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

**Zhangqian Bi**[1*] **Yao Wan**[1*†] **Zheng Wang**[2] **Hongyu Zhang**[3] **Batu Guan**[1]
**Fangxin Lu**[1] **Zili Zhang**[4] **Yulei Sui**[5] **Hai Jin**[1*] **Xuanhua Shi**[1*]
[1]Huazhong University of Science and Technology [2]University of Leeds
[3]Chongqing University [4]Shanghai Jiao Tong University [5]University of New South Wales
{zqbi,wanyao,hjin,xhshi}@hust.edu.cn



## Abstract

*Large Language Models* (LLMs) have shown remarkable progress in automated code generation. Yet, LLM-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. We present CoCoGen, a new code generation approach that uses compiler feedback to improve the LLM-generated code. CoCoGen first leverages static analysis to identify mismatches 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 the project context and consistently outperforms the existing retrieval-based code generation baselines.

## 1 Introduction

*Large Language Models* (LLMs), especially those pre-trained on code, as demonstrated by tools such as GitHub Copilot (Microsoft, 2024), Amazon's CodeWhisperer (Amazon, 2023), and ChatGPT (OpenAI, 2023), are revolutionizing how developers approach programming by automatically generating code for given contexts (*e.g.*, natural-language descriptions or surrounding incomplete code). While existing LLM-based code generation tools excel in code generation within a small-scale and isolated context (*i.e.*, a single file), integrating LLM-based code generation into real-world software projects remains challenging (Li et al., 2024).

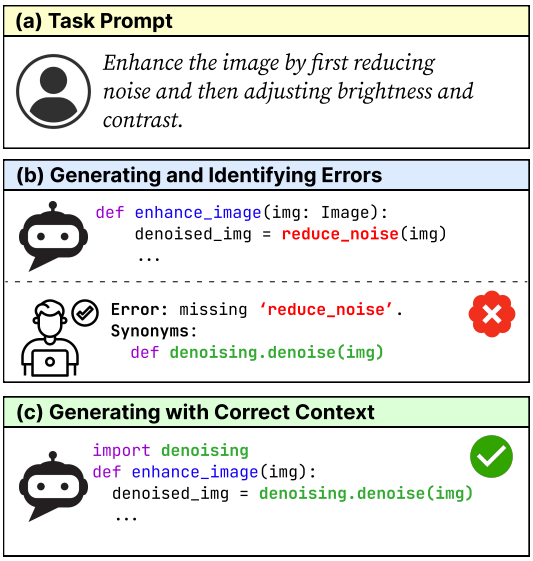


Figure 1: LLM-based code generation example. (**a**) task prompt; (**b**) wrong solution and error identification; (**c**) correct solution utilizing project context

Practical code generation for software repositories is often associated with broader project-level contexts declared in other repository files due to the demands of modularity, structure, and comprehension in software management (Kemerer, 1995). Solely providing the task requirements and surrounding incomplete code can lead LLMs to overlook the complex hierarchies of APIs, classes, data structures, or type constraints specific to a software project's repository, risking omitting essential logic during code generation (Gifany et al., 2013).

As a motivating example, consider the case shown in Figure 1. In this case, we use the OpenAI 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

[*]Also with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China.

[†]Yao Wan is the corresponding author.

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 empirically show later in (Section 2.2), 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., 2024), 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, 2023) 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), showing the effectiveness of the proposed CoCoGen method (Section 5).

The source code and dataset used in this paper are available at: https://github.com/CGCL-codes/naturalcc/tree/main/examples/cocogen.

## 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, and data structures). We denote this input as $x$. Given $x$, it is first converted to a sequence of tokens $\boldsymbol{x} = [x_1, \ldots, x_{|\boldsymbol{x}|}]$, and a generative *Language Model* (LM) $p_{\text{LM}}(\boldsymbol{x})$ predicts 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 $\boldsymbol{y} = [x_{|\boldsymbol{x}|+1}, \ldots, x_{|\boldsymbol{x}|+|\boldsymbol{y}|}]$ is computed as a prod-

Table 1: 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' |

uct of next-token distributions given left context:

$$p(y|x) = \prod_{t=|\boldsymbol{x}|+1}^{|\boldsymbol{x}|+|\boldsymbol{y}|} p_{\text{LM}}(x_t|x_{1:t}) \quad (1)$$

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:

$$p_{\text{LM}}(y|x) = p(y|x, \{\langle x_i, y_i\rangle\}_{i\leq m}) \quad (2)$$

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 context window length $w$, without encompassing 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, as demonstrated by an increase in the pass rate from 31.6% with CodeT5 (Wang et al., 2021) to 94.7% with the state-of-the-art Code Llama (Rozière et al., 2023) on the widely-used HumanEval benchmark (Chen et al., 2021). However, a recent study (Yu et al., 2024) shows that existing LLMs for code generation struggle to generate code snippets that are dependent on the project contexts, such as private APIs, classes, data structures, or type constraints. To this end, various benchmarks, including ClassEval (Du et al., 2024), CoderEval (Yu et al., 2024), and CrossCodeEval (Ding et al., 2023), have been devised to assess the performance of LLMs 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., 2024). This dataset comprises 85 function-level tasks and 145 project-level tasks. Specifically, we select the GPT-3.5-Turbo (OpenAI, 2023) as a target LLM for function-level code generation. Based on it, we select a state-of-the-art method called RepoCoder (Zhang et al., 2023a). RepoCoder retrieves project-level context as an augmentation

