Iterative Refinement of Project-Level Code Context for Precise Code Generation with Compiler Feedback

Anonymous ACL submission

Abstract

Large language models (LLMs) have shown remarkable progress in automated code generation. Yet, incorporating LLM-based code generation into real-life software projects poses challenges, as the generated code may contain errors in API usage, class, data structure, or missing project-specific information. As much of this project-specific context cannot fit into the prompts of LLMs, we must find ways to allow the model to explore the project-level code context. To this end, this paper puts forward a novel approach, termed CoCoGen, which iteratively refines the project-level code context for precise code generation, guided by the compiler feedback. In particular, CoCoGen first leverages compiler techniques to identify a mismatch between the generated code and the project's context. It then iteratively aligns and fixes the identified errors using information extracted from the code repository. We integrate CoCoGen with two representative LLMs, i.e., GPT-3.5-Turbo and Code Llama (13B), and apply it to Python code generation. Experimental results show that CoCoGen significantly improves the vanilla LLMs by over 80% in generating code dependent on project context, and consistently outperforms the existing retrieval-based code generation baselines.[1]

1 Introduction

Large language models (LLMs), especially those pre-trained on code, as demonstrated by tools such as GitHub Copilot (Friedman, 2021), Amazon's CodeWhisperer (Amazon, 2023), and ChatGPT (OpenAI, 2023a), are revolutionizing how developers approach programming by automatically generating code based on natural language intent and the context of surrounding code. While existing LLM-based code generation tools excel in handling small code samples, such as completing the current line or the body of a method or giving examples of a standard API, integrating LLM-based code generation into real-life software projects remains challenging (Li et al., 2024).

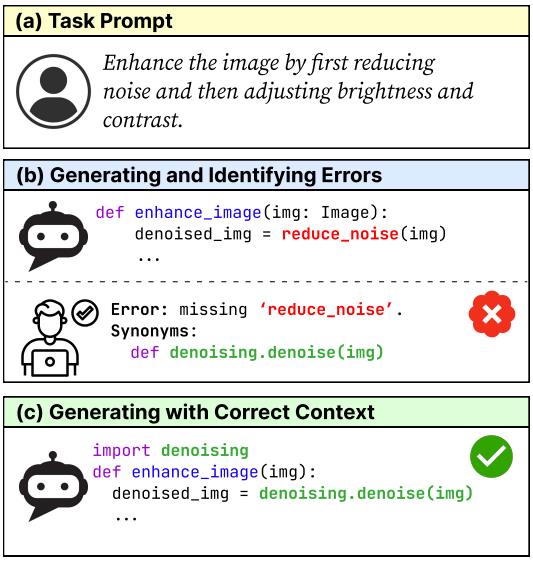

Figure 1: Example illustrating limitations of LLM-based code generation: (a) task prompt, (b) wrong solution and error identification, and (c) correct solution utilizing project context.

Practical code generation for software projects requires using project-specific contexts like classes, methods, and data structures due to modularity, maintenance, and code independence requirements. However, the repository's size often exceeds the capacity of the prompt, challenging LLMs to comprehend extensive contexts (Liao et al., 2023; Li et al., 2024). Also, studies have demonstrated that irrelevant context in prompts potentially compromises the model performance (Shi et al., 2023). Inputting only the task requirements can lead LLMs to overlook project-specific APIs, classes, data structures, or type information specific to a software project's repository, risking omitting essential logic during code generation (Gifany et al., 2013).

As a motivation example, consider the the case shown in Figure 1. In this case, we use the Ope-

[1] All the source code and dataset are available at: https://anonymous.4open.science/r/CoCoGen-4721/

nAI GPT-3.5-Turbo API to generate an image-enhancing function by first reducing the notice and then adjusting brightness and contrast. Given the prompt in Figure 1a, the LLM will produce a code snippet of calling a `reduce_noise` method as depicted in Figure 1b. Although the generated code follows a standard workflow specified in the task prompt, it leads to a compilation error in our application context because `reduce_noise` is not implemented. This is unsurprising as the prompt does not provide enough project-level context for the LLM, and can be fixed by providing the project-specific function `denoising.denoise` as context, as illustrated in Figure 1c. While a carefully engineered prompt may resolve this issue, it is not always possible for the user to generate such prompts, and the project context may be too large to fit into the prompt. As we will show later in the paper, such errors are commonly found when applying LLMs to repository-level code generation (Li et al., 2024).

This paper investigates a way to effectively integrate LLM-based code generation with existing code implementations within a software project. Our solution is to leverage project-level contextual information, such as project-specific implementation of classes, methods, and data structures, to reduce compilation errors and improve code quality. Directly incorporating the entire project code into a language model is infeasible due to model input sequence length limitations. Instead, we use compiler-based analysis to post-process the model-generated code by first detecting discrepancies between the generated code and the project's context. We then utilize information extracted from the project code base to rectify the mismatches in using modules, APIs, and classes. Our approach combines well-established compiler techniques with emerging generative methods, allowing software developers to leverage the power of LLMs without being overwhelmed and discouraged by the frequent compilation and semantic errors in the model-generated code.

We present CoCoGen, a method to allow an LLM to leverage the code repository of a software project to enhance the quality of the generated code. For a given LLM-generated code sample, CoCoGen first compiles it and identifies context-related errors. It then retrieves the related context from the code repository to fix the errors. This iterative generation and verification process proceeds repetitively until no error is identified in the generated solution. We demonstrate that CoCoGen enhances the accuracy of generation (as indicated by pass rates) by bridging the gap between the repository context and the intended solution.

