The Last Dependency Crusade: Solving Python Dependency Conflicts with LLMs

Antony Bartlett
Delft University of Technology
Delft, The Netherlands
a.j.bartlett@tudelft.nl

Cynthia Liem
Delft University of Technology
Delft, The Netherlands
c.c.s.liem@tudelft.nl

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
a.panichella@tudelft.nl

***Abstract*— Resolving Python dependency issues remains a tedious and error-prone process, forcing developers to manually trial compatible module versions and interpreter configurations. Existing automated solutions, such as knowledge-graph-based and database-driven methods, face limitations due to the variety of dependency error types, large sets of possible module versions, and conflicts among transitive dependencies. This paper investigates the use of Large Language Models (LLMs) to automatically repair dependency issues in Python programs. We propose `PLLM` (pronounced "plum"), a novel retrieval-augmented generation (RAG) approach that iteratively infers missing or incorrect dependencies. `PLLM` builds a test environment where the LLM proposes module combinations, observes execution feedback, and refines its predictions using natural language processing (NLP) to parse error messages. We evaluate `PLLM` on the Gistable `HG2.9K` dataset, a curated collection of real-world Python programs. Using this benchmark, we explore multiple PLLM configurations, including six open-source LLMs evaluated both with and without RAG. Our findings show that RAG consistently improves fix rates, with the best performance achieved by `Gemma-2 9B` when combined with RAG. Compared to two state-of-the-art baselines, `PyEGo` and `ReadPyE`, `PLLM` achieves significantly higher fix rates; +15.97% more than `ReadPyE` and +21.58% more than `PyEGo`. Further analysis shows that `PLLM` is especially effective for projects with numerous dependencies and those using specialized numerical or machine-learning libraries.**

***Index Terms*—Python, dependency conflicts, large language models, retrieval-augmented generation**

I. Introduction

Python, introduced in 1991, has become one of the most widely used programming languages [1], due in part to its extensive ecosystem of reusable modules. These modules, once imported, become project dependencies that must be managed over time. As Python's ecosystem grew, so did the complexity of dependency management, prompting the development of tools such as `distutils` [2], `setuptools` [3], and `pip` [4], which are based on the Python Package Index (`PyPI`) to install direct and transitive dependencies.

Yet despite these advances, many Python programs still fail to run out-of-the-box [5], [6], typically due to missing, incompatible, or conflicting dependencies. A *dependency conflict* occurs when two or more modules require different and incompatible versions of the same dependency. This can arise directly—when a specified version is unavailable or mismatches another requirement—or transitively, when nested dependencies bring incompatible constraints. Such conflicts often result in runtime errors, broken APIs, or failed installations. This problem is particularly acute in machine learning and scientific computing, where libraries frequently depend on hardware-specific versions (e.g., CUDA) or require tightly coupled version combinations [7]. Even minor version updates can introduce breaking changes [8], highlighting the need for precise and automated dependency resolution.

Existing automated solutions rely on knowledge graphs (*e.g.*,, `PyEGo` [9], `ReadPyE` [10]), but these struggle with complex dependencies and require frequent updates.

We introduce `PLLM`, an LLM-based method that resolves dependency conflicts through iterative repair. `PLLM` combines Retrieval-Augmented Generation with build feedback, addressing hallucinations through concrete error feedback [11]. To explore this, we introduce `PLLM` (pronounced "plum"), a novel LLM-based method that automatically resolves dependency issues through an iterative repair process.

We evaluate `PLLM` on the Gistable `HG2.9K`, a curated benchmark of real-world challenging Python programs commonly used to assess dependency resolution methods. This benchmark is widely used in the literature to evaluate the performance of dependency conflict resolution techniques [9], [10], [12]. The following research questions guide our study:

RQ1: *To what extent can current LLMs infer working dependencies from a given Python file?*
RQ2: *How does `PLLM` compare to state-of-the-art knowledge-based dependency resolution techniques?*
RQ3: *Under what conditions does `PLLM` outperform traditional methods?*

We evaluated `PLLM` with six LLMs, observing Gemma-2 9B with RAG as the most performant. Once evaluated against the entire `HG2.9K`, `PLLM` was found to significantly outperform knowledge-graph baselines `PyEGo` and `ReadPyE`, achieving +21.58% and +15.97% more fixes respectively.

`PLLM` is particularly effective for projects with intricate dependency structures, excelling at complex dependencies such as those for machine learning (e.g., `tensorflow`) and numerical computing (e.g., `scipy`), contributing the most unique conflict resolutions overall.

II. Background and Related Work

A. Existing Approaches

Knowledge graphs have been widely used to encode relationships among Python modules. Horton and Parnin [12] introduced `DockerizeMe`, which constructs a knowledge graph using `Libraries.io` data stored in Neo4J to infer required dependencies and resolve transitive dependencies.

Building on this work, `PyEGo` [9] expands `DockerizeMe` with package documentation and release metadata, using 256,000 nodes and 1.9 million relationships.

`ReadPyE` [10] combines naming similarity with optimization algorithms for module matching. By iteratively validating dependency choices and adjusting module selections `ReadPyE` finds solutions based on validation logs. Although this method improves matching accuracy, it and `PyEGo` require regular updates to maintain the graph's relevance, a common limitation among knowledge-graph-based tools.