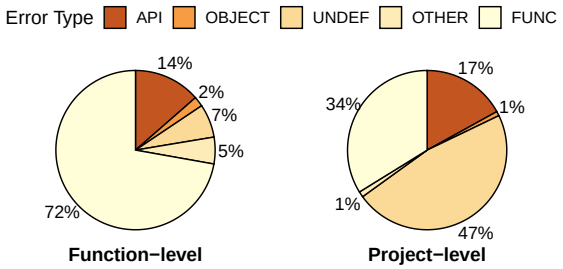


Figure 2: Distribution of error types in the generated solutions on CoderEval dataset

and incorporates the five code fragments with the highest similarity scores, as determined by dense passage retrieval (Karpukhin et al., 2020), into the prompt for improved code generation.

As the code is generated, we compile it and collect any errors reported by the compiler or encountered during testing. We have generated 10 candidate solutions for a task, comprising 850 solutions for function-level tasks and 1450 for project-level tasks. We report the Pass@10 rate as 53.57% for function-level tasks and 39.73% for project-level tasks. The code that does not pass the test has been taken into error analysis. 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 reveals 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 sup-

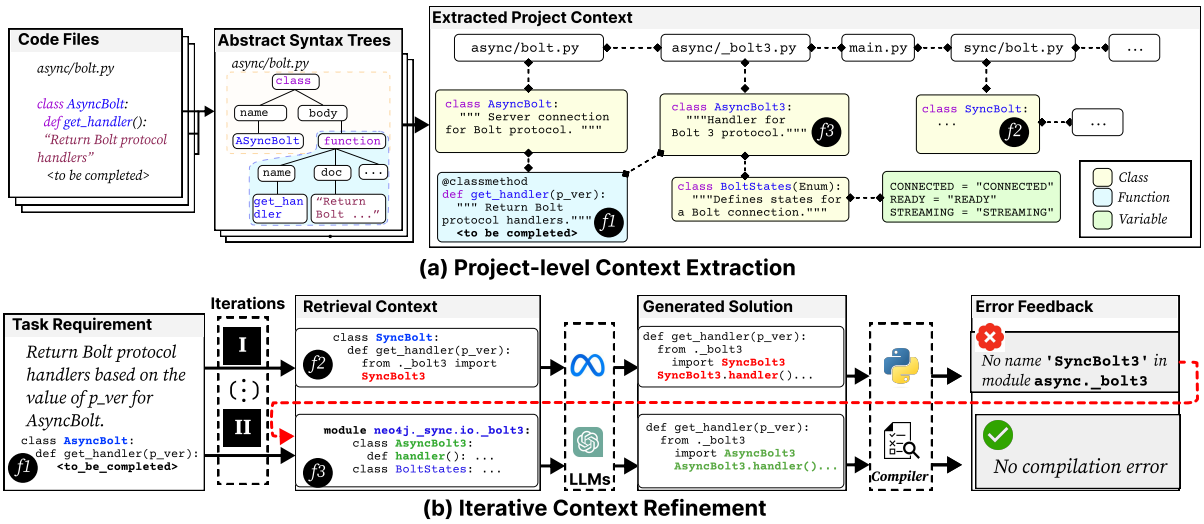


Figure 3: Overview of the CoCoGen method. (**a**) the project-level code context extraction process; (**b**) iterative refinement to fix compiler-reported errors

plying the relevant project context. Experiments demonstrate that CoCoGen not only fixes these two types of error but also mitigates other compilation errors by providing context feedback on error messages to the code LM, leading directly to an improvement in prediction accuracy.

## 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 generated solutions incrementally, ensuring they evolve towards an error-free state that seamlessly aligns with the codebase of the software project.

### 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 (Aho et al., 2006) 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 3a'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 3a 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 allows CoCoGen to find the function from the project syntactically using the function's qualified name `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., 2024; 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).

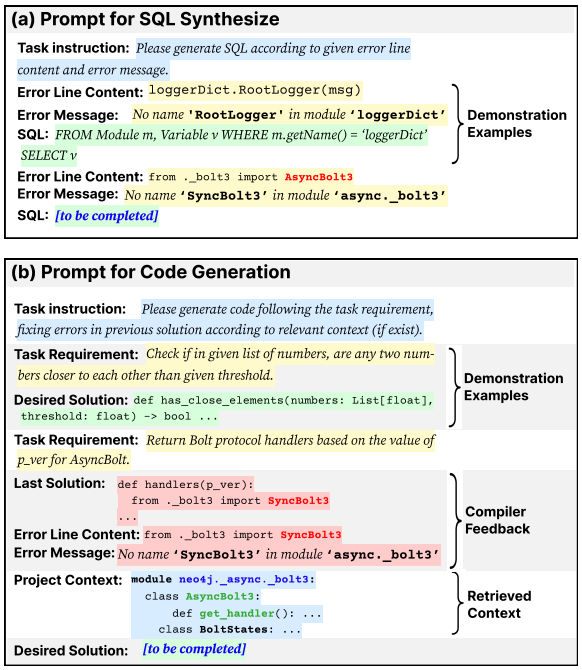


Figure 4: Prompt examples for (**a**) SQL synthesize and (**b**) code generation

**Structural Search.** 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 4a. Several examples of paired compiler feedback and SQL queries are provided as demonstrations for in-context learning*.

