DI-BENCH: Benchmarking Large Language Models on Dependency Inference with Testable Repositories at Scale

Anonymous ACL submission

Abstract

Large Language Models have advanced automated software development, however, it remains a challenge to correctly infer dependencies, namely, identifying the internal components and external packages required for a repository to successfully run. Existing studies highlight that dependency-related issues cause over 40% of observed runtime errors on the generated repository. To address this, we introduce DI-BENCH[1], a large-scale benchmark and evaluation framework specifically designed to assess LLMs' capability on dependency inference. The benchmark features 600 repositories with testing environments across Python, C#, Rust, and JavaScript. Extensive experiments with textual and execution-based metrics reveal that the current best-performing model achieves only a 42% execution pass rate, indicating significant room for improvement. DI-BENCH establishes a new viewpoint for evaluating LLM performance on repositories, paving the way for more robust end-to-end software synthesis.

1 Introduction

Large Language Models (LLMs) have revolutionized automated software development, scaling from function-level code completion (GitHub, 2023) to repository-level code synthesis (Wang et al., 2024; Qian et al., 2024; Ibrahimzada et al., 2024). A pivotal yet often overlooked step in this process is ensuring that generated repositories are fully executable. This requires accurate inference and integration of all necessary dependencies, both internal (across project components) and external (from package ecosystems). Without robust dependency inference, even the most advanced code generation solutions risk failing at runtime, impeding further iteration, evaluation, and reliable deployment.

As illustrated in Figure 1, dependency inference involves understanding the intricate relationships within the codebase and mapping out the external packages required for execution. Such dependencies are typically documented in configuration files that may vary from language to language (see Appendix C). Correctly reconstructing these relationships is a foundational capability: it not only ensures that code generation tools produce functional and self-contained repositories, but it also informs deeper reasoning about project architecture and build systems (PyPI, 2024; crates.io, 2024). Consequently, mastering dependency inference is a critical leap forward for enabling robust, end-to-end software synthesis and maintenance.

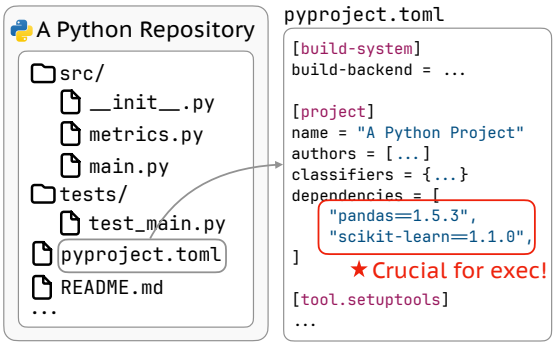

Figure 1: An example of Python project dependencies.

Despite the significance of dependency inference, current LLM-based approaches struggle in this area. Works like ChatDev (Qian et al., 2024) and DevBench (Li et al., 2024a)—pioneers in repository-level generation using multi-agent LLM systems—have reported that dependency-related issues (e.g., missing or incorrectly specified modules) account for over 50% of their observed runtime errors. MetaGPT (Hong et al., 2024) also demonstrates that missing or incorrectly generated dependencies represent one of the most significant hallucinations when LLMs attempt to generate the entire project. These challenges highlight the difficulty that state-of-the-art models face in accurately navigating build systems and package repositories. Although existing repository-level bench-

[1]code: https://github.com/DIBench/DIBench

marks such as SWEBench (Jimenez et al., 2023), RepoBench (Liu et al., 2023), and DevBench (Li et al., 2024a) offer valuable insights into a model's ability to handle large contexts and generate code at scale, none focuses on systematically evaluating dependency inference capabilities.

To address this critical gap, we introduce DI-BENCH, the first comprehensive repository-level benchmark dedicated to dependency inference. DI-BENCH comprises 600 verified repositories, including 400 regular-sized and 200 large-sized, across four popular programming languages (Python, C#, Rust, and JavaScript). Each repository is carefully curated to assess a model's ability to identify both internal and external dependencies. We pair this dataset with a rigorous, multi-faceted evaluation framework. Beyond measuring textual matching accuracy between model-generated and ground-truth dependencies, we propose a novel CI-based execution evaluation by reusing each repository's intrinsic Continuous Integration (CI) pipelines as automated test harnesses. This approach enables scalable and objective assessment of end-to-end executability, eliminating the costly and error-prone need for manual environment setup.

Through comprehensive experiments involving various LLMs and prompting strategies, we observed that even the best-performing LLM achieved only a 42% executability rate. This finding highlights *significant room for future improvement* in this area. Further analysis revealed that several factors influence performance, including the dependency amount and repository size. Notably, issues such as hallucination and challenges related to dependency metadata emerged as critical bottlenecks that adversely affect model performance.

In summary, our contributions are as follows:

- **DI-BENCH Benchmark:** We introduce a pioneering, large-scale, dependency-focused benchmark featuring 600 repositories spanning 4 popular programming languages. It establishes a new standard for evaluating LLMs' capabilities in realistic, repository-scale scenarios.

- **Dual-Use CI Infrastructure:** We leverage CI workflows not only to identify executable repositories during dataset curation but also to serve as a reliable, fully automated test environment. By using CI pipelines, we ensure that dependency checks remain robust, scalable, and faithful to real-world development practices.

- **Granular Evaluation Metrics:** We combine coarse-grained runtime executability measures with fine-grained precision and recall on inferred dependencies. This dual-layered approach enables systematic analysis of both functional correctness and textual accuracy with richer insights.

By spotlighting dependency inference and offering a dedicated benchmark, our work lays the foundation for advancing LLMs toward robust, end-to-end repository-level software synthesis.

2 Related Works

Repository-level coding tasks have attracted increasing attention in recent years. Many benchmarks (Zhang et al., 2023; Liu et al., 2023; Ding et al., 2023) center on code completion tasks at various granularities - from individual lines and API calls to entire function implementations. SWE-Bench and its variant (Jimenez et al., 2023; Yang et al., 2024) challenge LLMs and LLM-powered systems with real-world scenarios, using issues and pull requests from popular Python repositories on GitHub. While these benchmarks focus on assistant-like tasks, recent work explores LLMs' capabilities in complete project generation. DevBench (Li et al., 2024a) decomposes the development process into distinct stages and evaluates AI performance at each stage. Agent-As-a-Judge (Zhuge et al., 2024) introduces DevAI, innovatively employing LLM agents as evaluators of development outcomes.