In contrast to graph-based methods, Mukherjee *et al.* [13] proposed `PyDFix`, which uses regex-based parsing of Python error logs to infer missing dependencies. By analyzing build failures and iteratively patching the environment, `PyDFix` provides a lightweight, runtime-driven alternative. However, regex-based methods are inherently brittle: even small changes in log formatting can break the parsing logic, limiting the generality and robustness of such approaches.

While these methods have contributed significantly to the field, they suffer from key limitations: knowledge graphs require large databases and frequent updates, while regex methods are brittle to format changes [14]–[16].

`PLLM` uses an LLM within a RAG pipeline to dynamically resolve dependencies by retrieving PyPI metadata and incorporating build error feedback, avoiding static infrastructure and reducing model hallucination risks [11]. This adaptive pipeline reduces the need for pre-built infrastructure and improves generalization across a broad range of dependency error types.

B. Datasets of Python Dependency Conflicts

Evaluating dependency repair techniques requires datasets that include real-world Python code with known dependency issues. The most widely used benchmark is the Gistable dataset, introduced by Horton and Parnin [6]. This dataset, collected from GitHub's Gist platform[1], contains over 10,000 Python gists—programs that vary in complexity. The authors found that only 24.4% of these gists could be executed without modification, while 52% failed due to missing imports. The Gistable dataset[2] includes a challenging subset of 2,891 "hard" gists that fail due to missing imports and remain unrunnable without fixes. This subset has become the gold standard for evaluating dependency-fixing approaches.

III. Our Approach

`PLLM` is a prompt-based LLM system supporting multiple open-source backends. Figure 1 illustrates our five-stage pipeline combining RAG and LLMs for iterative Python dependency resolution.

```
Prompt:
"Given a python file:\n{raw_file}\nReturn a list of Python
    ↪ modules and python version required to run. Output
    ↪ JSON based on the schema {format_instructions}"
```

Listing 1: Prompt for inferring information from a given Python file.

```
{ "python_modules": [{"module": "<String>", "version": "<
    String>"}], "python_version": "<String>" }
```

Listing 2: JSON Schema: Dependencies and Python version.

- *Stage A*: Inferring module names and Python version.
- *Stage B*: Inferring module versions.
- *Stage C*: Docker-based build and validation.
- *Stage D*: Error analysis and classification.
- *Stage E*: Feedback loop and iterative refinement.

We now detail each stage, describe the prompts used, and explain how RAG contributes to each step.

A. Inferring Module Names and Python Version (Stage A)

`PLLM` prompts the LLM with the input Python file to infer required modules and Python version (Listing 1). Placeholders like `raw_file` and `format_instructions` (JSON schema in Listing 2) make prompts modular and reusable.

When RAG is enabled, `PLLM` complements the LLM output by running a regex-based static analysis to extract all import statements from the input. These are merged with the LLM-inferred modules to form a more complete list of module/ dependencies to consider for conflict fixes. We further elaborate on RAG in Section III-F.

B. Inferring Module Versions (Stage B)

Stage B prompts the LLM to assign versions to each identified module. Using RAG, we provide PyPI metadata to avoid hallucinated versions (Listing 3); without RAG, the LLM infers versions from context (Listing 4). Standard library modules are filtered out, and import/install name mismatches are resolved via curated mapping, e.g., `sklearn` → `scikit-learn`, `bs4` → `beautifulsoup4` as shown Listing 5.

The prompt with RAG in Listing 3 contains a larger set of placeholders, as we provide the LLM with specific dependency information for guidance. In the Listing, `'module_name'` represents the name of the module (dependency) obtained from the initial prompt (Stage A). When RAG is enabled, we provide `'module_versions'`, a comma-separated list of available versions sourced from PyPI, for the given dependency. We also provide the LLM with a list of previously attempted versions `'previous_versions'` to prevent using versions that already resulted in unsuccessful conflict fixes. To promote diversity and avoid local optima, we request the LLM to sample versions evenly across the available list. Finally, the `'format_instruction'` prompts the LLM to return both the dependency name and the newly chosen version. When RAG is disabled, Stage B uses a much simpler prompt. An

