
Class-Level Code Generation from Natural Language Using Iterative, Tool-Enhanced Reasoning over Repository

Ajinkya Deshpande, Anmol Agarwal, Shashank Shet,
Arun Iyer, Aditya Kanade, Ramakrishna B Bairi & Suresh Parthasarathy

Microsoft Research India
{ariy,kanadeaditya,ram.bairi,supartha}@microsoft.com

Abstract

Large Language Models (LLMs) have demonstrated significant potential in code generation tasks, achieving promising results at the function or statement level in various benchmarks. However, the complexities associated with creating code artifacts like classes, particularly within the context of real-world software repositories, remain underexplored. Existing research often treats class-level generation as an isolated task, neglecting the intricate dependencies and interactions that characterize real-world software development environments. To address this gap, we introduce **RepoClassBench**, a comprehensive benchmark designed to rigorously evaluate LLMs in generating complex, class-level code within real-world repositories. RepoClassBench includes "Natural Language to Class Generation" tasks across Java and Python, from a selection of public repositories. We ensure that each class in our dataset not only has cross-file dependencies within the repository but also includes corresponding test cases to verify its functionality. We find that current models struggle with the realistic challenges posed by our benchmark, primarily due to their limited exposure to relevant repository contexts. To address this shortcoming, we introduce **Retrieve-Repotools-Reflect (RRR)**, a novel approach that equips LLMs with static analysis tools to iteratively navigate and reason about repository-level context in an agent-based framework. Our experiments demonstrate that RRR significantly outperforms existing baselines on RepoClassBench, showcasing its effectiveness across programming languages and in various settings. Our findings emphasize the critical need for code generation benchmarks that incorporate repository-level dependencies to more accurately reflect the complexities of software development. Furthermore, our work illustrates the benefits of leveraging specialized tools to enhance LLMs' understanding of repository context. We plan to make our dataset and evaluation harness public.

1 Introduction

Using Large Language Models (LLMs) to generate code has garnered significant attention in recent years for its potential to streamline software development processes by automatically translating natural language descriptions into executable code snippets. Several code-specific models, like CodeGen (Nijkamp et al., 2023), WizardCoder (Luo et al., 2023), CodeLlama (Rozière et al., 2024), StarCoder (Li et al., 2023), DeepSeekCoder (Guo et al., 2024) have been proposed to this end.

While much of the focus in this domain has been on generating code units such as functions or statements, the specific task of generating classes has received comparatively less attention. Two of the most popular benchmarks HumanEval (Chen et al., 2021) and MBPP (Odena et al., 2021), for instance, focus on function generation. While useful, the problems in these datasets are short and standalone, and existing works have been able to show good

performance on these benchmarks. LATS (Zhou et al., 2023) for instance reports a 94.4% accuracy on HumanEval, and 81.1% accuracy on MBPP.

To address both of these issues, ClassEval (Du et al., 2023) proposes a benchmark for class generation. The 100 classes in the ClassEval dataset were handcrafted such that they contain inter-method dependencies, i.e. a method could reference another method in the same class. Using this dataset, they showed that, LLMs have a harder time generating code with these kind of dependencies than standalone functions of the kind present in HumanEval or MBPP.

While an important contribution, the problems proposed in ClassEval are still standalone when taking the class as a single unit. The only dependencies from outside the class are from well known libraries that the LLM is likely to have memorized. This narrow focus overlooks the complex dependencies that classes may have on other components within a codebase, presenting a gap in our understanding of code generation techniques’ practical applicability. A much more useful problem is to consider the generation of a new class that depends on code from across a repository.

To address this gap, we make an attempt at creating a dataset to explore the task of generating classes within the context of code repositories, where classes may interact with other code entities within a larger codebase. Specifically, we collect 130 Java classes from 10 repositories and 97 Python classes from 10 repositories to create RepoClassBench. Each class is present in the context of a real-world repository and has dependencies from the repository. Additionally, we make sure that each class has corresponding test cases that pass on the ground truth, and ensure sufficient coverage.

To be able to solve the problems in this dataset, the model has to both, understand the functionality required from each method in the class and reason about how to use repository-dependencies to achieve the same. We provide an evaluation of existing code-generation techniques in this setting, and demonstrate their poor performance. Specifically, BASICPROMPTING either hallucinates identifiers or avoids the dependencies, REFLEXION is able to reason about the error, but does not have enough context to fix it, and RAG-based approaches are able to find similar snippets from across the repo but fail to bring in other kinds of dependencies that are required by the class. Taking a step forward, we address the shortcoming of these methods, by proposing a novel method called RRR and show significant gains. Specifically, RRR leverages existing programming language tools to retrieve precise information from across the repository. With the injection of pointed repository context through these tools, the model is able to fix the error observed during the feedback-reflection stage.

By bridging these gaps, our study seeks to contribute to a deeper understanding of LLMs’ potential in generating classes within real-world coding scenarios, with implications for the development of more effective code generation techniques in the future. Our contributions are three-fold:

- We contribute the first benchmark RepoClassBench for class-level code generation in realistic environment of an existing repository, with 130 java classes spanning 10 repositories and 97 python classes spanning 10 repositories.
- We propose a novel method called RRR that equips LLMs with static analysis tools to iteratively navigate and reason about repository-level context in an agent-based framework, and provide a comparison with existing methods.
- We contribute 6 repository tools, based on our observations of common errors experienced by code agents in this setting.

2 Related Work

Large Language Models have seen wide success on various coding tasks. Many benchmarks have been created to assess their performance. CoNaLA (Yin et al., 2018), consisting of 500 examples is a statement-level benchmark where the target of each example contains one statement. HumanEval (Chen et al., 2021) and MBPP (Odena et al., 2021) are two widely

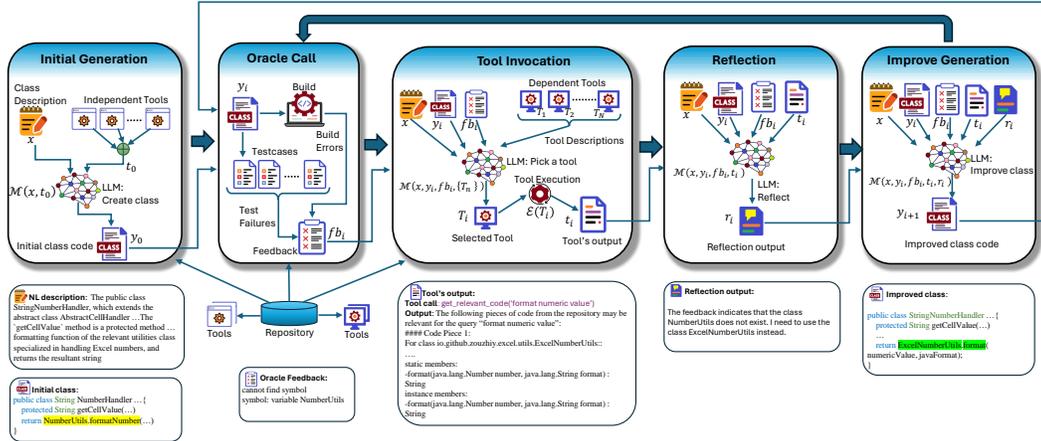


Figure 1: Flowchart illustrating the procedural framework of RRR. RRR utilizes the natural language description of the class and the outputs of independent tools to create an initial attempt. This attempt is evaluated by an oracle that pinpoints specific errors. Subsequently, RRR uses repository tools to gather information to rectify errors. It then reflects on feedback and tool insights to refine the attempt. This iterative cycle persists until all test cases pass or the maximum allowed number of oracle calls is reached.

used datasets, for function level code-generation, consisting of 164 and 974 tasks respectively. At the class-level, ClassEval (Du et al., 2023) has been proposed with 100 class generation problems, where the input is the class skeleton. However, these are all independent code-generation problems. Although ClassEval includes inter-method dependencies, they are all present within the same class. The external references come from well-known libraries that the LLM is likely to have memorized. In real world repositories, code includes complex inter-dependencies from other files in the repository. RepoBench (Liu et al., 2023), CoderEval (Zhang et al., 2024) and MGD (Agrawal et al., 2023) are attempts to move closer to this setting, and show that existing models perform much better on the standalone setting than the non-standalone setting. However they explore line and function level tasks in the context of a repository, whereas RepoClassBench explores the generation of non-standalone classes within the context of a repository. There are two aspects to solving our dataset, retrieving the right context, and reasoning to generate the code.

