ENVBENCH: A BENCHMARK FOR AUTOMATED ENVIRONMENT SETUP

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin,[*] Egor Bogomolov, Yaroslav Zharov
JetBrains Research
Correspondence to alexandra.eliseeva@jetbrains.com

ABSTRACT

Recent advances in Large Language Models (LLMs) have enabled researchers to focus on practical repository-level tasks in software engineering domain. In this work, we consider a cornerstone task for automating work with software repositories—environment setup, *i.e.*, a task of configuring a repository-specific development environment on a system. Existing studies on environment setup introduce innovative agentic strategies, but their evaluation is often based on small datasets that may not capture the full range of configuration challenges encountered in practice. To address this gap, we introduce a comprehensive environment setup benchmark ENVBENCH. It encompasses 329 Python and 665 JVM-based (Java, Kotlin) repositories, with a focus on repositories that present genuine configuration challenges, excluding projects that can be fully configured by simple deterministic scripts. To enable further benchmark extension and usage for model tuning, we implement two automatic metrics: a static analysis check for missing imports in Python and a compilation check for JVM languages. We demonstrate the applicability of our benchmark by evaluating three environment setup approaches, including a simple zero-shot baseline and two agentic workflows, that we test with two powerful LLM backbones, GPT-4o and GPT-4o-mini. The best approach manages to successfully configure 6.69% repositories for Python and 29.47% repositories for JVM, suggesting that ENVBENCH remains challenging for current approaches. Our benchmark suite is publicly available at https://github.com/JetBrains-Research/EnvBench. The dataset and experiment trajectories are available at https://jb.gg/envbench.

1 INTRODUCTION

Recent advances in Large Language Models (LLMs) have enabled their application across many domains, including software engineering (Fan et al., 2023). Their capabilities in reasoning and interaction with external environments (Liu et al., 2024b; Wang et al., 2024b), as well as in efficient processing of large amounts of information (Wang et al., 2024a), have allowed researchers to tackle practical repository-level software engineering tasks, such as code generation (Liu et al.; Zhao et al., 2024), code editing (Jimenez et al., 2024), and code understanding (Ma et al., 2024; Luo et al., 2024; Liu et al., 2024a).

In this work, we focus on another repository-level task that programmers face regularly—*environment setup*, *i.e.*, configuring the system to work with an arbitrary software project, for instance, a freshly cloned GitHub repository. It usually entails installing the dependencies but might include arbitrary project-specific steps, such as installing additional system packages, setting the correct environment variables, and more. A well-maintained project should be straightforward to set up, however, in practice, it is not always the case. For instance, setting up the repository is perceived to be the most challenging part of reproducing Natural Language Processing (NLP) research results, according to Storks et al. (2023), it may take up to several hours. Similarly, a survey conducted by Aghajani et al. (2020) reveals that incomplete documentation of installation, deployment, and release processes is considered a significant issue by 68% of developers, and 63% report inappropriate installation instructions as a prevalent concern. Moreover, automating environment setup

[*]Work done during internship at JetBrains Research.

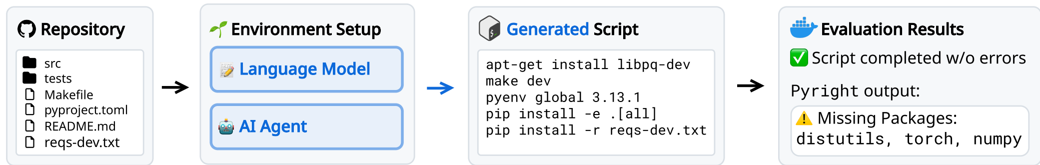

Figure 1: Overview of the workflow with ENVBENCH. The process begins with cloning a target repository. Next, the repository is passed as an input to an environment setup approach, which then produces a shell script to set up the repository as an output. Internally, it could be, for instance, a single LLM request or an AI agent building a script dynamically. Finally, in our evaluation suite, we execute the produced script and verify the environment is correctly configured through static analysis and compilation checks.

could enable scaling of the execution-based benchmarks, which currently often require significant manual effort to select a set of executable repositories (Jimenez et al., 2024).

As of now, few studies have considered environment setup as a standalone task. There are numerous works that include environment setup as a part of a larger task—for instance, scientific reproduction (Siegel et al., 2024; Bogin et al., 2024) or solving machine learning problems (Tang et al., 2024). However, to the best of our knowledge, there are only two works specifically on environment setup concurrent to ours (Milliken et al., 2024; Bouzenia & Pradel, 2024). While these works represent important progress in automating environment setup, they primarily focus on novel agentic strategies rather than on comprehensive benchmarking. That manifests in a limited number of the software projects and technologies covered in the respective datasets. For instance, the dataset from Milliken et al. (2024) features 40 Python repositories, and Bouzenia & Pradel (2024) includes 10 repositories for each of the considered languages (Python, Java, C, C++, and JavaScript).

Taking this into consideration, we introduce a novel environment setup benchmark—ENVBENCH. It features a diverse set of projects, covering Python (329 repositories) and JVM languages such as Java and Kotlin (665 repositories in total). We implement two automatic metrics to verify that the environment is set up correctly — static analysis to obtain the number of missing imports (*i.e.*, the number of import statements across the codebase that couldn't be resolved via static analysis due to the corresponding package not being installed) for Python and a compilation check for JVM languages. We ensure that the projects included in our benchmark present genuine configuration challenges by implementing simple deterministic shell scripts and excluding repositories that can be correctly configured by these scripts alone. Finally, we evaluate three environment setup approaches with two powerful LLMs, GPT-4o and GPT-4o-mini. Our set of baselines includes a simple zero-shot setting, a ReAct (Yao et al., 2023) agentic workflow with access to the Bash terminal similar to Bouzenia & Pradel (2024) (Bash Agent), and an agentic setting following Milliken et al. (2024) (Installamatic Agent).

Our findings show that the Bash Agent with GPT-4o achieves the highest success rates, correctly configuring 29.47% of the JVM repositories and 6.69% of the Python repositories. Although environment setup for Python remains challenging, LLM-based approaches still reduce the number of missing imports compared to the deterministic script for many repositories, demonstrating their potential. Additionally, we observe that LLM-based approaches that are not explicitly provided with error feedback commonly produce erroneous environment setup scripts. This aligns with previous findings (Milliken et al., 2024) that several generation attempts with error feedback significantly improve environment setup capabilities.

Our benchmark suite is publicly available at `https://github.com/JetBrains-Research/EnvBench`. The dataset and experiment trajectories are available at `https://jb.gg/envbench`.

2 RELATED WORKS

Environment setup, *i.e.*, creating a functioning development environment for a software repository, is a vital task in software development. The steps involved vary a lot between programming languages

and even between different repositories coming from the same technology stack, thus complicating the automation of this task. Moreover, obtaining a subset of executable repositories from a vast dataset is a common step for execution-based benchmarks. In most recent benchmarks, a check if a repository is executable is done semi-automatically, with a significant amount of manual work required (Jimenez et al., 2024; Jain et al., 2024; Tang et al., 2024; Zhao et al., 2024).