[1]https://gist.github.com/
[2]https://github.com/gistable/gistable

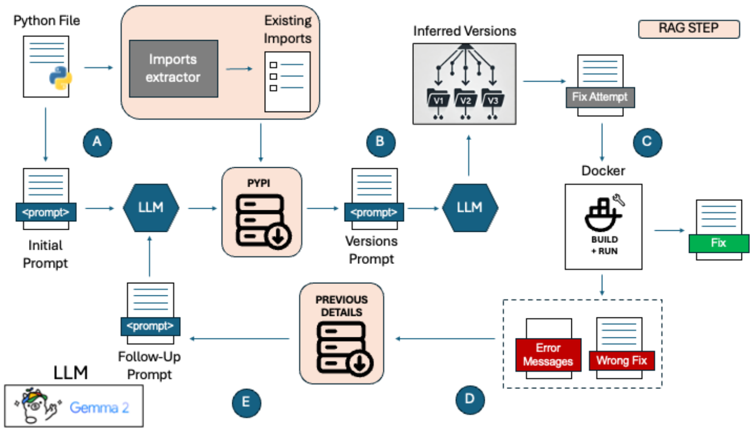

Fig. 1: Overview of PLLM, which encompasses five main stages: (A) extracting import statements from the input Python file, (B) prompting the model to infer modules and Python versions, (C) generating candidate dependency fixes using PyPI metadata, (D) validating candidate fixes through Docker build and execution, and (E) providing error-based feedback to the LLM for iterative refinement.

```
Prompt:
"Given a comma-separated list of 'Module versions' for the
    ↪ '{module_name}' module, from oldest to newest:\n{
    ↪ module_versions}\nPerform equally distanced sampling
    ↪ to return a version from the given versions,
    ↪ excluding previously used versions ({
    ↪ previous_versions}). Return the information with the
    ↪ format {format_instructions}"
```

Listing 3: Prompt with RAG for selecting a version from available PyPI releases.

```
Prompt:
"Infer a possible working version of the '{module_name}'
    ↪ module for Python {python_version}.\nReturn the
    ↪ information with the format {format_instructions}"
```

Listing 4: Prompt without RAG (LLM infers version from context).

example of such a prompt can be seen in Listing 4. In this prompt, we only give the LLM the module name, python version and format instructions, allowing it to infer a version only with this information.

C. Docker-based Build and Validation (Stage C)

PLLM validates predictions by building and executing the Python file in Docker containers. We construct Dockerfiles with the inferred Python version and modules (Listing 6), using unique names for concurrent execution. Success is indicated by a valid configuration; failure triggers error analysis.

D. Error Analysis and Classification (Stage D)

Stage D analyzes build/execution errors to guide LLM corrections. Building on prior work in automated environment inference [9], [10], [12], this hybridized approach reduces over-prompting–the bottleneck of PLLM–by identifying eight common error types that arise during dependency resolution: VersionNotFound, DependencyConflict, ImportError, ModuleNotFound, AttributeError, InvalidVersion, NonZeroCode, and SyntaxError.

```
import sklearn
Must be installed as 'scikit-learn'
```

Listing 5: Discrepancy between import and install names.

```
FROM python:3.6
WORKDIR /app
RUN ["pip","install","--upgrade","pip"]
RUN ["pip","install","--trusted-host","pypi.python.org
    ↪ ","--default-timeout=100","keras==2.0.9"]
RUN ["pip","install","--trusted-host","pypi.python.org
    ↪ ","--default-timeout=100","tensorflow==2.4.4"]
COPY snippet.py /app
CMD ["python", "/app/snippet.py"]
```

Listing 6: Dockerfile.

A key insight from our development is that combining multiple failure signals into a single prompt often leads to LLM confusion or hallucinations. Therefore, we adopt a *multi-step prompting strategy* that isolates specific messages and crafts focused prompts, improving interpretability and response accuracy.

For example, ImportError, the most common error, is handled with a specialized prompt to identify the missing module (Listing 7).

An ImportError indicates a missing module that was not installed. We provide the LLM with the error, requesting it return the offending module name. This ensures the LLM handles noisy logs and correctly extracts the relevant information. Stage D thus acts as the diagnostic core, linking runtime feedback to model-driven correction.

E. Feedback Loop and Iterative Refinement (Stage E)

Stage E uses error feedback to iteratively refine dependency suggestions. Each cycle attempts to repair by inferring (i) missing modules, (ii) different versions, or (iii) additional dependencies until success or a predefined iteration limit is reached. We maintain a history of previously attempted combinations to prevent redundant retries while continuing to

```
Prompt:
"Given the following ImportError:\n{error_msg}\nIdentify
    ↪ the module causing the error.\nThe module is usually
    ↪  mentioned in a statement like 'from x import y'.\
    ↪ nReturn just the module name using the format {
    ↪ format_instructions}"
```

Listing 7: Prompt for identifying the missing module from an ImportError.

verify the existence and naming of modules on PyPI. This stage is imperative, as it also allows `PLLM` to indirectly locate and address transitive dependencies.

F. Retrieval-Augmented Generation (RAG) Implementation

Our RAG implementation addresses LLM hallucination by retrieving real-time PyPI metadata (550,000+ modules). However, raw metadata can contain considerable unwanted information, that can mislead models. To address this, we filter the metadata to within the release window of the Python version being used, as well as the explicit Python version support declared by the module. This ensures that only relevant versions are considered, reducing noise and improving inference accuracy. Filtered metadata is provided as comma-separated version lists, which proved most effective for LLM prompting. Furthermore, in situations where no suitable versions are found, we fall back to the latest release of the module.

G. Parallel Multi-Version Execution

A distinctive feature of `PLLM` is its capacity to validate against multiple Python versions simultaneously. For example, if the LLM predicts Python 3.5 and we have allowed for a range of 2, we execute against versions 2.7, 3.4, 3.5, 3.6, and 3.7 in parallel, increasing success likelihood across deployment scenarios.