Reasoning: To improve the generation of LLMs, various iterative refinement techniques have been proposed. Self-refine (Madaan et al., 2023) attempts to use the LLM as it’s own critic and produces successively better outputs. Reflexion (Shinn et al., 2023) incorporates test-case feedback while generating the reflection on its output. LATS (Zhou et al., 2023) uses the LLM as an agent to explore a tree of solutions, using compiler and test feedback as observations.

Retrieval: While reasoning-enhanced methods, in themselves, may be useful for standalone generations, they are not sufficient when external context is needed. This is especially true, when the context consists of private data, unseen during pretraining. Under this paradigm Retrieval-Augmented-Generation methods like REALM (Guu et al., 2020), ATLAS (Izacard et al., 2022), RetGen (Zhang et al., 2021), FLARE (Jiang et al., 2023) retrieve relevant context, usually by considering snippets with the highest similarity score with the query. Similarly, in the code setting RLPG (Shrivastava et al., 2023) trains a model to predict the relevant context source, but relies on there being a “hole” in the code, whereas there is no such hole in the NL to new class setting. Additionally the RLPG model was trained for Java, whereas for the other languages new models would need to be trained. This adds additional cost of constructing new training data and the actual training of new models. RepoCoder (Zhang et al., 2023) has been proposed to perform iterative retrieval and generation. While such similarity based RAG methods can retrieve “similar” context, they fails to effectively retrieve “dependency” context. Further discussion can be found in RQ2.

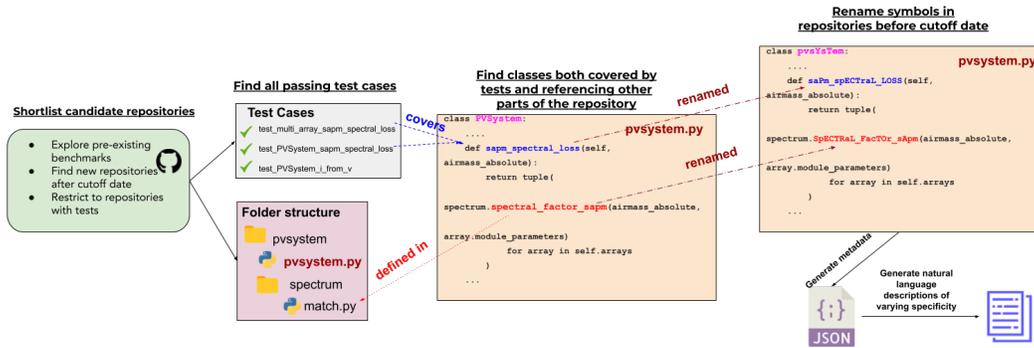


Figure 2: The dataset creation pipeline involved shortlisting candidate repositories, noting passing test cases, finding classes covered by passing test cases (which make external references) and finally mitigating memorization issues if necessary, using paraphrasing.

In our method, we leverage repository-level tools to allow the LLM explore the repository, as an alternative retrieval mechanism, in addition to using test-case feedback. This is along the lines of several works that have explored equipping the LLM with tools like ReACT (Yao et al., 2023) and ToolFormer (Schick et al., 2023). However to our knowledge, this is the first work that curates tools-specifically for repository-exploration.

Hence, we propose a benchmark that addresses the problem of class generation in the context of a repository, address a gap in the span of existing benchmarks, and also propose a novel method that integrates retrieval and reasoning, mitigating the shortcomings of existing methods.

3 Dataset: RepoClassBench

RepoClassBench is a benchmark featuring repositories from Github across languages: Java and Python. The task is to synthesize a complete class within a repository based on a natural language description, utilizing the context from other files within the same repository.

Current benchmarks face two primary limitations: (1) they (Du et al., 2023) typically focus on generating small localized code snippets, which do not accurately represent the complex tasks software engineers encounter, often requiring a comprehensive understanding of the entire codebase; (2) they (Liu et al., 2023) rely on metrics such as exact-match or cosine-similarity to the ground truth for evaluation, rather than assessing the functionality of the generated code through test cases. We mitigate these issues by designing a benchmark where every task corresponds to a class-generation problem where the LLM needs to synthesize the class based on the natural language specification of the class. We ensure that every class in our benchmark makes use of external references in the repository and is covered under test cases.

3.1 Benchmark Construction

Stage 1 - Shortlisting repositories: Our benchmark includes repositories both before and after the cutoff-date of the models we evaluate on. For JAVA we start with repositories considered in the MGD (Agrawal et al., 2023) dataset. For Python, we adapt the popular benchmark SWEBench (Jimenez et al., 2024) and also shortlist popular repositories which were first created on Github after Sept 2021. We filter out those repositories which we are unable to build and run. (Details in E.1.1)

Stage 2 - Shortlisting classes: Within each repository, we identify all classes that pass the existing test cases. We retain only those classes that (a) reference other parts of the repository within their body, and (b) have methods covered by test cases. To accommodate

the context length limitations of large language models (LLMs), we exclude classes whose implementations exceed 3,000 tokens (excluding docstrings). Additionally, we limit our selection to classes defined in the global namespace. (Details in E.1.2)

Stage 3 - Dataset paraphrasing: For repositories available before the LLMs’ training data cutoff, we undertake a paraphrasing initiative, altering the names of most symbols to prevent models from completing tasks through mere memorization. (Details in E.1.3)

Stage 4 - Generating natural language specification: We break the information within each class into varying levels of granularity and record it as metadata. The complete metadata fields are listed in Table E.1.3. Methods are categorized by three information levels: (1) Signature, detailing input and output types; (2) Docstring, providing a high-level function description; (3) Body, outlining full implementation and logic, including external references. We prompt GPT-4 to generate the natural language description of the class by providing it varying granularity of information extracted as a subset of the metadata (refer to Table E.1.3). Hence, two types of natural language description in our dataset are:-

1. DETAILED: This includes details from the entire class body (excluding imports) and prompts GPT-4 to create an NL description.
2. SKETCHY: This omits method bodies from the prompt, leading GPT-4 to generate an NL description without low-level implementation specifics or explicit external references.

In the SKETCHY setting, since GPT-4 does not receive the method bodies, the resulting natural language (NL) descriptions lack detailed implementation specifics and explicit mentions of the external references used during the method’s development. Consequently, the SKETCHY NL descriptions present a higher level of difficulty compared to the DETAILED versions. To foster community engagement and further research, we make the metadata used for constructing these prompts publicly available. This allows others to create NL descriptions with varying degrees of specificity and ambiguity to challenge the models’ capabilities. Example of the difference in prompts to GPT-4 for them can be found in Prompt 1.

Some **statistics** about our dataset can be found in Table 1. Distribution of tasks across different repositories can be found in: Figure 3 and Figure 4.

	Java	Python
Num. of tasks	130	97
Length of DETAILED NL description	1475.98 / 286.89	3245.23 / 771.77
Length of SKETCHY NL description	1481.69 / 269.81	2633.20 / 607.64
Length of classes	2080 / 452.69	4663.76 / 1070.49
Num. of TCs directly covering the classes	5.48	42.94
Num. of unique Ext. Refs	3.51	7.06
Num. of funcs in the class	3.1	9.29
Num. of funcs covered in at least one TC	2.85	4.84
Num. of funcs making at least one Ext. Refs	2.28	4.84

Table 1: Dataset High level Statistics. Each row represents an average over all the tasks in the dataset. The cells with / represent the <number of characters> / <number of tokens using gpt-3.5 tokenizer>. TC = Test Cases, funcs = functions, Ext. Refs = References from other files in the repository

4 Method

To address the challenges presented by our benchmark, we propose Retrieve-Repotools-Reflect (RRR), an innovative method that enhances Large Language Models (LLMs) with static analysis tools. This approach enables the LLMs to iteratively explore and understand

the context of a code repository through an agent-based framework. RRR leverages repository navigation and reasoning capabilities to effectively synthesize code that aligns with the broader structure and dependencies of the repository.

