))
317                    output = subprocess.run(["/DyPyBench/scripts/run-lex-test.sh %s
                           %s %s %s" %(proj_name, proj_no, proj_test_folder, args.
                           timeout)
318                    ], shell=True, capture_output=True, timeout=args.timeout)
319                    #if output needs to be printed on the console then comment
                           above and uncomment below
320                    """output = subprocess.run(["/DyPyBench/scripts/run-lex-test.sh
                           %s %s %s %s" %(proj_name, proj_no, proj_test_folder, args.
                           timeout)
321                    ], shell=True, stderr=subprocess.STDOUT, timeout=args.timeout)
                           """

323    if args.pycg:
```

```
324        projects = args.pycg
325        if 0 in projects:
326            projects = [x for x in range(1,51)]
327        for project in projects:
328            if(project < 0 or project > 50):
329                print("Project number should be between 1 and 50")
330            else:
331                proj_name = str(data[project - 1][1])
332                proj_no = str(data[project - 1][0])
333                proj_flags = str(original_data[project - 1][1])
334                if(proj_flags == "rt"):
335                    proj_test_folder = str(original_data[project - 1][3])
336                elif(proj_flags == "t"):
337                    proj_test_folder = str(original_data[project - 1][2])
338                elif(proj_flags == "r"):
339                    proj_test_folder = ""

341                flag = "folder"
342                if proj_test_folder.__contains__(".py"):
343                    flag = "file"

345                if args.save:
346                    output = subprocess.run(["/DyPyBench/scripts/run-pycg.sh %s %s
                           %s %s %s" %(proj_name, proj_no, proj_test_folder, flag, args
                           .timeout)
347                    ], shell=True, stdout=open(args.save,'a+',1), stderr=subprocess
                           .STDOUT, timeout=args.timeout)
348                else:
349                    output = subprocess.run(["/DyPyBench/scripts/run-pycg.sh %s %s
                           %s %s %s" %(proj_name, proj_no, proj_test_folder, flag, args
                           .timeout)
350                    ], shell=True, capture_output=True, timeout=args.timeout)
351                    #if output needs to be printed on the console then comment
                           above and uncomment below
352                    """output = subprocess.run(["/DyPyBench/scripts/run-pycg.sh %s
                           %s %s %s %s" %(proj_name, proj_no, proj_test_folder, flag,
                           args.timeout)
353                    ], shell=True, stderr=subprocess.STDOUT, timeout=args.timeout)
                           """
```

Listing B.3: Bash Script for Project Selection

```
1  #root directory
2  ROOT_DIR=$(pwd)


4  #read URL_FILE
5  URL_FILE=$ROOT_DIR/text/github-url.txt


7  # Create project folder to keep all the projects together inside one parent
       folder
8  PROJ_DIR=$ROOT_DIR/Project
9  #if folder already present, then delete the folder
10 if [[ ! -d "$ROOT_DIR/$PROJ_DIR" ]]
11 then
12     mkdir -p "$ROOT_DIR/$PROJ_DIR"
13 fi
14 cd "$ROOT_DIR/$PROJ_DIR"


16 #run a while loop for all projects
17 idx=1
18 while read line
19 do
20     parts=($line)
21     URL=${parts[0]}
22     FLAGS=${parts[1]}
23     if [[ $FLAGS == "rt" ]]
24     then
25         REQ_FILE=${parts[2]}
26         TEST_SUITE=${parts[3]}
27     elif [[ $FLAGS == "t" ]]
28     then
29         TEST_SUITE=${parts[2]}
30     fi


32     #change to working directory
33     cd $PROJ_DIR


35     #create directory for project
36     mkdir -p "project$idx"


38     #clone the repo to project directory
39     git clone "$URL" "project$idx"
40     cd "project$idx"
```

```
42    #create virtual env name .vm
43    virtualenv .vm

45    #activate virtual env
46    if [[ -d ".vm/local" ]]
47    then
48        source .vm/local/bin/activate
49    elif [[ -d ".vm/bin" ]]
50    then
51        source .vm/bin/activate
52    else
53        echo "Unable␣to␣create␣virtual␣env"
54        exit
55    fi

57    #install using pip install .
58    echo "Running␣pip␣install␣."
59    pip install .