However, existing works have not adequately addressed build configuration evaluation: code completion tasks (Zhang et al., 2023; Liu et al., 2023; Ding et al., 2023) do not include build files in their generation targets, and issue-fixing benchmarks like SWE-Bench (Jimenez et al., 2023) contain only 1% of patches related to build configurations. In repository-level code generation tasks (Li et al., 2024a; Zhuge et al., 2024), build file generation is merely treated as one subtask without specialized evaluation. Prior research in dependency inference (Ye et al., 2022; damnever, 2024) has predominantly focused on Python ecosystems using traditional program analysis techniques, while lacking broader language coverage. Our study fills in this gap by providing a benchmark specifically designed for evaluating dependency inference capability across multiple mainstream languages.

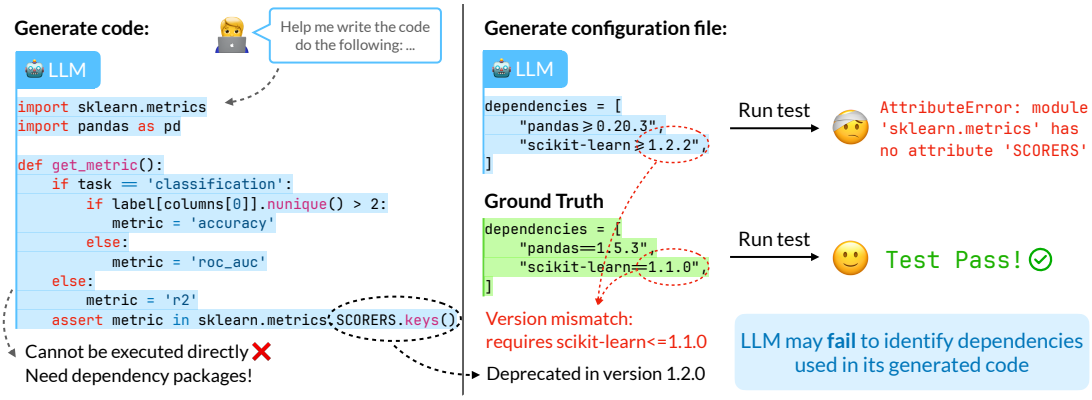

Figure 2: An example of incorrectly identifying dependencies used in code.

3 Dependency Inference

Although many studies focus on repository code generation with LLMs recently, there exists a significant gap between the generated code and the *executable* and *operational* software, *Dependency*. In this paper, we adapt *Dependency Inference*, which aims to generate a list of dependencies based on the source code. As shown in Figure 2, the code generated by LLM cannot be executed directly without installing the required dependencies; However, it is non-trivial to identify the correct dependencies using LLMs. The illustrated example shows that the LLM generates dependencies with a wrong version (should be '*scikit-learn==1.1.0*' rather than '*scikit-learn>=1.2.2*'), resulting in execution failure.

Automatic and accurate dependency inference makes end-to-end code development possible by installing the inferred dependencies for execution. Furthermore, it can enable key scenarios like fully-automated evaluation and iterative code improvement with execution feedback. Besides the repository, dependency inference can be also applied to small code snippets like Python Notebook, incremental code changes, and *etc*.

Formally, the task is formulated as below: Given a software repository containing many source code files and build configuration files where dependency-related sections are masked, the dependency inference task aims to generate a list of inferred dependencies to fill into the configuration. Formally, we define the task as:

$$\mathcal{F} : (R, \{b_1^m, b_2^m, ..., b_k^m\}) \rightarrow \{b_1, b_2, ..., b_k\}, \quad (1)$$

where R denotes the repository including all source files, b_i^m is a build configuration file with dependency masked/removed, b_i is a build configuration with the inferred dependencies. For example, in a Python project, given a `pyproject.toml` file with masked dependency sections and all source code files, the task is to edit `pyproject.toml` file to specifying all dependencies required by the project.

4 DI-BENCH

Focused on the task of *dependency inference*, we introduce DI-BENCH, a meticulously curated, large-scale benchmark dataset and evaluation framework at the repository level. DI-BENCH encompasses 600 real-world, testable repository instances across 4 programming languages, providing a comprehensive platform for assessing LLM-based methods in identifying and managing repository dependencies.

4.1 Statistics & Features

DI-BENCH's instances, sourced from real-world repositories, are categorized into two subsets based on repository size: *regular* and *large*. The *regular* subset includes repositories with fewer than 120k tokens[2], ensuring they fit within the context length limits of recent LLMs. It comprises 400 instances (100 per language) with an average of 12.1 dependencies. The *large* subset consists of 200 repositories (50 per language) with more than 120k tokens and the average dependency count is 29.3. Table 2 provides detailed statistics of DI-BENCH, while Figure 6 illustrates the overall distribution of token and dependency counts using Kernel Density Estimation (KDE) curves. The dataset exhibits a wide size distribution, with smaller repositories being more prevalent. Table 1 provides a comparative analysis of the features distinguishing DI-BENCH from existing code task benchmarks. The unique attributes of DI-BENCH include:

Beyond Code. DI-BENCH focuses on a crucial

[2]Token counts are calculated using the Llama 3.2 tokenizer.

Table 1: Comparison of features between existing benchmarks and DI-BENCH

Benchmark	Task	Evaluation	Scope	Languages	#Repo	Curation
MBPP (Austin et al., 2021)	Code Generation	Unit Tests	Function	Python	N/A	Manual
HumanEval (Chen et al., 2021)	Code Generation	Unit Tests	Function	Python	N/A	Manual
ClassEval (Du et al., 2023)	Code Generation	Unit Tests	Class	Python	N/A	Manual
RepoEval (Zhang et al., 2023)	Code Completion	Textual & Unit Tests	Repo-level	Python	14	Manual
RepoBench (Liu et al., 2023)	Retrieval & Completion	Only Textual	Repo-level	Python, Java	1,669	Automated
CrossCodeEval (Ding et al., 2023)	Code Completion	Only Textual	Repo-level	Python, Java, C#, TS	1,002	Automated
EvoCodeBench (Li et al., 2024b)	Code Generation	Unit Tests	Repo-level	Python	25	Automated
RepoMasterEval (Wu et al., 2024)	Code Completion	Unit Tests	Repo-level	Python, TS	6	Manual
DI-BENCH	**Dependency Inference**	**Textual & Test Suite**	**Repo-level**	**Python, Rust, C#, JS**	**600**	**Automated**

Table 2: Statistical summary of DI-BENCH

Subset	Lang	#Files	#LoC	#Tokens	#Deps.	#Tests
Regular	Python	31.1	3.1K	31K	5.9	46.6
	Rust	20.0	3.4K	32K	10.8	21.0
	C#	69.7	4.1K	39K	26.2	29.8
	JS	15.1	1.6K	15K	5.6	42.0
	Avg.	*33.9*	*3.0K*	*29K*	*12.1*	*34.9*
Large	Python	268.3	45.6K	519K	11.8	547.3
	Rust	94.3	23.6K	279K	45.2	153.4
	C#	252.2	23.9K	238K	43.4	132.6
	JS	139.8	26.6K	383K	15.9	291.1
	Avg.	*214.1*	*33.3K*	*387K*	*29.3*	*281.1*

challenge in real-world software development: dependency inference. This essential aspect is often overlooked in existing studies.