IV. Experimental Setup

The *goal* of this study is to evaluate the effectiveness of `PLLM` in resolving Python dependency conflicts for real-world Python programs. To this aim, we formulated the following research questions:

RQ1: *To what extent can current LLMs infer working dependencies from a given Python file?*

RQ2: *How does `PLLM` compare to state-of-the-art knowledge-based dependency resolution techniques?*

RQ3: *Under what conditions does `PLLM` outperform traditional methods?*

RQ1 evaluates current LLM capabilities for dependency inference. **RQ2** and **RQ3** compare `PLLM` against knowledge-based approaches to determine contexts where `PLLM` excels while identifying potential limitations.

A. LLM Model Selection

To ensure openness and reproducibility in our approach, we evaluated six open-source LLMs via Ollama[3]: three `Gemma2` [17] versions (3B, 9B, 27B), `DeepSeek-R1` [18], `Llama3.1` [19], and `MistralAI` [20].

[3]https://ollama.com

B. Dataset

We validated our approach using the `HG2.9K` dataset, which has become the standard benchmark for evaluating Python dependency resolution tools [9], [10]. This dataset, originally created as part of the `DockerizeMe` paper by Horton and Parnin [12], is a curated subset of the broader Gistable corpus [6]. It contains $2,891$ "hard" Gists—-Python programs collected from GitHub's Gist platform–that still fail with `ImportError` even after applying Gistable's naïve resolution strategy. As such, it represents realistic and challenging dependency drift scenarios in Python.

To answer **RQ1**, we created a subset of the `HG2.9K` to reduce computational cost while preserving result validity. The subset, containing 422 randomly selected Gists, was executed five times per LLM with different seeds to account for stochasticity [11]. By using power analysis [21], we yield a confidence level of 95% and a margin of error below 4.5% for binary outcomes. For **RQ2** and **RQ3**, we evaluated the best `PLLM` configuration (as identified in **RQ1**) and the baselines on the full `HG2.9K` dataset to ensure a comprehensive comparison.

C. Implementation and Parameter settings

We implemented `PLLM` using Python 3.11, Langchain 0.2.1, and Ollama 0.2.0. Experiments ran in docker containers on an an AMD EPYC 7713 64-core Processor running at 2.6 GHz (256 CPUs) with an available Nvidia A40 GPU (48GB GDDR6 memory).

Our implementation supports a set of configurable parameters to optimize evaluation and test execution. The key parameters relevant to this study include: `--model=gemma2`, `--temp=0.7`, `--loop=10`, `--range=1`, and `--rag=True`.

The `model` parameter specifies which LLM backend to use. This can be set to any model available in the Ollama model library[4]. For this study, we primarily used `Gemma2`, though other models were evaluated as discussed in Section IV-A.

The `temp` parameter controls the temperature for the LLM, with higher values (up to 1.0) encouraging more creative responses. We set this to 0.7 to strike a balance between determinism and variation in output. While the default temperature is not standardized—OpenAI's GPT-4 technical report [22] uses 0.6 arbitrarily, and Ollama's documentation[5] lists both 0.7 and 0.8 as defaults—we selected 0.7 based on its initial status as Ollama's default during early development of `PLLM`.

As described in Section III-C, `PLLM` uses an iterative loop to refine dependency suggestions. We set the `loop` value to 10 based on empirical trade-offs between runtime and success rate. Lower values led to incomplete resolutions, while higher values increased runtime with diminishing returns.

The `range` parameter defines how many Python versions are validated in parallel (see Section III-G). A value of 1 triggers validation across three versions (e.g., $v{-}1$, v, $v{+}1$), offering both breadth and efficiency.

[4]https://ollama.com/library
[5]https://github.com/ollama/ollama/blob/main/docs/modelfile.md

Lastly, the rag flag controls whether Retrieval-Augmented Generation is enabled. It is set to True by default but was explicitly disabled when evaluating **RQ1** to isolate the performance of the LLMs without external metadata assistance.

1) Baseline settings

We evaluated PLLM against two state-of-the-art baselines introduced in prior work: PyEGo [9] and ReadPyE [10] (see Section II). Both baselines were executed using the default settings recommended in their respective repositories. Each required Docker-based execution and the creation of a Neo4j knowledge graph. PyEGo provided a database dump in its repository, while ReadPyE offered this as a separate artifact. Two knowledge graphs are available in ReadPyE; for consistency, we selected **KG0**, which focuses solely on Python module inference. This aligns with the scope of PLLM, which does not attempt to resolve OS-level dependencies. It is worth noting that PyEGo does fix both Python and system-level dependencies, but for evaluation purposes, we restrict comparisons to Python modules only.

D. Evaluation Criteria

To answer **RQ1**, we performed a comparative analysis of various LLMs to understand their performance within the PLLM workflow. As part of this analysis, we executed these models with RAG enabled and disabled to understand the effects of RAG in the pipeline. We executed each model a total five times against a subset of the HG2.9K dataset, calculating the average number of fixes discovered.

The top-performing model from **RQ1** was then used in **RQ2** to compare PLLM against the two baselines, PyEGo and ReadPyE, across the full HG2.9K dataset. Following the evaluation criteria established in DockerizeMe [12], we define a successful fix as a Python program that executes without critical runtime errors such as ImportError, ModuleNotFoundError, AttributeError, or SyntaxError.