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. The detailed process of constructing and querying such SQL database is refer to Appendix G.

**Semantic Search.** In addition to returning the project-level context retrieved by the SQL query, we also enhance the acquired context by retrieving similar code snippets using dense passage retrieval (Karpukhin et al., 2020). In the initial search round, no compilation error is reported, and CoCoGen utilizes the task description string for retrieval. In subsequent searches, CoCoGen employs the error report and the corresponding error line.

Given a natural-language query $q$, CoCoGen first converts it into an embedding vector by utilizing an encoder network, as follows:

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

CoCoGen utilizes a pre-trained Transformer network (Vaswani et al., 2017) as the encoder. After generating the query vector, CoCoGen calculates the cosine similarity between the query and embedding vectors of each context entry $\boldsymbol{h}_c$. This similarity measure is defined as:

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

and 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 the 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 iterative until no error is reported by the compiler, resulting in an error-free solution that aligns with the project's environment.

It is noteworthy that CoCoGen does not take into account FUNC errors, which arise during execution despite successful compilation. CoCoGen focuses on addressing compilation errors, which constitute 66% of the total errors in the context of project-level code generation, as shown in Figure 6.

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

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 splits of the CoderEval dataset

| 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** |

# 4 Experimental Setup

## 4.1 Models and Datasets

To validate the effectiveness of CoCoGen, we select GPT-3.5-Turbo and Code-Llama 13B base[†] language models for investigation. The technical details of models and their invocation for inference are presented in Appendix A.

We conduct experiments using the Python split of the CoderEval benchmark (Yu et al., 2024), referred to as CoderEval-Python. It is a 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 categorizes 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 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.

We also evaluate CoCoGen on two function-level code generation benchmarks, namely HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021), as well as a project-level code completion benchmark named CrossCodeEval (Ding et al., 2023), to further validate its generalizability across other coding tasks. the statistics and experimental results of these benchmarks can be found in Appendix B and Appendix F.

[†] https://huggingface.co/codellama/CodeLlama-13b-hf

## 4.2 Baseline Methods

CoCoGen can function seamlessly and 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, 2023) 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., 2024). 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 project contexts aligned with the task instructions semantically through embedding similarity, 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 Evaluation Metrics

We employ the Pass@$k$ metric (Chen et al., 2021; Yu et al., 2024) for code generation tasks, and em-

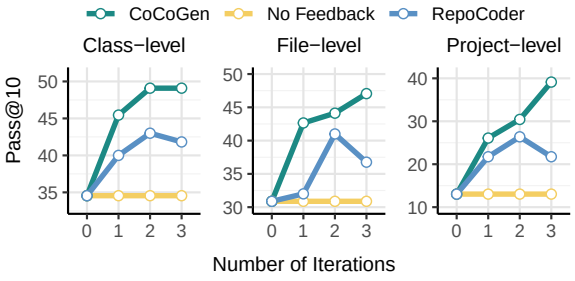


Figure 5: Pass@10 of CoCoGen, RepoCoder, and No Feedback baseline across three dependency levels

ploy the code exact match (C-EM), code edit similarity (C-ES), identifier exact match (I-EM), and identifier F1 score (I-F1) to evaluate both the code match rate and the identifier match rate for code completion tasks, follows Ding et al. (2023). We present the details of the Pass@$k$ metric for code generation. For metrics used in the code completion task, readers are referred to Appendix C.

**Pass@$k$.** Following previous studies (Chen et al., 2021; Yu et al., 2024), 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, assessed against various baselines, on the CoderEval (Yu et al., 2024) and the CrossCodeEval (Ding et al., 2023) dataset, respectively. This table shows that the CoCoGen can significantly outperform other baselines on the project-level code generation task. This trend persists, with a few exceptions noted specifically in terms of Pass@1. We attribute such exceptions to variations in the generated solutions. Selecting a significantly larger $n$ (*e.g.*, 1000), as discussed in (Li et al., 2022), stabilizes the result and eliminates these expectations. Moreover, it becomes evident that models incorporating contextual information, such as ReACC, RepoCoder, and CoCoGen, exhibit a noteworthy performance superiority over the no context (*i.e.*, direct) model, thereby affirming the practical value of the context in project-level code generation.

