Accepted: 13 November 2025
Revised: 28 October 2025
Received: 08 January 2025

RESEARCH-ARTICLE

CATCODER: Repository-Level Code Generation with Relevant Code and Type Context

ZHIYUAN PAN, Zhejiang University, Hangzhou, Zhejiang, China

XING HU, Zhejiang University, Hangzhou, Zhejiang, China

XIN XIA, Zhejiang University, Hangzhou, Zhejiang, China

XIAOHU YANG, Zhejiang University, Hangzhou, Zhejiang, China

CatCoder: Repository-Level Code Generation with Relevant Code and Type Context

ZHIYUAN PAN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XING HU*, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIN XIA, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIAOHU YANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

Large language models (LLMs) have demonstrated remarkable capabilities in code generation tasks. However, *repository-level code generation* presents unique challenges, particularly due to the need to utilize information spread across multiple files within a repository. Specifically, successful generation depends on a solid grasp of both general, context-agnostic knowledge and specific, context-dependent knowledge. While LLMs are widely used for the context-agnostic aspect, existing retrieval-based approaches sometimes fall short as they are limited in obtaining a broader and deeper repository context. In this paper, we present **CatCoder**, a novel code generation framework designed for statically typed programming languages. CatCoder enhances repository-level code generation by integrating relevant **c**ode **a**nd **t**ype context. Specifically, it leverages static analyzers to extract type dependencies and merges this information with retrieved code to create comprehensive prompts for LLMs. To evaluate the effectiveness of CatCoder, we adapt and construct benchmarks that include 199 Java tasks and 90 Rust tasks. The results show that CatCoder outperforms the RepoCoder baseline by up to 14.44% and 17.35%, in terms of compile@k and pass@k scores. In addition, the generalizability of CatCoder is assessed using various LLMs, including both code-specialized models and general-purpose models. Our findings indicate consistent performance improvements across all models, which underlines the practicality of CatCoder. Furthermore, we evaluate the time consumption of CatCoder in a large open source repository, and the results demonstrate the scalability of CatCoder.

CCS Concepts: • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Large Language Model, Code Generation, Repository Context

1 Introduction

Large language models (LLMs) [33, 36, 39, 47] have been widely applied in various software engineering-related fields, and they have shown remarkable performance in code generation tasks [14, 17]. Recently, repository-level code generation has attracted much attention [32, 40, 48, 53]. It refers to the generation of a specific function within a repository and is thus more practical than standalone code generation tasks (i.e., generating functions with no dependencies). This practical scenario introduces unique challenges, as the target code often depends on definitions and usage patterns distributed across multiple files, rather than being confined to a single file. When provided with only the *in-file* context for such a task, an LLM may hallucinate and produce code that appears

*Corresponding Author

Authors' Contact Information: Zhiyuan Pan, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, zy_pan@zju.edu.cn; Xing Hu, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, xinghu@zju.edu.cn; Xin Xia, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, xin.xia@acm.org; Xiaohu Yang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, yangxh@zju.edu.cn.

ACM 1557-7392/2025/12-ART
https://doi.org/10.1145/3779217

plausible but fails to compile or execute correctly within the project (e.g., misusing an API or referencing an undefined identifier). Therefore, a key problem in repository-level code generation is how to provide the LLM with the necessary *cross-file* context so that it can generate valid and consistent code.

Existing studies have tried to tackle this problem by utilizing repository information with retrieval-based approaches. Zan et al. [51] train a dual-encoder model to retrieve API information from library documentation. Lu et al. [34] build an external code database to retrieve similar code. Shrivastava et al. [40] propose RLPG, in which they train a classifier to predict prompt proposals taken from repository files. Zhang et al. [53] propose RepoCoder, which iteratively retrieves similar code in a repository. However, these approaches still have limitations. First, methods that depend on external resources or require training specialized retrieval models often lack generalizability across unseen repositories or programming languages. Second, the retrieved code may fail to capture all necessary field accesses and API calls required for correct generation. This issue is particularly pronounced in repositories with minimal code duplication [53], or when the desired functionality is novel within the codebase. These limitations underscore the need for more comprehensive modeling of repository context.

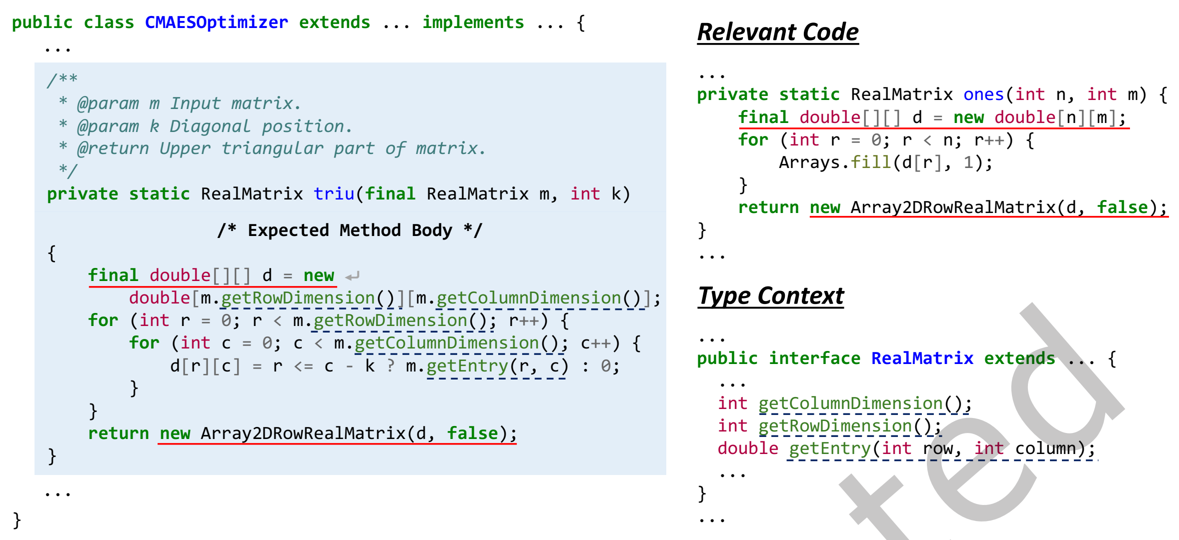
In this paper, we aim to address the aforementioned limitations by capturing a broader repository context for the LLM. We observe that, in addition to code retrieval, the types associated with a function can also serve as valuable references for LLMs. Specifically, these related types provide a rich set of fields and methods from which the LLM can select when generating code, thereby enhancing its understanding of repository-specific details. We refer to this explicit information of types as *type context*. The type context encapsulates localized knowledge embedded in nested scopes, serves as a complement to code retrieval, and helps LLMs develop a deeper understanding of the repository structure. Our work focuses on constructing type context for *statically typed* programming languages[1], where precise type information can be reliably extracted and leveraged. In contrast, dynamically typed languages (e.g., Python and JavaScript) determine variable types at runtime, making type information less accessible and more ambiguous. To extract type context, we utilize existing static analyzers.

Building on the concept of *type context*, we propose a new repository-level code generation framework named **CatCoder**. By leveraging the combination of relevant **c**ode **a**nd **t**ype context, CatCoder aims to improve the generation of code that is both logically sound and correctly integrated within its local environment. The framework operates in three stages. First, it retrieves relevant code from the repository. Next, it invokes a static analyzer to obtain dependent types for the target function and constructs a textual representation of the type context. Finally, it integrates both the retrieved code and the type context into a single prompt and generates the target function using an LLM. CatCoder runs on top of a frozen LLM and does not require external databases or additional model training.

To evaluate the effectiveness of CatCoder, we adapt and construct repository-level code generation benchmarks that include 199 Java tasks and 90 Rust tasks in total. We compare CatCoder with vanilla LLM, the In-File context (i.e., contents within the same file), and RepoCoder [53], a state-of-the-art framework for repository-level code generation, in terms of compile@k and pass@k [17] scores (i.e., compilation rate and test passing rate). To validate the generalizability of CatCoder, we evaluate CatCoder with various LLMs, including code-specialized models and general-purpose models. To assess the scalability of CatCoder, we conduct extensive evaluations on a large open-source repository, measuring the time consumption of sequential code generation tasks. The results on the benchmarks indicate that CatCoder substantially outperforms the vanilla LLM and the In-File context. In addition, it outperforms RepoCoder by up to 14.44% and 17.35%, in terms of compile@k and pass@k. The results of the generalizability study demonstrate that CatCoder improves the performance of all selected LLMs. Furthermore, the results of the scalability study underline the efficiency and practicality of CatCoder in large repositories.

Contributions. In summary, we make the following contributions:

[1]Statically typed programming languages require variable types to be declared at compile time, such as C, C++, Java, and Rust.



```
public class CMAESOptimizer extends ... implements ... {
  ...
  /**
   * @param m Input matrix.
   * @param k Diagonal position.
   * @return Upper triangular part of matrix.
   */
  private static RealMatrix triu(final RealMatrix m, int k)
                 /* Expected Method Body */
  {
      final double[][] d = new ↵
          double[m.getRowDimension()][m.getColumnDimension()];
      for (int r = 0; r < m.getRowDimension(); r++) {
          for (int c = 0; c < m.getColumnDimension(); c++) {
              d[r][c] = r <= c - k ? m.getEntry(r, c) : 0;
          }
      }
      return new Array2DRowRealMatrix(d, false);
  }
  ...
}
```

Relevant Code

```
...
private static RealMatrix ones(int n, int m) {
    final double[][] d = new double[n][m];
    for (int r = 0; r < n; r++) {
        Arrays.fill(d[r], 1);
    }
    return new Array2DRowRealMatrix(d, false);
}
...
```

Type Context

```
...
public interface RealMatrix extends ... {
  ...
  int getColumnDimension();
  int getRowDimension();
  double getEntry(int row, int column);
  ...
}
...
```

Fig. 1. The two essential sources of local context for generating the triu method. (1) **Relevant Code** (the ones method) provides a crucial, repository-specific pattern for instantiating a RealMatrix object. (2) **Type Context** provides the equally crucial API definition for the RealMatrix interface, including methods like getEntry needed for element access.