4.1 Phases of RRR

The procedural framework of RRR is illustrated visually in Figure 1 and outlined algorithmically in Algorithm 1. During the initial generation phase, the LLM \mathcal{M} makes an initial "guess" y_1 based on the class description x and output from invocations of the *independent tool* t_0 : $y_1 = \mathcal{M}(x, t_0)$. Given the limited information available at this stage, the LLM may resort to hallucinating identifiers and other code-structures. (Prompt in G)

The oracle call entails passing the generated code y_i to the oracle \mathcal{O} , to receive *oracle feedback* fb_i , $fb_i = \mathcal{O}(y_i)$. If the attempt exceeds the maximum number of oracle calls or successfully passes all test cases, the loop terminates and returns y_i . Otherwise, the oracle feedback errors fb_i are utilized by the LLM agent in subsequent phases to refine its generation.

While the oracle feedback identifies problems in the code, it lacks guidance on error resolution. To address this, the LLM requires repository context. This context is provided through carefully curated tools, allowing the LLM to explore the repository and retrieve relevant information. Based on the class description x , current generation y_i and feedback fb_i , the model generates a *set of tool calls* \mathbb{T}_i : $\mathbb{T}_i = \mathcal{M}(x, y_i, fb_i)$. The executor \mathcal{E} takes these tool calls and produces outputs t_i : $t_i = \mathcal{E}(\mathbb{T}_i)$. (Prompt in G)

Based on the feedback from the oracle fb_i and tool outputs t_i , the LLM generates a reflection r_i on the encountered errors and necessary actions to rectify them, using hints from the tool outputs $t_{dependent}$. $r_i = \mathcal{M}(x, y_i, fb_i, t_i)$ This reflection serves as a hint for the subsequent stage. (Prompt in G)

In the improved generation phase, leveraging the last attempt's y_i , oracle feedback fb_i , tool outputs t_i , and reflection r_i , the LLM makes another attempt at code generation y_{i+1} . $y_{i+1} = \mathcal{M}(x, y_i, fb_i, t_i, r_i)$ (Prompt in G)

After the improved generation, the attempt gets passed back to the "Oracle call" phase and the loop continues.

4.2 Tools

In RRR, tools are categorized as either independent or dependent based on their need for reasoning. **Independent tools** operate without considering the current state of the RRR loop and are automatically invoked during the initial generation phase. Our suite includes a single independent tool, 'get_related_snippets'. On the other hand, tools requiring reasoning over the current state of the RRR loop are classified as **dependent tools**. Our dependent toolset contains get_imports, get_class_info, get_signature, get_method_body and get_relevant_code. More information about the tools can be found in Table 4.2.

5 Experimental Results

5.1 Baselines

Apart from RRR, we test other important baselines (summarized in Table 8) on our newly constructed benchmark. In BASICPROMPTING the LLM is expected to generate code solely based on the Natural Language Description. In NAIVERAG, inputs include the Natural Language Description and top-snippets retrieved from repository when queried using the Natural Language Description. REFLEXION incorporates Oracle feedback to iteratively improve the generation. We also use REPOCODER, where the initial generation uses snippets retrieved using the Natural Language Description as the query, and subsequent iterations use snippets retrieved using the previous code-generation as the query. Summary of the baseline can be found in 8.

Tool	Description
<code>get_related_snippets</code>	<i>Type: Independent.</i> Segments the repository into snippets and returns the top 5 snippets based on cosine similarity with the class description.
<code>get_imports</code>	<i>Type: Dependent.</i> Suggests imports for all the undefined symbols in the current generation, scanning the repository for potential source files defining the symbol and recommending import statements. <i>Input args:</i> No input
<code>get_class_info</code>	<i>Type: Dependent.</i> Locates the class definition in the repository and gathers information about its members, including inherited members, providing detailed information about each member. <i>Input args:</i> class name
<code>get_signature</code>	<i>Type: Dependent.</i> Returns the signature of the requested method, displaying signatures of all methods with the same name if they exist in the same class. <i>Input args:</i> class name, method name
<code>get_method_body</code>	<i>Type: Dependent.</i> Returns the method definition of the requested method, truncating the output if it is too large, and showing the definition for each method with the same name if they exist. <i>Input args:</i> class name, method name (where class name is the class of which the method is a member. Class name is left as None for global methods.)
<code>get_relevant_code</code>	<i>Type: Dependent.</i> Allows specific queries to retrieve code structures using embedding similarity scores, returning the top 3 structures based on cosine similarity using UnixCoder embeddings. <i>Input args:</i> natural language query

Table 2: Table containing descriptions of the tools used in RRR. The Type indicates whether reasoning is required (dependent) or not (independent) for the invocation.

5.2 Metrics

For each task in our benchmark we use three metrics to measure performance. Pass@K measures the percentage of the tasks for which there is at least one correctly generated solution (passing all test cases) among the top K samples generated by the LLM (Chen et al., 2021). For our experiments, we simply set the total number (denoted as n) of samples generated by an LLM to 1, and then calculate Pass@1 for the LLM. For completeness, in RQ 7, we also measure Pass@ 1, 2, 3, setting n=6 for the JAVA dataset. We also use TR (Test Rate) which measures the mean of the fraction of test cases passed for all generations across all tasks. Finally, for JAVA, since we have access to a compiler, we also measure CR, or the compilation rate which measures the percentage of tasks for which the LLM generated code that successfully compiled.

5.3 Research Questions

Through our experiments we aim to answer the following RQs (RQs 5-8 in Appendix): **RQ1**- How does RRR perform compared to the baselines, under the DETAILED and SKETCHY settings ? **RQ2** - Where do similarity-based retrieval methods fail? **RQ3** - What is the impact of test feedback on performance? **RQ4** - What are the challenges faced by RRR? **RQ5** - How important is each tool for our method? **RQ6** - How does number of iterations in RRR and baselines impact their performance? **RQ7** - How does increased sampling impact the performance our RRR and the baselines? **RQ8** - Does performance depend on whether the repository might have been included in the training dataset of the LLM?

5.3.1 RQ1 - Comparative analysis of RRR and baselines

We analyzed Table 5.3.1 and Table 4, comparing RRR’s performance with baselines. To explore the use of different LLMs, for JAVA, GPT-3.5 was used; for PYTHON, GPT-4 was employed. RRR consistently outperforms baselines across all metrics. BASICPROMPTING performs the worst without feedback or context, with hardly any generations passing

Method	JAVA			PYTHON	
	P@1	TR	CR	P@1	TR
BASICPROMPTING	1.54	1.54	2.31	1.03	2.40
REFLEXION	3.85	5.04	5.38	7.22	14.36
NAIVERAG	11.54	12.15	14.62	13.40	14.08
REPOCODER	40.77	43.38	46.92	22.68	25.59
RRR	54.62	63.22	70.77	27.84	36.92

Table 3: Performance numbers expressed in percentage, for the baselines and RRR on the DETAILED version of the dataset. P@1 represents the Pass@(1,1) metric, TR is the mean test-pass rate across all tasks, and CR is the mean compilation rate across tasks. RRR performs much better than the baselines.

Method	JAVA			PYTHON	
	P@1	TR	CR	P@1	TR
BASICPROMPTING	1.54	1.54	2.31	0.00	1.43
REFLEXION	2.31	3.04	5.38	0.00	0.24
NAIVERAG	8.46	8.46	10.00	0.00	13.38
REPOCODER	34.62	39.17	44.62	7.14	10.06
RRR	48.46	54.72	64.62	7.14	21.89

Table 4: Performance numbers expressed in percentage, for the baselines and RRR on the SKETCHY version of the dataset. RRR performs much better than the baselines.

test cases. REFLEXION slightly improves with oracle feedback but lacks repository context, resorting to hallucinating identifiers and limited repository utilization.

To add the repository context one might consider dumping the entire repository in the prompt. However, the token count in JAVA and PYTHON repositories can exceed 50k, surpassing LLM context windows, and dumping entire repositories into prompts is impractical. To tackle these issues, methods that employ retrieval can be used. There’s a noticeable performance jump from REFLEXION to NAIVERAG, further improved with REPOCODER, due to more relevant retrieved snippets. While REPOCODER is the best performing baseline, it has two major drawbacks. Firstly, oracle feedback is not used, and secondly, the REPOCODER snippets retrieve “similar” lines of code from the repository, and not dependencies, thereby missing crucial information. RQ2 explores this second point in greater detail. Conversely, RRR retrieves dependency context, combining repository context and oracle feedback intelligently. It queries specific repository information to address oracle feedback, consistently outperforming baselines across languages and metrics. Still, there are cases where RRR fails test cases, which we analyze in RQ4.