Table 3: Pass rates of CoCoGen with components ablated, based on GPT-3.5-Turbo model using the CoderEval-Python dataset

| Method | Pass@1 | Pass@5 | Pass@10 |
|---|---|---|---|
| CoCoGen | 28.01 | **43.01** | **46.58** |
| - w/o CF and SQL (RepoCoder) | **28.72** | 34.44 | 36.30 |
| - w/ CF, w/o SQL, w/o Semantic | 25.69 | 37.50 | 41.78 |
| - w/ CF and SQL, w/o Semantic | 26.37 | 38.31 | 41.78 |
| - w/ CF and Semantic, w/o SQL | 27.39 | 40.02 | 44.45 |

## 5.2 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. The improvement substantiates the efficacy of our suggested iterative refinement process, demonstrating its ability to enhance the generated code through multiple iterations progressively.

## 5.3 Ablation on Components

To explore the impact of each component within CoCoGen on overall performance, we remove various components from CoCoGen, including the compiler, SQL retriever, and semantic retriever. Notably, the configuration that excludes the compiler and SQL, relying solely on semantic retrieval, is known as RepoCoder (Zhang et al., 2023a).

Table 3 demonstrates the pass rates of CoCoGen across CoderEval-Python. The data indicates that incorporating compiler feedback does improve accuracy, although not markedly substantial. The reason is that highlighting compilation errors alone does not provide related context for resolving them, which is important in project-level coding problems. Also, compared to RepoCoder—which relies solely on semantic retrieval, integrating compiler feedback with project-specific context results in a stable performance improvement. Results on each data split can be found in Appendix F.2.

Further analytical experiments, including the

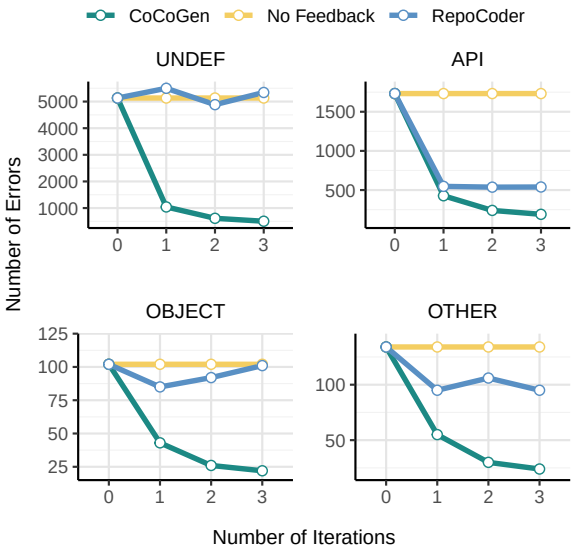


Figure 6: Compilation errors fixed per iteration of CoCoGen, RepoCoder, and No Feedback baselines

evaluation of benefits from solely compiler feedback, the performance of CoCoGen on project-level code completion and function-level code generation, and the efficiency of static analysis and SQL queries, can be found in Appendix F.

### 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. 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 are notably reduced from 5,133 to 1,042 after one iteration. Additionally, it is observed that the RepoCoder baseline, which operates without compiler feedback, manages to rectify API and syntax errors, corroborating the findings in Zhang et al. (2023a). Nonetheless, RepoCoder proves 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 focus on scenarios where compilation is successful, but execution fails. In the case illustrated in Figure 7, CoCoGen incorrectly excludes the `microseconds` field and erroneously adds a `month` field. This error stems from ambiguously

**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 example of a runtime error in CoCoGen's generated code for a CoderEval test case

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 E. The observations inspire us to integrate a comprehensive reference comprising documentation, web search results, and code execution log to provide explicit guidelines for code generation 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 (Woods, 1973) and structured prediction (Zelle and Mooney, 1996; Zettlemoyer and Collins, 2005). In recent years, the development of LLMs 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), and StarCoder (Li et al., 2023), alongside commercial offerings like GPT-3.5 (OpenAI, 2023) and GitHub Copilot (Microsoft, 2024). These models and tools have demonstrated significant promise in enhancing code generation capabilities.

**Project-level Code Generation.** Generating accurate code within a project poses challenges due to the modular design of software engineering, which results in cross-file dependency patterns (Parnas, 1972). Early works augmented N-gram, RNN, and LSTM models with an additional cache model to track project-level changes (Tu et al., 2014; Hellendoorn and Devanbu, 2017). Pashakhanloo et al. (2023) transformed projects into a relational database and proposes a graph walking method to traverse this database. Zan et al. (2022) first used

private API documentation to improve code generation, and Zan et al. (2023) investigated how various components of API documentation influence prediction accuracy. Shrivastava et al. (2023) proposed a code completion framework using a classifier to filter useful repository-level prompt proposals. Lu et al. (2022); Zhang et al. (2023a) proposed to use single or multiple levels of retrieval-augmented generation mechanisms for code generation. Liao et al. (2023) proposed A$^3$-CodGen, which utilizes local, global, and third-party-library information for better context retrieval. (Yu et al., 2024; Liu et al., 2023; Zhou et al., 2022; Ding et al., 2023) proposed benchmarks and datasets for repository-level code generation tasks.