For each approach, we also recorded the time taken to infer a working environment, as we were particularly interested in understanding the efficiency of each method in resolving dependency conflicts. Accurate timing information provides insight into the practical usability of these approaches, particularly in scenarios where dependency resolution speed is crucial. Some adjustments were required to obtain timings for our baselines. For ReadPyE, more extensive modifications were necessary to achieve precise timing measurements. First, we recorded the inference time, which includes the stage where a Dockerfile was created with the inferred Python modules. Next, we validated the Dockerfiles, as no logging was initially provided to confirm whether a Dockerfile was executable. Each Dockerfile was built and run at this stage, analyzing the output and logging the time required for completion. We then combined these timings to determine which Dockerfiles were successfully executed. For PyEGo, we could infer considerable information from their output logs, which contained various data points, including the start and stop times for each program inference and whether the inference was successful. To account for the stochastic nature of LLMs, we executed PLLM 10 times, aggregating the data to understand the full extent of fixes. However, as both of our baselines utilize a static knowledge graph, these approaches only required a single run.

TABLE I: Total number of successful fixes for each of the models with RAG on and off. (G2-S:Gemma2-2B, G2:Gemma2-9B, G2-L:Gemma2-27B, DS:DeepSeek-R1-8B, L3.1:Llama3.1-8B, M:MistralAI)

RAG	G2-S	G2	G2-L	DS	L3.1	M
Enabled	86	203	187	166	197	194
Disabled	-	171	207	164	111	106

TABLE II: Distribution of successful fix counts per Gist in the HG2.9K subset dataset.

Model	RAG	5	4	3	2	1
DeepSeek-R1-8B	✓	80	55	36	36	30
Gemma2-2B	✓	33	31	21	22	38
Gemma2-9B	✓	166	22	12	19	25
Gemma2-27B	✓	146	28	17	10	23
Llama3.1-8B	✓	103	53	38	39	35
MistralAI	✓	122	49	23	31	35
DeepSeek-R1-8B	✗	75	53	43	36	33
Gemma2-2B	✗	-	-	-	-	-
Gemma2-9B	✗	119	33	18	23	27
Gemma2-27B	✗	165	21	19	25	21
Llama3.1-8B	✗	49	25	35	23	61
MistralAI	✗	53	24	25	31	33

To answer **RQ3**, we conducted a permutation test [23], a non-parametric alternative to the ANOVA test [24], that does not require the data to be normally distributed. This test allowed us to determine whether there is a significant difference in the number of successful fixes produced by PLLM, PyEGo, and ReadPyE. We also assessed the impact of various project co-factors on these outcomes. The permutation test provided insights into the statistical significance of differences in successful fixes among the approaches without relying on the normality assumptions of ANOVA. To further investigate specific patterns, we analyzed the frequency of successful fixes for particular types of modules (e.g., PyTorch) and for projects with varying numbers of dependencies. We examined the success frequencies across Python files with similar dependencies and different dependency counts. Additionally, we analyzed the intersection and unique sets of projects successfully fixed by PLLM, PyEGo, and ReadPyE, identifying both shared and unique successes for each approach.

The complete replication package, including the datasets, the source code, and results, is available on Figshare [6].

V. Empirical Results

In this section, we discuss the results of the experiment with respect to our research questions.

A. Results for RQ1

Table I indicates comparable performance across most models ($\sim$200 fixes), except Gemma2-2B which achieved less than half. RAG improved performance for most models, though

[6]10.6084/m9.figshare.29204693

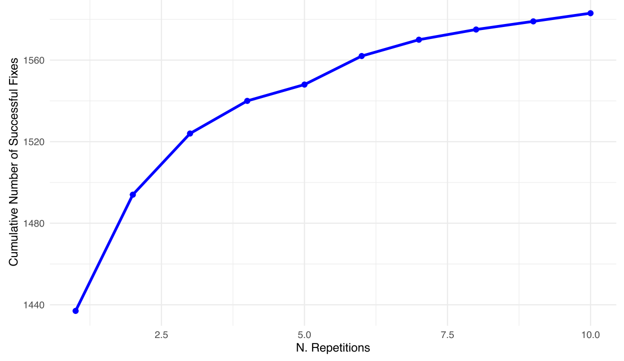

Fig. 2: Cumulative chart displaying the number of fixes found by PLLM across multiple runs.

TABLE III: Number successful and unsuccessful fixes produced by PLLM, PyEGo and ReadPyE. We also report the average time (in seconds) needed to generate the successful fixes alongside the interquartile range (IQR).

Method	# Fixed	# Unfixed	Fix Time (IQR)
PLLM	1583	1308	151.461 (221.30)
ReadPyE	1365	1526	62.77 (40.57)
PyEGo	1302	1589	4.025 (7.56)

Gemma2-27B performed slightly better without RAG. Considering both performance and consistency (Table II), we selected Gemma2-9B with RAG for subsequent experiments. This model provided an optimal balance between high success rates and computational efficiency, showing consistent performance across multiple runs, feasible for extensive evaluation.