We evaluate CoCoGen by applying it to the CoderEval benchmarking dataset (Yu et al., 2023), which consists of code generation tasks utilizing project-specific context. We test CoCoGen on two popular code generation models: the GPT-3.5-Turbo (OpenAI, 2023a) and Code Llama (Rozière et al., 2023). Experimental results demonstrate that CoCoGen significantly improves the repository-level code generation performance of different dependency levels, outperforming the baseline by over 80% relative pass rates in generating functions dependent on project-specific contexts. Moreover, our iterative method consistently enhances the performance of vanilla retrieval-augmented generation. We also provide a comprehensive analysis of the effectiveness and limitations of CoCoGen, offering insights for future research.

This paper makes the following contributions:

- An empirical study to analyze the error distribution in self-contained and repository-level code generation, highlighting the significance of precise and grounded program context in generating code at the project level (Section 2.2);
- A new iterative generation-verification-retrieval method that leverages the program compiler to eliminate context-related errors in repository-level code generation (Section 3);
- Extensive experiments and analysis based on two LLMs, i.e., GPT-3.5-Turbo and Code Llama (13B), validate the effectiveness of our proposed CoCoGen (Section 5).

2 Background

2.1 LLM-based Code Generation

Our work targets the code generation task, which produces source code from a natural-language description complemented by programming context (*e.g.*, project-specific APIs, data structures, and so on). We denote such input as x. Given x, it is first converted to a sequence of tokens $\boldsymbol{x} = [x_1, \ldots, x_{|\boldsymbol{x}|}]$, and a generation model $p_{\text{LM}}(\boldsymbol{x})$ predict new tokens sequentially. At each step t, the LM calculates the probability distribution of the next token as $p_{\text{LM}}(x_t|x_{1:t-1})$. The probability of generating a program y with token sequence

Table 1: Several typical errors reported in compilation and execution.

Error Type	Example
UNDEF	No name 'AsyncBolt5x0' in module 'neo4j._sync.io._bolt5'
API	No value for argument 'xmls' in function call
OBJECT	'function' object is not subscriptable
FUNC	The generated function not passes a test case
OTHER	Parsing failed: 'expected an indented block after function definition

$\boldsymbol{y} = [x_{|\boldsymbol{x}|+1}, \ldots, x_{|\boldsymbol{x}|+|\boldsymbol{y}|}]$ is computed as a product of next-token distributions given left context:

$$P(y|x) = \prod_{t=|\boldsymbol{x}|+1}^{|\boldsymbol{x}|+|\boldsymbol{y}|} p_{\text{LM}}(x_t|x_{1:t}) \,. \qquad (1)$$

For few-shot learning with large LMs, the generation is also often conditioned on a fixed set of m exemplars, $\{\langle x_i, y_i \rangle\}_{i \leq m}$. Thus, the LLM-based code generation can be formulated as:

$$P_{\text{LM}}(y|x) = P(y|x, \{\langle x_i, y_i \rangle\}_{i \leq m}) \,. \qquad (2)$$

Practically, the probability of the next token x_t depends on a fixed number of preceding tokens $x_{\max(1,t-w)} : x_{t-1}$, defined by the model's window length w, which may not encompass the entire software project's code base.

2.2 Error Analysis in Code Generation

The performance of simple function-level code generation has significantly improved, evidenced by an increase in the pass rate from a pass rate of 31.6% with CodeT5 (Wang et al., 2021) to 94.7% in the state-of-the-art Code Llama model (Rozière et al., 2023) on the widely-used HumanEval benchmark (Chen et al., 2021). However, a recent study (Yu et al., 2023) shows that existing LLMs for code generation struggle to generate code snippets that are dependent on the project context information, such as private APIs, types, and variables. To this end, various benchmarks, including ClassEval (Du et al., 2023) and CoderEval (Yu et al., 2023), have been devised to assess the performance of Language Models in generating context-dependent code within the project.

To better illustrate our motivation, we perform an empirical analysis of when the LLMs fail to generate complex code that is dependent on project-level context, on the CoderEval dataset (Yu et al., 2023). This dataset comprises 85 function-level tasks and 145 project-level tasks. Specifically, we select the GPT-3.5-TURBO (OpenAI, 2023a) as a target LLM for function-level code generation. Based on it, we choose a state-of-the-art method, termed RepoCoder (Zhang et al., 2023a), which retrieves the project-level context as augmentation and feeds 5 code fragments that have the highest DPR similarity score (Karpukhin et al., 2020) in prompt for better code generation.

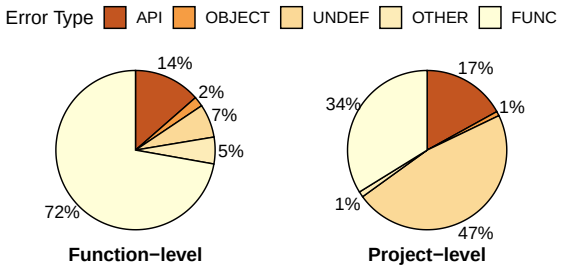

Figure 2: Distribution of error types in the generated solutions on CoderEval dataset.

As the code is generated, we compile the code and collect the errors reported by the compiler or in testing. Each generated code snippet contains precisely one type of error. If multiple errors are reported by the compiler, the most common error type is selected for analysis. The error distribution revealed that four specific types of errors constitute the majority of all errors encountered. We categorize these errors into: 1) *UNDEF*, involving Use of Undefined Symbol, 2) *API*, involving Incorrect Use of APIs, 3) *OBJECT*, involving Improper Use of an Object, 4) *FUNC*, involving Runtime or Functional Errors, and 5) *OTHER*, involving Other Syntax and Semantic Errors. Table 1 presents several errors encountered in compiling and testing. One example is the "*UNDEF*" error, where a variable `AsyncBolt5x0` is referenced but does not exist in the specified module.

Figure 2 illustrates the distribution of error types, under the scenario of function-level code generation and project-level code generation. From this figure, we can observe that the majority of errors are runtime or functional errors, accounting for 72% and 34% for function-level and project-level code generation, respectively. Furthermore, the *UNDEF* errors and the *API* errors account for substantially high portions of 21% and 64% for function-level and project-level code generation, respectively. COCOGEN addresses both types of errors associated with project context by supplying the relevant project context. Experiments demonstrate that COCOGEN not only fixes these two types of error but also mitigates other compilation errors by providing feedback on error messages to the language model, leading directly to an improvement in prediction accuracy.