**Post-processing of LLMs for Code Generation.** To identify correct code generated by LLM, researchers employ post-processing techniques to further rank and filter the generated code. Inala et al. (2022) trained a fault-aware neural ranker that ranks multiple code samples based on compilation feedback. AlphaCode (Li et al., 2022) and Shi et al. (2022) employed filtering methods based on the execution feedback. Zhang et al. (2023c) reranked LLM outputs based on back translation. SelfEdit (Zhang et al., 2023b) employed a generate-and-edit approach that utilizes execution results to improve the competitive programming task code quality. Chen et al. (2023) proposed a rubber duck debugging mechanism without human feedback, which identifies mistakes in generated code by investigating the execution results, and explaining the generated code in natural language.

## 7 Conclusion

In this paper, 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 LLMs. 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.

## Acknowledgements

This work is supported by the Major Program (JD) of Hubei Province (Grant No. 2023BAA024), and the National Natural Science Foundation of China under grant No. 62102157. This work is also partially supported by Huawei. Fangxin Lu is a visiting student at Huazhong University of Science and Technology, and she is from South-Central Minzu University for Nationalities.

## Limitations

In this paper, 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 requirements, 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 ensure 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


Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

Amazon. 2023. Ai code generator—amazon codewhisperer. Online. Accessed 1-Feb-2024.

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

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, A. X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. 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, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Advances in Neural Information Processing Systems*, volume 36, pages 46701–46723. Curran Associates, Inc.

Yangruibo Ding, Zijian Wang, Wasi U. Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. CoCoMIC: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445, Torino, Italia. ELRA and ICCL.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

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.

Vincent J. Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 763–773, New York, NY, USA. Association for Computing Machinery.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*, volume 35, pages 13419–13432. Curran Associates, Inc.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online. Association for Computational Linguistics.

Chris F. Kemerer. 1995. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering*, 1(1):1–22.

Vladimir I. Levenshtein. 1965. Binary codes capable of correcting deletions, insertions, and reversals. In *Doklady Akademii Nauk SSSR*, volume 163, pages 845–848. Soviet Union.

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

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, Muhtasham 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 Kunakov, 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. 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, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 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. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6227–6240, Dublin, Ireland. Association for Computational Linguistics.

Microsoft. 2024. Microsoft Copilot. Online. Accessed 1-Feb-2024.

OpenAI. 2022. New and improved embedding model. Online. Accessed 1-Feb-2024.

OpenAI. 2023. ChatGPT. Online. Accessed 1-Feb-2024.

David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Pardis Pashakhanloo, Aaditya Naik, Yuepeng Wang, Hanjun Dai, Petros Maniatis, and Mayur Naik. 2023. Codetrek: Flexible modeling of code using an extensible relational representation. In *Proceedings of 10th International Conference on Learning Representations*.

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. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. 2023. Repository-level prompt generation for large language models of code. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31693–31715. PMLR.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan,

Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 269–280, New York, NY, USA. Association for Computing Machinery.

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 *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. 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, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

William A. Woods. 1973. Progress in natural language understanding: an application to lunar geology. In *Proceedings of the 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, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Daoguang Zan, Bei Chen, Yongshun Gong, Junzhi Cao, Fengji Zhang, Bingchao Wu, Bei Guan, Yilong Yin, and Yongji Wang. 2023. Private-library-oriented code generation with large language models. *arXiv preprint arXiv:2307.15370*.

Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2022. When language model meets private library. In *Proceedings of Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 277–288, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

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, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023a. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, Singapore. Association for Computational Linguistics.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023b. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-Tau Yih, Daniel Fried, and Sida Wang. 2023c. Coder reviewer reranking for code generation. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 41832–41846. PMLR.

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

## A More Details on Investigated Large Language Models

In this paper, we have selected GPT-3.5-Turbo and Code-Llama 13B for investigation.

**GPT-3.5-Turbo (OpenAI, 2023).** It is a large-scale decoder-only model based on the Transformer architecture (Vaswani et al., 2017). It is pre-trained on a diverse array of data, encompassing both natural language and code, and can learn a specific task given an instruction and several demonstration examples. We perform the inference by invocating the model through an online API.

**Code Llama (Rozière et al., 2023).** It is a family of LLMs for code based on Llama 2 (Touvron et al., 2023) and further trained on 1TB code and natural language tokens. It can be used for code completion and generation following natural language instructions. For Code-Llama 13B, we utilize its base variant, specifically designed for general code generation and understanding.[‡]

[‡]The model can be accessed via https://huggingface.co/codellama/CodeLlama-13b-hf.

In the inference stage, we set the decoding temperature to 0.7, and adopt the top-$k$ sampling strategy. We implement the retrieval modules based on the `text-ada` model introduced by OpenAI (OpenAI, 2022), which is effective in both natural language search and code search. The embedding dimension is 1,536 for the `text-ada` model. 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 four 32GB Tesla V100 GPUs.

## B More Details on the Investigated Dataset

In this paper, we also use three recently proposed benchmarks evaluating the project-level code completion and function-level code generation ability. We provide the details of each benchmark.

**CrossCodeEval (Ding et al., 2023).** It is a project-level code completion benchmark that necessitates cross-file contextual understanding to complete the code accurately. CrossCodeEval is built on a diverse set of real-world, open-sourced, permissively-licensed repositories, including 471 repositories, 2,665 test cases across 1,368 files for Python language (Ding et al., 2023).