5.3.2 RQ2 - The contribution of similarity-based RAG

In this benchmark, repositories, typical of those on GitHub, contain numerous highly similar classes. RAG-based techniques excel over BASICPROMPTING or REFLEXION because they leverage these similarities. However, there’s a crucial distinction between “dependency context” and “similarity context.” Dependency context involves information from the repository about utilized code structures, while similarity context merely seeks similar code, which may not always be present.

To illustrate that REPOCODER’s gains largely stem from “similar” snippets, we remove all relatives of each class to be generated, defined as descendants of the grandparent except the immediate parent and itself. These relatives, often similar to the target class, are pulled in through REPOCODER snippets. Upon re-comparison with baselines (see Table 5), REPOCODER’s performance notably declines in both DETAILED and SKETCHY settings.

Method	JAVA- DETAILED			JAVA- SKETCHY		
	P@1	TR	CR	P@1	TR	CR
BASICPROMPTING	0.77	0.77	0.77	1.54	1.54	3.85
REFLEXION	2.31	2.88	3.85	1.54	2.36	4.62
NAIVERAG	8.46	8.46	10.00	4.62	6.60	8.46
REPOCODER	23.85	24.42	26.15	16.92	23.92	31.54
RRR	46.92	53.23	60.00	36.92	43.86	51.54

Table 5: Performance numbers expressed in percentage, for the baselines and RRR, after removing the “Relatives” from the DETAILED and SKETCHY versions of the Java dataset. While all retrieval-based methods suffer, RRR does not suffer as much as REPOCODER.

Conversely, RRR suffers less, indicating its reliance on “dependency context” for generation completion.

5.3.3 RQ3 - Importance of test feedback

Method	JAVA- Detailed			JAVA- Sketchy		
	P@1	TR	CR	P@1	TR	CR
BASICPROMPTING	1.54	1.54	1.67	1.67	1.73	2.69
REFLEXION	2.69	3.36	5.38	3.08	3.78	6.92
NAIVERAG	11.41	11.92	13.33	8.97	9.61	11.28
REPOCODER	37.05	40.12	45.00	29.74	36.77	44.62
RRR	56.15	62.32	71.92	41.67	51.76	63.46

Table 6: Performance numbers expressed in percentage, for the baselines and RRR, terminating the generation immediately after the compilation succeeds, on the DETAILED and SKETCHY versions of the Java dataset. There is a marginal decrease in performance, indicating that most functional requirements can be met simply by using the compiler as the oracle.

Examining the role of test feedback, we restrict the oracle to compiler feedback, applicable only for JAVA. In Table 6, baselines like BASICPROMPTING, NAIVERAG, and REPOCODER remain unchanged without oracle feedback. Methods utilizing test feedback show a slight decrease in performance, but still perform adequately. Code that compiles and aligns with functional descriptions tends to pass test cases, as they typically assess functional requirements. While test feedback aids in ambiguous cases, the LLM generally performs well with just compiler feedback.

5.3.4 RQ4 - Success and failure case analysis

Language	Reasoning Errors	Functional Ambiguity
JAVA- DETAILED	70%	30%
JAVA- SKETCHY	50%	50%

Table 7: Analyzing failure causes across a sample of 20 tasks from the Java dataset, errors are categorized as reasoning-related (in tool retrieval or code generation) or functional ambiguity-related. The table shows the percentage contribution of each error type to failure cases. In the DETAILED dataset, reasoning errors dominate, while in the SKETCHY version, functional ambiguity-related errors increase.

This section investigates instances where the Language Model (LLM) failed to pass test cases, identifying potential contributing factors. Notably, errors weren’t due to information

access limitations through tools; there was always a tool for repository information retrieval. Our analysis focuses on categorizing error types to guide future investigations for mitigation strategies.

Distinct error patterns emerged upon examination, broadly categorized as reasoning errors or functional ambiguity errors. Reasoning errors occur during tool retrieval or code generation, where the LLM fails to interpret or apply information correctly. Functional ambiguity errors arise when the LLM misinterprets terse natural language descriptions, leading to multiple interpretations or missing information. Table 5.3.4, a qualitative analysis of 20 failure cases, shows reasoning errors dominate in the DETAILED setting, while functional ambiguity increases in the sketchy setting. Additionally, the LLM struggles with lengthy textual inputs, with extended class length correlating significantly with decreased efficacy. Over the detailed JAVA dataset, test performance and class length had a Spearman correlation of -0.66, highlighting the challenge of reasoning over extensive texts. Identifying these failure cases sheds light on the dataset’s role in understanding LLM capabilities and limitations. By pinpointing error patterns and correlating them with variables like class length, our analysis sets the stage for future research on enhancing language model robustness and efficacy.

6 Discussion

RepoClassBench provides a previously underexplored setting, with unique challenges that require reasoning over the repository. We have further showed that previous methods that use similarity based retrieval have certain drawbacks, in terms of applicability and effectiveness. In solving this problem we proposed using tools to retrieve repository information, which is able to combine traditional embedding based retrieval (through the `get_related_snippets` and `get_relevant_code` tools) and static analysis tools. Through an iterative paradigm of refinement based on the tool outputs and oracle feedback, we showed that RRR performs well.

7 Conclusion

In conclusion, we propose a new benchmark on the underexplored setting of class-generation within a repository. We show that existing methods perform poorly and introduce a novel method called RRR that significantly improves on the baselines.

References

- Lakshya Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K Lahiri, and Sriram Rajamani. Monitor-guided decoding of code LMs with static analysis of repository context. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL <https://openreview.net/forum?id=qPubKxKvXq>.
- Ben Athiwaratkun, Sanjay Krishna Gouda, Zijian Wang, Xiaopeng Li, Yuchen Tian, Ming Tan, Wasi Uddin Ahmad, Shiqi Wang, Qing Sun, Mingyue Shang, Sujan Kumar Gonugondla, Hantian Ding, Varun Kumar, Nathan Fulton, Arash Farahani, Siddhartha Jain, Robert Giaquinto, Haifeng Qian, Murali Krishna Ramanathan, Ramesh Nallapati, Baishakhi Ray, Parminder Bhatia, Sudipta Sengupta, Dan Roth, and Bing Xiang. Multi-lingual evaluation of code generation models. 2022. doi: 10.48550/ARXIV.2210.14868. URL <https://arxiv.org/abs/2210.14868>.
- 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. Evaluating large language models trained on code. 2021.
- Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023.
- Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation, 2022.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y.K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024. URL <https://arxiv.org/abs/2401.14196>.
- Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Realm: retrieval-augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning, ICML'20*. JMLR.org, 2020.
- Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. Few-shot Learning with Retrieval Augmented Language Models. 2022. URL <http://arxiv.org/abs/2208.03299>.
- Zhengbao Jiang, Frank F. Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation. 2023.
- 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>.
- Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muh-tasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kurnakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.
- Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems, 2023.
- Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct, 2023.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegraffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023.

-
- Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *ICLR*, 2023.
- Augustus Odena, Charles Sutton, David Martin Dohan, Ellen Jiang, Henryk Michalewski, Jacob Austin, Maarten Paul Bosma, Maxwell Nye, Michael Terry, and Quoc V. Le. Program synthesis with large language models. In *n/a*, pp. n/a, n/a, 2021. n/a.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools, 2023.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.
- Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code, 2023.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL <https://openreview.net/forum?id=WE.vluYUL-X>.
- Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pp. 476–486. ACM, 2018. doi: <https://doi.org/10.1145/3196398.3196408>.
- Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.151. URL <https://aclanthology.org/2023.emnlp-main.151>.
- Yakun Zhang, Wenjie Zhang, Dezhi Ran, Qihao Zhu, Chengfeng Dou, Dan Hao, Tao Xie, and Lu Zhang. Learning-based widget matching for migrating gui test cases. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24. ACM, February 2024. doi: 10.1145/3597503.3623322. URL <http://dx.doi.org/10.1145/3597503.3623322>.
- Yizhe Zhang, Siqi Sun, Xiang Gao, Yuwei Fang, Chris Brockett, Michel Galley, Jianfeng Gao, and Bill Dolan. Joint retrieval and generation training for grounded text generation. *arXiv preprint arXiv:2105.06597*, 2021.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2023.