61    if [[ $FLAGS == "rt" ]]
62    then
63        if [[ $URL == "https://github.com/spotify/dh-virtualenv.git" ]]
64        then
65            sed -i.bak '0,/invoke==0.13.0/s//invoke/' dev-requirements.txt #fix
                   for dependency conflict issue
66        fi
67        echo "Running␣pip␣install␣requirements"
68        pip install -r $REQ_FILE
69    fi

71    #some projects need extra requirements for running test suites
72    if [[ $URL == "https://github.com/lorien/grab.git" ]]
73    then
74        pip install cssselect pyquery pymongo fastrq #required for running tests
75    elif [[ $URL == "https://github.com/psf/black.git" ]]
76    then
77        pip install aiohttp #required for running tests
78    elif [[ $URL == "https://github.com/errbotio/errbot.git" ]]
79    then
80        pip install mock #required for running tests
```

```
81    elif [[ $URL == "https://github.com/PyFilesystem/pyfilesystem2.git" ]]
82    then
83        pip install parameterized pyftpdlib psutil #required for running tests
84    elif [[ $URL == "https://github.com/wtforms/wtforms.git" ]]
85    then
86        pip install babel email_validator #required for running tests
87    elif [[ $URL == "https://github.com/geopy/geopy.git" ]]
88    then
89        pip install docutils #required for running tests
90    elif [[ $URL == "https://github.com/gawel/pyquery.git" ]]
91    then
92        pip install webtest #required for running tests
93    elif [[ $URL == "https://github.com/elastic/elasticsearch-dsl-py.git" ]]
94    then
95        pip install pytz #required for running tests
96    elif [[ $URL == "https://github.com/marshmallow-code/marshmallow.git" ]]
97    then
98        pip install pytz simplejson #required for running tests
99    elif [[ $URL == "https://github.com/pytest-dev/pytest.git" ]]
100   then
101       pip install hypothesis xmlschema #required for running tests
102   fi

104   #install pytest library
105   pip install pytest

107   #run test suite
108   if [[ $1 == "scikit-learn" ]]
109   then
110       pytest --import-mode=importlib $TEST_SUITE #tests for scikit-learn need
              importlib to locate conftest
111   else
112       pytest $TEST_SUITE
113   fi