Test Execution. DI-BENCH not only evaluates result correctness through textual matching but also executes project test suites, providing a straightforward and reliable evaluation.

Practical and Verified. The repository instances included in DI-BENCH are sourced from real-world projects on GitHub, thus making the benchmark both practical and challenging. Each project undergoes verification to ensure its validity.

Diverse Long Inputs. The dataset includes two subsets, regular and large, with a wide distribution of context lengths, ranging from small repositories with a few files to large projects with over 200 files.

Continually Updatable. We have developed a dataset curation pipeline that is fully automated, scalable, and continuously updatable, eliminating the need for manual annotation to set up environments and run tests.

Open Solution. Our evaluation framework features two complementary datasets: while both the regular and large sets welcome various approaches including language models and agentic systems, the large set presents additional challenges of model context limits, specifically motivating the exploration of novel methodologies.

4.2 Dataset Construction

Creating a dataset that supports execution-based evaluation at the repository level is challenging. Previous works often involve manual setting up environments and writing test scripts, which can require significant human and engineering effort and cannot scale up to larger datasets. As shown in Table 1, the existing largest repository-level benchmark supporting test execution contains only 25 repositories. To address this, we leverage GitHub Actions (GitHub, 2024)—a widely used continuous integration (CI) tool that allows developers to automate test execution through YAML configuration files. By reusing these developer-written CI workflows within repositories, we propose an automated curation pipeline that eliminates human engagement during the benchmark construction, ultimately resulting in a dataset of 600 testable repositories—24 times larger than the largest previous benchmark. With the large-scale dataset, we can provide more generalizable insights and more robust evaluations. Figure 3 illustrates steps to construct DI-BENCH with details listed below.

Repository Crawling. The goal of this phase is to collect GitHub repositories that meet the following criteria: 1) Written in one of the four programming languages: Python, C#, Rust, or JavaScript (The characteristics of these languages and their dependency configurations are detailed in Appendix C). These languages are popular, possess a standardized dependency packages ecosystem, and have clear standards for specifying dependencies. 2) Have more than 100 stars, serving as a quality filter criterion. 3) Repository size is less than 10MB to avoid extremely large repositories and maintain a manageable dataset size. 4) Most importantly, the repository must have GitHub Actions enabled, indicated by the presence of the `.github/workflows` folder. Repositories that meet these criteria proceed as candidate repos-

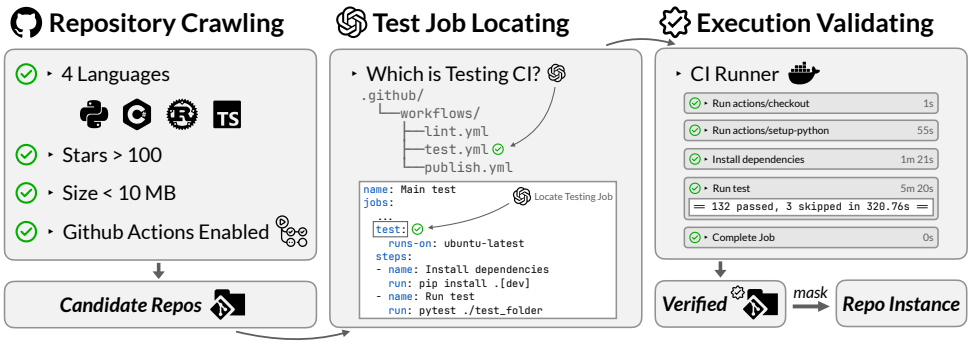

Figure 3: CI-based curation pipeline for DI-BENCH.

itories into subsequent phases.

Test Job Locating. Repositories often define multiple workflows to perform tasks unrelated to testing, such as linting and publishing. These tasks may also be defined within different jobs in the same workflow configuration file. Due to the lack of a specific naming convention, we introduce an LLM-assisted locating process to identify the specific jobs responsible for executing project tests. At the execution stage, only the test job will be run.

Execution Validating. We use `act` (nektos, 2024) as a local runner for GitHub Actions, enabling local execution of repository testing CI. In this phase, by executing the test jobs of candidate repositories, we obtain those that successfully follow the workflow and pass all tests as expected. The validation phase ensures that the selected repositories are correct and executable. This further highlights the advantage of our proposed CI-based testing approach: fully automated and scalable.

Dependency Masking. After the validation, we utilized an automated script to remove the sections specifying dependencies in the configuration files. We further performed sanitization by removing any existing dependency lock files (e.g., in JavaScript) to prevent potential ground truth leakage and ensure proper execution. This process ultimately produced the instances included in DI-BENCH.

5 Experiment Setup

This section provides a detailed description of the experimental settings, including the LLMs, baseline methods, and evaluation metrics.

Baseline Methods. We designed three baseline systems with various prompting strategies to evaluate how LLMs perform in the *dependency inference* task, intentionally avoiding complex techniques such as agent-based methods.

- All-In-One: The approach concatenates all the source code of a repository into a single query for model generation. It serves as a straightforward yet computationally intensive baseline.
- File-Iterate: The method processes each individual file in the repository to generate dependencies, with the results subsequently aggregated to feed into the model for generating the final output. This simulates a modular and distributed reasoning approach.
- Imports-Only: The approach collects all import-related statements from the code base as the input context to LLMs using *tree-sitter* (tree-sitter, 2024). For Python and JavaScript, we extract all `import` statements; for C# and Rust, we extract all `use` statements. More details about tree-sitter are provided in the Appendix D.4.

Metrics. we use textual and execution-based metrics, and the fake rate in evaluation.