A Algorithm

Algorithm 1 RRR

```
1: procedure RRR( $x$ : Natural language description,  $MAX\_CALLS$ : Maximum iterations,
    $\mathcal{O}$ : Oracle function,  $\mathcal{M}$ : Language Model Agent,  $\mathcal{E}$ : Tool executor)
2:    $y_1 = \mathcal{M}(x, t_0)$   $\triangleright$  Initial guess using NL description, independent tool outputs
3:   for  $i = 1$  to  $MAX\_CALLS$  do  $\triangleright$  Iterate over generation attempts
4:      $fb_i = \mathcal{O}(y_i)$   $\triangleright$  Get feedback from oracle
5:     if all test cases pass in  $fb_i$  then
6:       return  $y_i$ 
7:     end if
8:      $\mathbb{T}_i = \mathcal{M}(x, y_i, fb_i)$   $\triangleright$  Generate tool calls based on the error
9:      $t_i = \mathcal{E}(\mathbb{T}_i)$   $\triangleright$  Execute tool calls
10:     $r_i = \mathcal{M}(x, y_i, fb_i, t_i)$   $\triangleright$  Generate reflection
11:     $y_{i+1} = \mathcal{M}(x, y_i, fb_i, t_i, r_i)$   $\triangleright$  Generate improved code
12:  end for
13:  return  $y_{i+1}$   $\triangleright$  Return the latest generated code
14: end procedure
```

B Baselines

Approach	External Context shown	Testcase feedback for refinement	Multiple iterations	Tools access
BASICPROMPTING	×	×	×	×
REFLEXION	×	✓	✓	×
NAIVERAG	✓	×	×	×
REPOCODER	✓	×	✓	×
RRR	✓	✓	✓	✓

Table 8: A breakdown of the different approaches compared in this work. RRR ticks all the boxes and thus has advantages from each method.

C Experimental Setup

We utilized the GPT-3.5-turbo-instruct model, through the Azure OpenAI endpoint, configured with a temperature of 0.2, for the JAVA experiments. Similarly, we utilized the GPT-4 model, through the Azure OpenAI endpoint, configured with a temperature of 0.2, for the PYTHON experiments. Default values were maintained for all other parameters. All experiments were conducted on a machine with an AMD EPYC 7V13 64-Core Processor running at 2.45GHz, 216GB of RAM, and an NVIDIA A100 80GB GPU. The experiments were executed using Python 3.11.8 and PyTorch 2.0.1.

D Additional RQs

D.0.1 RQ5 - Tool statistics

While it is hard to evaluate exactly how much each tool contributed to the the success of our method, a proxy for this measurement could be to analyze the frequency with which each tool is called, in all the cases where the generation passed all test cases. Thus, in Table D.0.1 we count the number of times each tool was called, for all the successful generations, and express these counts as percentages. However, we notice an LLM-related idiosyncrasy, where the model is strongly biased to using the tools shown to it in the few-shot examples.

Method	imports	class_info	method_body	relevant_code	signature
DETAILED	32.50	37.69	27.50	1.15	1.15
Sketchy	30.64	36.66	28.06	2.75	1.89
DETAILED _{NoRelatives}	30.69	40.04	25.93	1.59	1.76
Sketchy _{NoRelatives}	29.76	37.42	26.26	4.38	2.19

Table 9: The frequency distribution of tool calls, expressed in percentage, for the various configurations of the Java dataset. The LLM is biased to utilize the tools shown in the few-shot example (`get_imports`, `get_class_info`, `get_relevant_code` for JAVA), and makes superfluous calls using them, adding noise to the true frequency distribution. There is however, an expected increase in the frequency of certain tools like `get_relevant_code`, moving from DETAILED to SKETCHY and Relatives to No Relatives.

Even in cases where certain tools are not required, it forces itself to create a reason for using it and makes superfluous calls, adding noise to the frequency count. To illustrate this point more clearly, consider Table Table D.0.1. Here, each cell represents the percentage of tasks, across which the corresponding tool was called at least once (assuming at least one round of tool-invocation happened). As visible from the table, the tools present in the few-shot example (`get_imports`, `get_class_info`, `get_relevant_code` for JAVA), were called in every example. The other tools get called much less frequently. This motivates careful selection of the few-shot examples, to choose tools which would be required to solve the most frequently observed errors.

Method	imports	class_info	method_body	relevant_code	signature
DETAILED	100.00	100.00	100.00	5.36	8.93
SKETCHY	100.00	100.00	100.00	20.75	16.98
DETAILED _{NoCousins}	100.00	100.00	100.00	11.76	17.65
SKETCHY _{NoCousins}	100.00	100.00	100.00	30.95	21.43

Table 10: Percentage of tasks across which, the corresponding tool has been used at least once. Note that tasks that pass in the first iteration, without needing to call any tools are not considered here. The tools which were mentioned in the few-shot example dominate the table, being called at least once in each task. Many of these calls end up being superfluous.

D.0.2 RQ6 - Number of iterations

Language	1	2	3	4	5
DETAILED+Cousins	11.54	36.15	49.23	53.08	54.62
SKETCHY+Cousins	7.69	25.38	39.23	46.15	48.46
DETAILED+No Cousins	7.69	26.15	35.38	43.08	46.92
SKETCHY+No Cousins	4.62	20.77	28.46	33.85	36.92

Table 11: Pass@1 performance across the 5 iterations of RRR. While the performance improves across iterations, the increase in performance at each iteration diminishes.

To examine the effect of the number of iterations on the performance, we measure the performance at each iteration of RRR. While, by definition, the performance is strictly non-decreasing with the number of iterations, the performance delta decreases between iterations. Thus, there exists a trade-off between the computational resources required for LLM inference and the gain in performance. While, in our case we terminate the generation at 5 iterations, for other datasets, depending on the complexity of the class, more iterations may be required.

D.0.3 RQ7 - Multiple generations

To test the effect of sampling multiple generations, we also report the performance numbers over multiple generations on the java dataset. Specifically, for $n = 6$ generations, we calculate the Pass@1, 2, 3 scores. Table 12 shows that the trends remain, and RRR outperforms the baselines significantly under all configurations.

Method	JAVA- Detailed			JAVA- Sketchy		
	P@1	P@2	P@3	P@1	P@2	P@3
BASICPROMPTING	1.54	1.54	1.54	1.67	1.77	1.86
REFLEXION	3.72	4.72	5.27	3.33	4.06	4.41
NAIVERAG	11.41	11.73	11.86	8.97	9.87	10.31
REPOCODER	37.18	43.89	47.25	31.41	36.65	39.29
RRR	58.21	65.38	68.28	48.46	55.68	58.91

Table 12: Pass @ 1,2,3 scores for RRR setting the number of generations n=6

D.0.4 RQ8 - Does performance depend on whether the LLM has seen the dataset before?

The Python dataset has tasks from a repository called Litestar which was created on Github after the training-date cutoff for the GPT models used for evaluation in this paper. We notice that the trend across different methods remains the same. The smaller size of the classes from Litestar as compared to the other Python repositories might be one of the reasons why all the methods perform better on tasks from Litestar.

Method	PYTHON- Litestar		PYTHON- All	
	P@1	TR	P@1	TR
BASICPROMPTING	2.44	2.44	1.03	2.49
REFLEXION	14.63	19.50	7.22	14.36
NAIVERAG	26.83	27.07	13.40	14.08
REPOCODER	36.59	40.04	22.68	25.59
RRR	41.46	48.67	27.84	36.92

Table 13: Performance scores on the Python dataset for (1) all the tasks, (2) for only the tasks from Litestar

E RepoClassBench

E.1 Benchmark Construction

E.1.1 Repository Selection

Our dataset comprises repositories from two distinct categories: **(Type 1)** well-established repositories such as ‘scikit-learn’, ‘requests’, ‘pydicom’, which have been present on GitHub since before September 2021; and **(Type 2)** repositories that were created on GitHub after the cutoff date of the language models (LMs) we are using, i.e., September 2021, ensuring that the LMs have not been exposed to these repositories during their training or fine-tuning phases. We detail the language-specific selection process below:

- **Java:** All Java repositories included in our study are from (Type 2). To construct the Java dataset, we utilized the existing dataset compiled by MGD Agrawal et al., 2023.