(1) We propose a novel framework for statically-typed programming languages, CatCoder, to generate repository-level code by highlighting relevant code and type context in code repositories.
(2) To the best of our knowledge, we are the first to construct a repository-level code generation benchmark for Rust, to evaluate the effectiveness of CatCoder on a minor programming language.
(3) We evaluate CatCoder using Java and Rust benchmarks, and results indicate that CatCoder outperforms the baselines.
(4) We release the replication package[2] of CatCoder, including the source code of CatCoder and our benchmarks, to facilitate future research.

Paper Organization. The remainder of the paper is structured as follows. Section 2 introduces the motivation of CatCoder using an example. Section 3 presents the details of CatCoder. Section 4 describes the experimental setups, including baselines, evaluation metrics, benchmarks, and implementation details. Section 5 analyzes the experimental results and provides answers to the research questions. Section 6 discusses the failure cases of CatCoder and threats to validity. Section 7 summarizes the related work. Section 8 concludes the paper with possible future work.

2 Motivating Example

In this section, we present a realistic development scenario to illustrate the core idea behind CatCoder. The motivating example will demonstrate that synthesizing information from both relevant code and type context is crucial for successful generation.

Consider the task of implementing the helper method `triu` within the `CMAESOptimizer` class from the widely used *Apache Commons Math* library, as shown in Fig. 1. As described in its docstring, the `triu` method is intended to extract the upper triangular part of a given matrix. Its signature is

[2]https://github.com/pan2013e/catcoder

`private static RealMatrix triu(final RealMatrix m, int k)`, where m is the input matrix and k specifies the diagonal offset.

Based on its context-agnostic knowledge of coding, an LLM can infer the general algorithmic principle: the implementation should return a new matrix with the same dimensions, where an element at position (r, c) is copied from m if the column index c is greater than or equal to $r + k$; otherwise, the element is set to zero. However, the requirement of context-dependent knowledge poses challenges to the LLM. The `RealMatrix` type is an interface in *Apache Commons Math* and cannot be treated as a nested `double` array. Proper interaction with a `RealMatrix` instance requires familiarity with repository-specific API conventions, such as using `getEntry(row, column)` to access elements. Furthermore, creating a new `RealMatrix` instance requires using a specific concrete implementation, `Array2DRowRealMatrix`, and understanding the correct constructor usage. An LLM must therefore grasp these repository-specific details to produce valid code.

Table 1. Different information sources for the `triu` task

Information Source	**Key Insight Provided**	**Sufficient Alone?**
Info.1 `ones` method	Idiomatic instantiation: `new Array2DRowRealMatrix(...)`	✘
Info.2 `RealMatrix` API	Correct API access: `m.getRow/ColumnDimension(...)` `m.getEntry(...)`	✘
CatCoder (Info.1 + Info.2)	Complete, correct implementation	✔

To generate a correct implementation, an LLM must solve two distinct subproblems, that is, (1) how to create the new `RealMatrix` to be returned, and (2) how to read data from the input `RealMatrix m`:

(1) As shown in Fig. 1, the `ones` method is retrieved due to its shared return type and location. From this snippet, an LLM can learn a non-trivial pattern for object creation: instantiate a new `Array2DRowRealMatrix` by first creating a primitive `double` array and passing it to the constructor. This form of relevant code retrieval reflects a common practice among developers, who often refer to similar methods when writing repository-level code.

(2) In addition, Fig. 1 illustrates the available APIs of the `RealMatrix` type, presented as the type context. Because both the target method `triu` and the surrounding class `CMAESOptimizer` reference `RealMatrix`, its API information provides essential insights. For instance, an LLM can learn how to access dimensions and elements correctly by referencing the provided method signatures. This mirrors another common development strategy: navigating related types (e.g., parameter or return types) to discover usable fields and methods.

Table 1 categorizes the two aforementioned information sources and their key insights. We abstract them as follows:

Info.1 Relevant code snippets that share similarities in method name, parameters, return types, or docstrings. These examples often perform semantically related or symmetric operations and thus provide reusable implementation patterns.

Info.2 Type context, defined as the available fields and methods of related types, offering clues for accessing or manipulating object instances correctly.

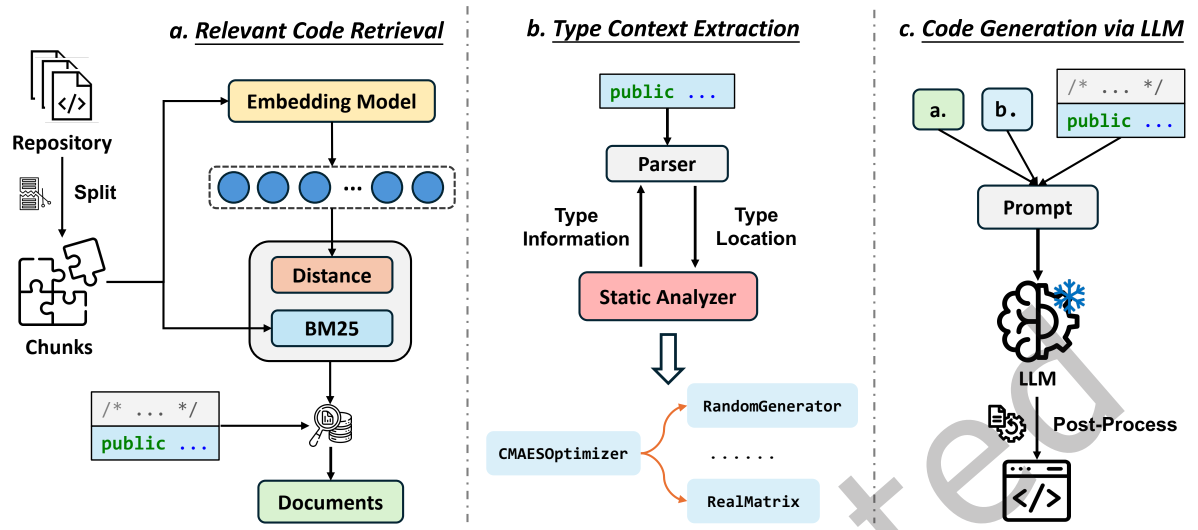


Fig. 2. Overview of our approach

This motivating example highlights that both **Info.1** and **Info.2** are indispensable. The absence of either is likely to lead to incorrect or incomplete code generation. While prior work [34, 53] has leveraged retrieval-augmented generation to utilize relevant code (Info.1), they typically overlook Info.2 (type context). As Fig. 1 shows, retrieved code alone may not fully capture the required API usage. Moreover, in repositories that lack code duplication, relying solely on retrieval is less effective. In such cases, the absence of type context increases the risk of hallucination, such as referencing non-existent methods or fields. To address this gap, CatCoder integrates both Info.1 and Info.2. Our framework is carried out in the following steps:

Step a. Relevant Code Retrieval. We retrieve similar code in a repository based on the query (i.e., the docstrings and signature of the function to be completed).

Step b. Type Context Extraction. We utilize static analyzers to determine the dependent types and then create type context by extracting the fields and method signatures.

Step c. Code Generation via LLM. We merge the results in Step a and Step b into a comprehensive prompt, and invoke an LLM to obtain the generated code.

3 Proposed Approach

The framework of CatCoder is presented in Fig. 2. CatCoder takes a source code repository and an empty function (along with its docstrings, if it exists). The output of CatCoder is the completed code of the function.

In addition to Fig. 2, Algorithm 1 presents the pseudocode of the workflow of CatCoder. CatCoder contains two core components: (1) *Relevant Code Retrieval* and (2) *Type Context Extraction*. By utilizing these components, CatCoder is able to gather two kinds of referential information for the LLM. After post-processing the raw output of the LLM, CatCoder returns the generated code.

Algorithm 1: CatCoder

```
Input: Code repository $R$, Empty function $f$
Output: The completed code of function $f$
1 $docs$ = code_retrieval($R$, $f$)                    /* Section 3.1 */
2 $contexts$ = type_context($R$, $f$)                  /* Section 3.2 */
3 $prompt$ = prepare($f$, $docs$, $contexts$)          /* Section 3.3 */
4 $code$ = LLM($prompt$)
5 return postprocess($code$)
```

3.1 Relevant Code Retrieval

3.1.1 Code Splitting. We traverse all the source code files in the repository and split them into smaller chunks. Starting with complete files, we set a maximum chunk size and recursively split larger fragments into smaller ones that fit in the chunk.

As the structure of the source code is often complex, we cannot split it at an arbitrary position. For example, we should not break in the middle of a statement line or some important code elements (e.g., function definitions). Otherwise, the resulting contents would be less meaningful and difficult to understand. Therefore, we split the code only before the following positions (sorted by priority in descending order):

(1) Type definitions (e.g., `public class Main` in Java)
(2) Function definitions (e.g., `public static void main(String[] args)` in Java)
(3) Control flow statements (e.g, *if*, *for* and *while* statements in Java)
(4) New lines (i.e., `\n` and `\r\n`)

This rule follows the hierarchy of many programming languages and helps protect code structures during splitting. After splitting, we obtain a corpus of code chunks[3] to perform retrieval.

3.1.2 Hybrid Retrieving. In the second step, we build retrievers to search for relevant code. To achieve better search performance, we follow previous research [35] and ensemble two kinds of retrievers: a *sparse* one and a *dense* one. The sparse retriever corresponds to keyword matching, and the dense retriever corresponds to semantic search. Each retriever searches and returns top-k[4] relevant documents (i.e., split code chunks), after a query is constructed by concatenating docstrings and the function signature.