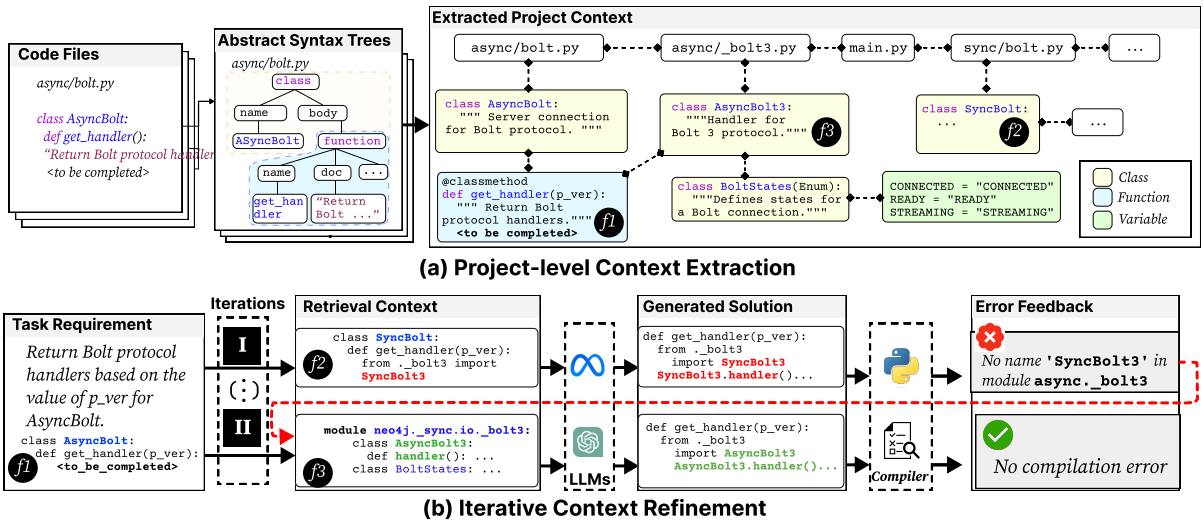

Figure 3: Overview of the CoCoGen method: (a) the project-level code context extraction process, and (b) iterative refinement to fix compiler-reported errors.

3 Methodology

3.1 Overview

Figure 3 depicts the workflow of CoCoGen, consisting of two crucial components: 1) a method for extracting project-level code context through both syntactic and semantic approaches, and 2) a component responsible for iterative generation and evaluation of solutions. This process refines the solutions incrementally, ensuring they evolve towards an error-free state that seamlessly aligns with the project's environment.

3.2 Project-Level Code Context Extraction

Supposing that the code generation tools are activated at a specific juncture. In light of the natural language requirement and the code produced by LLMs after an initial iteration, our objective is to extract the semantic context of the generated code from the project's code base.

Unlike plain texts, source code has syntactic structures that enable precise identification of elements in a project. Thus, in practice, we employ syntax-directed program analysis (Harrold and Rothermel, 1996) at various points throughout the offline stage to extract the code context at the project level. We initially employ a parser to transform each source code file within the project into an Abstract Syntax Tree (AST), extracting tree nodes that correspond to classes, functions, or variables. Subsequently, if a node of these types is found to be a child of another node (*e.g.*, the function `get_handler` and the class `AsyncBolt` in Figure 3(a)'s AST) , an edge is created from the parent node to the child node, establishing a hierarchical relationship.

Take the function f_1 from Figure 3(a) as an example. From this figure, we can see that both its semantics (*e.g.*, its docstring), and its syntactic relation between the parent class `AsyncBolt` and file `async/bolt.py` are captured. This establishes finding the function from the project syntactically using its qualified name `async/bolt.py:AsyncBolt.get_handler`, or semantically according to its docstring *"Return Bolt protocol handlers."*.

3.3 Retrieval-Augmented Code Generation

We leverage project-level code context in the retrieval-augmented generation paradigm (Zhang et al., 2023a; Ding et al., 2022; Karpukhin et al., 2020), which has been widely adopted to integrate factual knowledge into LLMs and address hallucination issues. In practice, we commence by extracting project-level context from the database through the construction of a Structured Query Language (SQL) query. Following this, we enhance the acquired context by retrieving similar code snippets based on the dense passage retrieval techniques (Karpukhin et al., 2020).

Based on the compiler feedback, we aim to retrieve the relevant project-level context from all extracted ones. We implement this by transforming the textual compiler feedback into an SQL query using the ChatGPT. The prompt used is presented in Figure 4(a). Here, several examples of paired compiler feedback and SQL queries are provided

(a) Prompt for SQL Synthesize

Task instruction: *Please generate SQL according to given error line content and error message.*

Demonstration Examples

Error Line Content: `loggerDict.RootLogger(msg)`

Error Message: *No name* **'RootLogger'** *in module* **'loggerDict'**

FROM Module m, Variable v WHERE m.getName() = 'loggerDict'

SELECT v

Error Line Content: `from ._bolt3 import AsyncBolt3`

Error Message: *No name* **'SyncBolt3'** *in module* **'async._bolt3'**

SQL: ***[to be completed]***

(b) Prompt for Code Generation

Task instruction: *Please generate code following the task requirement, fixing errors in previous solution according to relevant context (if exist).*

Demonstration Examples

Task Requirement: *Check if in given list of numbers, are any two numbers closer to each other than given threshold.*

Desired Solution: `def has_close_elements(numbers: List[float], threshold: float) -> bool ...`

Task Requirement: *Return Bolt protocol handlers based on the value of p_ver for AsyncBolt.*

Last Solution:

```
def handlers(p_ver):
  from ._bolt3 import SyncBolt3
...
```

Compiler Feedback

Error Line Content: `from ._bolt3 import SyncBolt3`

Error Message: *No name* **'SyncBolt3'** *in module* **'async._bolt3'**

Retrieved Context

Project Context:

```
module neo4j._async._bolt3:
  class AsyncBolt3:
    def get_handler(): ...
  class BoltStates: ...
```

Desired Solution: ***[to be completed]***

Figure 4: Prompt examples for SQL synthesize and code generation.

as demonstrations for in-context learning[2].