Figure 2 shows the cumulative fixes for the Gemma2-9B model against the entire HG2.9K, over ten independent runs. A single run achieves ~1,440 fixes, with additional runs uncovering more unique solutions due to LLM stochasticity. Even with a single run, PLLM outperforms both baselines (see more details in the next subsection).

B. Results for RQ2

Table III reports the number of successful and unsuccessful fixes produced by PLLM, PyEGo, and ReadPyE, and the average time (in seconds) required to generate successful fixes, along with the interquartile range (IQR). As observed, PLLM outperforms both baselines in terms of the number of successful fixes, achieving a total of 1583 successful fixes. This is significantly higher than the 1365 and 1302 successful fixes produced by ReadPyE and PyEGo, respectively.

Compared to the baselines, average fix time is higher for PLLM, with an average time of 151.46 seconds (compared to 62.77 seconds for ReadPyE and 4.025 seconds for PyEGo). The higher IQR for PLLM indicates a broader range of fix times, largely due to the iterative nature of our approach, which continues searching until a successful set of dependencies is found. We argue that this additional time is justified by the higher success rate achieved by PLLM, and an average time of two minutes remains reasonable for achieving a successful fix with Dockerfile generation and execution.

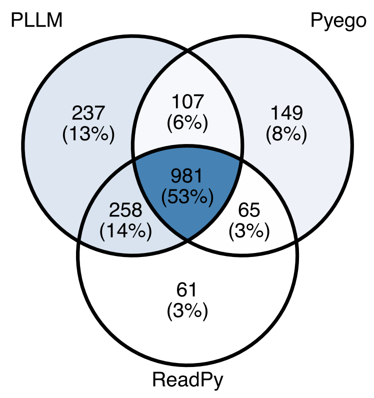

Fig. 3: Intersections and difference sets for the projects successfully fixed by PLLM, PyEGo, and ReadPyE.

TABLE IV: Percentage of successful fixes produced by PLLM and PyEGo for projects with a different number of dependencies. The best values are highlighted in gray color

# Project Dependencies	PLLM	PyEGo	ReadPyE
0-2	0.5478	0.4846	0.5046
3-4	0.5500	0.4162	0.4517
5-6	0.5410	0.2551	0.2806
7+	0.5333	0.3555	0.2222

To further analyze the effectiveness of our approach, we examined cases where PLLM succeeded while both baselines failed. Figure 3 visualizes the overlap of successful fixes across all three approaches, illustrating the unique effectiveness of PLLM. Specifically, all three approaches were successful on 53% of all Gists, while PLLM uniquely succeeded on 13% (representing 237 Gists) that neither baseline could fix. There is a distinct overlap of 981 Gists, representing cases where all three approaches were successful. The combined success across all approaches reaches 1858 Gists. Considering these 1858 as the total potentially fixable Gists within our validation system, PLLM successfully fixed 85% of the fixable Gists in the dataset. This finding strongly supports the effectiveness of our approach in reliably inferring working dependencies from a Python file and highlights the potential of hybrid approaches, which are part of our future agenda.

C. Results for RQ3

Permutation tests confirm PLLM's statistical superiority ($p < 0.01$) and significant interaction with dependency count. Table IV shows PLLM maintains consistent success (53-55%) regardless of dependency count, while baselines degrade from 50% to 32% (PyEGo) and <25% (ReadPyE), indicating knowledge-graph methods struggle with complex projects.

Table VI displays examples of Gists in which the various approaches passed and failed. We focus here on examples where PLLM was either better or worse than both baselines (*e.g.,* PLLM passed, PyEGo, ReadPyE failed, *etc.*). This allows us to dig deeper into the results of these specific data points to understand why the LLM succeeds or fails.

TABLE V: Percentage of successful fixes for the top-15 most frequent Python modules in our dataset for `PLLM`, `PyEGo`, and `ReadPyE`.

Module Name	**# Projects**	`PLLM`	`PyEGo`	`ReadPyE`
numpy	572	69.06	57.17	53.50
django	382	86.65	65.18	79.84
scipy	296	79.39	58.78	62.50
requests	168	66.67	50.60	52.38
pillow	166	76.51	69.88	71.08
matplotlib	162	64.81	62.96	54.94
tensorflow	140	83.57	51.43	62.14
scikit-learn	136	76.47	55.88	66.91
tensorflow-gpu	135	82.96	52.59	68.15
opencv-python	85	64.71	49.41	24.71
pandas	73	68.49	63.01	57.53
cython	69	55.07	33.33	39.13
image	69	23.19	18.84	18.84
pyyaml	69	66.67	66.67	71.01
theano	67	68.66	26.87	37.31
keras	63	77.78	14.29	42.86

In Gist `019fd5c706e0bc94879f`, `PLLM` and `PyEGo` both identified `rx` and `twisted` as required modules. In contrast, `ReadPyE` incorrectly inferred `urx`, likely due to a semantic match in its knowledge graph. This resulted in an `ImportError` and failure. While `PyEGo` selected the correct modules, its choice of `rx` version also failed at runtime. All three approaches selected the same version of `twisted`. `PLLM` required a few iterations to converge on a compatible version of `rx`, ultimately producing a successful configuration.