❶ *Sparse Retrieval.* We choose the bag-of-words model for sparse retrieval, and use the BM25 function [38, 43] to rank the set of documents, based on the query terms. Given a document D and a query Q with terms $t_1, t_2, \cdots, t_n$, the BM25 score can be defined as follows:

$$\text{BM25}(D,Q) = \sum_{t \in Q} \text{IDF}(t,D) \cdot \frac{\text{TF}(t,D) \cdot (k_1 + 1)}{\text{TF}(t,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{|D|_{\text{avg}}}\right)}$$

where $\text{IDF}(t,D)$ is the inverse document frequency weight, $\text{TF}(t,D)$ is the term frequency (i.e., the number of appearing times in the document), $|D|$ is the length of the document, and k_1 and b are hyperparameters. By default, the values of k_1 and b are set to 1.5 and 0.75. $\text{IDF}(t,D)$ can be calculated by:

[3]We call them "documents" later in this subsection.
[4]k is a hyperparameter, and we describe the settings in Section 4.4.4.

$$\mathrm{IDF}(t, D) = \log \frac{N_D - \mathrm{DF}(t, D) + 0.5}{\mathrm{DF}(t, D) + 0.5},$$

where N_D is the number of documents and $\mathrm{DF}(t, D)$ is the document frequency (i.e., the number of documents that contain the term).

In practice, after splitting the code files from a repository, we apply the aforementioned BM25 function for a query and obtain a list of scores. After that, we select documents with the top-k highest scores.

❷ *Dense Retrieval.* We employ an embedding model for dense retrieval. The model maps natural language sentences and code snippets into dense vector spaces. To select the most relevant documents, our intuitive idea is to find the nearest neighbors around the embedding vector of the query text. Thus, we use *Squared Euclidean Distance* to measure the proximity between embedding vectors in high-dimensional spaces, and this metric can be calculated by:

$$d^2(\mathbf{p}, \mathbf{q}) = \|\mathbf{p} - \mathbf{q}\|^2 = \sum_{i=1}^{n} (p_i - q_i)^2,$$

where n is the number of dimensions in the embedding vectors. A lower value of $d^2(\mathbf{p}, \mathbf{q})$ indicates that $\mathbf{p}$ and $\mathbf{q}$ are more similar.

In practice, we utilize the embedding model and generate vector representations of all documents and query text. Then we calculate $d^2(\mathbf{p}, \mathbf{q})$ between the query embedding and all document embeddings and select the top-k nearest documents.

3.1.3 Result Ranking. In the third step, we ensemble the two retrievers and re-rank their outputs using the *Reciprocal Rank Fusion* (RRF) [18] algorithm. We assign weights to the retrievers and calculate the weighted RRF score[5] for each retrieved document D:

$$\mathrm{RRF}(D) = \sum_{i=1}^{n} \mathbf{1}(D) \cdot \frac{w_i}{r(D) + 60},$$

where w_i is the weight assigned to a certain retriever, $\mathbf{1}(D)$ is an indicator function[6], and $r(D)$ is the original rank (if exists) of D in the outputs of this retriever.

After applying the RRF algorithm, we deduplicate the documents and sort them in descending order based on the RRF scores. Now we obtain the results of the *Relevant Code Retrieval* component.

3.2 Type Context Extraction

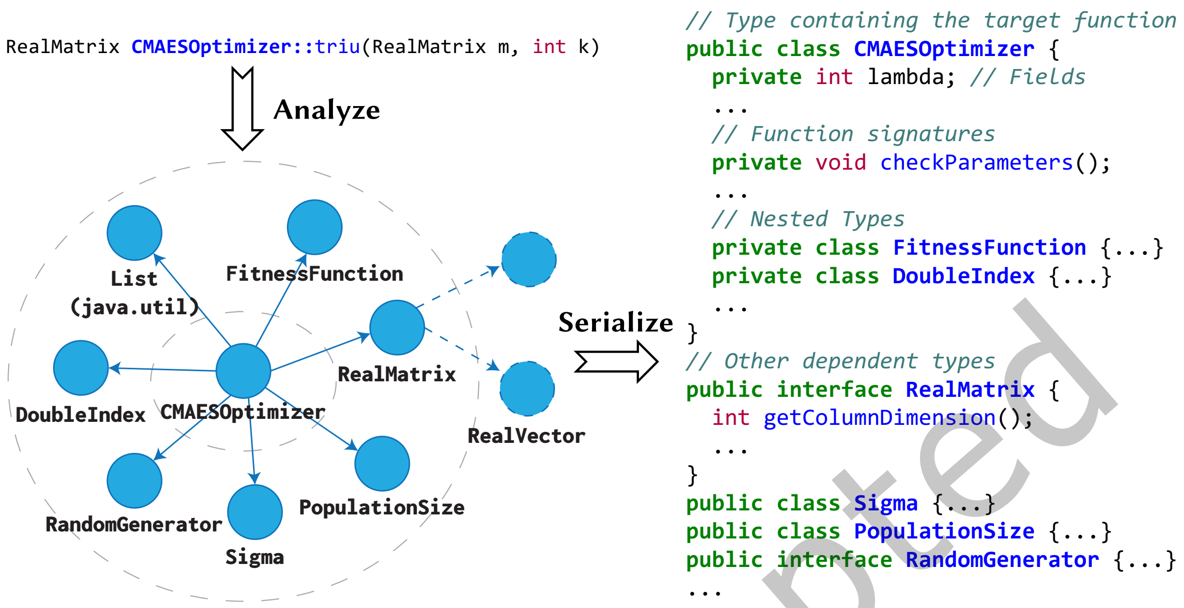
Here, we present the type context and the way to extract type contexts from code repositories using static analyzers[7].

3.2.1 Type Context. To capture the contextual relationships between types within a code repository, Fig. 3 presents a directed dependency graph $G = (V, E)$, in which each vertex $v \in V$ corresponds to a type defined in the repository (e.g., a class or interface). A directed edge $(A, B) \in E$ is created from type A to type B, if A references or depends on B (i.e., B occurs in the body of A). The dependency graph is able to identify clusters of contextually related types and provide types that an LLM is likely to use and refer to when generating function code.

[5]We assume that a retriever does not output duplicate documents. However, duplicate documents across retrievers are acceptable.

[6]The value is 1 if D is one of the outputs of this retriever, otherwise the value is 0.

[7]The word "type" is an abstract concept and might be different in various languages. For example, it refers to *class*, *interface*, and *enum* in Java; whereas in Rust, it refers to *struct* and *enum*.



```
// Type containing the target function
public class CMAESOptimizer {
  private int lambda; // Fields
  ...
  // Function signatures
  private void checkParameters();
  ...
  // Nested Types
  private class FitnessFunction {...}
  private class DoubleIndex {...}
  ...
}
// Other dependent types
public interface RealMatrix {
  int getColumnDimension();
  ...
}
public class Sigma {...}
public class PopulationSize {...}
public interface RandomGenerator {...}
...
```

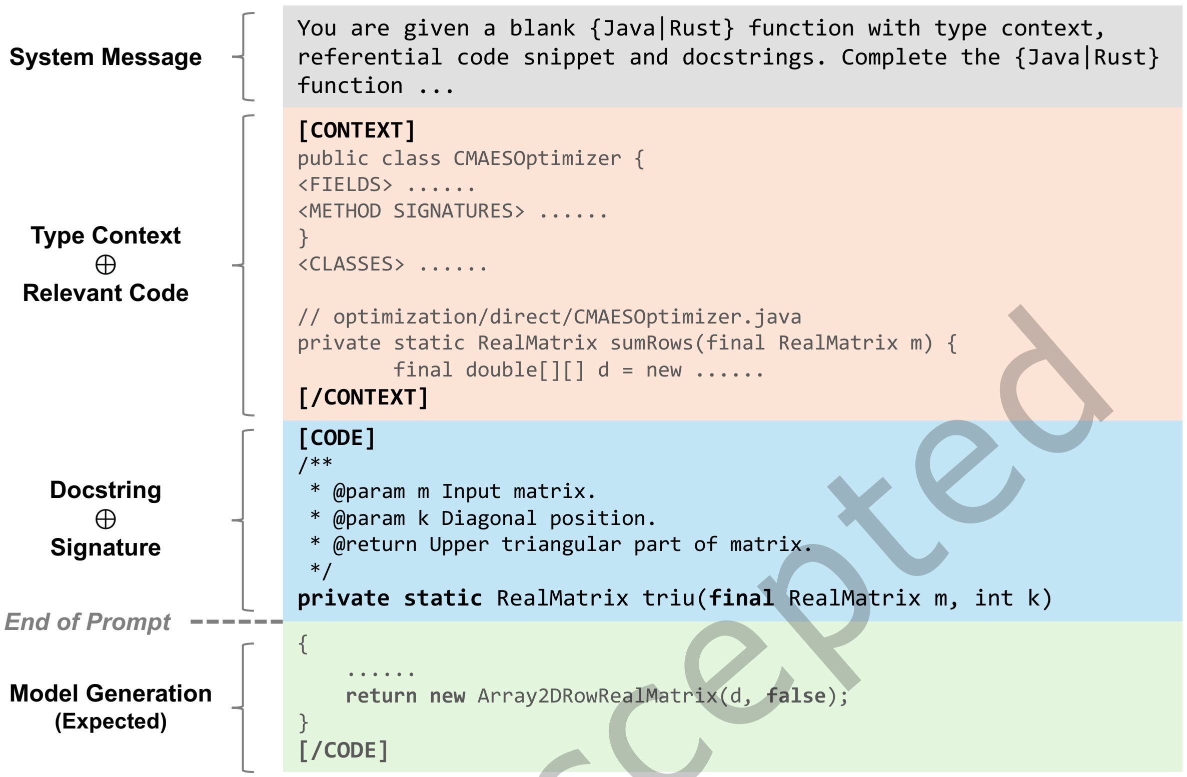
Fig. 3. Dependency Graph and Type context of the `triu` method in class `CMAESOptimizer`.