- **Python:** For (Type 1) repositories, we adapted the established SWEBench (Jimenez et al., 2024) benchmark. To mitigate the risk of dataset contamination, we paraphrased the symbols in these repositories as described in Section E.1.3. To assemble a pool of (Type 2) repositories, we identified the most starred Python repositories on GitHub created after the LMs’ cutoff date. We then excluded repositories without any mention of ‘pytest’ in their files, assuming the absence of test cases. Many of the remaining repositories were associated with LMs and appeared to require an ‘OPENAPI_KEY’ to execute tests. Consequently, we excluded repositories containing the keywords ‘OPEN_API’, ‘LLM’, and ‘GPT’. From the remaining candidates, we selected the top three repositories that did not seem to be related to LMs based on their title or description. This process yielded three repositories: ‘dosisod/refurb’, ‘pyscript/pyscript’, and ‘litestar-org/litestar’.

E.1.2 Task instance construction from each repository

For each repository identified in the previous section, we first ensure that we can successfully build the repository (for Java) and that all the necessary environment installations are in place (for Python). We provide the necessary scripts to install such conda environments wherever applicable. Once these prerequisites are met, we refer to the current state of the repository as ‘R’ and begin the process of shortlisting candidate classes. An ideal class for inclusion in our dataset should exhibit two key properties: (1) it utilizes context from the repository, and (2) its correctness can be verified through test cases. We define these properties in more detail as follows:

- **Uses Repository-Level Context:** For a given class ‘C’, there can be four types of references in its body:
 1. References defined in external libraries outside the repository.
 2. References to other members within the class ‘C’ itself (e.g., method ‘M1’ of class ‘C’ calling method ‘M2’ of the same class ‘C’).
 3. References to entities defined in the same file but outside the body of class ‘C’.
 4. References to entities defined elsewhere within the repository.

For our study, we categorize references of type (3) and (4) as **EXTERNAL REFERENCES**.

- **Covered Under Test Cases:** For repository R , let P denote the set of test cases that pass in the current state. Given a test case T and a class C from our benchmark tasks, we define:

$$\begin{aligned} \text{DirectCoverage}(T) &= \text{the set of classes/functions directly invoked in the body of } T, & (1) \\ \text{IndirectCoverage}(T) &= \text{the set of classes/functions not directly invoked in the body} \\ &\quad \text{of } T \text{ but if left unimplemented, } T \text{ would fail} & (2) \end{aligned}$$

We then determine the number of unique test cases that directly cover class C or any of its members. To ensure that the inclusion of a class in our prompts does not exceed the LLM’s context length limit, we exclude classes whose body exceeds 3000 tokens after the removal of docstrings. Additionally, we limit our benchmark to classes defined in the global namespace.

To confirm that each class possesses both of the aforementioned properties, we apply the following filtering criteria across different languages:

- **Java:** We require that at least two-thirds of the methods in a class are referenced in the combined bodies of all corresponding test cases to ensure adequate code coverage. Additionally, the class must contain at least one external reference.
- **Python:** We require at least two methods that are (a) directly covered by some test case and (b) make an external reference.

E.1.3 Dataset Paraphrasing

For each repository ‘R’, we begin by compiling a list of all identifiers that appear in at least one class within our benchmark. To paraphrase these identifiers, we apply a case-flipping technique to their original names. For example, the identifier ‘encode’ would be transformed to ‘eNcoDe’. This transformation is applied consistently across all Python and Cython files in the repository. (Athiwaratkun et al., 2022) has shown the LMs are not robust to errors when prone to natural language descriptions with randomly flipped characters.

To avoid inadvertently altering identifiers from external libraries, we exclude certain common identifiers from this process. For instance, we would not modify the identifier ‘items’ to prevent the expression ‘my_dict.items()’ from being incorrectly changed to ‘my_dict.ItEms()’.

We verify the success of our paraphrasing by ensuring that the majority of test cases that passed prior to the paraphrasing continue to pass afterwards. This approach to paraphrasing serves a dual purpose: (1) it preserves the semantic meaning of the original identifiers, and (2) it prevents the LLM from relying on rote memorization of its training data to complete tasks in our benchmark. A sample instance of paraphrased code can be found in 2.

Level	Field	DETAILED	SKETCHY
Class Level Info	Class Name and file path	✓	✓
	Import statements	✗	✗
	Member variables (with initializations)	✓	✓
	Class signature	✓	✓
	Decorators	✓	✓
	Parent Class names	✓	✓
Method Level Info	Method Signature	✓	✓
	Method decorators	✓	✓
	Method Docstrings	✓	✓
	Method Body	✓	✗

Table 14: Components of the metadata

E.2 Task components

Repository setup: At the start of each task, the repository is reset to its original state, ensuring all components are aligned with the ground truth. The class targeted by the task, along with its associated imports, is then removed from the repository.

Model input: The model receives the NL description of the class and is tasked with generating the complete class body, including any necessary import statements.

Testcase Feedback: We identify test cases that pass in the repository’s ground truth state and also reference the class or its members within their test functions. An incorrect class implementation could lead to failures in these tests. Let’s call this set EXPECTED_TO_PASS. The model’s output is assessed against this specific set of test cases.

Evaluation metrics: To evaluate the model’s generated code, we insert it into the repository at the location of the original class implementation and run the relevant test cases (ie EXPECTED_TO_PASS). Our evaluation metrics include: To score the generation, we use the following metrics:

- **Testcase pass rate (TR):** This is the fraction of test cases from EXPECTED_TO_PASS which pass when the model’s code is introduced into the repository.
- **Compilation Rate (CR):** This is a binary value between 1 and 0 depending on whether the repository was able to be built after the model’s code was introduced in the repository. (Applicable only to Java dataset)

F Tools

Broadly speaking, the tools utilized in RRR are classified as either dependent or independent, depending on whether their invocation requires reasoning. Independent tools do not necessitate reasoning about the current state of the RRR loop. These tools are automatically called during the initial generation phase. Our independent toolset contains a single tool `get_related_snippets`. On the other hand, tools requiring reasoning over the current state of the RepoReflexion loop are classified as dependent tools. Our dependent toolset contains `get_imports`, `get_class_info`, `get_signature`, `get_method_body` and `get_relevant_code/`.

F.1 Independent Tools

Independent tools do not necessitate reasoning about the current state of the RRR loop. These tools are automatically called during the initial generation phase.

- **get_related_snippets**: This tool addresses the common scenario of multiple similar classes within a repository. It segments the repository into snippets and returns the top 5 snippets based on cosine similarity with the class description. Since repositories often contain near-identical classes, the LLM agent benefits from examining these implementations.

F.2 Dependent Tools

Tools requiring reasoning over the current state of the RepoReflexion loop are classified as dependent tools.

- **get_imports** (Parameters: Empty): This tool suggests imports all the undefined symbols in the current generation. It scans the repository for potential source files defining the symbol and recommends import statements. If multiple sources are possible, it outputs all options for the LLM agent to choose from. The import tool can be helpful to resolve "symbol not found errors".
- **get_class_info** (Parameter 1= Class Name): This tool locates the class definition in the repository and gathers information about its members, including inherited members. The tool provides detailed information about each member, such as parameters, return type, access specifier, and whether the member is static or abstract. This tool can help when a method/variable in a class returns a "symbol not found" error. In cases of multiple classes with the same name, the tool lists information for each. To manage prompt length, the tool ranks members based on cosine similarity with the thought produced just before invocation, displaying the top k results, where k is set to 10. In case multiple classes with the same name exist, it shows the info for each of them. In case the LLM passes multiple classes, it shows the info for each of them.
- **get_signature** (Parameter 1= Class Name, Parameter 2= Method Name): This tool returns the signature of the requested method. In case multiple methods with the same name exist in the same class (overloading), it displays the signatures of all of them. This could help when the number or parameters or the types of the parameters were hallucinated, leading to incorrect method calls.
- **get_method_body** (Parameter 1= Class Name, Parameter 2= Method Name): This tool checks the source code and returns the method definition of the requested method. In case it is too large, it truncates the output. In case multiple methods with the same name are available, it shows the definition for each of them. In case the definition is unavailable (due to the method being in an external library), it shows the signature instead. This tool can be invoked to address the situation where the exact implementational logic of a method is required to fix an error.
- **get_relevant_code** (Parameter 1= Query String): While all the tools till now can help gather more information about symbols that the LLM already has knowledge about, a large part of writing code involves dependencies that the coder is unaware of.

For instance, there may be methods in utility classes that can be re-used instead of writing the logic from scratch. To aid in the search for relevant clues across the repository, this tool allows the LLM to make specific queries that retrieve code structures using embedding similarity scores. The tool considers three types of code-pieces, classes, independent methods (not present in a class) and snippets. For classes, the methods are stripped of their bodies before the encoding, for independent methods (not present in a class), the body of the method is used, and for snippets, the snippets are used directly to generate the embeddings. The tool returns the top 3 structures on the basis of the cosine similarity score.