- Textual Accuracy: This metric assesses whether the generated dependencies align with the ground truth from a textual matching perspective. We compute the *Precision* (ratio of correct dependencies among model-generated ones), *Recall* (Ratio of correct dependencies among all ground-truth ones), and *F1* (The harmonic mean of the above).
- Executability Rate: This metric measures whether the project can be successfully built and executed through CI testing pipeline with the generated dependencies. A score of 1 is assigned if all tests passed successfully; otherwise, a score of 0 will be given. Whether the tests pass is the most direct and reliable indicator of the correctness of the generated dependencies.
- Fake Rate: This metric represents the proportion of the generated dependencies that cannot

Table 3: Performance of benchmark methods across programming languages and repository sizes on GPT-4o, where Exec denotes the executability rate, P/R/F1 denote Precision, Recall, and F1-score, FR denotes Fake Rate, which is the lower, the better. Note that the Large repositories cannot fit into the All-In-One method (denoted with '-').

Lang	Method	Regular					Large				
		Exec	P	R	F1	FR	Exec	P	R	F1	FR
Python	All-In-One	42.0	62.6	72.9	67.4	2.7	-	-	-	-	-
	File-Iterate	29.0	39.1	74.3	51.3	4.4	8.0	19.5	35.3	25.1	6.4
	Imports-Only	36.0	57.5	73.9	64.7	3.7	18.0	36.9	46.9	41.3	23.1
Rust	All-In-One	11.0	93.7	74.7	83.2	0.9	-	-	-	-	-
	File-Iterate	8.0	74.8	76.1	75.4	1.2	2.0	45.0	69.1	54.5	6.2
	Imports-Only	4.0	89.0	65.4	75.4	1.1	2.0	84.9	51.0	63.7	12.1
C#	All-In-One	13.0	60.6	39.5	47.8	3.5	-	-	-	-	-
	File-Iterate	5.0	28.0	34.1	30.8	6.5	0.0	20.4	33.0	25.2	6.0
	Imports-Only	3.0	52.4	29.5	37.8	5.0	0.0	49.1	19.2	27.6	6.3
JavaScript	All-In-One	42.0	86.4	66.7	75.3	4.6	-	-	-	-	-
	File-Iterate	32.0	52.0	61.2	56.3	2.9	16.0	33.6	54.5	41.6	3.5
	Imports-Only	22.0	73.0	45.7	56.2	6.0	8.0	47.9	17.5	25.7	2.8

be found in the package ecosystem (for external dependencies) or in the local repository directory (for internal dependencies). It highlights the hallucination issue in LLMs, where non-existent dependencies or versions are generated.

Models. Since code repositories are usually very long, we choose popular LLMs that support 128k context windows as the backbone models, including GPT-4o (OpenAI, 2024b), GPT-4o-mini (OpenAI, 2024a), Llama 3.1-Instruct (Grattafiori et al., 2024), DeepSeek-Coder-V2-Lite-Instruct (MoE) (Guo et al., 2024) and Qwen-Coder-V2.5-Instruct series (Hui et al., 2024).

6 Experimental Results

6.1 Performance of Baseline Methods

We start by conducting preliminary experiments utilizing three baseline systems — All-In-One, File-Iterate, and Imports-Only on GPT-4o and GPT-4o-Mini (Table 7 in Appendix B.1), encompassing both the regular and large subsets. Table 3 shows the results with several key insights:

Challenging Nature of Dependency Inference Dependency inference presents a significant challenge for contemporary LLMs. In the regular subset (< 120k tokens), even the best-performing models achieved executability rates of below 50% for scripting languages such as Python and JavaScript and only around 10% for compiled languages like Rust and C#. These findings underscore the limitations of current models in accurately inferring dependencies in various languages.

Impact of Repository Size Large repositories, characterized by extensive contexts and complex dependency structures, are more challenging for dependency inference. Executability rates in the large subset were markedly lower across all baseline methods compared to the regular subset. For File-Iterate and Imports-Only, the performance gap was especially evident, indicating the difficulty of adapting these methods to large repositories.

Importance of Models and Prompting Strategies The choice of backbone LLMs and the construction of prompts play crucial roles in determining performance outcomes. For instance, the All-In-One approach with GPT-4o, by merging the entire code base into a single query, consistently outperformed other methods on executability. However, this approach does not work for larger repositories. While the File-Iterate and Imports-Only methods can process large repositories, their performance significantly declined without the full code context. The finding reveals the trade-off between prompting strategies and repository sizes to achieve optimal performance on dependency inference.

Hallucination Issues A recurring issue across all methods was the generation of hallucinated dependencies, i.e., non-existent packages or versions, as indicated by the Fake Rate. Specifically, we observed a remarkably higher Fake Rate on large subset with Imports-Only method. In Section 6.3, we will show that the hallucination adversely affected the executability.

Table 4: Model performance across programming languages with the All-In-One approach on DI-BENCH.

Language	Model	Size	Exec	P	R	F1	FR
Python	GPT-4o	-	42.0	62.6	72.9	67.4	2.7
	GPT-4o-mini	-	26.0	57.1	56.3	56.7	1.9
	Llama-3.1-8B-Instruct	8B	13.0	30.1	39.3	34.1	4.2
	DeepSeek-Coder-V2-Lite-Instruct	16B(MoE)	17.0	48.0	44.8	46.3	18.4
	Qwen2.5-Coder-7B-Instruct	7B	22.0	55.7	41.9	47.8	5.3
Rust	GPT-4o	-	11.0	93.7	74.7	83.2	0.9
	GPT-4o-mini	-	7.0	76.1	49.3	59.8	1.1
	Llama-3.1-8B-Instruct	8B	1.0	58.1	38.1	46.0	11.3
	DeepSeek-Coder-V2-Lite-Instruct	16B(MoE)	2.0	75.8	40.6	52.8	2.8
	Qwen2.5-Coder-7B-Instruct	7B	6.0	71.6	41.4	52.5	2.1
C#	GPT-4o	-	13.0	60.6	39.5	47.8	3.5
	GPT-4o-mini	-	4.0	42.1	22.5	29.3	11.5
	Llama-3.1-8B-Instruct	8B	0.0	14.9	8.4	10.7	21.2
	DeepSeek-Coder-V2-Lite-Instruct	16B(MoE)	1.0	33.6	7.0	11.6	9.2
	Qwen2.5-Coder-7B-Instruct	7B	1.0	22.6	17.2	19.6	14.7
JavaScript	GPT-4o	-	42.0	86.4	66.7	75.3	4.6
	GPT-4o-mini	-	17.0	83.3	31.0	45.1	2.4
	Llama-3.1-8B-Instruct	8B	9.0	67.4	16.5	26.6	1.4
	DeepSeek-Coder-V2-Lite-Instruct	16B(MoE)	17.0	81.8	31.1	45.1	2.3
	Qwen2.5-Coder-7B-Instruct	7B	17.0	81.6	42.7	56.1	3.4

6.2 Performance of Varying Models

Due to space constraints, we focus on reporting benchmarking results on the DI-BENCH Regular dataset using the All-In-One approach in the following sections. This method was selected due to its superior performance as shown in Table 3, as well as its simplicity and capability to reflect a zero-shot setting for the dependency inference task.