Approaches. Given the inherent variability of environment setup of arbitrary repositories, non-ML automatic approaches are limited and usually tied to a specific ecosystem. For instance, there are tools that can gather external dependencies from the source code of Python repositories (Gruber & Fraser, 2023; Vadim Kravcenko, 2014), yet configuring the system to ensure successful installation is out of their scope. Recently, two AI agents for environment setup were introduced. Milliken et al. (2024) propose INSTALLAMATIC capable of successfully setting up 21 out of 40 considered Python repositories. Bouzenia & Pradel (2024) introduce EXECUTIONAGENT that correctly configures 33 out of 50 considered repositories across 5 programming languages (Python, Java, C, C++, and JavaScript).

Benchmarks. Several recent works have tackled scientific reproduction (Siegel et al., 2024; Bogin et al., 2024) or solving machine learning problems (Tang et al., 2024) — while both tasks may include environment setup, the evaluation is conducted in an end-to-end fashion, offering little insight into the challenges the current LLMs face during environment setup. To the best of our knowledge, there are two existing benchmarks tailored specifically for the environment setup task — we'll refer to them by the names of accompanying approaches, $\text{INSTALLAMATIC}_{\text{bench}}$ (Milliken et al., 2024) and $\text{EXECUTIONAGENT}_{\text{bench}}$ (Bouzenia & Pradel, 2024), respectively. $\text{INSTALLAMATIC}_{\text{bench}}$ contains 40 Python repositories. Milliken et al. (2024) manually inspect the repositories and provide ground truth installation-relevant context and an exemplar Dockerfile for each. The expected output is a Dockerfile, and the success metric is for at least one test to pass with the generated Dockerfile. $\text{EXECUTIONAGENT}_{\text{bench}}$ covers 5 programming languages (Python, Java, C, C++, and JavaScript) with 10 repositories for each programming language. Bouzenia & Pradel (2024) select the repositories with CI logs available, providing the ground truth results of test suite execution for each repository. The expected outputs are both a Dockerfile that specifies system configuration and a shell script that sets up the environment and runs tests. For evaluation, the authors consider three metrics: success build rate, success test rate, and deviation from the ground truth in terms of the number of passing, failing, and skipped tests. The first two metrics require manual inspection. In addition, both works focus on rather popular projects: Milliken et al. (2024) consider repositories with at least 1000 stars on GitHub, while Bouzenia & Pradel (2024) — with at least 100.

Our contributions. Compared to existing benchmarks, our benchmark covers a broader range of 994 repositories across three programming languages (Python, Java, Kotlin) and two distinct ecosystems (Python and JVM). We apply more relaxed star filters (minimum 10) and exclude repositories that can be configured using simple deterministic scripts, ensuring genuine environment setup challenges. Unlike benchmarks relying on test execution, we use static analysis (Python) and compilation checks (JVM) to verify successful setup. Similar to Bouzenia & Pradel (2024), we employ shell script output format for the environment setup approaches, however, we provide predefined Dockerfiles to ensure that base system configuration remains consistent across all approaches.

3 ENVBENCH BENCHMARK

In this section, we describe ENVBENCH—our benchmark for environment setup task. We consider repositories written in Python or in the JVM-based languages (specifically, Java and Kotlin), representing two popular[1] yet fundamentally different technology stacks. Refer to Appendix A.1 for additional information about the repositories in our benchmark. ENVBENCH is available at `https://jb.gg/envbench`. Our evaluation suite and other associated code are available at `https://github.com/JetBrains-Research/EnvBench`.

[1]For instance, from GitHub's Octoverse report from October 2024, Python is the most used language on GitHub, and Java is the 4th: `https://github.blog/news-insights/octoverse/octoverse-2024/`

3.1 TASK DEFINITION

We present an overview of the expected workflow within our benchmark in Figure 1.

Input and output. In our benchmark, the input for an environment setup approach is the full repository contents, and the expected output is a shell script that configures the repository. Both the processing of repository contents and the script generation method are integral to the approach. For example, the process could involve a single LLM request, where the repository context is gathered using a predefined algorithm, or an LLM agent that dynamically explores the repository and executes shell commands via provided tools to generate the script.

Evaluation metrics. Given the differences between Python and JVM languages, we implement two distinct metrics to evaluate whether the repository environment was configured correctly. For Python, we run a popular static analysis tool pyright (Microsoft, 2019) and count the number of reported errors related to missing dependencies (specifically, of `reportMissingImports` type). For JVM languages, we try to build the repository via either Gradle (`gradle build` command) or Maven (`mvn compile` command), check if the attempt was successful (for both), and report the number of errors in build tool output (for Maven) . In the process of the benchmark construction, we verify that the included JVM-based repositories use either Gradle or Maven build tools based on the presence of the configuration files. Both metrics can finish execution with non-zero exit codes if the script for configuring a repository is incorrect. In most of the considered configurations, our metrics allow both a binary success indicator (zero exit code and zero reported errors) and a continuous measure based on the reported errors per repository that could mitigate the presence of the repositories that are objectively infeasible to set up successfully.

In contrast with our approach, the previous works on environment setup adopted test suite-based metrics (Milliken et al., 2024; Bouzenia & Pradel, 2024) that can be considered closer to the real use-cases. Following Bouzenia & Pradel (2024), execution-based metric for the environment setup could consist of three criteria: successfully building (or installing) the project, being able to run the test suite, and test suite working as expected. Each step depends on all the previous finishing successfully. In our formulation, we cover build (installation) step, discovering most problems that environment setup approaches could face except those that only appear during runtime. Our metrics are more lightweight compared to test suite-based, enabling us to scale our benchmark while our experiments demonstrate that ENVBENCH remains challenging even for powerful LLM-based environment setup approaches (refer to Section 5 for the results of our experiments). As environment setup methods advance, ENVBENCH could be extended with test suite-based metrics, ensuring it remains challenging and closely aligned with real-world use cases.

3.2 EVALUATION SUITE

Environment setup inherently means performing actions that modify system configuration. To prevent automated approaches from possibly corrupting a host system, we implement an evaluation suite where solutions for each repository are launched in a Docker container (Docker, Inc., 2013). The overall evaluation process for each repository is as follows. Evaluation suite expects as an input the name of the repository, the revision at which the repository should be considered, and the environment setup shell script for the repository, supposedly, produced by an environment setup approach in advance. We clone the repository into a Docker container, execute the provided shell script, and, if the script execution finishes successfully, execute the metric for the corresponding language. We release two base Docker images, for Python and for JVM languages, that provide a minimal set of relevant tools. More details about our evaluation suite Docker configuration can be found in Appendix A.2.