F.3 Implementation details

For Python, all the tools were implemented using a combination of Jedi and Tree-Sitter. For Java, the EclipseJDTLS Language-server was used.

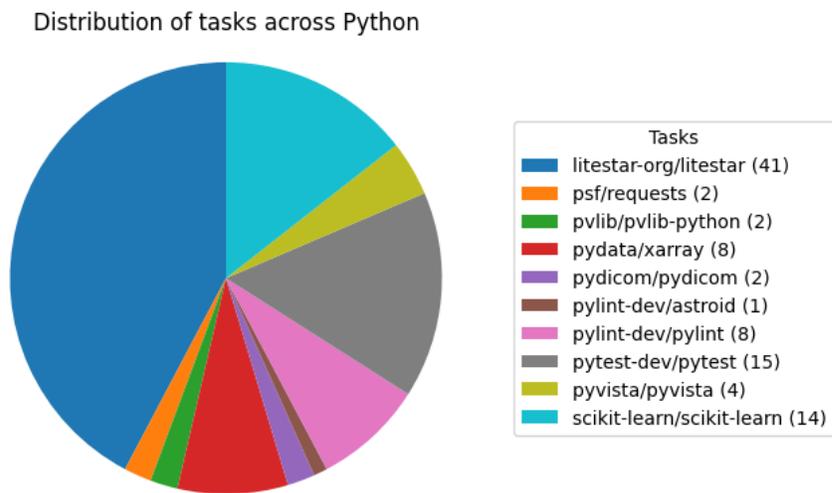


Figure 3: Distribution of the tasks across the various repositories in the Python dataset.

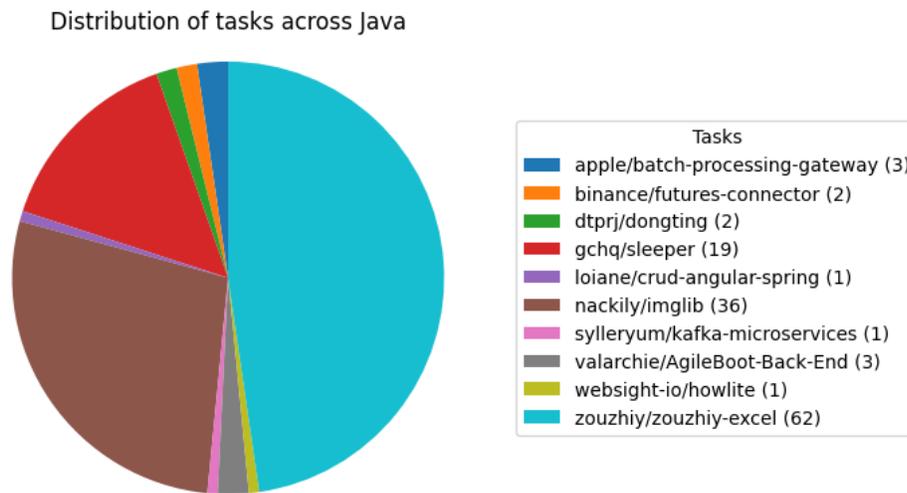


Figure 4: Distribution of the tasks across the various repositories in the Java dataset.

G Prompt Templates

Initial Prompt Template

This is the first prompt that is used fed to the LLM. The intent here is code generation, using a provided description and a few relevant code snippets retrieved from the repository. Relevance is measured as cosine similarity between computed UniXCoder Guo et al. (2022) embeddings.

```
Solve the below class-generation tasks (include all necessary
imports):
# Question 1
Below are some referential code fragments from other files.
{FS_EXAMPLE_SNIPPETS}
Based on the above, generate the following class
File: {FS_EXAMPLE_FILE_PATH}
Description: {FS_EXAMPLE_DESCRIPTION}
Generated Code:
```{LANGUAGE}
{FS_EXAMPLE_CODE}
```

# Question 2
Below are some referential code fragments from other files.
{SNIPPETS}
Based on the above, generate the following class
File: {FILE_PATH}
Description: {DESCRIPTION}
Generated Code:
```{LANGUAGE}
```

### Tool Invocation Prompt Template

This is the next prompt in the pipeline and is meant to take the generated code, along with oracle feedback. Additionally, language specific, curated few-shot examples are provided to act as a template of the expected structure of the output.

```
You are a {LANGUAGE} coding assistant. Fix the error in the code
by interleaving Thought and Action. `Thought` can be used to
reason about the current situation/error. You have been also
provided a set of tools/actions to get information about the
various parts of the repository. Here is a list of available
actions/tools:
(1) get_class_info(class_name): retrieves a list of available
methods or properties for a given class `class_name` if it
exists. Also returns the constructor.
(2) get_signature(class_name, method_name): which returns the
signature of the specified method `method_name` in the class `
class_name`, including its parameter names and types, if it
exists.
(3) get_method_body(class_name, method_name): which returns the
body of the specified method if it exists.
(4) get_relevant_code(search_string): which returns potentially
relevant pieces of code from the repository corresponding to
the `search_string`.
(5) get_imports(): which returns import suggestions for all the
undefined symbols in the code.

These are the only permitted actions/tools. Note that you cannot
call these actions on the class that needs to be generated
since it does not exist yet. That the same tool can also be
called multiple times with different arguments.
```

---

Here are some examples.

```
Question 1:
File: {FS_EXAMPLE_FILE_PATH}
Description: {FS_EXAMPLE_DESCRIPTION}"
Previous faulty implementation:
```{LANGUAGE}
{FS_EXAMPLE_PREVIOUS_IMPL}
```

Feedback from previous faulty implementation:
{FS_EXAMPLE_PREVIOUS_IMPL_FEEDBACK}

Available tools (the same tool can be called more than once but
atmax 3 tool calls can be made): get_class_info(class_name)
get_signature(class_name, method_name) get_method_body(
class_name, method_name) get_relevant_code(search_string)
get_imports()
{fs_example_previous_impl_tool_call}

Question 2:
File: {FILE_PATH}
Description: {DESCRIPTION}
Previous faulty implementation:
```{LANGUAGE}
{PREVIOUS_IMPL}
```

Feedback from previous faulty implementation:
{PREVIOUS_IMPL_FEEDBACK}

Available tools (the same tool can be called more than once but at
max 3 tool calls can be made): get_class_info(class_name),
get_signature(class_name, method_name), get_method_body(
class_name, method_name), get_relevant_code(search_string)
get_imports()
<tool_usage_thoughts>
```

## Feedback Reflection Prompt Template

You are a {LANGUAGE} writing assistant. You are supposed to generate a {LANGUAGE} class based on the description of the class provided to you. You will be given your previous faulty implementation and feedback about what went wrong based on a few test cases. Your goal is to write a few sentences to explain why your implementation is wrong as indicated by the tests. You will need this as a hint when you try again later. Only provide the few sentence description in your answer, not the implementation.

```
Question 1:
Below are some potentially relevant pieces of information.
{FS_EXAMPLE_TOOL_OBSERVATIONS}
Based on the above, generate the following class
File: {FS_EXAMPLE_FILE_PATH}
Description: {FS_EXAMPLE_DESCRIPTION}
Previous faulty implementation:
```{LANGUAGE}
{FS_EXAMPLE_PREVIOUS_IMPL_V2}
```

Feedback from previous faulty implementation:
{FS_EXAMPLE_PREVIOUS_IMPL_FEEDBACK_V2}
Reflection on previous faulty implementation:
```

```

```{FS_EXAMPLE_PREVIOUS_IMPL_REFLECTION_V2}```

# Question 2:
Below are some potentially relevant pieces of information.
{TOOL_OBSERVATIONS}
Based on the above, generate the following class
File: {FILE_PATH}
Description: {DESCRIPTION}
Previous faulty implementation:
```{LANGUAGE}
{PREVIOUS_IMPL}
```

Feedback from previous faulty implementation:
{PREVIOUS_IMPL_FEEDBACK}
Reflection on previous faulty implementation:```
"""