Specifically, in this section, we evaluate various models and report the results in Table 4. The table reveals that open-sourced models achieve significantly lower performance compared to models like GPT-4o. Notably, the Qwen-7B model demonstrates superior performance across all metrics, outperforming the other two open-sourced models and occasionally even surpassing the GPT-4o-mini model. Additionally, we find that although the open-sourced models can achieve moderate textual accuracy, their executability rate is considerably lower. This highlights the challenge of not only understanding and generating correct textual dependencies but also ensuring that the generated code can be executed successfully.

In addition, we vary the model sizes of the QWen2.5-Coder-Instruct series with the All-In-One method, where the model size ranges from 3B, 7B, 14B to 32B. We observed that the model in general achieves better performance when increasing the model size. The results and more analysis are presented in Appendix B.2

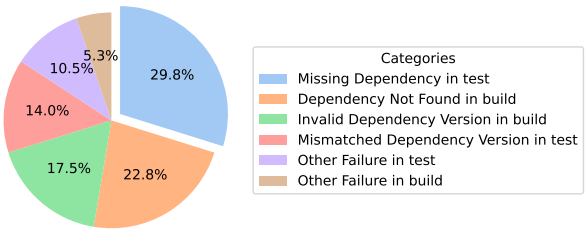

Figure 4: Distribution of failure categories (GPT-4o, All-In-One setting, Python).

Failure Categories and Distribution. To better understand why the execution failed with model-generated dependencies, we manually analyzed the failure cases of GPT-4o, the best-performing model, under the All-In-One setting in Python. As shown in Figure 4, the most common failure category is "*Missing Dependency in Test*", which means that the model missed to generate some dependencies that are required during the testing evaluation. Additionally, "*Dependency Not Found in Build*" and "*Invalid Dependency Version in Build*" also account for a significant proportion. These indicate that the model-generated dependency specifications either include nonexistent packages or specify nonexistent versions, leading to failures when installing generated dependencies.

6.3 Further Analysis and Ablation Study

In this section, we conduct further analysis with focuses on how repository size and the amount

Table 5: Execution success improvement by replacing predicted dependency metadata with oracle metadata.

Language	Exec	Exec (with Orac.)	Δ
Python	42.0	54.0	+28.6%
Rust	11.0	38.0	+245.5%
C#	12.0	15.0	+25%
JavaScript	42.0	65.0	+54.8%

of dependencies affect dependency inference performance, as well as the impact of dependency metadata and the hallucination issue.

Challenges in dependency inference for larger repositories with more dependencies. As illustrated in Figure 5, inference accuracy decreases significantly as the number of dependencies grows. This trend is consistent across all languages, particularly those with complex dependency structures like Rust and JavaScript. The decline in performance is attributed to the difficulty of maintaining accurate dependency mappings as their quantity increases, highlighting spaces for future enhancement of LLMs. Besides, we made further analysis about how the repository size affect the performance on the regular dataset and results are depicted in Figure 8 (Appendix B.3). We observed a negative correlation between repository size and model performance, which aligns with the finding obtained in Table 3. This suggests that long-context reasoning (Hsieh et al., 2024; Bai et al., 2024) remains a significant challenge for LLMs, as longer input contexts lead to increased complexity.

Reasoning the dependency metadata is a bottleneck. In previous experiments, we found that while textual accuracy was relatively high, the executability rate was significantly lower. For example, GPT-4o achieved a precision of 62.6% and recall of 72.9% on Python, while the executability rate was only 42.0%. We suspect this discrepancy arises from incorrect metadata generation in dependencies, such as package version constraints, extra features and so on, (examples can be found in Appendix C). To validate this hypothesis, we replaced the predicted dependencies with oracle metadata and observed a notable increase in the executability rate. As shown in Table 5, the Python executability rate improved from 42.0% to 54.0%, representing a relative increase of 28%. This demonstrates the importance of accurate dependency metadata for successful execution of dependency configurations.

Hallucination hurts the executability. We observed all models generate hallucinated dependencies that do not exist, as indicated by the fake rate. Although the fake rate was relatively low, excluding the hallucinated dependencies can improve the executability, as shown in Table 6. These improvements, though modest, reinforce the need for more accurate dependency predictions. Hallucination issues remain one of the primary obstacles to improving the reliability of dependency inference systems.

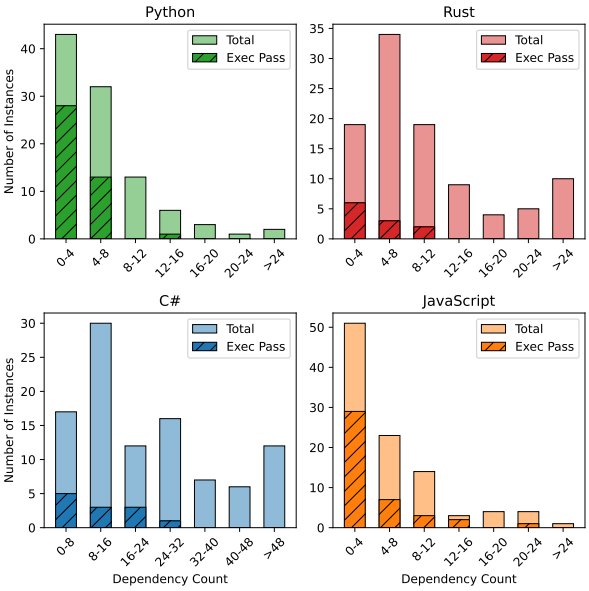

Figure 5: Execution pass rate w.r.t dependency count.

Table 6: Impact of hallucination on exeutability rate.

Language	Exec	Exec w.o. Fake Dep.	Δ
Python	42.0	43.0	+2.4%
Rust	11.0	13.0	+9.1%
C#	12.0	13.0	+8.3%
JavaScript	42.0	42.0	+0%

7 Conclusion

We introduce DI-BENCH, the first benchmark dedicated to dependency inference across 600 repositories in four programming languages: Python, C#, Rust, and JavaScript. In addition to measuring textual accuracy, we propose a novel CI-based evaluation that incorporates actual tests execution. Extensive experiments on various open-source and proprietary LLMs demonstrate that even the most advanced models struggle to infer dependencies accurately, highlighting opportunities for future advancements. We believe this study lays the groundwork for repository-level code development, with dependency inference serving as a pivotal step toward fully automated code generation.