3.3 DATA COLLECTION AND FILTERING

Data Collection. To construct ENVBENCH, we start by obtaining a diverse list of GitHub repositories using a dedicated tool GitHub Search (Dabic et al., 2021). Our selection criteria include the primary language of the repository being either Python, Java, or Kotlin, repository having a permissive license, and a set of quality filters designed to exclude projects that might introduce biases (Kalliamvakou et al., 2014): at least 1000 commits; at least 10 issues; at least 10 contributors; at least 10 stars; last commit made not earlier than January 1st, 2024. That way, we focus on ma-

ture and active projects, but include both popular and those that have received less recognition. We then clone the source code of the repositories that are still available at the time of July 2024, which resulted in 2,590 repositories for Python and 1,688 for JVM languages.

Filtering by Contents. We study the configuration files present in the repositories and observe that the majority of Python projects use either pip (Python Packaging Authority, 2008) or Poetry (Python Poetry, 2018) to manage the dependencies, and the presence of other dependency managers is negligible. Similarly, the majority of JVM projects use either Gradle (Gradle, Inc., 2010) or Maven (Apache Software Foundation, 2004) build tools. So, as the next step, we filter out the repositories that either do not contain configuration files associated with the respective dependency managers in their root directory or contain several. This way, we avoid including monorepos — *i.e.*, repositories that contain several different projects from the point of view of configuration — from our sample, as setting up a monorepo is a more challenging task, where evaluation can be quite ambiguous. After this step, we retain 743 repositories for Python and 1,487 for JVM languages.

Additionally, we filter out repositories that contain configuration files related to Docker (Docker, Inc., 2013) (*e.g.*, `Dockerfile` or `docker-compose.yml`). We rely on Docker to sandbox evaluation and experiments (more details in Section 3.2 and Section 4, respectively), and running Docker within a Docker container poses significant technical challenges. Furthermore, Docker is often employed to simplify and encapsulate environment setup, potentially bypassing key challenges. Following this filtering step, we obtain 391 repositories for Python and 977 for JVM languages.

Baseline Filtering. Intuitively, for many repositories environment setup can be as simple as running `pip install -r requirements.txt`. To ensure that our benchmark remains non-trivial, we implement two fully deterministic shell scripts, one for Python and one for JVM languages (refer to Appendix A.3 for details). At the core, the algorithms are similar: (1) analyze configuration files to determine the required dependency manager and Python/Java version; (2) verify if the correct Python/Java version is installed, and install it if necessary; and (3) install packages using the specified dependency manager (for Python; for JVM, project builds are excluded as they occur during evaluation).

We run the baseline scripts and assess their performance using the evaluation suite outlined in Section 3.2. For Python, the script successfully (as defined in Section 3.1) sets up 62 repositories (15.9%), while for JVM languages — 309 repositories (31.6%). From the sample obtained on the previous step, we were unable to process 3 repositories. After filtering out these unprocessable repositories, as well as those successfully configured by the baseline scripts, we obtain a final dataset of 329 Python repositories and 665 JVM repositories.

4 Experimental setup

4.1 Dataset & Metrics

Our dataset, EnvBench, features 329 Python and 665 JVM repositories. Its construction is described in details in Section 3. We implemented two language-specific metrics that rely on either static analysis (for Python) or compilation check (for JVM languages) to confirm if the repository was configured correctly (refer to Section 3.1). These metrics output the number of observed errors, however, depending on a shell script produced by an environment setup approach, they might also exit prematurely with a non-zero exit code. We consider two metrics: *pass@1*, a binary measure of success—where success is defined as both the exit code and reported errors being zero—and *avgErrs*—the average number of reported errors per repository—which quantifies the extent to which the setup was completed. Note that *avgErrs* can only be computed for the repositories where the environment setup script finished execution with a zero exit code.

4.2 Baselines

We consider three LLM-based baselines in our experiments. For each baseline, we run experiments with two proprietary LLMs: GPT-4o and GPT-4o-mini. Refer to Appendix B for implementation details of each baseline.

Zero-shot LLM. We construct a prompt with information about a repository (including a directory structure, README contents, and the contents of the configuration files) and system configuration for our evaluation suite (Section 3.2) and send a single request to an LLM, asking it to generate a shell script that correctly configures the environment for the given repository.

Installamatic Agent. We consider environment setup agent INSTALLAMATIC introduced by Milliken et al. (2024). INSTALLAMATIC comprises two stages: (1) a search phase, where the agent explores the repository contents, and (2) a build/repair phase, where the agent generates and tests a Dockerfile, with several regeneration attempts allowed in case of failure. We slightly adapt INSTALLAMATIC to match our evaluation setup (refer to Section 3.2 for details) by asking the agent to generate a shell script instead of a Dockerfile, providing agent with extra information about the system configuration for our evaluation suite, and crafting a separate prompt for JVM repositories. We also do not allow regeneration attempts and report the results with the first version of the script produced by the agent, which is one of the settings considered by Milliken et al. (2024). In this formulation, INSTALLAMATIC can be considered an extension of Zero-shot LLM that employs an agent for context gathering instead of a predefined prompt template; however, shell script generation is still conducted as a single LLM request.

Bash Agent. We consider a ReAct (Yao et al., 2023) agent that approaches the tasks iteratively, producing thoughts and actions at each step based on previous observations. As the available actions, we provide a single `execute_bash_command` tool that allows interaction with the system via shell commands. Due to safety considerations, we execute the commands issued by the agent inside a Docker container. Compared to Zero-shot LLM and INSTALLAMATIC, this baseline unites both context gathering and shell script generation in an iterative and dynamic process fully controlled by an LLM agent. Additionally, Bash Agent can be considered as a simplified version of the EXECUTIONAGENT introduced by Bouzenia & Pradel (2024), as it is similar, but lacks a few components such as meta-prompting and summarization of the shell commands output.

5 RESULTS & DISCUSSION

Baseline	Model	JVM		Python	
		pass@1 ↑	**avgErrs** ↓ **(Maven)**	**pass@1** ↑	**avgErrs** ↓ **(overlap)**
Zero-shot LLM	GPT-4o	8.57% 57/665	480.50 30	5.47% 18/329	54.89
	GPT-4o-mini	11.13% 74/665	202.97 72	4.56% 15/329	151.30
Installamatic Agent	GPT-4o	1.35% 9/665	**21.43** 65	4.86% 16/329	108.93
	GPT-4o-mini	3.01% 20/665	33.53 32	2.74% 9/329	83.57
Bash Agent	GPT-4o	**29.47%** 196/665	26.84 216	**6.69%** 22/329	**52.00**
	GPT-4o-mini	26.77% 178/665	24.77 205	5.47% 18/329	79.89

Table 1: Main experimental results. **pass@1** — percentage of correctly set up repositories, *i.e.*, repositories where our metric returned zero exit code and reported zero issues. **avgErrs** — average number of the reported errors per repository. For JVM, we only report **avgErrs** for repositories using Maven build tool. Note that **avgErrs** for each baseline can be calculated only over the repositories where the environment setup script from the corresponding baseline finished with a zero exit code. For Python, we report **avgErrs** calculated on 44 repositories where *all* baselines were able to produce scripts finishing with a zero exit code. For JVM, there are no such repositories, so we specify the number of the repositories across which **avgErrs** is calcualted for each baseline. The symbol ↑ indicates that higher values in the current column are better, while ↓ indicates that lower values are better.