```

Code Generation Prompt Template

You are a {LANGUAGE} writing assistant. You are supposed to generate a {LANGUAGE} class based on the description of the class provided to you. You will be given your previous faulty implementation, feedback about what went wrong based on a few test cases, and a hint to change the implementation appropriately. Use these to provide a correct implementation of the class (include all necessary imports).

```

# Question 1:
Below are some potentially relevant pieces of information.
{FS_EXAMPLE_TOOL_OBSERVATIONS}
Based on the above, generate the following class
File: {FS_EXAMPLE_FILE_PATH}
Description: {FS_EXAMPLE_DESCRIPTION}
Previous faulty implementation:
```{LANGUAGE}
{FS_EXAMPLE_PREVIOUS_IMPL_V2}
```

Feedback from previous faulty implementation:
{FS_EXAMPLE_PREVIOUS_IMPL_FEEDBACK_V2}
Reflection on previous faulty implementation:
{FS_EXAMPLE_PREVIOUS_IMPL_REFLECTION_V2}
Corrected code based on feedback and reflection:
```{LANGUAGE}
{FS_EXAMPLE_CODE}
```

# Question 2:
Below are some potentially relevant pieces of information.
{TOOL_OBSERVATIONS}
Based on the above, generate the following class
File: {FILE_PATH}
Description: {DESCRIPTION}
Previous faulty implementation:
```{LANGUAGE}
{PREVIOUS_IMPL}
```

Feedback from previous faulty implementation:
{PREVIOUS_IMPL_FEEDBACK}
Reflection on previous faulty implementation:
{PREVIOUS_IMPL_REFLECTION}
Corrected code based on feedback and reflection:
```{LANGUAGE}"""

```

---

## Diff 1

Listing 1: Here, the diff represents the difference in the context provided to GPT-4 while generating NL description for the class in the Sketchy and Detailed settings. The green portion represents the extra details added for the prompt for the Detailed setting as compared to the Sketch setting.

```
--- file1_before.txt 2024-03-30 03:21:05.716952669 -0700
+++ file1_after.txt 2024-03-30 03:11:00.743386767 -0700
@@ -1,66 +1,92 @@
Class signature: class Accept:
Class full name: litestar.datastructures.headers.Accept

Functions accessible:
<Function details for function no. 0>
Function signature: def __init__(self, accept_value: str) -> None:
Function fqdn: litestar.datastructures.headers.Accept.__init__
Decorators:
Function docstring: Initialize the Accept header with an accept_value.
+Function body: def __init__(self, accept_value: str) -> None:
+ self._accepted_types = [MediaTypeHeader(t) for t in accept_value.split(",")]
+ self._accepted_types.sort(key=lambda t: t.priority, reverse=True)
</function details>

<Function details for function no. 1>
Function signature: def best_match(self, provided_types: List[str], default: Optional[
 str] = None) -> Optional[str]:
Function fqdn: litestar.datastructures.headers.Accept.best_match
Decorators:
Function docstring: """Find the best matching media type for the request.

Args:
 provided_types: A list of media types that can be provided as a response. These
 types
 can contain a wildcard ``*`` character in the main- or subtype
 part.
 default: The media type that is returned if none of the provided types match.

Returns:
 The best matching media type. If the matching provided type contains wildcard
 characters,
 they are replaced with the corresponding part of the accepted type.
 Otherwise the
 provided type is returned as-is.
"""
+Function body: def best_match(self, provided_types: List[str], default: Optional[str] = None) -> Optional[str]:
+ types = [MediaTypeHeader(t) for t in provided_types]
+
+ for accepted in self._accepted_types:
+ for provided in types:
+ if provided.match(accepted):
+ # Return the accepted type with wildcards replaced
+ # by concrete parts from the provided type
+ result = copy(provided)
+ if result.subtype == "*":
+ result.subtype = accepted.subtype
+ if result.maintype == "*":
+ result.maintype = accepted.maintype
+ return str(result)
+ return default
</function details>

<Function details for function no. 2>
Function signature: def accepts(self, media_type: str) -> bool:
Function fqdn: litestar.datastructures.headers.Accept.accepts
```

---

```

Decorators:
Function docstring: """Check if the request accepts the specified media type.

 If multiple media types can be provided, it is better to use :func:`best_match`.

Args:
 media_type: The media type to check for.

Returns:
 True if the request accepts ``media_type``.
"""
+Function body: def accepts(self, media_type: str) -> bool:
+ return self.best_match([media_type]) == media_type
</function details>

<Function details for function no. 3>
Function signature: def __len__(self) -> int:
Function fqdn: litestar.datastructures.headers.Accept.__len__
Decorators:
Function docstring: Returns the number of accepted types.
+Function body: def __len__(self) -> int:
+ return len(self._accepted_types)
</function details>

<Function details for function no. 4>
Function signature: def __getitem__(self, key: int) -> str:
Function fqdn: litestar.datastructures.headers.Accept.__getitem__
Decorators:
Function docstring: Returns the accepted type at the given index.
+Function body: def __getitem__(self, key: int) -> str:
+ return str(self._accepted_types[key])
</function details>

<Function details for function no. 5>
Function signature: def __iter__(self) -> Iterator[str]:
Function fqdn: litestar.datastructures.headers.Accept.__iter__
Decorators:
Function docstring: Returns an iterator over the accepted types.
+Function body: def __iter__(self) -> Iterator[str]:
+ return map(str, self._accepted_types)
</function details>

Class variables accessible:
* __slots__ = ("_accepted_types",) | defined in class `litestar.datastructures.headers.
 Accept`

Instance variables accessible:
* _accepted_types

Properties accessible: None

```

## Diff 2

Listing 2: A diff file showing the changes in the body of a candidate class after a symbol-renaming based paraphrase attempt.

```

--- file2_before.py 2024-03-30 03:51:38.882430546 -0700
+++ file2_after.py 2024-03-30 03:51:03.053152249 -0700
@@ -1,2 +1,2 @@
-class SESSiON(SessionREDIRECTmiXIn):
+class Session(SessionRedirectMixin):
 """A Requests session.
 #: :class:`Session <Session>`.
- self.headers = DeFauLt_hEAdERS()

```

---

```

+ self.headers = default_headers()

@@ -41,3 +50,3 @@
#: Event-handling hooks.
- self.hooks = DEFAULT_HOOKS()
+ self.hooks = default_hooks()

@@ -79,3 +88,3 @@
#: may be any other ``cookielib.CookieJar`` compatible object.
- self.cookies = COOKIJAR_FROM_DICT({})
+ self.cookies = cookiejar_from_dict({})

@@ -83,4 +92,4 @@
self.adapters = OrderedDict()
- self.mount('https://', HTTPAdapter())
- self.mount('http://', HTTPAdapter())
+ self.mount("https://", HTTPAdapter())
+ self.mount("http://", HTTPAdapter())

@@ -106,7 +115,7 @@
if not isinstance(cookies, cookielib.CookieJar):
- cookies = COOKIJAR_FROM_DICT(cookies)
+ cookies = cookiejar_from_dict(cookies)

Merge with session cookies
- merged_cookies = MERGE_COOKIES(
- MERGE_COOKIES(REQUESTSCOOKIEJAR(), self.cookies), cookies)
+ merged_cookies = merge_cookies(
+ merge_cookies(RequestsCookieJar(), self.cookies), cookies)

@@ -198,3 +209,3 @@
- settings = self.MERGE_ENVIRONMENT_SETTINGS(
+ settings = self.merge_environment_settings(
 prep.url, proxies, stream, verify, cert

@@ -221,5 +232,5 @@
kwargs.setdefault('allow_redirects', True)
- return self.REQUEST('GET', url, **kwargs)
+ return self.request('GET', url, **kwargs)

- def OPTIONS(self, url, **kwargs):
+ def options(self, url, **kwargs):
 r"""Sends a OPTIONS request. Returns :class:`Response` object.

@@ -373,17 +384,18 @@
 no_proxy = proxies.get('no_proxy') if proxies is not None else None
- env_proxies = Get_ENVIRON_PROXYES(url, no_proxy=no_proxy)
+ env_proxies = get_envIRON_proxies(url, no_proxy=no_proxy)
 proxies.setdefault(k, v)

Merge all the kwargs.
- proxies = MERGE_SETTING(proxies, self.proxies)
- stream = MERGE_SETTING(stream, self.stream)
- verify = MERGE_SETTING(verify, self.verify)
+ proxies = merge_setting(proxies, self.proxies)
+ stream = merge_setting(stream, self.stream)
+ verify = merge_setting(verify, self.verify)

```