**HumanEval (Chen et al., 2021).** It is a function-level code generation benchmark that consists of 164 programming problems with corresponding Python solutions. These problems cover a range of difficulty levels and programming concepts. HumanEval is widely used in evaluating LMs for code generation tasks due to its focus on real-world coding scenarios and its comprehensive test cases.

**MBPP (Austin et al., 2021).** It is a function-level code generation benchmark consisting of 974 crowd-sourced Python programming problems. These problems are designed to be solvable by entry-level programmers and cover programming fundamentals and standard library functionalities. Each problem includes a task description, a code solution, and several automated test cases.

## C More Details on the Evaluation Metrics

In this work, we use several evaluation metrics for method generation and line completion tasks. This includes:

**Exact Match.** This metric assesses whether each character of the model's predicted code exactly matches each character of the correct answer. All characters are the same yields $EM = 1$, while any discrepancy results in $EM = 0$. This is a strict all-or-nothing metric, where a single character difference results in a score of 0. The average value of EM on all test cases is reported.

**Edit Similarity (Levenshtein, 1965).** This metric quantifies how dissimilar two strings are to one another. It is measured by counting the minimum number of operations required to transform one string into the other:

$$\mathrm{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) & \text{if } \mathrm{head}(a) = \mathrm{head}(b), \\ 1 + \min \begin{cases} \mathrm{lev}(\mathrm{tail}(a), b) \\ \mathrm{lev}(a, \mathrm{tail}(b)) \\ \mathrm{lev}(\mathrm{tail}(a), \mathrm{tail}(b)) \end{cases} & \text{otherwise} \end{cases} \tag{6}$$

**F1 Score (Schütze et al., 2008).** This metric evaluates the balance between precision and recall for the retrieved identifiers. The F1 score is based on the number of identifiers that both the prediction and the ground truth share. Precision is defined as the ratio of the shared identifiers to the total number of identifiers in the prediction, while recall is the ratio of the shared identifiers to the total number of identifiers in the ground truth. The F1 score is then calculated as the harmonic mean of the recall rate and the precision rate:

$$F_1 = \frac{2}{\mathrm{recall}^{-1} + \mathrm{precision}^{-1}} \tag{7}$$

## D 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

Table 4: 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 5: 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 |

Table 6: Pass rates of CoCoGen on Class Runnable test cases

| Data Split | Class Runnable | | |
|---|---|---|---|
| Method | Pass@1 | Pass@5 | Pass@10 |
| CoCoGen | 28.00 | 44.92 | **49.09** |
| - w/o CF and SQL (RepoCoder) | **35.45** | 40.46 | 41.82 |
| - w/ CF, w/o SQL, w/o Semantic | 30.00 | 42.58 | 45.45 |
| - w/ CF and SQL, w/o Semantic | 30.36 | 44.61 | 49.09 |
| - w/ CF and Semantic, w/o SQL | 31.45 | **45.00** | 47.27 |

distinct error types, and only 12 of these are reported frequently on the CoderEval-Python 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 4.

## E More Details About the Error Cases

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. The first factor is the occurrence of degenerate solutions, such as overly simplistic or redundant code. This phe-

**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 8: An error case of degenrate solution

**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 9: An error case of misinterpreting task requirements

Table 7: Pass rates of CoCoGen on File Runnable test cases

| Data Split | File Runnable | | |
|---|---|---|---|
| Method | Pass@1 | Pass@5 | Pass@10 |
| CoCoGen | **30.29** | **43.58** | **47.06** |
| - w/o CF and SQL (RepoCoder) | 29.41 | 34.61 | 36.76 |
| - w/ CF, w/o SQL, w/o Semantic | 26.03 | 37.41 | 42.65 |
| - w/ CF and SQL, w/o Semantic | 26.76 | 37.68 | 39.71 |
| - w/ CF and Semantic, w/o SQL | 27.35 | 39.90 | 45.45 |

Table 8: Pass rates of CoCoGen on Project Runnable test cases

| Data Split | Project Runnable | | |
|---|---|---|---|
| Method | Pass@1 | Pass@5 | Pass@10 |
| CoCoGen | **21.30** | **36.73** | **39.13** |
| - w/o CF and SQL (RepoCoder) | 16.96 | 19.57 | 21.74 |
| - w/ CF, w/o SQL, w/o Semantic | 14.35 | 25.62 | 30.43 |
| - w/ CF and SQL, w/o Semantic | 15.65 | 25.12 | 30.43 |
| - w/ CF and Semantic, w/o SQL | 17.83 | 28.50 | 34.78 |

nomenon is discovered and explored in (Zhang et al., 2023c). Figure 8 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 9 depicts an instance where the assignment is to leverage the pre-existing `Point` class; instead, the LM disregards this specification and redundantly recreates the class. This underscores the challenge LLMs face in accurately comprehending prompts and generating appropriate solutions.