We present the primary results of our experiments in Table 1. Bash Agent with GPT-4o as a backbone is the best-performing environment setup approach in terms of pass@1 both for JVM and for Python, successfully setting up 29.47% and 6.69% of the considered repositories, respectively (with GPT-4o). Zero-shot LLM is significantly worse for JVM repositories (11.13% with GPT-4o-mini) and slightly worse for Python (5.47% with GPT-4o). Although Installamatic Agent incorporates more advanced context gathering than Zero-shot LLM, it ranks as the lowest baseline among considered, with 3.01% pass@1 (with GPT-4o-mini) for JVM and 4.86% (with GPT-4o) for Python. We hypothesize that the underlying reason might be that the prompts in Installamatic mostly encourage considering natural language documents, and those are likely to be of higher quality in the original dataset considered by Milliken et al. (2024) due to the popularity filter (the average number of stars is 19k as compared to 1.9k in our dataset).

Our second metric, avgErrs, indicates 52.00 missing imports per repository for the best-performing Python baseline, Bash Agent with GPT-4o, and 21.43 errors per repository for Installamatic Agent with GPT-4o for JVM repositories that use Maven build tool. However, avgErrs can only be computed in cases when baseline-produced environment setup scripts finished execution without errors. We study the exit codes of the produced shell scripts for both Python and JVM and observe that the considered baselines can struggle to achieve that, complicating the comparison via avgErrs (see Appendix C.3 for more details). This problem is more prominent for JVM than for Python.

Baseline	Model	Repositories			Avg. decrease ↑	Avg. increase ↓
		Less ↑	Same	More ↓		
Zero-shot LLM	GPT-4o	**53%** 76/144	34% 50/144	**13%** 18/144	59%	487%
	GPT-4o-mini	39% 67/172	42% 73/172	19% 32/172	56%	589%
Installamatic Agent	GPT-4o	42% 61/145	34% 49/145	24% 35/145	**63%**	286%
	GPT-4o-mini	31% 36/115	46% 53/115	23% 26/115	54%	303%
Bash Agent	GPT-4o	31% 71/228	52% 119/228	17% 38/228	53%	**206%**
	GPT-4o-mini	41% 92/226	41% 93/226	18% 41/226	51%	211%

Table 2: Comparison of missing imports per repository for the considered baselines on the Python sample, relative to the deterministic script described in Section 3.3. **Less**, **Same**, and **More** indicate the number of repositories where the baseline approach resulted in fewer, the same, or more missing imports, respectively. **Avg. decrease** and **Avg. increase** columns represent the average percentage reduction or increase in missing imports for repositories where the baseline outperformed (**Less**) or underperformed the deterministic script (**More**). Statistics are calculated only on repositories where both the baseline and the deterministic script exited with a zero exit code, with the set of repositories varying for each baseline. The symbol ↑ indicates that higher values in the current column are better, while ↓ indicates that lower values are better.

To further break down the performance of the environment setup baselines for Python, we compare them with expert-produced scripts as an upper bound and with a simple deterministic script that we implemented during benchmark construction as a lower bound. For the former, we consider 30 randomly sampled repositories and observe a significant gap between expert-produced scripts and all the considered baselines (see Appendix D for more details). Our expert-produced scripts achieve a pass@1 of 66.7% on that sample as compared to 10.0% pass@1 achieved by the best baseline. For the latter, we report the comparison results in Table 2. For a relatively small percentage of repositories (from 24% for Installamatic Agent with GPT-4o to 13% for Zero-shot LLM with GPT-4o), the scripts produced by the baseline methods result in more missing imports than the deterministic script. However, the average increase in the number of missing imports per repository relative to the deterministic script in this case is significant, at least 200% for all the considered

baselines. These repositories could provide valuable insights into the limitations of current methods and serve as a basis for developing more robust environment setup strategies in future work.

On the other hand, the percentage of the repositories where the scripts produced by the considered baselines outperform the deterministic script is higher for all the considered baselines (from 31% for Bash Agent with GPT-4o to 53% for Zero-shot LLM with GPT-4o). In this case, the average decrease in the number of missing imports per repository relative to the deterministic script is around 50%-60% for all the considered baselines. Overall, while fully correct environment setup for Python remains challenging, the considered baselines show potential compared to a simple deterministic script.

6 LIMITATIONS

Our work has several important limitations that we list below.

Docker Support. We explicitly exclude projects that require Docker for their setup from our benchmark. While Docker is increasingly common in modern development workflows, including such projects would add complexity to our evaluation setup and metrics. Future work could explore extending our approach to handle a wider range of projects.

Data Contamination. Data contamination is another potential concern, as the LLMs we use were trained on public code repositories that may overlap with our benchmark dataset. This could lead to the models having prior exposure to setup instructions for some of the repositories during pre-training. However, given that environment setup guidelines are often not explicitly documented, and even when they exist, they require advanced reasoning capabilities to interpret and follow correctly, we believe this does not significantly impact our findings. As the construction of our benchmark does not require significant manual effort, it can be updated in the future by incorporating newer repositories, thereby reducing the risk of data contamination as LLM training datasets evolve.

OSS Code Quality. The quality of open-source code and documentation varies significantly across repositories. Some repositories may have incomplete or outdated documentation, making environment setup particularly challenging. Others may be completely invalid or impossible to set up due to missing critical files, broken dependencies, or incompatible configurations. Our results may be influenced by this inherent variability in code quality and repository validity, though this reflects real-world conditions that automated tools need to handle. To mitigate this concern, we employ both a binary success metric and the number of reported errors per repository that quantifies the extent to which the environment setup was completed. As a part of future work, our benchmark could be further manually verified to identify and remove invalid samples.

Static Analysis and Compilation. Our evaluation relies on static analysis for Python and compilation checks for JVM languages rather than actual execution metrics like test suite runs. While this provides a reasonable proxy for environment setup success, it may miss runtime issues that would only appear during execution. However, this approach allows us to evaluate environment setup for a larger number of repositories efficiently while avoiding the complexity of test suite execution, which makes our benchmark more representative and allows possibly reusing the built infrastructure for further research that might involve not only evaluation but training of environment setup approaches. Additionally, we validate the robustness of the proposed metrics by manually implementing setup scripts for 30 randomly sampled Python repositories and studying metrics' behavior (see Appendix D for more details).