In Fig. 3, we build the dependency graph for the `triu` method in class `CMAESOptimizer`, starting from its own class. As the graph spans, the vertex set and the edge set will eventually converge, and we can obtain the types that are relevant to `CMAESOptimizer`. To pass this structural information to the LLM, we create **type context** from the graph by extracting the fields and function signatures within the types and concatenating them into a string. Specifically, for the string representation, we follow the format proposed by Tufano et al. [44], which has been applied in the field of unit test case generation. The right part of Fig. 3 shows the resulting textual type context for the target method (simplified for illustration).

In large repositories, the full dependency graph can become vast, making the resulting type context too long to fit in an LLM's context window. However, the relevance between types tends to decrease with the distance between them in the graph. For example, in Fig. 3, programmers working on class `CMAESOptimizer` are more likely to directly use the API of `RealMatrix`, a direct neighbor, than the API of `RealVector`, which is an indirect neighbor at a distance of 2. Therefore, in practice, we constrain the scope of type context creation to the direct neighbors of the initial types in the graph. We also prune types from standard libraries, as we assume LLMs possess general knowledge of these libraries from their pre-training.

3.2.2 Building Type Context via Static Analyzer. To get detailed information (i.e., fields and methods) of a type from its name, some straightforward heuristics might be used, e.g., locating imports or looking for similar file names. However, these heuristics are not sound and do not work with all statically typed programming languages. Thus, we need to perform an analysis to accurately navigate to a type and find its definitions and implementations. Many language servers serve as static analyzers and provide this type-based analysis, such as clangd [2] for C++, Eclipse JDT [5] for Java, and rust-analyzer [10] for Rust.

Specifically, to build the type context for a function code generation task, we first determine the set of relevant types (as Section 3.2.1 describes). We start with the signature and form a set of types, and the initial element is



```
You are given a blank {Java|Rust} function with type context,
referential code snippet and docstrings. Complete the {Java|Rust}
function ...
[CONTEXT]
public class CMAESOptimizer {
<FIELDS> ......
<METHOD SIGNATURES> ......
}
<CLASSES> ......

// optimization/direct/CMAESOptimizer.java
private static RealMatrix sumRows(final RealMatrix m) {
        final double[][] d = new ......
[/CONTEXT]
[CODE]
/**
 * @param m Input matrix.
 * @param k Diagonal position.
 * @return Upper triangular part of matrix.
 */
private static RealMatrix triu(final RealMatrix m, int k)
{
    ......
    return new Array2DRowRealMatrix(d, false);
}
[/CODE]
```

Fig. 4. Our prompt design. The actual prompt ends right after the signature.

the type to which the current function belongs. Then we implement a parser that works collaboratively with the static analyzer. The static analyzer provides detailed type information to the parser. The parser then searches for a new occurring type, adds it to the set, and queries the analyzer with its location (i.e., byte offset or line and column offsets in a file). This interactive process continues until all direct neighbors of the initial set of types have been discovered. Finally, as described in Section 3.2.1, we concatenate the fields and method signatures within the set of types and obtain the type context.

3.3 Code Generation via LLM

Finally, we describe how to utilize the previous results and invoke the LLM to generate code. Fig. 4 illustrates our prompt design, in which there are two fields: `CONTEXT` and `CODE`. To construct a prompt, we follow the context format in previous work [32, 40, 53], and fill the `CONTEXT` field by concatenating the type context and the retrieved code chunks. Then we place the docstrings and the signature of the method at the beginning of the `CODE` field. The prompt serves as input to a frozen LLM[8], and we expect the model to complete the `CODE` field by generating the method body (i.e., the **green** part in Fig. 4). In addition, to post-process the raw output of the LLM, we employ several procedures[9], including truncating texts, prepending function signatures, and removing Markdown syntax.

[8]We discuss the details of selected models and model access in Section 4.4.
[9]The post-processing procedures may vary with different LLMs and programming languages.

4 Experimental Setup

In this section, we first formulate four research questions (RQs). Based on the RQs, we describe the details of the baselines, evaluation metrics, benchmarks, and implementation of our approach.

We aim to answer the following research questions:

RQ1 How effective is our approach at generating function code in a repository, compared with the baselines? And does our approach generalize to different LLMs?

RQ2 How do the two components in our approach (i.e., relevant code retrieval and type context extraction) contribute to the overall effectiveness, respectively?

RQ3 Is our approach generalizable to various large language models? And how do different models perform on the benchmark?

RQ4 Is our approach efficient when generating a large number of functions in large repositories?

In brief, we evaluate the **effectiveness** of our approach in RQ1, and then conduct an **ablation study** in RQ2. In RQ3, we study the **generalizability** of CatCoder and the performance of different models. Finally, in RQ4, we evaluate the **scalability** of our approach. For implementation and evaluation of effectiveness, we select two statically typed programming languages, Java and Rust, since Java is one of the most widely used programming languages [12], and Rust is a new system-level language that attracts the attention of many developers [11].

4.1 Baselines

As described in Section 3.3, Fig. 4 shows the prompt template used in our evaluation. As the `CONTEXT` field is the only variable, we compare our approach with methods that fill different contexts in the prompt. For the repository-level code generation task, there are three types of context: (1) *No context*, (2) *Local context*, and (3) *Cross-file context*. Therefore, we design the following baselines for comparison in RQ1:

(1) **Vanilla**: In this method, we prompt LLMs without any repository context. In this case, all natural language descriptions are given, but the `CONTEXT` field in Fig. 4 is left empty.
(2) **In-File**: In this method, we consider code generation as a left-to-right file completion task. We extract code before the function to be generated and put it in the `CONTEXT` field.
(3) **RepoCoder**: RepoCoder [53] is an approach that augments the prompt by iteratively retrieving referential code fragments, and it is originally proposed for Python code repositories. We adopt their idea and implement this baseline for Java and Rust based on our retrieval pipeline. Specifically, the initial iteration uses natural language descriptions (i.e., docstrings) as the query for code retrieval. From the second iteration, it uses the previously generated code for retrieval.

4.2 Evaluation Metrics

For the function code generation task, we argue that metrics based on text similarity are not accurate, and it is essential to use metrics based on compilation and execution results. Therefore, we use compilation rate and test passing rate, which evaluate syntax correctness and functional correctness, respectively. Specifically, to calculate such a rate, we follow Chen et al. [17] and use the unbiased *score@k*. It is defined as

$$\text{score@}k = \mathop{\mathbb{E}}_{\text{Tasks}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right],$$

where n is the number of code samples generated per task and k is the parameter in "score@k" ($n \geq k$). Based on this general formula, we define two detailed metrics that correspond to compilation rate and test passing rate:

- **compile@*k***. In the formula, c refers to the number of syntactically correct code samples. A code sample is syntactically correct if and only if it passes compiler checks and compiles.

- **pass@k**. In the formula, c refers to the number of functionally correct code samples. A code sample is functionally correct if and only if it passes the tests in the benchmark.

We calculate score@k values for each task in the benchmark, and the final score@k for a benchmark is the mean value of all score@k values.

4.3 Benchmarks

4.3.1 Java Benchmark. There are several publicly available repository-level benchmarks for Java [13, 19, 50, 52]. However, it is challenging to use many of them to evaluate CatCoder because they either lack the raw repository code or are not designed for function code generation tasks. Considering the challenges, the CoderUJB [52] benchmark is a suitable base benchmark that contains 238 code generation tasks, and we successfully adapt it for our evaluation. In the adaptation process, we filter out a few functions that satisfy one of the following patterns:

(1) The length of function code exceeds the max output limit (i.e., 512 tokens in our settings).
(2) The function does not have any modifiers or annotations. This helps us to perform post-processing more easily.
(3) The function is standalone, which means that it can be moved to a new class[10] and still compiles. The removal of standalone functions makes the benchmark more complex and difficult, and helps better evaluate the performance of repository-level code generation.

We apply the filtering rules sequentially: 17 functions are filtered out by the first rule, 5 by the second, and 17 by the third. As a result, after removing these 39 functions, we obtain 199 coding tasks. As the methods in the benchmark are mined by the CoderUJB authors from the Defects4J [27] dataset, it is convenient to evaluate the correctness of the generated code using the existing extensive unit test methods. Specifically, we "patch" the repository with generated code and run the `defects4j test -t ...` command.

4.3.2 Rust Benchmark. To our knowledge, the only code generation benchmark for the Rust programming language is MultiPL-E [16], which translates the original HumanEval [17] and MBPP [14] benchmarks. However, this benchmark is designed for standalone function code generation tasks and thus is not suitable for our evaluation. In this case, we construct a new benchmark named *RustEval*.

Table 2. The crates used in our Rust benchmark

Crate	# Downloads	Crate	# Downloads
bit-vec	54M	sdl2	1.7M
curv-kzen	161K	splay_tree	72K
dlv-list	14M	sshkeys	189K
gptman	514K	unix_path	86K
mathru	202K	vec_map	81M
multiset	99K	without-alloc	259K
rocket_http	4.2M		

As shown in Table 2, we collect 13 repositories of various categories from *crates.io* [4], the official Rust package registry. In these repositories, we obtain 90 functions with docstrings that satisfy the official guidelines [6] (i.e., have detailed descriptions and at least one documentation test). To test the correctness of the generated code, we utilize the documentation tests in the docstrings by running the `cargo test --doc ...` command.

Table 3. Statistics of our benchmarks, including number of tasks, average number of lines of code (NLOC) per task, and average cyclomatic complexity number (CCN) per task.

	Java	**Rust**
# Tasks	199	90
Avg. NLOC	16.59	10.68
Avg. CCN	4.59	2.64

4.3.3 Benchmark Statistics. Table 3 presents the statistics of our benchmarks, where we assess their complexity in terms of the average number of lines of code (NLOC) and cyclomatic complexity number (CCN) per task. Compared to the code completion task in our baseline paper, which requires generating only a single line of code, our function code generation task involves producing more lines and deeper code structures, as reflected in the NLOC and CCN values shown in Table 3. In addition, compared to the Java benchmark, the Rust benchmark is less syntactically verbose and structurally simpler. We attribute this to idiomatic Rust's reduced boilerplate and more concise abstractions. However, from a semantic standpoint, this does not imply that code generation is easier, as the underlying logic can still be dense and complex.