Limitations

Our study acknowledges several limitations. ❶ Due to constraints in computing resources, our evaluation primarily focused on five mainstream models, selecting smaller model sizes. While these models are sufficiently representative, broadening the scope to include a greater variety of LLMs with diverse sizes could potentially enrich our findings. ❷ In our experiments, we employed the GPT-4o and GPT-4o mini models, which operate as black boxes. The outputs may vary due to potential model upgrades or fluctuations in resources. To mitigate this issue, we provide the dates of the model versions used as a reference and set the temperature to 0 to ensure more consistent outputs. ❸ Test coverage for each repository may not be exhaustive, meaning some test cases might not encompass every possible code path. However, as the tests were developed by project contributors, the results are expected to reflect practical settings accurately.

Ethics Considerations

We take ethical considerations very seriously, and strictly adhere to the ACL Ethics Policy. The dataset were collected from open-source GitHub repositories, most of which have clear licenses. While we respect the efforts of each repository author and comply with the respective licenses, we cannot guarantee that all specific requirements of individual repositories have been accounted for. All the data used in our work is publicly accessible and does not involve any ethical concerns.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. 2024. Longbench: A bilingual, multitask benchmark for long context understanding. *Preprint*, arXiv:2308.14508.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

crates.io. 2024. The Rust community's crate registry. https://crates.io/.

damnever. 2024. A tool to generate requirements.txt for Python project, and more than that. https://github.com/damnever/pigar.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Preprint*, arXiv:2310.11248.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

GitHub. 2023. GitHub Copilot – Your AI pair programmer. https://github.com/features/copilot.

GitHub. 2024. GitHub Actions: Automate your workflow from idea to production. https://github.com/features/actions.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, and et al. Abhishek Kadian. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations (ICLR)*.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. 2024. Ruler: What's the real context size of your long-context language models? *Preprint*, arXiv:2404.06654.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Repository-level compositional code translation and validation. *arXiv preprint arXiv:2410.24117*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. 2024a. Devbench: A comprehensive benchmark for software development. *Preprint*, arXiv:2403.08604.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024b. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2404.00599*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

nektos. 2024. act: Run your GitHub Actions locally. https://nektosact.com/.

OpenAI. 2024a. GPT-4o mini: advancing cost-efficient intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/.

OpenAI. 2024b. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/.

PyPI. 2024. Find, install and publish Python packages with the Python Package Index. https://https://pypi.org/.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. Chatdev: Communicative agents for software development. *Preprint*, arXiv:2307.07924.

tree-sitter. 2024. Tree-sitter. https://tree-sitter.github.io/tree-sitter/.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *Preprint*, arXiv:2407.16741.

Qinyun Wu, Chao Peng, Pengfei Gao, Ruida Hu, Haoyu Gan, Bo Jiang, Jinhe Tang, Zhiwen Deng, Zhanming Guan, Cuiyun Gao, et al. 2024. Repomastereval: Evaluating code completion via real-world repositories. *arXiv preprint arXiv:2408.03519*.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. 2024. Swe-bench multimodal: Do ai systems generalize to visual software domains? *Preprint*, arXiv:2410.03859.

Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-based environment dependency inference for python programs. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1245–1256.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570*.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. 2024. Agent-as-a-judge: Evaluate agents with agents. *Preprint*, arXiv:2410.10934.

A Distribution of DI-BENCH Dataset on Token Count and Dependency Amount

Figure 6 illustrates the distribution of token and dependency counts across different programming languages (Python, Rust, C#, and JavaScript) for both Regular and Large repositories. For Regular repositories, the token count distribution shows that Python and Rust have a higher density at lower token counts, indicating that these languages typically have smaller codebases. In contrast, C# and JavaScript display a more spread-out distribution, suggesting a wider range of codebase sizes. When examining Large repositories, the token count distribution shifts substantially, with all languages showing a lower density, highlighting the increased complexity and size of codebases in larger repositories.

The dependency count distribution for Regular repositories reveals that most dependencies are concentrated in the lower range across all languages, with Python and Rust having slightly higher densities at lower counts. For Large repositories, the dependency count distribution shows a similar pattern but with slightly higher densities for C# and JavaScript, indicating these languages tend to have more dependencies in larger codebases.

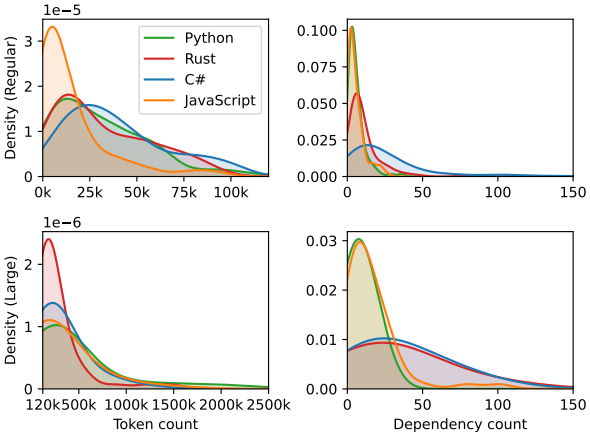

Figure 6: Distribution of token and dependency count.

B Additional Experiments

B.1 Performance of Baseline Methods with GPT-4o-mini

Table 7 presents the performance of various benchmark methods across different languages and repository sizes (Regular and Large) on GPT-4o-mini. Notably, the effectiveness of these methods varies significantly between Regular and Large repositories, with performance generally declining as repository size increases. Python and Rust show relatively higher performance in Regular repositories compared to C# and JavaScript, which struggle more consistently across both repository sizes. Furthermore, the Imports-Only method for Python and File-Iterate method for Rust stand out with comparatively better performance in Regular repositories. The results indicate that while some methods perform well in smaller repositories, there is a significant drop in effectiveness in larger repositories, underscoring the importance of optimizing methods to handle different repository scales efficiently. The conclusion aligns with the findings we obtained in Section 6.1. Besides, the variability suggests that a one-size-fits-all approach is insufficient, and tailored strategies are necessary to maintain high performance across different contexts.

B.2 Performance When Varying the Model Size

Figure 7 presents the performance of All-In-One approach on Regular dataset with different sizes of Qwen2.5-Coder-Instruct models. We observed a general trend where larger models consistently improved executability and textual accuracy metrics across four languages. Besides, when increasing the model size for compiled languages ike Rust and C#, textual accuracy increases sharply, but the executability remains relative low, demonstrating the great value of our execution-based evaluation in benchmarking.