# F More Analytic Experiments

## F.1 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 conduct experiments using the GPT-3.5-Turbo and subsequently reported the average score across the entire CoderEval-Python dataset consisting of six levels of context dependencies, as shown in Table 5. 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.

Table 9: Pass rates of CoCoGen on varying iterations

| **Iteration** | **Class Level** | **File Level** | **Project Level** |
|---|---|---|---|
| $i = 0$ | 34.55 | 30.88 | 13.04 |
| $i = 3$ | **49.09** | **47.06** | **39.13** |
| $i = 10$ | 47.27 | 45.59 | **39.13** |

Table 10: C-EM, C-ES, I-EM, and I-F1 scores based on the GPT-3.5-Turbo on the CrossCodeEval dataset

| Category | Code Match | | Identifier Match | |
|---|---|---|---|---|
| Method | C-EM | C-ES | I-EM | I-F1 |
| Direct | 1.91 | 50.51 | 3.60 | 32.27 |
| ReAcc | 5.48 | 54.03 | 10.02 | 38.49 |
| RepoCoder | 8.52 | 55.05 | 13.02 | 41.05 |
| CoCoGen | **9.08** | **55.31** | **14.11** | **42.34** |

## F.2 More Results on the Ablation Study

To further examine the performance of each component of CoCoGen across different dependency levels of test cases, we evaluate CoCoGen at various dependency levels from the CoderEval-Python test suite. The results are presented in Table 6, Table 7, and Table 8. The results show that utilizing compiler feedback and structural queries consistently demonstrates performance improvements across most scenarios. Furthermore, it achieves better performance when stronger cross-file dependencies are present (*i.e.*, at the file-level and project-level tasks), indicating that CoCoGen can accurately capture the context across the project.

## F.3 Varying the Number of Iterations

To investigate whether iteration could continuously improve performance, we increase the number of iterations to ten. As shown in Table 9, with more iterations, the performance of CoCoGen reaches a plateau. This indicates that there are still errors that cannot got repaired by CoCoGen, leaving for further research.

## F.4 CoCoGen on Project-level Code Completion

We also evaluate CoCoGen on project-level code completion tasks. Although CoCoGen is originally designed for code generation tasks, the compiler feedback may also benefit this code completion task. The results are presented in Table 10. From the table, we can see that the compiler feedback does improve performance, although not as significantly as it does in the code generation task.

Table 11: The performance of CoCoGen across varying iterations on the HumanEval benchmark (CE: Compilation Errors, CE%: their proportion among all generated solutions)

| | Pass@1 | Pass@5 | Pass@10 | CE | CE(%) |
|---|---|---|---|---|---|
| Direct | 61.52 | 80.56 | 83.15 | 40 | 2.38% |
| CF $i = 1$ | 71.46 | 82.46 | 85.20 | 7 | 0.42% |
| CF $i = 2$ | 71.40 | 82.56 | 85.98 | 3 | 0.18% |
| CF $i = 3$ | 71.65 | 82.49 | 85.36 | 3 | 0.18% |

Table 12: The performance of CoCoGen across varying iterations on the MBPP benchmark (CE: Compilation Errors, CE%: their proportion among all generated solutions)

| | Pass@1 | Pass@5 | Pass@10 | CE | CE(%) |
|---|---|---|---|---|---|
| Direct | 49.71 | 58.50 | 60.81 | 185 | 1.90% |
| CF $i = 1$ | 52.95 | 59.58 | 61.85 | 11 | 0.11% |
| CF $i = 2$ | 53.06 | 59.87 | 62.01 | 9 | 0.09% |
| CF $i = 3$ | 52.79 | 59.80 | 62.09 | 7 | 0.07% |

### F.5 CoCoGen on Function-level Code Generation

To assess the efficacy of CoCoGen in function-level code generation, we evaluate it on two commonly used datasets: HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). We retain the compiler feedback model in CoCoGen, and conduct tests in a 0-shot scenario, generating 10 code samples for each of the 163 test cases for HumanEval and 974 cases for MBPP, totaling 11,370 samples. The generation and refinement loop iterates 3 times, as shown in Table 11 and Table 12.

The data presented in the table illustrates that compilation feedback improves the accuracy of the generated code, although its efficacy is somewhat limited. We manually review 7 compilation errors not resolved in HumanEval after the first iteration, and find that 6 out of 7 instances are caused by the LLM mistakenly generating code segments enclosed in Markdown code block markers (```), resulting in compilation failures. Additionally, another instance of compilation error is identified in the following generated solution:

```
 def is_simple_power(x,
n):  return x > 0 and (x
== 1 or (n != 1 and x ==
n**int(round(math.log(x, n))))
```

This code snippet contains five left parentheses and four right parentheses in the `return` statement, causing the syntax error. Interestingly, despite the compiler has indicated this syntax error, it is not rectified over three iterations.