115   ((idx++))
116   deactivate

118 done < "$URL_FILE"
```

Bibliography

[1] *ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4*. URL: `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[2] Hadeel Alsolai and Marc Roper. "A systematic literature review of machine learning techniques for software maintainability prediction". en. In: *Information and Software Technology* 119 (Mar. 2020), p. 106214. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2019.106214`.

[3] Maurício Aniche et al. "The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring". In: *IEEE Transactions on Software Engineering* 48.4 (Apr. 2022), pp. 1432–1450. ISSN: 1939-3520. DOI: `10.1109/TSE.2020.3021736`.

[4] Muhammad Ilyas Azeem et al. "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis". en. In: *Information and Software Technology* 108 (Apr. 2019), pp. 115–138. ISSN: 0950-5849. DOI: `10.1016/j.infsof.2018.12.009`.

[5] Thoms Ball. "The concept of dynamic analysis". en. In: *ACM SIGSOFT Software Engineering Notes* 24.6 (Nov. 1999), pp. 216–234. ISSN: 0163-5948. DOI: `10.1145/318774.318944`.

[6] *Benchmark*. en. Page Version ID: 1127768447. Dec. 2022. URL: `https://en.wikipedia.org/w/index.php?title=Benchmarking&oldid=1127768447`.

[7] *Benchmark (computing)*. en. Page Version ID: 1137083952. Feb. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Benchmark_(computing)&oldid=1137083952`.

[8] Stephen M. Blackburn et al. "The DaCapo benchmarks: java benchmarking development and analysis". In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. OOPSLA '06. New York, NY, USA: Association for Computing Machinery, Oct. 2006, pp. 169–190. ISBN: 978-1-59593-348-5. DOI: `10.1145/1167473.1167488`. URL: `https://doi.org/10.1145/1167473.1167488`.

[9] *Boost.Benchmarks*. URL: `https://www.boost.org/doc/libs/1_81_0/libs/json/doc/html/json/benchmarks.html`.

[10] Hudson Borges and Marco Tulio Valente. "What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform". en. In: *Journal of Systems and Software* 146 (Dec. 2018), pp. 112–129. ISSN: 01641212. DOI: `10.1016/j.jss.2018.09.016`.

[11] *CVE-Search*. Python. Apr. 2023. URL: `https://github.com/cve-search/cve-search`.

[12] *Docker*. en-US. May 2022. URL: `https://www.docker.com/`.

[13] *Docker Hub*. URL: `https://hub.docker.com/`.

[14]   Aryaz Eghbali and Michael Pradel. "DynaPyt: A Dynamic Analysis Framework for Python".
        In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Sym-
        posium on the Foundations of Software Engineering*. ESEC/FSE 2022. Singapore, Singa-
        pore: Association for Computing Machinery, 2022, pp. 760–771. ISBN: 9781450394130. DOI:
        10.1145/3540250.3549126. URL: https://doi.org/10.1145/3540250.3549126.

[15]   *GitHub*. en. URL: https://github.com.

[16]   *Java Microbenchmark Harness (JMH)_2023*. Java. Feb. 2023. URL: https://github.com/
        openjdk/jmh.

[17]   Rui Lima, António Miguel Rosado da Cruz, and Jorge Ribeiro. "Artificial Intelligence Applied
        to Software Testing: A Literature Review". In: *2020 15th Iberian Conference on Information
        Systems and Technologies (CISTI)*. June 2020, pp. 1–6. DOI: 10.23919/CISTI49556.2020.
        9141124.

[18]   *List of tools for static code analysis*. en. Page Version ID: 1149963228. Apr. 2023. URL: https:
        //en.wikipedia.org/w/index.php?title=List_of_tools_for_static_code_analysis&
        oldid=1149963228.

[19]   Fang Liu et al. "Multi-task learning based pre-trained language model for code completion".
        In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software
        Engineering*. ASE '20. New York, NY, USA: Association for Computing Machinery, Jan.
        2021, pp. 473–485. ISBN: 978-1-4503-6768-4. DOI: 10.1145/3324884.3416591. URL: https:
        //doi.org/10.1145/3324884.3416591.

[20]   Matthias S. Müller et al. "SPEC MPI2007—an application benchmark suite for parallel sys-