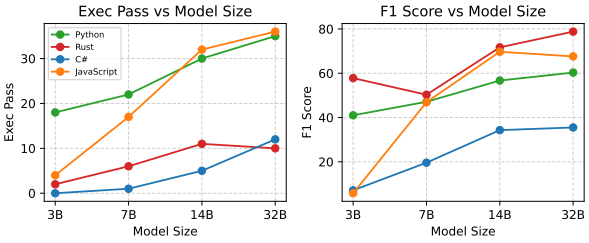

Figure 7: Model performance across programming languages for Qwen2.5-Coder-Instruct

B.3 Performance When Varying the Repository Size

In Table 3, we can observe that that dependency inference performance deteriorates when applied to larger datasets. However, whether this finding is generally applicable is unconfirmed. Therefore, we conducted a further analysis within the regular dataset which contains repositories of varying

Table 7: Performance of benchmark methods across programming languages and repository sizes on GPT-4o-mini (Continue to Table 3 with a different model)

Lang	Method	Regular					Large				
		Exec	P	R	F1	FR	Exec	P	R	F1	FR
Python	All-In-One	26.0	57.1	56.3	56.7	1.9	-	-	-	-	-
	File-Iterate	21.0	42.6	62.4	50.7	2.7	14.0	31.0	27.6	29.2	4.2
	Imports-Only	30.0	59.4	60.2	59.8	1.7	18.0	45.4	32.3	37.7	3.1
Rust	All-In-One	7.0	76.1	49.3	59.8	1.1	-	-	-	-	-
	File-Iterate	4.0	75.2	60.8	67.2	1.6	0.0	36.5	45.6	40.5	4.4
	Imports-Only	1.0	77.9	49.9	60.8	1.2	0.0	63.9	23.9	34.8	4.0
C#	All-in-One	3.0	41.1	18.1	25.2	12.5	-	-	-	-	-
	File-Iter	3.0	25.4	22.9	24.1	15.2	0.0	19.2	13.6	15.9	5.8
	Pattern-Retrieve	3.0	44.1	23.3	30.5	6.4	0.0	34.7	14.1	20.1	6.2
JavaScript	All-In-One	18.0	83.3	31.0	45.1	2.4	-	-	-	-	-
	File-Iterate	17.0	45.1	26.2	33.1	6.7	2.0	26.4	20.4	23.0	3.1
	Imports-Only	13.0	65.4	18.9	29.3	1.2	4.0	54.5	11.3	18.8	1.2

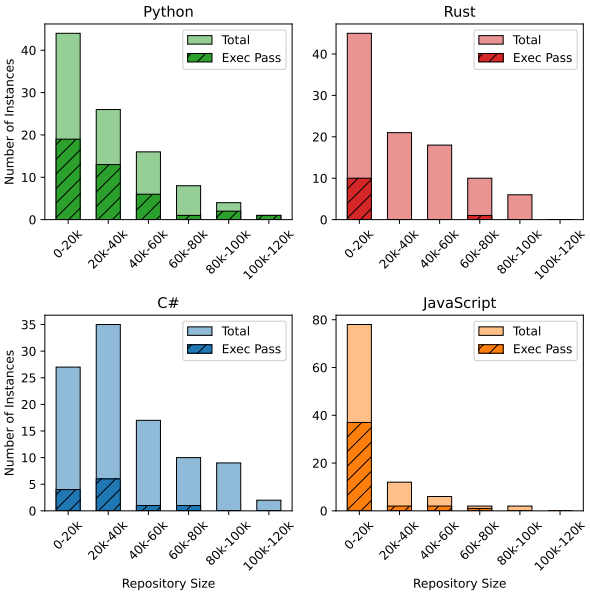

Figure 8: Execution pass rate w.r.t repository size

sizes. The results are depicted in Figure 8, showing a decline in executability rates as repository size increases. Hence, there exists a negative correlation between repository size and model performance both between and within datasets. This suggests that long-context reasoning remains a significant challenge for LLMs, as longer input contexts lead to increased complexity in managing the project dependencies. This finding aligns with previous studies on long-context reasoning (Hsieh et al., 2024; Bai et al., 2024).

C Example of Configuration Files

This section introduces the types of configuration files for the four languages involved in this paper. These files specify project dependencies and serve as carriers for storing inference results. They also exercise the capabilities of LLMs to interact with modern programming languages build systems, it is important to provide a clear demonstration here.

Python (Figure 9) `pyproject.toml` is the configuration file used by most Python projects. It includes sections for specifying metadata such as package names and authors, defining project dependencies, and configuring various development tools.

```toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "spam-eggs"
version = "2020.0.0"
dependencies = [
  "httpx",
  "gidgethub[httpx]>4.0.0",
  "django>2.1; os_name != 'nt'",
  "django>2.0; os_name == 'nt'",
]
requires-python = ">=3.8"
authors = [
  {name = "Pradyun Gedam", email =
  "pradyun@example.com"},
]

[project.optional-dependencies]
gui = ["PyQt5"]

[project.urls]
Homepage = "https://example.com"

[project.scripts]
spam-cli = "spam:main_cli"
```

Figure 9: An example of `pyproject.toml` in Python

Rust (Figure 10) `Cargo.toml` is the configuration file used in Rust projects. A single repository may contain multiple local crates, each with its own `Cargo.toml`, requiring proper configuration of internal dependency references.

```toml
[package]
name = "example_project"
version = "0.1.0"
authors = ["Your Name <your.email@example.com>"]
categories = ["CLI", "web"]
workspace = "./workspace"

[dependencies]
# External Dependency
serde = { version = "1.0", features = ["derive"]}
tokio = { version = "1.0", features = ["full"]}
regex = "1.5"
# Internal Dependency
my_local_crate = { path = "../my_local_crate" }

[dev-dependencies]
mockito = "0.30"

[build-dependencies]
cc = "1.0"

[features]
default = ["serde", "regex"]
extras = ["tokio", "mockito"]

[lib]
name = "example_lib"
path = "src/lib.rs"
crate-type = ["rlib", "cdylib"]
```

Figure 10: An example of `Cargo.toml` in Rust

C# (Figure 11) Similar to Rust projects, C# repositories are often structured as solutions containing multiple internal projects. Each project uses a `.csproj` configuration file to specify external and internal dependencies and configure compilation options.

JavaScript (Figure 12) `package.json` is the configuration file used in JavaScript projects, particularly those managed with Node.js. It defines metadata such as the project name, version, and description, and specifies dependencies, scripts, and entry points for the project.