7 CONCLUSION

This work presents ENVBENCH—a benchmark for evaluating automated environment setup methods, addressing the limited scale of previous environment setup datasets by covering 329 Python and 665 JVM repositories. Our evaluation suite is based on static analysis for Python and compilation checks for JVM, enabling a systematic assessment of environment setup strategies.

We evaluate three environment setup methods, including two AI agents, with two powerful LLMs as the backbones. Our results demonstrate that environment setup remains challenging for those methods, with the best-performing method successfully configuring only 29.47% of JVM and 6.69%

of Python repositories. One key challenge is the generation of erroneous scripts, and we leave further investigation and mitigation to future work.

Our benchmark and the associated code are publicly available, providing a scalable platform for further research. In the future, it could be additionally manually verified and extended to incorporate new software repositories, more programming languages, and runtime-based evaluation metrics.

REFERENCES

Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. Software documentation: the practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, pp. 590–601, New York, NY, USA, October 2020. Association for Computing Machinery. ISBN 978-1-4503-7121-6. doi: 10.1145/3377811.3380405. URL https://doi.org/10.1145/3377811.3380405.

Apache Software Foundation. Apache maven. https://github.com/apache/maven, 2004. Accessed: 2025-01-21.

Ben Bogin, Kejuan Yang, Shashank Gupta, Kyle Richardson, Erin Bransom, Peter Clark, Ashish Sabharwal, and Tushar Khot. SUPER: Evaluating Agents on Setting Up and Executing Tasks from Research Repositories, September 2024. URL http://arxiv.org/abs/2409.07440. arXiv:2409.07440 [cs] version: 1.

Islem Bouzenia and Michael Pradel. You name it, i run it: An llm agent to execute tests of arbitrary projects. *arXiv preprint arXiv:2412.10133*, 2024.

Ozren Dabic, Emad Aghajani, and Gabriele Bavota. Sampling projects in github for msr studies. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pp. 560–564. IEEE, 2021.

Docker, Inc. Docker: Empowering app development for developers. https://www.docker.com/, 2013. Accessed: 2025-01-21.

Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pp. 31–53. IEEE, 2023.

Gradle, Inc. Gradle. https://github.com/gradle/gradle, 2010. Accessed: 2025-01-21.

Martin Gruber and Gordon Fraser. FlaPy: Mining Flaky Python Tests at Scale, May 2023. URL http://arxiv.org/abs/2305.04793. arXiv:2305.04793 [cs].

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2e: Turning any github repository into a programming agent environment. In *ICML*, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pp. 92–101, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328630. doi: 10.1145/2597073.2597074. URL https://doi.org/10.1145/2597073.2597074.

LangChain. Langgraph: Build resilient language agents as graphs. https://github.com/langchain-ai/langgraph. Accessed: 2025-02-08.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025*, 2024a.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024b.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*.

Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. Repoagent: An llm-powered open-source framework for repository-level code documentation generation. *arXiv preprint arXiv:2402.16667*, 2024.

Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. How to understand whole software repository? *arXiv preprint arXiv:2406.01422*, 2024.

Microsoft. Pyright: Static type checker for python. `https://github.com/microsoft/pyright`, 2019. Accessed: 2025-01-21.

Louis Milliken, Sungmin Kang, and Shin Yoo. Beyond pip install: Evaluating llm agents for the automated installation of python projects. *arXiv preprint arXiv:2412.06294*, 2024.

Python Packaging Authority. pip. `https://github.com/pypa/pip`, 2008. Accessed: 2025-01-21.

Python Poetry. Poetry. `https://github.com/python-poetry/poetry`, 2018. Accessed: 2025-01-21.

Zachary S. Siegel, Sayash Kapoor, Nitya Nagdir, Benedikt Stroebl, and Arvind Narayanan. CORE-Bench: Fostering the Credibility of Published Research Through a Computational Reproducibility Agent Benchmark, September 2024. URL `http://arxiv.org/abs/2409.11363`. arXiv:2409.11363 [cs].

Shane Storks, Keunwoo Yu, Ziqiao Ma, and Joyce Chai. Nlp reproducibility for all: Understanding experiences of beginners. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10199–10219, 2023.

Xiangru Tang, Yuliang Liu, Zefan Cai, Yanjun Shao, Junjie Lu, Yichi Zhang, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Liang Chen, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yin Fang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark Gerstein. ML-Bench: Evaluating Large Language Models and Agents for Machine Learning Tasks on Repository-Level Code, August 2024. URL `http://arxiv.org/abs/2311.09835`. arXiv:2311.09835 [cs].

Vadim Kravcenko. pipreqs: Generate pip requirements.txt file based on imports of any project. `https://github.com/bndr/pipreqs`, 2014. Accessed: 2025-02-06.

Xindi Wang, Mahsa Salmani, Parsa Omidi, Xiangyu Ren, Mehdi Rezagholizadeh, and Armaghan Eshaghi. Beyond the limits: A survey of techniques to extend the context length in large language models. *arXiv preprint arXiv:2402.02244*, 2024a.

Yanlin Wang, Wanjun Zhong, Yanxian Huang, Ensheng Shi, Min Yang, Jiachi Chen, Hui Li, Yuchi Ma, Qianxiang Wang, and Zibin Zheng. Agents in software engineering: Survey, landscape, and vision. *arXiv preprint arXiv:2409.09030*, 2024b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

A BENCHMARK

In this section, we provide further details about ENVBENCH, evaluation suite, and the deterministic scripts we used for benchmark construction.

A.1 BENCHMARK STATISTICS

Additional information about the repositories in our benchmark is presented in Table 3.

Table 3: Statistics for the repositories from our benchmark.

Language	General Statistics		Dependency Managers Distribution			
	Avg. Stars	Avg. # Files	Pip	Poetry	Gradle	Maven
Python	1469	779	82.06% 270/329	17.94% 59/329	–	–
JVM	2079	2647	–	–	59.70% 397/665	40.30% 268/665

A.2 DOCKER ENVIRONMENT

We use Docker (Docker, Inc., 2013) for both evaluation suite and experiments to safely isolate the execution of LLM-produced scripts from the host system. We implement two Docker environments, one for Python and one for JVM languages, where we preinstall commonly needed tools. We use preconfigured universal Docker images since it saves time by having standard tools preinstalled, ensures that all approaches operate in the same reproducible environment, and mitigates common issues (*e.g.*, our preliminary experiments showed that all the considered approaches struggle to install Android SDK if it is not available on the system beforehand). We base the Docker images on `ubuntu:22.04` and use non-interactive mode for all tools. The Dockerfiles are available in our GitHub repository: `https://github.com/JetBrains-Research/EnvBench`.