For instance, consider the compiler feedback: *No name* `'SyncBolt3'` *found in module* `'async._bolt3'`, the resulting SQL query generated by ChatGPT is as follows:

```
FROM Module m, Class c
WHERE m.contains(c)
and m.getName() = 'async._bolt3'
SELECT m, c
```

Using this SQL query, the code snippets that involve the implementations of `AsyncBolt3` will be returned from our constructed database.

Semantic Search. In addition to returning the project-level level retrieved by the SQL query, we also enhance the acquired context by retrieving similar code snippets using dense passage retrieval.

Specifically, given a natural language query q, we first convert it into an embedding vector by utilizing an encoder network, as follows:

$$\mathbf{h}_q = \text{ENCODER}(q)\,. \tag{3}$$

In our experiments, the pre-trained Transformer (Vaswani et al., 2017) is adopted as the encoder network.

[2]Only one demonstration example is illustrated in Figure 3(b), the full list of examples can be found in Appendix C.

After generating the query vector, we calculate the cosine similarity between the query and embedding vectors of each context entry $\boldsymbol{h}_c$. This similarity measure is defined as:

$$sim(\mathbf{h}_q, \mathbf{h}_c) = \frac{\mathbf{h}_q^\intercal \mathbf{h}_c}{||\mathbf{h}_q|| \cdot ||\mathbf{h}_c||}\,. \tag{4}$$

The top-n entries exhibiting the highest similarity to the query are retrieved as results.

3.4 Refinement with Compiler Feedback

Figure 3(b) showcases the iterative refinement pipeline. Given the task requirement and partial function f_1, a semantic retrieval is activated to identify similar functions. Specifically, the function f_2, which provides equivalent functionality in synchronous scenario, is identified. Utilizing both the retrieved context and the prompt illustrated in Figure 4(b), the language model generates a solution; However, the generated output mistakenly invokes `SyncBolt3` due to its intended use in asynchronous scenarios, not aligning with the synchronous scenario in f_2 The compiler's feedback highlights this error. With this feedback, CoCoGen conducts a hybrid structural and semantic search, leading to the discovery of the correct function f_3 for asynchronous scenarios. Incorporating the error details and context into the next iteration ensures accurate function invocation. This process goes iteratively until no error is reported by the compiler, resulting in an error-free solution that aligns with the project's environment.

4 Experimental Setup

4.1 Datasets

To validate the effectiveness of CoCoGen, we conducted experiments using the CoderEval benchmark (Yu et al., 2023), a recently developed benchmark designed to evaluate models within realistic software development scenarios. Without loss of generalizability, we concentrate on the Python programming language within this dataset. This benchmark categories 230 test samples into six levels of context dependency: 1) *self-contained*: built-in types/functions, no imports required; 2) *slib-runnable*: standard libraries/modules, no installation needed; 3) *plib-runnable*: publicly-available libraries on PyPI/Maven; 4) *class-runnable*: code outside the function but within a class; 5) *file-runnable*: code outside the class but within the file; and 6) *project-runnable*: code in other source

Table 2: Pass rates of CoCoGen based on two LLMs, i.e., GPT-3.5-Turbo and Code Llama (13B), assessed against various baselines across different data splits of CoderEval.

Data Split	Class Runnable			File Runnable			Project Runnable		
Method	Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10	Pass@1	Pass@5	Pass@10
LLM: GPT-3.5-Turbo									
Direct	8.73	12.57	14.55	19.85	27.62	30.88	9.57	12.08	13.04
ReACC	20.36	33.27	38.18	17.65	28.92	33.82	11.30	19.53	21.74
RepoCoder	**35.45**	40.46	41.82	29.41	34.61	36.76	16.96	19.57	21.74
CoCoGen	28.00	**44.92**	**49.09**	**30.29**	**43.58**	**47.06**	**21.30**	**36.73**	**39.13**
LLM: Code Llama (13B)									
Direct	18.91	30.65	34.55	18.53	27.82	29.41	5.22	8.70	13.04
ReACC	20.36	33.27	38.18	**17.65**	27.61	33.82	11.30	19.53	21.74
RepoCoder	17.82	35.22	40.00	15.00	28.31	32.35	**16.09**	21.36	21.74
CoCoGen	**26.36**	**39.42**	**41.82**	17.06	**29.39**	**33.82**	13.04	**28.04**	**34.78**

files. We concentrate on the last three dependency types, where the solutions are dependent on project-specific contexts. There are 55, 68, and 23 tasks associated with each dependency level, respectively.

4.2 Baseline Methods

CoCoGen can function seamlessly be integrated into existing LLMs, requiring only black-box access to these models. In this paper, we select two state-of-the-art LLMs for code generation, namely GPT-3.5-Turbo (OpenAI, 2023a) and Code Llama (13B) (Rozière et al., 2023), as our base models. To validate the effectiveness of CoCoGen, we compare it with the following baselines:

▷ **Direct** Generation (Yu et al., 2023). This line of method denotes directly inputting the task requirements into LLM for code generation, without providing additional context.

▷ **ReACC** (Lu et al., 2022). We employ the retrieval-augmented generation technique introduced in this baseline for code generation tasks. More precisely, we retrieve semantic contexts aligned with the task instructions and leverage them to augment the prompts of LLMs for better code generation.