It can be found that dependency management constitutes the majority of the configuration files.

D Experimental Settings

D.1 Baseline All-In-One

In All-In-One, our baseline approach feeds the entire codebase as input context to the LLM and processes the task through a single LLM call. The

```xml
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
   <RootNamespace>ExampleProject</RootNamespace>
  </PropertyGroup>

  <!-- External dependency -->
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json"
    Version="13.0.1" />
  </ItemGroup>

  <!-- Internal project reference -->
  <ItemGroup>
    <ProjectReference
    Include="..\Internal\Internal.csproj" />
  </ItemGroup>

</Project>
```

Figure 11: An example of `example.csproj` in C#

```json
{
  "name": "example-project",
  "version": "1.0.0",
  "description": "A simple example of
  package.json",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "jest"
  },
  "keywords": ["nodejs"],
  "author": "Your Name",
  "license": "MIT",
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "jest": "^29.0.0"
  }
}
```

Figure 12: An example of `package.json` in JavaScript

model simultaneously generates all build configurations, which are then parsed to obtain the updated build files. The complete prompt template used for this approach is detailed in Appendix D.1.

Prompt Template

```
Edit the build files to include all necessary
dependency-related configurations to ensure the
project builds and runs successfully. Output a
copy of each build file.

You will receive four sections of information to
configure dependencies in build files:
1. **Project Structure**: A tree structure
representing the project's layout.
2. **Environment Specifications**: Details about
the operating system and language SDK where the
project will run.
```

````
3. **Source Code**: The full source code of the
project.
4. **Build Files**: Build files missing
dependency configurations, which you will need to
update.

!Important Notes:
1. The project may include multiple build files.
Ensure you update all of them with the necessary
dependency configurations.
2. Only edit the files listed in the "Build
Files" section.
3. Limit your edits strictly to dependency
configurations within the build files.

To suggest changes to a file you MUST return the
entire content of the updated file.
You MUST use this *file listing* format:

path/to/filename.js
```
// entire file content ...
// ... goes in between
```

Every *file listing* MUST use this format:
- First line: the filename with any originally
provided path; no extra markup, punctuation,
comments, etc. **JUST** the filename with path.
- Second line: opening ```
- ... entire content of the file ...
- Final line: closing ```

To suggest changes to a file you MUST return a
*file listing* that contains the entire content
of the file.
*NEVER* skip, omit or elide content from a *file
listing* using "..." or by adding comments like
"... rest of code..."!
Create a new file you MUST return a *file
listing* which includes an appropriate filename,
including any appropriate path.

--- Begin of Project Structure ---
{project_structure}
--- End of Project Structure ---

--- Begin of Environment Specifications ---
{env_specs}
--- End of Environment Specifications ---

--- Begin of Source Code ---
{src_section}
--- End of Source Code ---

--- Begin of Build Files ---
{build_section}
--- End of Build Files ---
````

D.2 Baseline Imports-Only

Imports-Only follows the same prompting strategy as All-In-One with a single LLM call. The key distinction lies in the input composition: while All-In-One includes the complete codebase, Imports-Only only incorporates the import statements from source files in the input context. This selective approach focuses the model's attention on the most dependency-relevant code segments. We leverage tree-sitter to extract import statements across different programming languages, with detailed usage information provided in Appendix D.4.

D.3 Baseline File-Iterate

File-Iterate employs a two-stage prompting strategy. In the first stage, it processes source files individually, applying the same prompt template which is detailed in Appendix D.1 as previous baselines but with a single file as context per LLM call. This generates separate build files edits for each source file. In the second stage, for each build file, we merge its various updates from the first stage using a dedicated LLM call. The merge prompt template is detailed in Appendix D.3. The final output consists of the comprehensively updated build files derived from this two-stage process.

Prompt For Merge Build File Edits

````
Here is a list of edits to a project's build
files, which is generated by add \
dependency configuration according to each
source file.
Edit the build files to merge all edits in the
"Build File Edits" section \
to ensure the project builds and runs
successfully. Output a copy of the build file.

You will receive four sections of information to
configure dependencies in build files:
1. **Project Structure**: A tree structure
representing the project's layout.
2. **Environment Specifications**: Details about
the operating system and language SDK where the
project will run.
3. **Build File Edits**: A list of edited build
file, which you will need to merge.
4. **Build File**: Build files missing dependency
configurations, which you will need to update
based on above edits.

To suggest changes to a file you MUST return the
entire content of the updated file.
You MUST use this *file listing* format:

path/to/filename.js
```
// entire file content ...
// ... goes in between
```

Every *file listing* MUST use this format:
- First line: the filename with any originally
provided path; no extra markup, punctuation,
comments, etc. **JUST** the filename with path.
- Second line: opening ```
- ... entire content of the file ...
- Final line: closing ```

To suggest changes to a file you MUST return a
*file listing* that contains the entire content
of the file.
````

```
*NEVER* skip, omit or elide content from a *file
listing* using "..." or by adding comments like
"... rest of code..."!
Create a new file you MUST return a *file
listing* which includes an appropriate filename,
including any appropriate path.

--- Begin of Project Structure ---
{project_structure}
--- End of Project Structure ---

--- Begin of Environment Specifications ---
{env_specs}
--- End of Environment Specifications ---

--- Begin of Build File Edits---
{build_file_edits}
--- End of Build Files Edits---

--- Begin of Build File ---
{build_section}
--- End of Build File ---
```

D.4 Tree-sitter

Tree-sitter is a parsing system widely used in code analysis that generates concrete syntax trees for source code. In our implementation, we utilize Tree-sitter to extract import statements and dependency-related code segments across different programming languages. Tree-sitter's language-agnostic nature and robust parsing capabilities enable our system to maintain consistent analysis quality across Python, JavaScript, Rust, and other supported languages. Tree-sitter queries provide a powerful pattern-matching language for searching syntax trees. The query language allows precise targeting of syntax tree patterns using a declarative, S-expression-based syntax. Below is the queries we used to extract import statements.

```
# Python
[(import_statement) (import_from_statement)]
@import

# Rust
(use_declaration) @use

# C#
(using_directive) @use

# JavaScript
(import_statement) @import
```

D.5 Model Serving

For GPT-4o and GPT-4o-mini, we utilize the specific versions gpt-4o-20240806 and gpt-4o-mini-20240718, accessed through the OpenAI API. For open-source models, we employ checkpoints available on Hugging Face and serve them using VLLM across 4 A100 GPUs. The decoding strategy is configured as greedy decoding with a maximum output token limit of 8,000.