Python Environment: We preinstall:

- pyenv with Python 3.8-3.13
- Poetry for dependency management
- uv for package installation
- pyright for static analysis
- pipenv
- conda/miniconda

JVM Environment: We preinstall:

- sdkman with Java 11.0.20-tem
- Maven
- Gradle
- Node.js and npm
- Android SDK

A.3 Deterministic Scripts

We implement two deterministic scripts that can handle the simplest environment setup cases. The scripts rely on repository contents to determine the required dependency manager and Python/Java version requirements. The exact scripts are available in our GitHub repository: `https://github.com/JetBrains-Research/EnvBench`.

Python Script:

- Detects environment type by checking for environment.yml (Conda), uv.lock (uv), or poetry.lock (Poetry)
- Creates and activates the appropriate virtual environment
- Installs dependencies by searching for requirements.txt, setup.py, pyproject.toml, setup.cfg, or Pipfile
- Exits with error code if no recognized configuration is found

JVM Script:

- Detects build system by checking for pom.xml (Maven) or build.gradle (Gradle)
- Determines Java version from build files or defaults to Java 11
- Runs appropriate build command (mvn install or gradle build)
- Handles common build flags like skipping tests or offline mode

B LLM BASELINES IMPLEMENTATION DETAILS

In this section, we share details on the implementations of the considered LLM-based environment setup baselines. The implementations are available in our repository: `https://github.com/JetBrains-Research/EnvBench`.

For **Zero-shot LLM**, we first follow predefined steps to collect relevant context for each repository. Specifically, for both languages, we provide the following information: directory structure (`tree`, `ls -R`), documentation contents (README, installation guides, Markdown files), environment information (Dockerfile), deterministic script (described in Appendix A.3) for corresponding language as an example. For Python, we provide: common configuration files (`setup.py`, `pyproject.toml`), dependency specifications (`requirements.txt`), Python version requirements (if present in configuration files), contents of `__init__.py` files. For JVM, we provide: build files (`pom.xml`, `build.gradle`, `settings.gradle`), dependency and lock files, Java version requirements (if present in build files), build tool wrapper scripts, `module-info.java` files.

After collecting the context, we employ it to prompt LLM to generate an environment setup shell script.

Bash Agent is a ReAct (Yao et al., 2023) agent that is provided access to a terminal of a Docker container to perform environment setup (refer to Appendix A.2 for details about Docker configuration). The agent is equipped with a single `execute_bash_command` tool that accepts a command and returns stdout contents and stderr contents after the command execution. We use LangGraph (LangChain) framework to implement Bash Agent. The agent is allowed up to 30 iterations, and the execution finishes prematurely if the LLM does not use the tool in its response. We set a 360 seconds timeout for the execution of each issued command. We allow up to 5000 characters in the output of each command to avoid overly long non-informative contexts and return the first and the last half of the output in case it exceeds this value. To obtain a resulting shell script, we include all the executed commands that finished with exit code zero.

For **Installamatic Agent**, we follow the original setting (Milliken et al., 2024) excluding the repair stage: the agent is allowed one full *Documentation Gathering* stage iteration and one full *Dockerfile Build* stage iteration. During *Documentation Gathering* stage, Installamatic Agent explores the repository via given tools until `finish_search` tool is called, and the expected output is the list of the files considered to be installation-relevant. During *Dockerfile Build* stage, the agent is allowed to explore installation-relevant files via the same set of tools until `submit_summary` tool is called; via this tool, the agent produces a natural language summary of the information required to set up the current repository, and afterwards, the agent generates a Dockerfile. We reuse the original prompts, however, reformulate the task to generate a shell script instead, and extend the prompts with information about our evaluation suite Docker configuration. We use LangGraph (LangChain) framework to implement Installamatic Agent.

C Evaluation Results

In this section, we provide additional results from our experiments.

C.1 Case Study

In this section, we present examples of the scripts generated by the agents and the zero-shot LLM. In Figure 2, we compare the performance of the baselines for the Python project `tablib` that is a format-agnostic tabular dataset library.

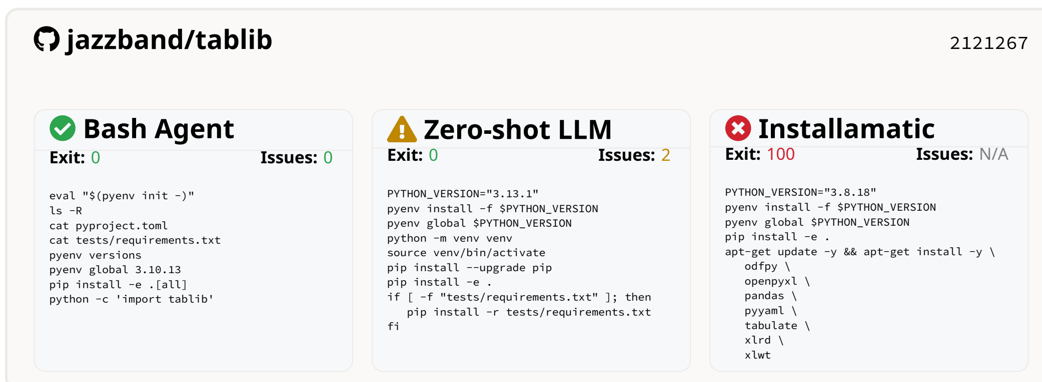

Figure 2: Baselines comparison for the Tablib repository. GPT-4o-mini is used for all baselines. The commented lines have been removed for brevity.

The `tablib` repository represents a typical Python project setup scenario where:

- No explicit installation instructions are provided in the README
- No separate `setup.py` or installation script exists
- Dependencies and project metadata are managed through `pyproject.toml`
- Optional dependencies are defined as "extras" that can be installed with `.[group_name]`

This common configuration requires the installation tool to properly parse the `pyproject.toml` file to determine Python version requirements and dependencies.

The Bash Agent successfully installs the package by:

- Reading `pyproject.toml` to determine Python version requirements (≥ 3.9)
- Installing all optional dependencies via `.[all]`, which includes all file format support
- Using minimal necessary steps without superfluous operations

The Zero-shot LLM approach finishes without errors but misses optional dependencies:

- Uses Python 3.13.1, which is compatible (≥ 3.9)
- Adds unnecessary complexity with virtual environment creation and pip upgrade
- Locates and installs test requirements that are not needed in this case

The Installamatic Agent fails (exit code 100) by:

- Using Python 3.8.18, which violates the package's requirement of ≥ 3.9
- Installing dependencies through apt instead of using pip, which results in a non-zero exit code

This example illustrates common challenges in automated environment setup for Python projects. Even with a relatively standard project structure using modern tooling (`pyproject.toml`), multiple failure points exist around version compatibility, dependency resolution, and installation method

selection. The lack of standardized installation procedures across Python projects, combined with the variety of dependency management approaches, makes automated setup a complex task requiring careful consideration of project-specific requirements and configurations.

C.2 Costs

Table 4 presents the token usage and cost statistics for our experiments, calculated using OpenAI API prices as of February 6, 2025. Analysis of these statistics reveals two key patterns.