Gist `'477a9dcd198439ef2def'` is an example of where `PLLM` was unsuccessful. By observing the dependency chosen by both baselines, we identify that `readability` is likely no longer available or has been replaced with `readability-lxml`. We can also observe the same scenario with `'7030355'` and the `mpd`, now `python-mpd2` dependency. These examples could be added to our JSON file mentioned in Section III-B. We also note a failure for `PLLM` with Gist `'4e5035242b8e4b07ff3a'`. This is a little more interesting as the LLM has `pymongo` listed as a dependency. However, unlike the other approaches, it has also attempted to install `pip` and `bson`, which have likely contributed to the failure in this instance.

Finally, we examine Gists `'3153844'` and `'c2dfe57-72ba3cd16c1be17ba42b7db66'` in which `PLLM` was the only successful approach. Here, we witness another example of `ReadPyE` attempting a dependency with an alternative name. In this instance, the original dependency `graphite` was successfully installed by `PLLM`, and no dependency was even attempted by `PyEGo`. The final Gist `'c2dfe5772ba3cd16c1be17ba42b7db66'` has overlap between the baselines, but ultimately, the lack of `tensorflow` in `PyEGo` and the incorrect `tensorflow` dependency for `ReadPyE` led to unsuccessful conflict resolutions for both approaches.

These examples illustrate several core strengths of `PLLM`: iterative refinement, flexible version search, and responsiveness to real-time failure signals. However, they also expose failure modes tied to naming mismatches and unnecessary package inference—areas where targeted prompt tuning or expanded mapping rules could yield further gains.

VI. Threats to validity

Construct validity. A key aspect of our approach (`PLLM`) is allowing the LLM to dynamically infer Python versions from Python code and validate these along with adjacent Python versions. This flexible strategy addresses limitations in prior work [10], which rely on static Python versions during testing, potentially limiting both the adaptability and real-world relevance of the results. The only exception in our approach is Python 2.7, which is consistently validated due to its unique, legacy-specific versioning requirements.

Our method ensures that, once execution begins, the Python version remains unchanged, even if a `SyntaxError` occurs —an indicator of an incompatible Python version. Unlike approaches that dynamically switch versions mid-execution, our approach simulates real-world constraints more accurately.

Internal validity. For evaluation, we used a single LLM, Gemma2 [17], which showed strong performance in tests compared to other open-source LLMs. A small manual study was performed using the commercial Claude 4 Sonnet model [7]. Claude fixed 15 PLLM-failure cases, already handled by the baselines, but failed on all PLLM-only successes particularly with 3+ dependencies, often hallucinating versions or misusing import names. This supports our open-source choice.

To address the stochastic nature of LLM outputs, we followed existing guidelines [11] on assessing LLMs and executed `PLLM` multiple times, aggregating the results to ensure a more comprehensive assessment of its effectiveness. The model size also impacts internal validity; we selected the 9-billion-parameter version of Gemma2, compatible with both our local machine and the validation servers.

External validity. Our evaluation is based on the `HG2.9K` dataset, which has become a de facto standard in the Python dependency resolution literature. It has been used by both of our baselines [9], [10], and contains realistic, challenging Python programs (Gists) exhibiting dependency drift, making it well suited for assessing module-level inference capabilities.

While `ReadPyE` has also been evaluated on other datasets, we focused on `HG2.9K` for two key reasons. First, it enables direct, fair comparison with prior work. Second, it allows us to isolate and evaluate the core question of this paper: how well can LLMs infer and resolve dependency configurations from Python source code? This controlled scope provides a focused evaluation of dependency inference capabilities, which is fundamental to broader dependency resolution challenges. Expanding to other datasets would improve the generalizability of our findings, and this is a natural next step for future work.

Conclusion validity. The `HG2.9K` dataset includes some Python programs that are unresolvable due to module deprecation. Additionally, our approach was validated only on x86 Docker Linux machines. This limitation restricts our ability to

[7]https://www.anthropic.com/claude/sonnet

TABLE VI: Example Gists showing where PLLM, PyEGo, and ReadPyE succeeded or failed in dependency resolution. Each cell lists inferred dependencies; grey cells denote successful runs. These cases illustrate complementary strengths and limitations of the different approaches.

Gist ID	PLLM	PyEGo	ReadPyE
019fd5c706e0bc94879f	rx;twisted	rx;twisted	twisted;urx
477a9dcd198439ef2def	urllib2;readability	readability-lxml	readability-lxml
3153844	graphite	NA	graphiti
4e5035242b8e4b07ff3a	pymongo;pip;bson	pymongo	pymongo
c2dfe5772ba3cd16c1be17ba42b7db66	keras;tensorflow	keras	keras;tensorflow-gpu
7030355	mpd	python-mpd2	python-mpd2

validate Gists in HG2.9K designed for other platforms, such as macOS or ARM-based systems. Hence, we were unable to validate if a Dockerfile generated by one of our baselines would execute correctly on these platforms.