4.4 Implementation

4.4.1 Approach. We implement CatCoder for Java and Rust, and the source code is written in Python and Rust. Specifically in *Relevant Code Retrieval*, we use MPNet V2 [1] as the embedding model. It is an open-source model fine-tuned from the base MPNet [41] model using massive training data, including the popular CodeSearchNet [25]. For evaluation on the benchmarks, the ground truth code in the repositories is removed during retrieval to prevent leakage. In *Type Context Extraction*, we use Eclipse JDT.LS [5] and rust-analyzer [10] as the static analyzers. For Java, our approach interacts with Eclipse JDT.LS through the Language Server Protocol [9]. For Rust, we use rust-analyzer as a third-party library and implement a binding between Rust and Python.

Table 4. Models used for evaluation in RQ3. The model acronyms correspond to the x-axis labels in Figure 6.

	Model ID	**Acronym**	**Open-Source?**
	Code Specialized		
[39]	*CodeLlama-7b-Instruct*	CL-7B	✓
[39]	*CodeLlama-13b-Instruct*	CL-13B	✓
[23]	*deepseek-coder-6.7b-instruct*	DS-6.7B	✓
[3]	*codegemma-7b-it*	CG-7B	✓
[15]	*CodeQwen1.5-7B-Chat*	CQ-7B	✓
	General Purpose		
[8]	*Meta-Llama-3-8B-Instruct*	ML-8B	✓

4.4.2 Model Selection. For RQ1 and RQ2, we perform the evaluation using *CodeLlama-13B-Instruct* [39] as the default LLM, due to its state-of-the-art performance among models of medium size. For RQ3, to evaluate the generalizability of CatCoder, we select a variety of state-of-the-art LLMs with different sizes, including

[10]We assume that all Java standard libraries are imported in advance when applying this rule.

Table 5. Evaluation results of different methods on two benchmarks, using *CodeLlama-13B-Instruct*

Method	**Java**						**Rust**					
	compile@k (%)			*pass@k* (%)			*compile@k* (%)			*pass@k* (%)		
	1	3	5	1	3	5	1	3	5	1	3	5
Vanilla	33.6	49.4	56.2	14.9	22.1	25.2	18.0	28.3	33.1	10.8	15.8	18.2
In-File	59.7	72.7	76.8	35.3	43.8	47.6	52.1	66.5	70.9	41.6	55.7	61.6
RepoCoder	63.0	74.1	78.2	41.0	46.9	49.0	63.7	73.4	76.4	49.4	59.5	63.0
CatCoder	**71.9**	**84.8**	**88.4**	**44.7**	**54.0**	**57.5**	**65.1**	**75.5**	**78.9**	**52.7**	**62.0**	**65.9**

code-specialized models and general-purpose models. The details of the models are listed in Table 4, in which the model IDs are taken from the official releases at HuggingFace[7] or the API documentations.

4.4.3 Environment and Model Access. The experiments are performed on a Linux server with capable NVIDIA GPUs. For open-source LLMs, we deploy a local API server based on vLLM [28], a unified inference and serving engine for LLMs. The weights of open-source models are available at HuggingFace [7]. To facilitate future research, we list URLs of the models in our replication package.

4.4.4 Configurations. In *Relevant Code Retrieval*, we set the maximum chunk size to 2,000 and k (i.e., the number of documents that a retriever should return) to 4 for Java repositories. We set the chunk size to 1,000 and k to 8 for Rust repositories. This is due to the observation that functions and docstrings in our Java repositories are generally longer. In addition, when performing re-ranking, we set the weights of the sparse and dense retrievers to 30% and 70%, respectively.

In evaluations, we set n in compile@k and pass@k to 10 and k to 1, 3, and 5. To control the randomness of the generated code [17], we set the temperature to 0.6 and the nucleus sampling [24] parameter (*top_p*) to 0.7. In addition, we set the maximum number of generated tokens to 512.

5 Experimental Results

In this section, we analyze the experiment results in detail and answer the research questions.

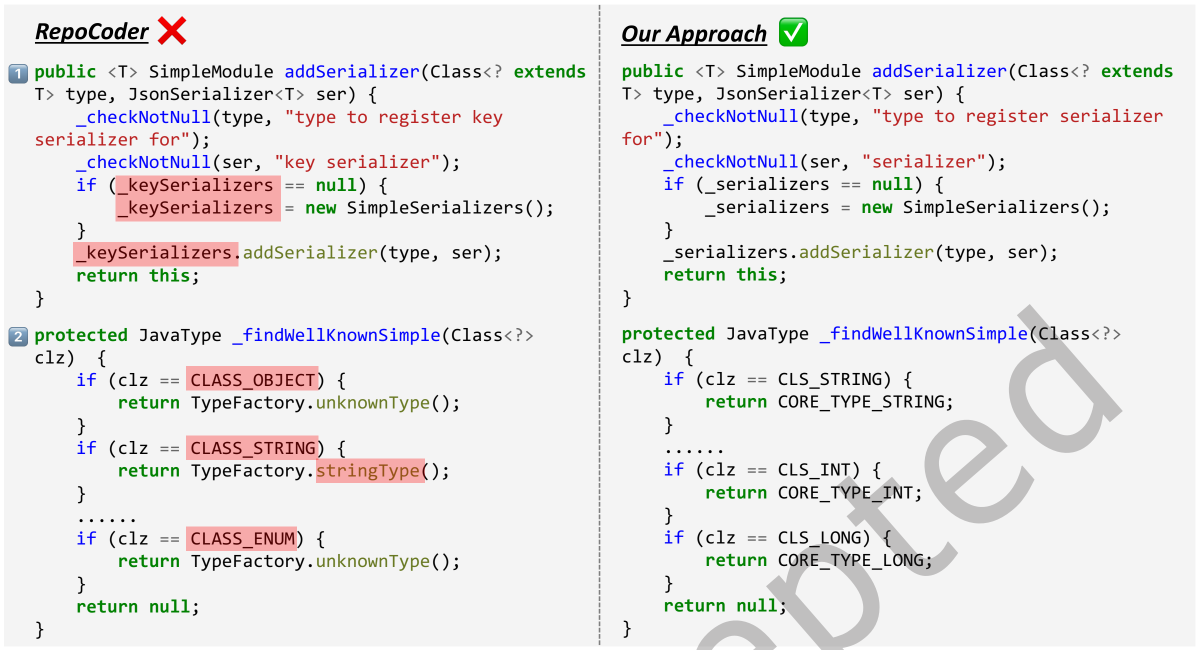
5.1 RQ1: Effectiveness

5.1.1 Results. The overall result is presented in Table 5. According to Table 5, we observe that the Vanilla baseline performs poorly as expected because the lack of repository context is likely to cause hallucinations. Next, we can conclude that the In-File baseline performs worse than RepoCoder and CatCoder, as in-file context is not enough to provide key information such as cross-file type structures and API usage. In addition, CatCoder outperforms RepoCoder. In terms of the Java benchmark, the compile@k and pass@k scores increase by up to 14.44% and 17.35%, respectively. For the Rust benchmark, the compile@k and pass@k scores increase by up to 3.27% and 6.68%.

5.1.2 Qualitative Analysis. To show the advantages of CatCoder over RepoCoder, we also perform a qualitative analysis of the results. Fig. 5 presents two cases, in which RepoCoder fails to generate the correct function code while CatCoder succeeds.

❶ **Case 1** (method `addSerializer`): The code generated by RepoCoder compiles but fails to pass unit tests because it uses the wrong field `_keySerializers` instead of the correct one (i.e., `_serializers`).

Upon checking the context provided to the LLM, we find that RepoCoder retrieves a similar method named `addKeySerializers`, which uses the field `_keySerializers` in its body. The LLM simply copies the code from `addKeySerializer` and causes the code to fail in unit tests. For CatCoder, it also retrieves the method



```
1 public <T> SimpleModule addSerializer(Class<? extends
T> type, JsonSerializer<T> ser) {
      _checkNotNull(type, "type to register key
serializer for");
      _checkNotNull(ser, "key serializer");
      if (_keySerializers == null) {
          _keySerializers = new SimpleSerializers();
      }
      _keySerializers.addSerializer(type, ser);
      return this;
  }
2 protected JavaType _findWellKnownSimple(Class<?>
clz)  {
      if (clz == CLASS_OBJECT) {
          return TypeFactory.unknownType();
      }
      if (clz == CLASS_STRING) {
          return TypeFactory.stringType();
      }
      ......
      if (clz == CLASS_ENUM) {
          return TypeFactory.unknownType();
      }
      return null;
  }
```

```
public <T> SimpleModule addSerializer(Class<? extends
T> type, JsonSerializer<T> ser) {
    _checkNotNull(type, "type to register serializer
for");
    _checkNotNull(ser, "serializer");
    if (_serializers == null) {
        _serializers = new SimpleSerializers();
    }
    _serializers.addSerializer(type, ser);
    return this;
}
protected JavaType _findWellKnownSimple(Class<?>
clz)  {
    if (clz == CLS_STRING) {
        return CORE_TYPE_STRING;
    }
    ......
    if (clz == CLS_INT) {
        return CORE_TYPE_INT;
    }
    if (clz == CLS_LONG) {
        return CORE_TYPE_LONG;
    }
    return null;
}
```

Fig. 5. Two tasks in the Java benchmark, where RepoCoder fails but CatCoder succeeds.

`addKeySerializer`, and the pattern of its generated code is the same as the code generated by RepoCoder. However, with the type context extracted by CatCoder, the LLM knows the existence of pairs (field `_serializers`, method `addSerializer`) and (field `_keySerializers`, method `addKeySerializers`). Therefore, we believe that this makes the LLM filter out some misleading reference code and choose the right field for usage.

❷ **Case 2** (method `_findWellKnownSimple`): The code generated by RepoCoder causes many compilation errors, as it contains non-existent fields and methods in the current scope.