Agent vs Zero-shot Token Usage. Agent-based approaches consume 5-10x more tokens compared to zero-shot LLM approaches, due to their need to reason about the environment state and process command outputs.

Language-specific Differences. JVM projects require significantly more tokens than Python projects across all approaches. This disparity primarily stems from the more verbose build and dependency resolution logs produced by JVM environments.

Despite these variations in token consumption, the overall costs remain reasonable - even the most token-intensive agent-based approach averages only $0.25 per repository.

	Baseline	Model	# tokens		Cost	
			Avg.	Total	Avg.	Total
JVM	Zero-shot LLM	GPT-4o	15.6k	10.1M	$0.042	$26.98
		GPT-4o-mini	15.5k	10.2M	$0.002	$1.60
	Bash Agent	GPT-4o	77k	51M	$0.20	$132.8
		GPT-4o-mini	135k	90M	$0.02	$13.8
	Installamatic Agent	GPT-4o	98k	65M	$0.25	$168.7
		GPT-4o-mini	56k	37M	$0.01	$6.1
Python	Zero-shot LLM	GPT-4o	11.0k	3.6M	$0.030	$10.00
		GPT-4o-mini	10.8k	3.6M	$0.002	$0.57
	Bash Agent	GPT-4o	59k	18M	$0.15	$47.4
		GPT-4o-mini	97k	32M	$0.01	$4.9
	Installamatic Agent	GPT-4o	57k	19M	$0.15	$50.1
		GPT-4o-mini	37k	12M	$0.01	$2.0

Table 4: Usage statistics. Avg. refers to average number per one repository.

C.3 Exit Codes

We report the results of the considered baselines in terms of exit codes of the produced environment setup scripts in Table 5. Here, we also include the results from the simple deterministic scripts described in Section 3.3 for comparison. We observe that the scripts finishing execution prematurely with a non-zero exit code is a relatively frequent issue for the considered baselines. For JVM, the deterministic script failed for only 1.35% of repositories, while LLM-based scripts showed higher failure rates, ranging from 17.74% (Bash Agent with GPT-4o) to 81.20% (Zero-shot LLM with GPT-4o). For Python, the deterministic script failed for 28.57% of repositories, but Bash Agent with GPT-4o reduced this to 5.78%, outperforming it by 22.79%.

We qualitatively explore a small sample of the generated environment setup scripts to identify possible root causes. For Python, the failures are at times due to the missing system dependencies or a mismatch in Python versions (*e.g.*, if a project relies on an older Python version than the one used on the system and vice versa). Similarly, mismatch in Java versions is a common cause for a non-zero exit code for JVM environment setup scripts. For both languages, the most common reason why the scripts produced by Zero-shot LLM and Installamatic Agent fail is the usage of tools that are

Baseline	Model	Exit Codes (JVM)		Exit Codes (Python)	
		Zero ↑	Non-Zero ↓	Zero ↑	Non-Zero ↓
Deterministic Script	—	**98.65%** 656/665	**1.35%** 9/665	71.43% 235/329	28.57% 94/329
Zero-shot LLM	GPT-4o	18.80% 125/665	81.20% 540/665	47.42% 156/329	52.58% 173/329
	GPT-4o-mini	27.97% 186/665	72.03% 479/665	55.62% 183/329	44.38% 146/329
Installamatic Agent	GPT-4o	33.83% 225/665	66.17% 440/665	50.76% 167/329	49.24% 162/329
	GPT-4o-mini	27.82% 185/665	72.18% 480/665	41.64% 137/329	58.36% 192/329
Bash Agent	GPT-4o	82.26% 547/665	17.74% 118/665	**94.22%** 310/329	**5.78%** 19/329
	GPT-4o-mini	77.29% 514/665	22.71% 151/665	93.31% 307/329	6.69% 22/329

Table 5: Distribution of the exit codes for the environment setup scripts produced by the considered baselines and the deterministic scripts described in Section 3.3. Zero exit code indicates that the environment setup script finished without errors, however, it is not enough to consider a repository to be set up correctly. The number of successfully set up repositories is reported in Table 1. By design of our benchmark, the deterministic scripts successfully set up *none* of the repositories. The symbol ↑ indicates that higher values in the current column are better, while ↓ indicates that lower values are better.

unavailable on the system. Compared to those two approaches, Bash Agent can receive immediate error feedback and either install the required tool or consider using another. However, we still observe that it sometimes fails to recover and continues trying to attempt the same failing action repeatedly.

C.4 Shell Scripts Analysis

Baseline	Model	Avg. # Lines		Avg. Execution Time	
		JVM	Python	JVM	Python
Zero-shot LLM	GPT-4o	53.36	56.78	176.22	302.35
	GPT-4o-mini	33.59	40.46	221.09	294.32
Installamatic Agent	GPT-4o	26.83	29.08	97.22	270.95
	GPT-4o-mini	33.71	22.13	221.08	225.94
Bash Agent	GPT-4o	13.09	9.64	200.55	180.11
	GPT-4o-mini	18.30	14.64	242.84	203.35

Table 6: Statistics for the environment setup shell scripts produced by the environment setup baselines. Average execution time is reported in seconds.

In Table 6, we provide additional statistics about the environment setup scripts generated by the considered approaches. The best-performing Bash Agent tends to produce the shortest scripts among considered baselines. Additionally, for Bash Agent with GPT-4o, we provide the list of the most frequently executed Bash commands across all repositories in our dataset (Python in Figure 3, JVM in Figure 4). For both languages, agents actively use file system exploration commands (e.g., `cat` and `ls`). There are also a lot of language-specific commands: Python agent uses `pyenv`, `pip`, `python` and `poetry` a lot, while JVM agent runs `sdk`, `./gradlew` and `maven`.

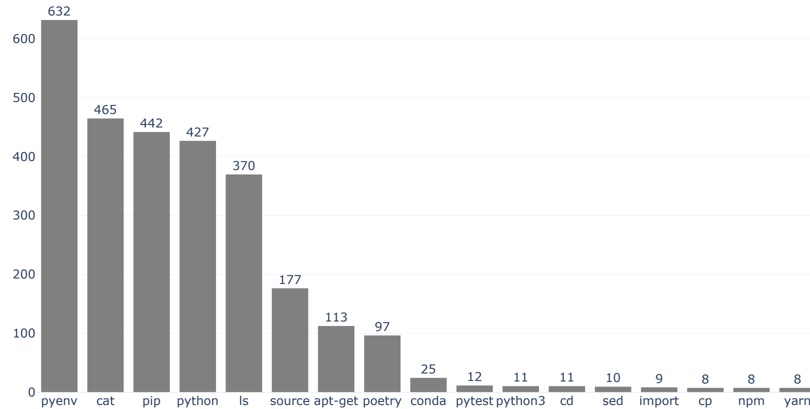

Figure 3: Most frequent Bash commands executed by Bash Agent with GPT-4o on Python dataset.