▷ **RepoCoder** (Zhang et al., 2023a). Similar to our work, the referenced baseline also proposes the iterative refinement of generated code. Specifically, it involves retrieving similar code snippets derived from the previously generated ones, and employing them to augment the prompts of LLMs. One distinguishing feature is that this baseline does not leverage compiler feedback.

4.3 Implementation Details

In the inference process, we set the decoding temperature to 0.7, and adopt a top-k sampling strategy. We implement the retrieval modules based on the text-ada model introduced by OpenAI (OpenAI, 2023b), which is effective in both natural language search and code search. The dimension for each embedding is 1,536. We retrieve at most 5 entries for each query. All the experiments in this paper are conducted on a Linux server with 128GB memory, with a single 32GB Tesla V100 GPU.

4.4 Evaluation Metrics

Following previous studies (Chen et al., 2021; Yu et al., 2023), we evaluate the functional correctness of the generated code by executing test cases. We employ the Pass@k metric, where k denotes the number of programs generated for each task. A task is solved if at least one solution passes all unit tests, and we report the overall proportion of solved tasks. To reduce sampling variance, we generate $n \geq k$ solutions (for this study, $n = 20$ and $k = 1, 5, 10$) for each task, count the number of correct solutions $c \leq n$ that pass the unit tests, and calculate the unbiased estimator:

$$\text{Pass@}k = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right]. \quad (5)$$

5 Results and Analysis

5.1 Overall Performance of CoCoGen

Table 2 shows the overall performance of CoCoGen based on two LLMs, i.e., GPT-3.5-Turbo and Code Llama (13B), assessed against various baselines across different data splits. This figure shows that the CoCoGen can significantly outperform other baselines on the project-level code generation across different data splits. This trend persists, with a few exceptions noted specifically in terms of Pass@1. We attribute such exceptions to variations

Table 3: Pass Rates on CoderEval-Python of each approach with or without compiler feedback.

Method	Pass@1	Pass@5	Pass@10
Direct	20.65	26.66	29.13
Direct+CF	32.34	38.43	40.43
ReACC	34.13	41.44	43.48
ReACC+CF	36.60	44.05	46.52
RepoCoder	36.82	40.73	42.17
RepoCoder+CF	38.00	45.38	48.26

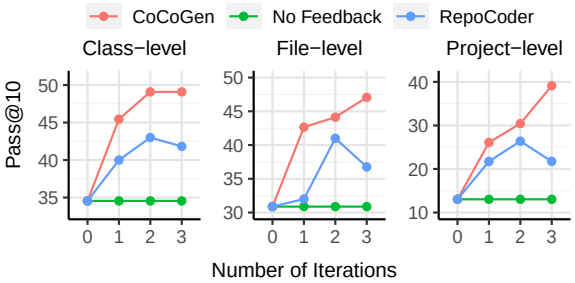

Figure 5: Performance of CoCoGen, RepoCoder, and No Feedback baseline across three dependency levels.

in the generated solutions. Selecting a significantly larger n (*e.g.*, 1000), as discussed in (Li et al., 2022), stabilizes expectations. Significantly, according to Code Llama (13B), when evaluated on the intricate project-runnable dataset, CoCoGen demonstrates superior performance compared to the state-of-the-art RepoCoder, achieving an 87.7% higher success rate at Pass@5 and a 79.9% improvement at Pass@10. Moreover, it becomes evident that models incorporating contextual information, such as ReACC, RepoCoder, and CoCoGen, exhibit a noteworthy performance superiority over the vanilla model, thereby affirming the practical value of contextual information.

5.2 Usefulness of Compiler Feedback

Here, we examine the usefulness of compiler feedback by integrating it into three baseline models: Vanilla, ReACC, and RepoCoder, each considered separately. We conducted experiments using the GPT-3.5-Turbo and subsequently reported the average score across the entire CoderEval dataset, as shown in Table 3. The table provides clear evidence that incorporating compiler feedback yields a notable enhancement in model performance for code generation. Specifically, a comparison between RepoCoder with and without compiler feedback reveals a substantial increase in Pass@1 from 36.82 to 38.00. Note that, the RepoCoder+CF here is equivalent to our proposed CoCoGen.

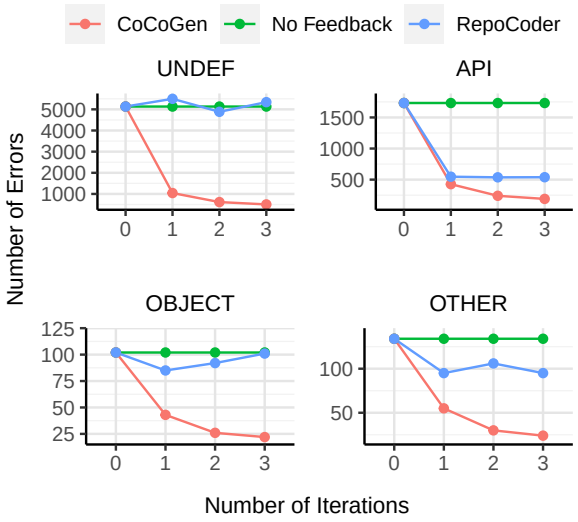

Figure 6: Compilation errors reported and fixed per iteration of CoCoGen, RepoCoder, and No Feedback baselines.

5.3 Effectiveness of the Iterative Refinement

Here, we investigate the effectiveness of iterative refinement in code generation with compiler feedback. We conduct an ablation analysis on both RepoCoder and CoCoGen, via removing or retaining the iterative refinement process. Figure 5 shows the performance of RepoCoder and CoCoGen, with respect to varying iterations, on different data splits. This figure clearly illustrates that as the number of iterations increases, the performance of CoCoGen also exhibits a corresponding improvement. It substantiates the efficacy of our suggested iterative refinement process, demonstrating its ability to enhance the generated code through multiple iterations progressively.