        tems using MPI". In: *Concurrency and Computation: Practice and Experience* 22.2 (2010),
        pp. 191–205. DOI: https://doi.org/10.1002/cpe.1535. eprint: https://onlinelibrary.
        wiley.com/doi/pdf/10.1002/cpe.1535. URL: https://onlinelibrary.wiley.com/doi/
        abs/10.1002/cpe.1535.

[21]   Matthias S. Müller et al. "SPEC OMP2012 — An Application Benchmark Suite for Paral-
        lel Systems Using OpenMP". en. In: *OpenMP in a Heterogeneous World*. Ed. by Barbara
        M. Chapman et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012,
        pp. 223–236. ISBN: 978-3-642-30961-8. DOI: 10.1007/978-3-642-30961-8_17.

[22]   James Newsome and Dawn Xiaodong Song. "Dynamic taint analysis for automatic detection,
        analysis, and signaturegeneration of exploits on commodity software." In: *NDSS*. Vol. 5.
        Citeseer. 2005, pp. 3–4.

[23]   Safa Omri and Carsten Sinz. "Deep Learning for Software Defect Prediction: A Survey".
        In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering
        Workshops*. ICSEW'20. New York, NY, USA: Association for Computing Machinery, Sept.
        2020, pp. 209–214. ISBN: 978-1-4503-7963-2. DOI: 10.1145/3387940.3391463. URL: https:
        //doi.org/10.1145/3387940.3391463.

[24] Michael Pradel and Koushik Sen. "DeepBugs: A Learning Approach to Name-based Bug Detection". In: arXiv:1805.11683 (Apr. 2018). arXiv:1805.11683 [cs]. DOI: `10.48550/arXiv.1805.11683`. URL: `http://arxiv.org/abs/1805.11683`.

[25] *PyCharm*. en. Page Version ID: 1147902283. Apr. 2023. URL: `https://en.wikipedia.org/w/index.php?title=PyCharm&oldid=1147902283`.

[26] *PyDev*. en. Page Version ID: 1147368803. Mar. 2023. URL: `https://en.wikipedia.org/w/index.php?title=PyDev&oldid=1147368803`.

[27] *Python (Programming Language)*. en. Page Version ID: 1136880732. Feb. 2023. URL: `https://en.wikipedia.org/w/index.php?title=Python_(programming_language)&oldid=1136880732`.

[28] *Python Performance Testing*. en. Sept. 2022. URL: `https://blog.sentry.io/2022/09/30/python-performance-testing-a-comprehensive-guide/`.

[29] Vitalis Salis et al. "PyCG: Practical Call Graph Generation in Python". In: arXiv:2103.00587 (Feb. 2021). arXiv:2103.00587 [cs]. URL: `http://arxiv.org/abs/2103.00587`.

[30] Tushar Sharma et al. "A Survey on Machine Learning Techniques for Source Code Analysis". In: arXiv:2110.09610 (Sept. 2022). arXiv:2110.09610 [cs]. URL: `http://arxiv.org/abs/2110.09610`.

[31] Beatriz Souza and Michael Pradel. "LExecutor: Learning-Guided Execution". In: arXiv:2302.02343 (Feb. 2023). arXiv:2302.02343 [cs]. DOI: `10.48550/arXiv.2302.02343`. URL: `http://arxiv.org/abs/2302.02343`.

[32] *sys — System-specific parameters and functions*. URL: `https://docs.python.org/3/library/sys.html`.

[33] *The Computer Language Benchmarks Game*. en. Page Version ID: 1132931747. Jan. 2023. URL: `https://en.wikipedia.org/w/index.php?title=The_Computer_Language_Benchmarks_Game&oldid=1132931747`.

[34] *The Python Performance Benchmark Suite — Python Performance Benchmark Suite 1.0.6 documentation*. URL: `https://pyperformance.readthedocs.io/`.

[35] *What Is Dynamic Analysis?* en. URL: `https://totalview.io/blog/what-dynamic-analysis`.

[36] Michael Wong. "C++ benchmarks in SPEC CPU2006". In: *ACM SIGARCH Computer Architecture News* 35.1 (Mar. 2007), pp. 77–83. ISSN: 0163-5964. DOI: `10.1145/1241601.1241617`.

[37] Zhaogui Xu et al. "Python predictive analysis for bug detection". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. New York, NY, USA: Association for Computing Machinery, Nov. 2016, pp. 121–132. ISBN: 978-1-4503-4218-6. DOI: `10.1145/2950290.2950357`. URL: `https://dl.acm.org/doi/10.1145/2950290.2950357`.

[38]   Jie M. Zhang et al. "Machine Learning Testing: Survey, Landscapes and Horizons". In: *IEEE Transactions on Software Engineering* 48.1 (Jan. 2022), pp. 1–36. ISSN: 1939-3520. DOI: `10.1109/TSE.2019.2962027`.

———————————————   ———————————————————————

Datum   **Unterschrift**