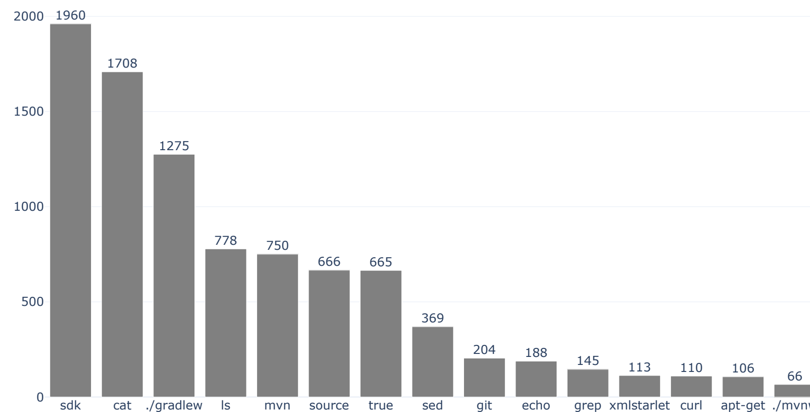

Figure 4: Most frequent Bash commands executed by Bash Agent with GPT-4o on JVM dataset.

D Expert-produced Scripts

We manually investigate the robustness of our proposed environment setup metric for Python on a small sample. Specifically, two authors with professional Python software development experience produce curated scripts for 30 randomly selected repositories from our Python sample, and we run our evaluation suite (Section 3.2) with those scripts. The results are presented in Table 7, and the exact outcomes for each repository are available in Table 8. For all the considered repositories the expert scripts finished with zero exit code and achieved **pass@1** of 66.7%[2] and **avgErrs** of 9.8, outperforming all considered environment setup baselines. Finally, we employ bootstrap resampling (10,000 iterations) to mitigate the small size of the manually processed sample and share the histogram of the distribution of the **avgErrs** for expert-produced scripts and for Bash Agent with GPT-4o-mini in Figure 5, which further confirms a wide gap between manually curated scripts and scripts generated by automatic environment setup methods.

[2]Issues that prevented successful setup include the presence of obsolete dependencies (*e.g.*, a legacy Python 2.x module in a Python 3.x codebase) and dynamically resolved imports can't be correctly processed via a static type checker.

Baseline	Model	pass@1	avgErrs	Zero Exit Code
Expert	—	**66.7%**	**9.8**	**100%** 30/30
Zero-shot LLM	GPT-4o	10.0%	12.0	43% 13/30
	GPT-4o-mini	6.7%	26.1	43% 13/30
Installamatic Agent	GPT-4o	6.7%	34.9	57% 17/30
	GPT-4o-mini	6.7%	52.9	33% 10/30
Bash Agent	GPT-4o	0.0%	42.8	93% 28/30
	GPT-4o-mini	10.0%	16.9	93% 28/30
Deterministic Script	—	0%	45.9	67% 20/30

Table 7: The results of the environment setup baselines, deterministic script and expert-produced environment setup scripts for 30 randomly sampled Python repositories.

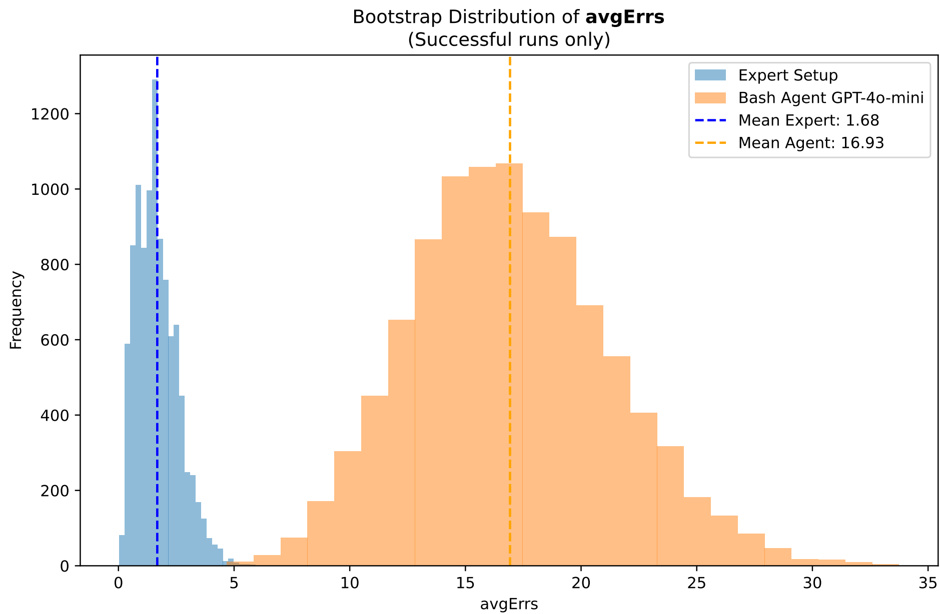

Figure 5: Histograms for **avgErrs**—the average number of missing errors per repository—for expert-produced scripts and for scripts from Bash Agent with GPT-4o-mini obtained via bootstrap resampling (10,000 iterations).

Repository	Expert	Zero-shot LLM		Installamatic Agent		Bash Agent	
		GPT-4o-mini	GPT-4o	GPT-4o-mini	GPT-4o	GPT-4o-mini	GPT-4o
biopsykit	2	-	-	-	-	4	2
cookiecutter	0	-	-	-	60	0	51
client	8	36	-	-	-	36	82
mov-cli	0	-	5	5	5	5	5
section-properties	0	95	-	4	-	6	6
skrub	1	-	36	-	9	56	96
python-holidays	0	0	0	0	0	0	14
guardrails	4	-	-	-	41	41	41
hydra	25	84	-	84	26	25	81
duckdb_engine	1	-	-	-	3	3	3
pytest-xdist	0	-	-	-	0	1	1
lobsterpy	0	0	0	-	-	7	8
lhotse	0	61	-	-	-	61	61
mpmath	0	1	1	-	-	1	1
spectrum-access-system	2	-	-	-	-	39	39
adaptix	0	-	0	0	-	94	144
ansible-zuul-jobs	0	-	4	-	6	6	6
photutils	0	5	-	142	-	2	138
stopstalk-deployment	248	-	-	-	-	-	-
cheroot	0	-	-	27	-	-	-
unixmd	0	-	-	-	50	13	2
extension-helpers	0	-	3	-	3	3	3
elife-bot	0	-	-	-	265	1	265
spotpy	2	11	-	-	-	8	60
bread	0	-	8	9	9	5	5
hazm	2	-	79	-	-	19	19
django-registration	0	1	1	-	1	1	1
custodian	0	7	9	17	106	9	13
lifelines	0	23	-	241	3	0	23
smart_open	0	15	10	-	7	28	28

Table 8: The number of missing imports for 30 randomly samples Python repositories for the expert-produced scripts and environment setup baselines.