In detail, we find that the non-existent `CLASS_XXX` fields come from another file, and RepoCoder retrieves them as context. Without knowing the type context of `JavaType` and current class (i.e., `TypeFactory`), the LLM accepts misleading references and generates a non-existent API call `stringType()`.

5.1.3 Comparison of Java and Rust. As shown in Table 5, the magnitude of improvement provided by CatCoder over the RepoCoder baseline differs between the two languages. On the Java benchmark, CatCoder achieves a relative improvement of up to 17.35% in pass@k, whereas on Rust, the maximum relative improvement is 6.68%. This suggests that CatCoder offers a more substantial advantage in the context of the evaluated Java projects. We attribute this difference to the characteristics of the languages and the structural properties of the benchmark projects. As indicated in Table 3, the Java benchmark is, on average, more structurally complex and verbose than the Rust benchmark. We argue that this increased complexity creates scenarios in which the type context provided by CatCoder becomes a more critical factor for successful code generation, as illustrated in Section 5.1.2.

5.1.4 Semantic Similarity vs. Test Pass Rate. In addition to compile@k and pass@k, we conduct an evaluation using CodeBLEU [37], a metric that lies between purely text similarity-based and execution-based metrics. Beyond traditional n-gram matching, CodeBLEU incorporates syntactic and semantic matching, thus better capturing the

Table 6. Results of CodeBLEU, using *CodeLlama-13B-Instruct*. **C**: CodeBLEU score; **N**: n-gram match score; **W**: weighted n-gram match score; **S**: syntax match score; **D**: dataflow match score.

Method	**Java**					**Rust**			
	C	*N*	*W*	*S*	*D*	*C*	*N*	*W*	*S*
Vanilla	0.36	0.23	0.30	0.41	0.37	0.25	0.16	0.21	0.39
In-File	0.51	0.40	0.46	0.58	0.55	0.38	0.29	0.33	0.54
RepoCoder	0.57	0.47	0.53	0.64	0.61	0.46	0.37	0.42	0.60
CatCoder	0.56	0.46	0.52	0.63	0.59	0.44	0.35	0.40	0.60

Table 7. Evaluation results of different variants on two benchmarks, using *CodeLlama-13B-Instruct*

Variant	**Java**						**Rust**					
	compile@k (%)			*pass@k* (%)			*compile@k* (%)			*pass@k* (%)		
	1	3	5	1	3	5	1	3	5	1	3	5
FULL	71.9	84.8	88.3	44.7	54.0	57.5	65.1	75.5	78.9	52.7	62.0	65.9
-CR	57.9	75.4	81.4	27.5	38.8	43.7	35.0	50.7	56.3	23.3	35.4	41.0
-TC	67.1	80.3	84.8	39.5	49.5	53.4	59.6	70.1	73.8	49.9	59.6	63.8

semantic similarity between code snippets. It is computed as:

$$\text{CodeBLEU} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}},$$

where BLEU is the n-gram match score, $\text{BLEU}_{\text{weight}}$ is the weighted n-gram match score, $\text{Match}_{\text{ast}}$ is the abstract syntax tree (AST) match score, and Match_{df} is the dataflow match score. The parameters α, β, γ, and δ are weighting coefficients.

We use an open-source implementation[11] of CodeBLEU, and the results are presented in Table 6. For the Java benchmark, the weights are set to $\alpha = \beta = \gamma = \delta = 0.25$. For the Rust benchmark, since the tool cannot extract dataflow from Rust code, we omit the dataflow match component and assign the weights to $\alpha = \beta = \gamma = 0.33$ and $\delta = 0$.

On both Java and Rust benchmarks, CatCoder and RepoCoder outperform the other two baselines, Vanilla and In-File, across CodeBLEU and all four sub-metrics, suggesting that relevant code retrieval helps LLMs generate code more similar to the ground truth. Notably, the scores of CatCoder and RepoCoder are very close, with RepoCoder even slightly outperforming CatCoder in CodeBLEU. However, since CatCoder achieves higher compile@k and pass@k scores, particularly on the Java benchmark, this indicates that RepoCoder sometimes produces code structurally similar to the ground truth but fails due to subtle issues. This underscores the importance of execution-based metrics in evaluating repository-level code generation.

Answer to RQ1: CatCoder is effective at generating repository-level function code, and it outperforms the RepoCoder baseline by up to 14.44% and 17.35% in terms of compile@k and pass@k scores. Besides, in qualitative analysis, we find that the type context is effective in filtering out misleading reference code.

5.2 RQ2: Ablation Study

To investigate the contribution of the components (as described in Section 3) to the effectiveness of CatCoder, we create the following variants:

[11] https://github.com/k4black/codebleu

(1) FULL: The complete version of CatCoder, exactly as Section 3 illustrates.
(2) -CR: *Relevant Code Retrieval* is removed.
(3) -TC: *Type Context Extraction* is removed.

We evaluate the above three variants on the same benchmarks as RQ1, also using *CodeLlama-13B-Instruct* as the underlying LLM. We compare the incomplete ones with FULL, and the results are illustrated in Table 7. We can conclude that CatCoder benefits from the two components, as the compile@k and pass@k scores of both incomplete variants drop noticeably. In detail, for the -CR variant, the compile@k and pass@k scores decrease by up to 19.43% and 38.35%, respectively, on the Java benchmark, and by up to 46.24% and 56.03% on the Rust benchmark. And for the -TC variant, the compile@k and pass@k scores drop by up to 6.63% and 11.57% respectively on the Java benchmark, and by up to 8.52% and 5.3% on the Rust benchmark.

Notably, removing *Type Context Extraction* leads to a smaller performance decline compared to removing *Relevant Code Retrieval*. This observation underscores the effectiveness of our hybrid retrieval mechanism. However, we emphasize that the importance of type context is not diminished. In many instances, particularly those involving functionalities that are symmetric to, or minor variations of, existing code in the repository, the retriever is able to find highly relevant examples. The presence of duplicated code snippets benefits retrieval-based generation by providing the LLM with strong, nearly complete templates for the target function. In such cases, the retrieved code is sufficient for the LLM to produce a correct solution, rendering the additional type context less critical. However, the performance gap between the -TC variant and FULL highlights a subset of more challenging problems where retrieval alone is insufficient. The qualitative examples in Fig. 5 illustrate a typical failure mode: the retrieved code may include distracting snippets or reference elements outside the current scope, resulting in generation errors. Therefore, we argue that our Type Context Extraction component is an essential and complementary mechanism.

Answer to RQ2: The two components of CatCoder, namely *Relevant Code Retrieval* and *Type Context Extraction*, improve the effectiveness of CatCoder.

5.3 RQ3: Generalizability

To assess whether the benefits of CatCoder are specific to a single LLM or generalize across different models, we conduct an extensive generalizability study. As described in Section 4.4.2, we select a diverse set of LLMs, including both code-specialized and general-purpose models of varying sizes (Table 4). We then evaluate the performance of the In-File, RepoCoder, and CatCoder approaches, as well as the CatCoder variants (-CR and -TC), using each of these models on both our Java and Rust benchmarks.

5.3.1 Results on Generalizability. Fig. 6 shows the pass@1 scores for these experiments. The results for both Java (Fig. 6a, 6c) and Rust (Fig. 6b, 6d) benchmarks suggest that CatCoder is able to improve the performance of every tested model.

On the Java benchmark, CatCoder improves the average pass@1 score by 13.73%, compared to RepoCoder (Fig. 6a). The improvement is universal, ranging from a significant increase for models like *DS-6.7B* to a more modest increase for models like *CL-13B*. Similarly, on the Rust benchmark (Fig. 6b), CatCoder consistently outperforms both In-File and RepoCoder for all LLMs. It improves the average pass@1 score by 8.4% compared to RepoCoder. Additionally, the relative performance ranking of the models remains largely consistent across both languages, with *CL-13B* and *DS-6.7B* generally at the top, and with *CG-7B* and *ML-8B* showing more modest results.

Furthermore, the ablation study results, when generalized across models (Fig. 6c, 6d), are consistent with the findings from RQ2. For both Java and Rust, the -CR variant (without Code Retrieval) consistently leads to the

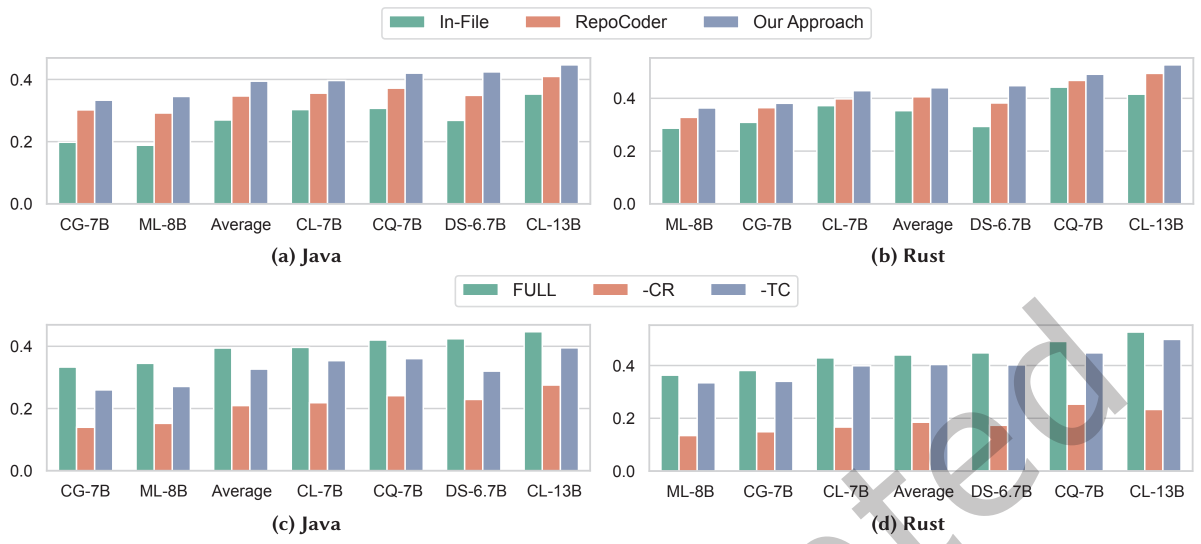


Fig. 6. **pass@1** performance comparison using various LLMs on the Java and Rust benchmarks. Subfigures (a) and (b) compare CatCoder with baseline methods—In-File and RepoCoder—while (c) and (d) compare CatCoder with its variants. Model acronyms on the x-axis are consistent with those listed in Table 4, and variant names refer to those introduced in RQ2. Models are sorted in ascending order based on the performance of CatCoder. The "Average" bar represents the mean pass@1 score across all evaluated models.

most significant performance degradation, while the -TC variant (without Type Context) shows a smaller but still noticeable drop. On average, across all models on the Java benchmark, the pass@1 score decreases by 46.89% without *Relevant Code Retrieval*, and decreases by 17.17% without *Type Context Extraction*. This pattern holds for Rust as well. On average, removing *Relevant Code Retrieval* decreases the pass@1 score by 57.94%, while removing *Type Context Extraction* leads to an 8.21% decrease.