As a result, PyEGo includes operating system-level dependencies by default. ReadPyE also provides this capability through a specific database dump. However, to keep the approaches consistent with PLLM, we configured ReadPyE to only fix Python-level dependencies with its default database.

VII. Conclusion and Future Work

This paper introduced PLLM, a novel LLM-driven approach for resolving Python dependency conflicts. Unlike traditional methods such as PyEGo and ReadPyE, which rely on precomputed knowledge graphs, PLLM uses prompt-based inference and feedback loops to recover a working environment from raw Python code and runtime errors. This eliminates the need for extensive external infrastructure, making the solution lightweight, adaptable, and easy to apply across environments.

Our evaluation on the HG2.9K benchmark demonstrates that PLLM outperforms the state-of-the-art solutions PyEGo and ReadPyE. PLLM achieves significantly more successful resolutions, performing consistently, even in cases with many dependencies. Qualitative analysis reveals that PLLM's success stems from its ability to refine its guesses based on runtime information —an advantage absent in static systems.

At the same time, both baselines still succeed in a subset of cases that PLLM fails to resolve. These are often linked to legacy dependencies, renamed packages, or system-level nuances more effectively handled by structured knowledge graphs. This points to a key insight: prompt-based and knowledge-based techniques offer complementary strengths. Rather than treating them as competing approaches, we see clear potential in hybrid approaches that unify LLM-driven reasoning with curated symbolic knowledge.

Future work includes extending PLLM to support system-level dependency resolution, improving prompt construction and dependency disambiguation, and exploring alternative RAG methods such as GraphRAG [25]. Finally, we plan to investigate hybrid architectures that combine prompt-based inference with knowledge graph approaches given their complementary nature as highlighted by our results.

References

[1] S. Cass, "The top programming languages 2024," *IEEE Spectrum*, 08 2024. [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2024

[2] K. W. Smith, *Cython: A Guide for Python Programmers*. " O'Reilly Media, Inc.", 2015.

[3] "Building and distributing packages with setuptools," 2024. [Online]. Available: https://setuptools.pypa.io/en/latest/setuptools.html

[4] "Python software foundation - the pip tool," 2024. [Online]. Available: https://pypi.org/project/pip/https://pypi.org/project/pip/

[5] D. Yang *et al.*, "From query to usable code: an analysis of stack overflow code snippets," ser. ICSE '16. ACM, May 2016. [Online]. Available: http://dx.doi.org/10.1145/2901739.2901767

[6] E. Horton and C. Parnin, "Gistable: Evaluating the executability of python code snippets on github," 2018, pp. 217–227.

[7] K. Huang *et al.*, "Demystifying dependency bugs in deep learning stack," ser. ESEC/FSE 2023, 2023, p. 450–462. [Online]. Available: https://doi.org/10.1145/3611643.3616325

[8] J. Dietrich *et al.*, "Dependency versioning in the wild." IEEE, 2019, pp. 349–359.

[9] H. Ye *et al.*, "Knowledge-based environment dependency inference for python programs," ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1245–1256. [Online]. Available: https://doi.org/10.1145/3510003.3510127

[10] W. Cheng *et al.*, "Revisiting knowledge-based inference of python runtime environments: A realistic and adaptive approach," *IEEE Transactions on Software Engineering*, vol. 50, no. 2, pp. 258–279, 2024.

[11] J. Sallou *et al.*, "Breaking the silence: the threats of using llms in software engineering," 2024, pp. 102–106.

[12] E. Horton and C. Parnin, "Dockerizeme: Automatic inference of environment dependencies for python code snippets," 2019. [Online]. Available: https://arxiv.org/abs/1905.11127

[13] S. Mukherjee *et al.*, "Fixing dependency errors for python build reproducibility," ser. ISSTA 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 439–451. [Online]. Available: https://doi.org/10.1145/3460319.3464797

[14] T. Zhang *et al.*, "System log parsing: A survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 8, pp. 8596–8614, 2023.

[15] J. Zhu *et al.*, "Tools and benchmarks for automated log parsing." IEEE, 2019, pp. 121–130.

[16] S. Messaoudi *et al.*, "A search-based approach for accurate identification of log message formats," 2018, pp. 167–177.

[17] G. Team *et al.*, "Gemma 2: Improving open language models at a practical size," 2024. [Online]. Available: https://arxiv.org/abs/2408.00118

[18] D.-A. Team *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[19] L. . Team *et al.*, "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[20] A. Q. Jiang *et al.*, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[21] M. F. Triola *et al.*, *Elementary statistics*. Pearson/Addison-Wesley Reading, MA, 2004.

[22] O. Team, "Gpt-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2303.08774

[23] M. J. Crawley, *Statistics: an introduction using R*. John Wiley & Sons, 2014.

[24] L. St *et al.*, "Analysis of variance (anova)," *Chemometrics and intelligent laboratory systems*, vol. 6, no. 4, pp. 259–272, 1989.

[25] J. Larson and S. Truitt, "Graphrag: Unlocking llm discovery on narrative private data," *Microsoft Research Blog*, 02 2024. [Online]. Available: https://www.microsoft.com/en-us/research/blog/graphrag-unlocking-llm-discovery-on-narrative-private-data/