5.4 Error Analysis and Case Study

We also perform an error analysis of the generated code in iterative generation. We follow the categorization of errors defined in Section 2.2, examples of each error type can be found in Table 1. We exclude *FUNC* errors because they are reported in execution instead of by the compiler. Figure 6 shows the distribution of errors resolved iteratively by our CoCoGen and two baselines. From this figure, we can see that the errors of various types can be effectively resolved after a single iteration of refinement. For instance, *UNDEF* errors were notably reduced from 5133 to 1042 after one iteration. Additionally, it was observed that the RepoCoder baseline, which operates without compiler feedback, managed to rectify API and syntax errors, corroborating the findings in Zhang et al. (2023a).

Task Requirement *Use the value in timedelta to generate the Structure class.*

def dehydrate_timedelta(value):
:param value:
:type value: timedelta

Context
```
class timedelta: self.days
                 self.seconds ...
class Structure:
  def __init__(self, tag, fields)
```

Reference Solution
```
def dehydrate_timedelta(value):
    days = value.days
    seconds = value.seconds
    microseconds = value.microseconds
    nanoseconds = microseconds * 1000
    return Structure(b\"E\", days, seconds, 0, nanoseconds)
```

CoCoGen's Solution:
```
def dehydrate_timedelta(value):
    months = 0
    days = value.days
    seconds = value.seconds
    nanoseconds = 1000 * value.microseconds
    return Structure(b\"E\", months, days, seconds, nanoseconds)
```

Figure 7: An error case from CoderEval-Python.

Nonetheless, it proved ineffective against *UNDEF* and *OBJECT* errors, likely due to the model's lack of awareness regarding these errors in the absence of compiler feedback.

To thoroughly evaluate CoCoGen's effectiveness, we focused on scenarios where compilation was successful, but execution failed. In the case illustrated in Figure 7, CoCoGen incorrectly excludes the `mileseconds` field and erroneously adds a `month` field. This error stems from ambiguously stated task requirements and the model's lack of familiarity with the `Structure` class's format, despite its definition being available, resulting in misinterpretation of the intended functionality. Further examples are detailed in Appendix A. This observation inspires us to integrate a comprehensive reference comprising both documentation and web search results to provide explicit usage guidelines in our future work.

6 Related Work

LLM-based Code Generation. Automated code generation has a history spanning several decades, with initial endeavors utilizing rule-based systems (Backus et al., 1957; Woods, 1973) and structured prediction (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005). In recent years, the development of large-scale language models has led to the emergence of many prominent models in coding tasks. These include open-access models such as DeepSeek Coder (Bi et al., 2024), Code Llama (Rozière et al., 2023), CodeGen (Nijkamp et al., 2022), StarCoder (Li et al., 2023), and CodeGeeX2 (Zheng et al., 2023), alongside commercial offerings like GPT-3.5 (OpenAI, 2023a), Gemini (Google, 2024), and LLM-powered coding assistant tools such as GitHub Copilot (Microsoft, 2024) and Tabnine (Tabnine, 2024). These models and tools have demonstrated significant promise in enhancing code generation capabilities.

Context-aware Code Generation. In the context of project-level code generation, harnessing information beyond the current file has been found useful. The project can be extracted structurally into various forms like relational database (Pashakhanloo et al., 2022), API dependency graph (Lyu et al., 2021), and class hierarchies (Zhang et al., 2021). Shrivastava et al. (2023) proposes a code completion framework using repository-level prompt proposals, and then applying a classifier to classify useful proposals. (Lu et al., 2022; Zhang et al., 2023a) propose to use single or multiple levels of retrieval-augmented generation mechanisms for code generation. (Yu et al., 2023; Liu et al., 2023; Zhou et al., 2022) proposes benchmarks and datasets for repository-level code synthesis tasks. In our work, we select two code models, namely GPT-3.5 and Code Llama, and retrieve project context for fixing context-related compilation errors, to enhance the precision of code generation.

7 Conclusion

In this work, we show how to use project-specific context information, indexed structural and semantic, to fix compilation errors generated by compilers and improve the quality of code generated by LMs. Our experimental results show the increased prevalence of errors related to project contexts in project-level code generation compared to function-level code generation. The presented CoCoGen can effectively fix the compilation errors by retrieving related context from the project, thus significantly improving the native LLM baselines on over 80% relative pass rates in generating functions dependent on project-specific contexts.

Limitations

In this study, we utilize compilation information as a means to validate programs. However, it is important to note that even programs that compile successfully can experience execution failures. Furthermore, the successful compilation of programs does not guarantee their safety for execution. As a result, while our findings indicate improvements in quality metrics through the correction of code to properly leverage context, it is imperative to undertake additional verification methods such as testing and manual review to ascertain the functional correctness of the generated code. The challenge of ensuring functional correctness of code encompasses various aspects, including compliance with task re-

quirements, adherence to pre/post-conditions and security requirements, and preserving robustness in generating code. With many questions unanswered, we hope our study can promote a broader view of utilizing computational linguistic technologies in the realm of automated software engineering.

Ethics Statements

We meticulously ensured that all code and models integrated into our research adhere to open access policies as outlined by the Creative Commons license. The methodology ensures full compliance with copyright and intellectual property laws, thereby eliminating any potential for infringement or unauthorized use of protected materials. By exclusively utilizing resources that are freely available and legally distributable, we maintain the highest standards of ethical conduct in research. This approach fosters an environment of transparency and respect for the intellectual property rights of others. Our commitment to these principles ensures that our work advances the frontiers of knowledge in a manner that is both legally sound and ethically responsible.

References

C Amazon. 2023. Ai code generator—amazon codewhisperer.

J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. 1957. The fortran automatic coding system. In *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability*, IRE-AIEE-ACM '57 (Western), page 188–198, New York, NY, USA. Association for Computing Machinery.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007*.

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*.

Nat Friedman. 2021. Introducing github copilot: your ai pair programmer. *URL https://github. blog/2021-06-29-introducing-github-copilot-ai-pair-programmer*.