In conclusion, our results support the generalizability of CatCoder. It consistently enhances the performance of a wide variety of LLMs on both Java and Rust, demonstrating that the architectural benefits of CatCoder are model-agnostic.

5.3.2 Comparison of different LLMs. The results of our generalizability study also offer insights into the current landscape of LLMs for code:

❶ Although the selected general purpose LLM (i.e., *ML-8B*) claims to be powerful in many aspects [8], it falls behind all selected code-specialized models except for *CG-7B* in our benchmarks. One example is *ML-8B* versus *CL-7B*. Although *CL-7B* is fine-tuned from an older foundation model (i.e., Llama 2 [42]), it outperforms the new model *ML-8B* (i.e., Llama 3 [8]). This indicates that code generation tasks benefit a lot from fine-tuning models using high-quality code datasets.

❷ From the trend in Fig. 6, although larger models tend to have a better absolute performance than smaller models, the improvement by CatCoder is larger on smaller models. For instance, the pass@1 score increase for models like *ML-8B*, *CQ-7B*, and *DS-6.7B* is more pronounced than the increase for *CL-13B*. This suggests that providing high-quality, structured context via frameworks like CatCoder can help smaller or less specialized models bridge the performance gap with larger, more powerful ones.

Answer to RQ3: CatCoder is generalizable to many different LLMs, as it improves the pass@1 scores of all selected LLMs, compared with both In-File method and RepoCoder. In addition, the performance of different LLMs on the benchmark provides valuable insights.

5.4 RQ4: Scalability

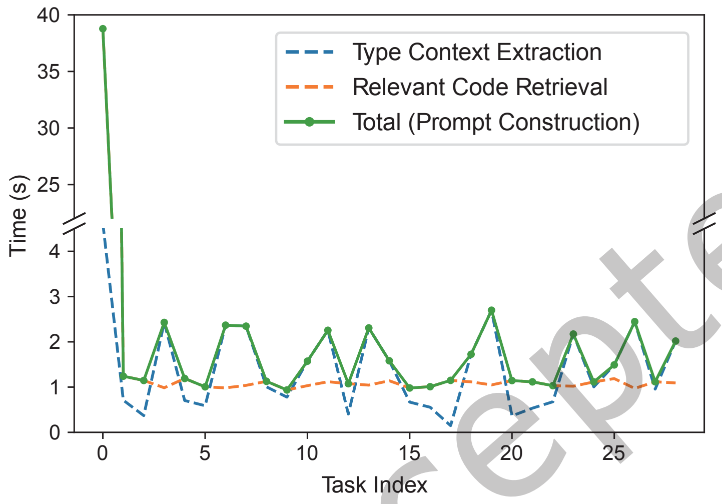


Fig. 7. Detailed latency of the components when constructing prompts for 29 code generation tasks sequentially in the *closure-compiler* project.

Table 8. Average latency of the components when constructing prompts for 29 code generation tasks sequentially in the `closure-compiler` project. CR: Relevant Code Retrieval, in Section 3.1; TC: Type Context Extraction, in Section 3.2.

Case	CR	TC	Total
Cold start (Task 0)	38.8s	4.62s	38.8s
Avg. after cold start	1.08s	1.32s	1.56s
Avg. speedup w/ cache	36x	3.5x	25x

In real-world software development, the latency of repository-level code generation tools is crucial, as they inherently process considerable volumes of contextual code in large and complex projects. Significant latencies would negate the advantages of automated assistance and lead to user frustration. Therefore, we propose this research question and investigate the scalability of CatCoder when applied to functions within large-scale code repositories.

5.4.1 Experimental Design. To robustly assess CatCoder's scalability, the *closure-compiler* repository[12] in our benchmark (Section 4.3) is selected for evaluation, by considering the following aspects:

[12]The Closure Compiler is a tool for checking and optimizing JavaScript code. The project is developed by Google and has attracted wide attention.

- This repository is the largest one in our benchmark and comprises approximately 400,000 lines of code, representing a non-trivial, real-world system. Thus, it can present a genuine scalability challenge for CatCoder.
- It is a well-known and actively developed open-source project. Thus, the evaluation is realistic and can reflect the types of complex dependencies, code structures, and API usage patterns that CatCoder would encounter in practical deployment.

Specifically, our benchmark contains 29 distinct function code generation tasks for the *closure-compiler* repository. For evaluation, we open this repository, invoke CatCoder on these code generation tasks sequentially, and examine the time consumption of each code generation task. This sequential execution simulates a common developer workflow where a programmer might work on several functions or address multiple issues within the same project during a single coding session.

For time consumption, we focus on the latency of CatCoder's core components (i.e., Relevant Code Retrieval and Type Context Extraction), while excluding the inference latency of the underlying LLM. This is due to that the time taken for LLM inference is highly dependent on external factors that are largely independent of CatCoder, including the specific LLM chosen, the underlying hardware, and the deployment environment. In addition, the length of prompts constructed by CatCoder does not scale with repository sizes, making the inference latency less important in scalability evaluation.

5.4.2 Analysis of Scalability. Fig. 7 shows the detailed latency of the two components of CatCoder when running each sequential code generation task in the *closure-compiler* repository. In addition, Table 8 shows the statistics of average latency. It is also worth noting that the two components are independent and can run in parallel. Thus, the total time is bounded by the slower component.

We can notice that CatCoder takes a longer time to process the very first code generation task in a repository that it has not previously analyzed. We refer to the first task as "cold start". The latency of cold start can be attributed to several setup operations in our approach:

- For Relevant Code Retrieval (CR), the setup operations mainly involve generating embedding vectors for code chunks, which are crucial for subsequent semantic search.
- For Type Context Extraction (TC), the operations include the static analyzer's setup processes, such as parsing the codebase and building representations of type hierarchies and dependencies.

As shown in Table 8, the measured cold start time is 38.8 seconds, with the Relevant Code Retrieval component as the main bottleneck. Considering that the *closure-compiler* repository is large, and it also takes many IDEs to index a new large repository for a long time, we believe that the cold start overhead of CatCoder is still acceptable for enabling its advanced context-aware code generation capabilities.

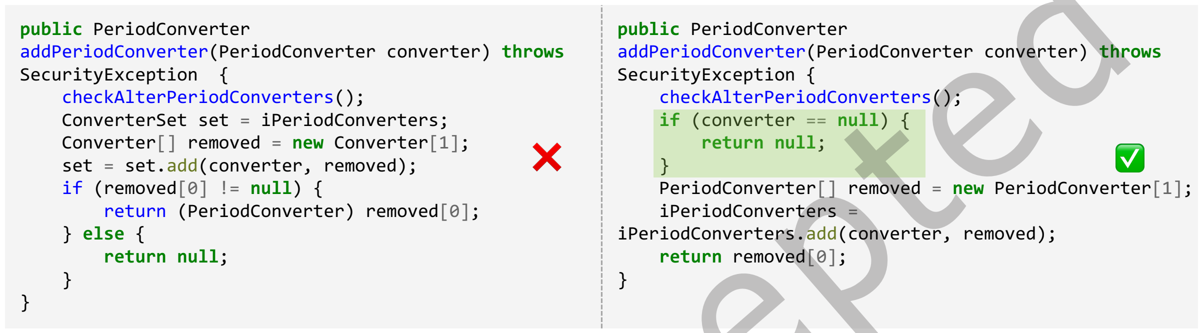
Following the initial cold start, CatCoder's performance for subsequent tasks improves significantly. Table 8 shows the reduction in average latency for tasks after the first one: Relevant Code Retrieval time drops to 1.08s, Type Context Extraction to 1.32s, and the total prompt construction time averages 1.56s. The results indicate an average 36x speedup for Relevant Code Retrieval, 3.5x speedup for Type Context Extraction, and 25x speedup for total time. In this case, after the initial waiting, users do not have to wait for a long time each time to receive a response in an interactive development workflow. This is due to CatCoder's caching mechanism, which stores intermediate data, including embedding vectors of code chunks and intermediate states of the static analyzer. The cached data is computationally expensive to generate, but generally stable across a sequence of related development tasks within a single coding session. The massive speedup after the cold start demonstrates the effective reuse of cache data.

Answer to RQ4: CatCoder shows practical scalability at a series of code generation tasks within large repositories. In the case of the *closure-compiler* repository, the caching mechanism of CatCoder is able to introduce a 25x average speedup after the cold start.

6 Discussion

In this section, we discuss the limitations of our approach and outline potential directions for future work, followed by a discussion of threats to validity.