In MBPP, we manually observe the 7 errors not fixed after the last iteration, and identify that two test tasks are responsible for all 7 errors: one named "sum" contributed to six errors, and another called "month_season" resulted in one error. We detail these error cases as follows:

```
 def month_season(month, day):
seasons =  "spring":  [(3, 20),
(6, 20)], "summer":  [(6, 21),
(9, 22)], "autumn":  [(9, 23),
(12, 20)], "winter":  [(12, 21),
(3, 19)]  for season, (start,
end) in seasons.items():  if
(month == start[0] and day >=
start[1]) or (month == end[0]
and day <= end[1]):  return
season return "Invalid input"
```

The error arises because `'start'` and `'end'` are not included in `seasons.items`, which appears strange. We speculate that this issue stems from the presence of functions with the same name within the training set.

```
 def sum(a, b):
common_divisors = [i for i
in range(1, min(a, b) + 1)
if a % i == 0 and b % i == 0]
return sum(common_divisors) if
common_divisors else 0
```

The root cause of this error lies in the model's confusion when a user-defined function name coincides with that of a system library function `sum`, representing a category of potential issues.

### F.6 Efficiency of Static Analysis

Utilizing static analysis tools to inspect code typically results in increased latency. To investigate whether the latency impacts the usability of CoCoGen, we measure its latency at each test case. We have observed that in experiments, the speed of static analysis and structural query is relatively fast. It is because the semantic checker (i.e., `pylint`), only analyzes the current file and its dependent files. The average latency is recorded at 1.27 seconds, with a minimum of 0.359 seconds and a maximum of 6.984 seconds. We deem this latency to be acceptable.

Table 13: Tables pre-computed by CoCoGen for error correction

| Table Name | Description | Element Example |
|---|---|---|
| **M** | Stores the a module and its hierarchy in project. | tests.unit.async_.work.__init__ |
| **M_C** | Stores a module and a class inside the module | Module neo4j._codec.packstream.v1.__init__, Class PackableBuffer |
| **M_C_CF** | Stores a class, its parent module, and its member functions. | Module neo4j.time.__init__, Class Clock, Function local_offset |
| **M_C_V** | Stores a class variable, its parent class and module. | Module neo4j._sync.io.tmphhoug1of, Class Bolt, Variable is_reset |
| **M_GF** | Store a global function and its parent module. | Module neo4j.time._arithmetic, Function nano_add |
| **M_GV** | Stores a global variable and its parent module. | neo4j.__init__, Global Variable TRUST_SYSTEM_CA_SIGNED_CERTIFICATES |

Table 14: 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` |

# G Technical Details of Structural Query System

## G.1 The Design Principle

In CoCoGen, we employ the *Structural Query Language* (SQL) to perform structured queries on code repositories. The SQL syntax used here is CodeQL, a specialized version designed for software repository mining, capable of conducting complex control-flow and data-flow analyses on a set of code files.

We extracted 11 error codes from the compiler (shown in Table 4), each corresponding to one of four error categories as in Section 2.2. For efficiency, two authors manually inspected the most frequently occurring error codes, precomputed six data tables from the project graph, and hard-coded the structural context retrieval procedure for the following error codes: `E0001` (syntax error), `E0602` (undefined-variable), `E1101` (no-member), `E0213` (no-self-argument), and `E0102` (function redefined). The tables to retrieve structural project context for these frequently occurred errors are presented in Table 13.

Whenever the compiler reports an error, if it is a frequently occurring error, its retrieval is done from these precomputed data tables. Otherwise, the CoCoGen invokes the LLM to generate an SQL query statement, which is then executed on the project graph.

In the CoderEval-Python benchmark, with iteration rounds set to 3, related project contexts of 93.2% of errors are successfully retrieved. For the remaining 6.8% of cases, there are two reasons for failure: first, the LLM does not follow the demonstrated CodeQL examples and generates queries that are syntactically or semantically incorrect, thus rejected by the CodeQL query system; second, the query does not return anything because the specified conditions in the query are not satisfied in project context.

### G.2 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 14. The four queries shown in the table are written manually by one of the authors, and designed to handle four representative types of compilation errors. These sampled error messages are reported by compilers when generating solutions without project context. The specific four messages are chosen randomly from the compilation log.

## H 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. To encode passages to vectors, we utilize `text-embedding-ada-002`, a text embedding model provided by OpenAI and accessible via online APIs (OpenAI, 2022).

**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

$databaseEntries \leftarrow \emptyset$
**for** each $sourceFile$ in SOURCEFILESET **do**
  $nodesForVisit \leftarrow \langle\rangle$
  $propertyPrefixSeq \leftarrow \langle\rangle$
  $astFile \leftarrow$ PARSER$(sourceFile)$
  $nodesForVisit$.ADD$(astFile.rootNode)$
  **while** $nodesForVisit$ is not empty **do**
    $currentNode \leftarrow nodesForVisit$.POP()
    **if** $currentNode$ is PREFIXMARK **then**
      $propertyPrefixSeq$.POP()
    **end if**
    **if** $currentNode.type$ is in [VARIABLETYPE, FUNCTIONTYPE, CLASSTYPE] **then**
      $nodeProperty \leftarrow$ GETPROPERTIES$(currentNode)$
      $nodeSchema \leftarrow \langle propertyPrefixSeq, currentNode, nodeProperty\rangle$
      $nodeEmbedding \leftarrow$ 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**