D Gifany, IS Amiri, M Ranjbar, and J Ali. 2013. Logic codes generation and transmission using an encoding-decoding system. *International Journal of Advances in Engineering & Technology*, 5(2):37.

Google. 2024. Gemini. https://deepmind.google/technologies/gemini/. [Online; accessed 1-Feb-2024].

Mary Jean Harrold and Gregg Rothermel. 1996. Syntax-directed construction of program dependence graphs. *Technical Report OSU-CISRC-5/96-TR32*.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Zhi Jin, Hao Zhu, Huanyu Liu, Kaibo Liu, Lecheng Wang, Zheng Fang, et al. 2024. Deveval: Evaluating code generation in practical software projects. *arXiv preprint arXiv:2401.06401*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Dianshu Liao, Shidong Pan, Qing Huang, Xiaoxue Ren, Zhenchang Xing, Huan Jin, and Qinying Li. 2023. Context-aware code generation framework for code repositories: Local, global, and third-party library awareness. *arXiv preprint arXiv:2312.05772*.

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

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722*.

Chen Lyu, Ruyun Wang, Hongyu Zhang, Hanwen Zhang, and Songlin Hu. 2021. Embedding api dependency graph for neural code generation. *Empirical Software Engineering*, 26:1–51.

Microsoft. 2024. Microsoft Copilot. https://www.microsoft.com/zh-cn/microsoft-copilot. [Online; accessed 1-Feb-2024].

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

OpenAI. 2023a. chatgpt. http://chat.openai.com. [Online; accessed 1-Feb-2023].

OpenAI. 2023b. New and improved embedding model. https://openai.com/blog/new-and-improved-embedding-model. [Online; accessed 1-Feb-2023].

Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2022. Codetrek: Flexible modeling of code using an extensible relational representation.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H Chi, Nathanael Schärli, and Denny Zhou. 2023. Large language models can be easily distracted by irrelevant context. In *International Conference on Machine Learning*, pages 31210–31227. PMLR.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pages 31693–31715. PMLR.

Tabnine. 2024. Tabnine. https://www.tabnine.com/. [Online; accessed 1-Feb-2024].

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of Advances in neural information processing systems*, pages 5998–6008.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

W. A. Woods. 1973. Progress in natural language understanding: an application to lunar geology. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, AFIPS '73, page 441–450, New York, NY, USA. Association for Computing Machinery.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. 2023. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. *arXiv preprint arXiv:2302.00288*.

John M. Zelle and Raymond J. Mooney. 1996. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, page 1050–1055. AAAI Press.

Luke S. Zettlemoyer and Michael Collins. 2005. Learning to map sentences to logical form: structured classification with probabilistic categorial grammars. In *Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence*, UAI'05, page 658–666, Arlington, Virginia, USA. AUAI Press.

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

Jiyang Zhang, Sheena Panthaplackel, Pengyu Nie, Raymond J Mooney, Junyi Jessy Li, and Milos Gligoric. 2021. Learning to generate code comments from class hierarchies. *arXiv preprint arXiv:2103.13426*.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023b. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x. *arXiv preprint arXiv:2303.17568*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2022. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

A More details about the Error Cases

In this section, we delve into additional cases where CoCoGen struggles to generate viable solutions, elucidating the underlying causes to foster further research. We identify two primary factors leading to execution errors despite successful compilation:

Task Requirement: *Returns the key in the form of int.*

def index(self, key):
:param key: a key
:return: index
:rtype: int

Context:

```
class Record():
    __keys: t.Tuple[str]
```

Reference Solution:

```
def index(self, key):
    if isinstance(key, int):
        if 0 <= key < len(self.__keys):
            return key
        raise IndexError(key)
    elif isinstance(key, str):
        try:
            return self.__keys.index(key)
        except ValueError:
            raise KeyError(key)
    else:
        raise TypeError(key)
```

CoCoGen's Solution:

```
def index(self, key):
    return int(key)
```

Figure A1: An error case of degenrate solution.

degenerate solutions, such as overly simplistic or redundant code. This phenomenon is discovered and explored in (Zhang et al., 2023b). Figure A1 exemplifies this issue, showcasing a solution that omits necessary validity checks and error handling, neither of which are specified in the task requirements, nor produced by CoCoGen's outputs. Our analysis reveals that degenerate solutions frequently pass compilation, rendering compiler-based verification ineffective.

Another prevalent mistake involves misinterpreting task requirements, resulting in solutions that lack logical coherence. Figure A2 depicts an instance where the assignment was to leverage the pre-existing `Point` class; instead, the language model disregards this specification and redundantly recreates the class. This underscores the challenge large language models (LLMs) face in accurately comprehending prompts and generating appropriate solutions.

B Errors Reported by the Compiler

We utilize `pylint` for error checking; it is a static code analyzer designed to inspect code within a project without executing it. The analyzer accepts the entire file containing the solution along with associated source files as input, and then it extracts errors that pertain directly to the lines in the generated solution from the entirety of identified errors. `pylint` generates a range of diagnostic messages, encompassing errors, warnings, recommendations for adhering to language conventions, and suggestions for refactoring to adhere to best coding practices. However, our focus is solely on the errors.

Each error is identified by an error code and described with an error message. The error code is a numerical identifier that reflects the error's nature, while the error message provides a concise description of the error. `Pylint` recognizes 133 distinct

Task Requirement: *Dynamically Generating Point Class.*

def dehydrate_timedelta(value):
:param value:
:type value: timedelta

Context:

```
class Point
    srid_table = {}
    srid_table_lock = Lock()
```

Reference Solution:

```
def point_type(name, fields, srid_map):
    def srid(self):
        try:
            return srid_map[len(self)]
        except KeyError:
            return None
    attributes = {\"srid\": property(srid)}
    for index, subclass_field in enumerate(fields):
        def accessor(self, i=index, f=subclass_field):
            try:
                return self[i]
            except IndexError:
                raise AttributeError(f)
        for field_alias in {subclass_field, \"xyz\"[index]}:
            attributes[field_alias] = property(accessor)
    cls = type(name, (Point,), attributes)
    with srid_table_lock:
        for dim, srid in srid_map.items():
            srid_table[srid] = (cls, dim)
    return cls
```

CoCoGen's Solution:

```
def point_type(name, fields, srid_map):
    fields_dict = {f: i for i, f in enumerate(fields)}
    srid = srid_map.get(len(fields), None)
    class Point:
        def __init__(self, *args):
            if len(args) != len(fields):
                raise ValueError(\"Wrong number of arguments\")
            self._coordinates = tuple(args)
            self._srid = srid
        @property
        def srid(self):
            return self._srid
        @srid.setter
        def srid(self, value):
            self._srid = value
        def __iter__(self):
            return iter(self._coordinates)
        def __eq__(self, other):
            return isinstance(other, Point) and
                self._coordinates == other._coordinates
        def __repr__(self):
            return f\"{name}({', '.join(
                        map(str, self._coordinates))})\"
    return Point
```

Figure A2: An error case of misinterpreting task requirements.

error types, and only 12 of these are reported frequently on the CoderEval dataset. We have categorized these errors into four groups based on their characteristics, detailed in Table 1. The *FUNC* error category is also included in the error distribution analysis but is excluded from the compiler feedback pipeline. This exclusion is because such errors can only be identified with execution, a process that extends beyond the scope of static analysis and is unsuitable for the real-time generation and refinement pipeline. The specific types of each error and corresponding error codes are documented in Table A1.

C Demonstration Examples of Structural Queries

We present a demonstration example in Section 3.3 to compose the structural query. The example focuses on identifying and addressing instances of missing or incorrectly utilized context entries. We detail several example error messages along with their corresponding structural queries. A comprehensive list of these examples can be found in Table A2.

Table A1: A complete list of frequently occurred errors reported by the compiler

Error Category	Error ID	Corresponding Error Code	Error Reason
UNDEF	UNDEF-P	E0401	Unable to import a Package.
	UNDEF-CM	E1101	A Class is accessed for an unexistent Member.
	UNDEF-API	E0611	A function or API cannot be found in a module.
	UNDEF-O	E0602	An undefined variable or Object is accessed.
API	API-TMA	E1121	A function call passes Too Many positional Arguments.
	API-IA	E1120	A function call passes Insufficient Arguments.
	API-WA	E1111	Assignment from the function that doesn't return anything.
		E1123	A function call passes a keyword argument which has no corresponding formal parameter.
OBJ	OBJ-NI	E1133	A Non-Iterable value is used in place where iterable is expected.
	OBJ-NC	E1102	An object being called is a Non-Callable object.
	OBJ-NS	E1136	A subscripted value does Not support Subscription.
OTHER	OTHER		Other errors reported by analyzer.

Table A2: A complete list of demonstration examples prompted to the language model

Error Type	Example Error Message	Action	Example Structural Query
UNDEF-P	Unable to import 'keys'	Confine the search scope in all modules	`from Module m where m.inSource() and v.getScope() = m select m`
UNDEF-CM	Instance of 'RootLogger' has no 'loggerDict' member	Confine the search scope in all members in the class	`from Module m, Class c, Function cf where m.inSource() and m.contains(c) and c.contains(cf) and cf.getScope() = c and c.getName = 'RootLogger' and not cf.isInitMethod() select m, c, cf`
UNDEF-API	No name 'AsyncBolt5x0' in module 'neo4j._sync.io._bolt5'	Confine the search scope in all names in the module	`from Module m, Variable v where m.inSource() and v.getScope() = m and m.getName() = 'neo4j._sync.io._bolt5' select m, v.getDefinition()`
API	No value for argument 'xmls' in function call 'dumpXML'	Return the information of the function	`from Module m, Function f where m.inSource() and m.contains(f) and f.getName() = 'dumpXML' select m, f`

D Algorithm for Project Database Construction

We present the comprehensive algorithm for constructing the project database and generating code with CoCoGen.

The algorithm for building the project database is detailed in Algorithm 1. It involves identifying source files in the project by extracting all files that end with a `.py` extension. To parse Python source files, we employ the `tree-sitter-python` parser for generating abstract syntax trees and `codeql-python` to extract the property of a context entry node. For passage encoding, we utilize `text-embedding-ada-002`, a model provided by OpenAI and accessible via an online API.

Algorithm 1 Project Database Construction

Require: SOURCEFILESET: A set of project source files
Require: PARSER: A parser for source files
Require: ENCODER: A passage encoder transforms text to numerical vector
Ensure: $databaseEntries$: Entries in the project database

```
databaseEntries ← ∅
for each sourceFile in SOURCEFILESET do
    nodesForVisit ← ⟨⟩
    propertyPrefixSeq ← ⟨⟩
    astFile ← PARSER(sourceFile)
    nodesForVisit.ADD(astFile.rootNode)
    while nodesForVisit is not empty do
        currentNode ← nodesForVisit.POP()
        if currentNode is PREFIXMARK then
            propertyPrefixSeq.POP()
        end if
        if currentNode.type is in [VARIABLETYPE, FUNCTIONTYPE, CLASSTYPE] then
            nodeProperty ← GETPROPERTIES(currentNode)
            nodeSchema ← ⟨propertyPrefixSeq, currentNode, nodeProperty⟩
            nodeEmbedding ← ENCODER(nodeSchema)
            databaseEntries.ADD([nodeSchema, nodeEmbedding])
            propertyPrefixSeq.PUSH(currentNode)
            nodesForVisit.PUSH(PREFIXMARK)
        end if
        for childNode in currentNode.CHILDS() do
            nodesForVisit.PUSH(childNode)
        end for
    end while
end for
```