6.1 Limitations of CatCoder



```
public PeriodConverter
addPeriodConverter(PeriodConverter converter) throws
SecurityException  {
    checkAlterPeriodConverters();
    ConverterSet set = iPeriodConverters;
    Converter[] removed = new Converter[1];
    set = set.add(converter, removed);
    if (removed[0] != null) {
        return (PeriodConverter) removed[0];
    } else {
        return null;
    }
}
```

```
public PeriodConverter
addPeriodConverter(PeriodConverter converter) throws
SecurityException {
    checkAlterPeriodConverters();
    if (converter == null) {
        return null;
    }
    PeriodConverter[] removed = new PeriodConverter[1];
    iPeriodConverters =
iPeriodConverters.add(converter, removed);
    return removed[0];
}
```

Fig. 8. A case that CatCoder fails to generate correct code. The first code is generated by CatCoder, and the second one is the ground truth.

6.1.1 *Limitation in the Underlying LLM.*

Apart from relevant code and type context, an LLM's internal capability of understanding general syntax and algorithmic principles is equally essential for correct repository-level code generation. Similar to many prior studies [32, 40, 48, 53], CatCoder operates on a frozen LLM. As such, this line of work is inherently limited in enhancing the model's logical reasoning abilities.

Fig. 8 presents a failure case produced by CatCoder. Although the generated code correctly uses all relevant APIs and closely resembles the ground truth implementation, it fails to pass some unit tests due to missing the logic for handling `null` pointers (highlighted in green in Fig. 8). This example illustrates that function code generation is inherently complex, and the effective use of repository context does not always guarantee correctness. A possible direction for future work could be fine-tuning LLMs on code repositories to enhance repository-level in-context learning, or adopting reinforcement learning approaches [29] to improve reasoning and coding capabilities.

6.1.2 *Limitation in Retrieval.*

In the Relevant Code Retrieval component, we adopt a hybrid retrieval strategy that combines sparse (BM25) and dense (embedding-based) methods to achieve high recall and precision. However, like any information retrieval system, it is not infallible. The retriever may still return incomplete or even misleading context. To some extent, the type context in CatCoder helps mitigate this limitation by supplying accessible API information that the LLM can cross-reference, as illustrated in Section 5.1.2. Possible future work includes enhancing the retrieval and reranking pipeline using powerful LLMs.

6.1.3 *Limitation in Static Analysis.*

Although Java is a statically typed programming language, it supports dynamic features such as reflection. The Java Reflection API enables dynamic class loading, object instantiation, and method invocation based on runtime strings. Despite the capabilities of the static analyzer we employ (i.e., Eclipse

JDT.LS, commonly used in IDEs), handling these dynamic features remains challenging. For example, if a target function is related to a field whose class is loaded reflectively at runtime (e.g., via the `Class.forName` method), the analyzer may fail to resolve the actual class and its members. In such cases, the resulting type context may omit information related to the truly dependent class. Therefore, a limitation is that CatCoder may produce *incomplete* type context when the repository heavily relies on dynamic features.

6.2 Threats to Validity

6.2.1 Internal Threats. In general, static analyzers may introduce *unsound* results, which can, in turn, affect dependent approaches. However, we argue that this is not a concern in CatCoder. Specifically, we use static analysis only to locate types and obtain their definitions and implementations. This form of analysis is lightweight and significantly more accurate than other forms, such as pointer or reflection analysis, as it operates solely on explicit type names in the code. Besides, the tools we use (i.e., Eclipse JDT and rust-analyzer) and the analysis we apply have been integrated into widely used IDEs and code editors (e.g., Eclipse and Visual Studio Code). Therefore, we consider further verification of the analysis results in CatCoder unnecessary, and we trust that our evaluation results remain valid.

6.2.2 External Threats. Our implementation of CatCoder is limited as it only supports Java and Rust. However, the proposed approach does not rely on detailed language specifications and is applicable to any statically typed programming language. In the future, to make CatCoder more practical, we plan to extend it to support more statically typed languages, for example, C++ and C#.

7 Related Work

7.1 Large Language Models for Code

Large language models for code (i.e., code LLMs) refer to a family of Transformer [45] models pre-trained or fine-tuned on massive code datasets. Previous work has shown the exceptional performance of code LLMs in various code-related tasks, such as code generation [17], test case generation [44], and program repair [49]. In general, pre-trained language models for code can be categorized into the following three types:

(1) **Encoder-only models**, such as CodeBERT [21] and GraphCodeBERT [22]. These models are pre-trained using Masked Language Modeling (MLM) techniques, and are popular and effective for tasks like classification, where a deep understanding of input data is required.
(2) **Encoder-Decoder models**, such as the CodeT5 family [46, 47]. These models first encode the input into an internal representation and then decode it into an output sequence.
(3) **Decoder-only models**, such as Codex [17], CodeGen [36], StarCoder [31] and CodeLlama [39]. These models are pre-trained using Next Token Prediction tasks and are powerful in generative tasks.

In our experiments, we select a variety of state-of-the-art code LLMs, all of which are decoder-only.

7.2 Benchmarks for Repository-Level Code Generation

Iyer et al. [26] are among the first to highlight the importance of the programmatic context and introduce the task of generating class member functions with natural language descriptions and class contexts. In summary, they construct Concode, a large training dataset consisting of Java classes from online code repositories, and propose an encoder-decoder model for code generation. With the development of large language models, many subsequent repository-level code generation benchmarks have arisen in academia. Du et al. [20] make an attempt to evaluate LLMs in the more challenging class-level code generation scenario and manually construct a benchmark named ClassEval, which consists of 100 class-level Python code generation tasks.

Compared with previous benchmarks, for example HumanEval [17] and MBPP [14], ClassEval is more pragmatic. However, it is still limited because, although the methods in the tasks are not standalone, the programmatic contexts are still restricted to very common libraries and methods within the same file. In real-world scenarios, a code repository generally spans a large number of files, making cross-file contexts essential for LLMs to generate the correct code [19].

Toward more realistic evaluation, subsequent benchmarks such as CoderEval [50], CrossCodeEval [19], and CoderUJB [52] are proposed, all of which are built on a set of real-world open-source repositories and utilize cross-file context for evaluation. A key novelty of CoderEval and CrossCodeEval lies in their ability to support multilingual code generation tasks, while CoderUJB innovatively spans five kinds of practical programming tasks.

As LLMs evolve, Liu et al. [32] point out data leakage and memorization problems, which may affect the integrity and trustworthiness of model evaluation results. To mitigate this issue, they propose RepoBench and regularly update the dataset to keep code samples up-to-date. Similarly, Li et al. [30] propose EvoCodeBench, which includes an automatic pipeline to evolve itself.

7.3 Repository-Level Code Generation with LLMs

Many existing approaches for repository-level code generation perform retrieval. Some work, such as [34, 51], searches for useful information from documentation or external databases. Specifically, Zan et al. [51] train a dual-encoder model, which encodes the natural language descriptions and API information, respectively, to retrieve possible APIs from the documentation of the library. Lu et al. [34] propose a retrieval-augmented generation approach named ReAcc, which retrieves similar code from a code database they built. However, these approaches are not flexible and general as their retrieval sources (e.g., API documentation and external databases) may not be available in other repositories.

Other retrieval-based approaches explore retrieving similar code snippets within the repositories. Shrivastava et al. [40] propose a framework named Repo-Level Prompt Generator to complete the code for holes in single lines. In the framework, they train a prompt proposal classifier that takes the repository files and a set of prompt proposals and outputs the predicted prompt proposal. Zhang et al. [53] propose RepoCoder, which utilizes repository information by iteratively retrieving similar code. In the initial iteration, it uses the natural language descriptions as query input, and in subsequent iterations, it uses previously generated code for queries.

Our work CatCoder combines retrieval-based techniques with a new kind of information called type context, and the comparison with RepoCoder in Section 5 shows that type context serves as essential auxiliary information for statically typed languages. In addition, CatCoder does not require model training or multiple generation iterations, making it more practical in real-world settings.

In addition to the above work, which primarily focuses on prompt engineering, several model training frameworks [23, 48] and decoding strategies [13] have been proposed to support repository-level code generation. As our approach CatCoder does not modify the underlying LLM, it can be combined with these frameworks to improve effectiveness.

8 Conclusion and Future Work

In this paper, we aim to address the lack of type dependency information in repository-level code generation for statically typed programming languages. We propose CatCoder, a code generation framework for statically typed programming languages, which improves the performance of LLMs by retrieving relevant code and extracting type context using static analyzers. CatCoder is systematically evaluated on Java and Rust benchmarks that are built from real-world open-source repositories. The results show that CatCoder outperforms the RepoCoder baseline by up to 17.35% in terms of pass@k scores. Further studies demonstrate the generalizability and scalability of CatCoder. We release the replication package of CatCoder, including the source code of CatCoder, the benchmark datasets, and the evaluation scripts. In future work, possible research attempts include investigating the data leakage problem of the benchmarks. These investigations may help further improve the applicability of our proposed approach in practical scenarios.

Acknowledgments

This research is supported by the Fundamental Research Funds for the Central Universities (No. 226-2025-00171).

References

[1] 2024. all-mpnet-base-v2. Available at: https://huggingface.co/sentence-transformers/all-mpnet-base-v2. Last accessed May 2024.

[2] 2024. clangd. Available at: https://clangd.llvm.org. Last accessed May 2024.

[3] 2024. CodeGemma: Open Code Models Based on Gemma. Available at: https://goo.gle/codegemma. Last accessed May 2024.

[4] 2024. crates.io: Rust Package Registry. Available at: https://crates.io/. Last accessed May 2024.

[5] 2024. Eclipse JDT LS. Available at: https://projects.eclipse.org/projects/eclipse.jdt.ls. Last accessed May 2024.

[6] 2024. How to write documentation - The rustdoc book. Available at: https://doc.rust-lang.org/rustdoc/how-to-write-documentation.html. Last accessed May 2024.

[7] 2024. Hugging Face. Available at: https://huggingface.co/. Last accessed May 2024.

[8] 2024. Introducing Meta Llama 3: The most capable openly available LLM to date. Available at: https://ai.meta.com/blog/meta-llama-3. Last accessed May 2024.

[9] 2024. Language Server Protocol Specification - 3.17. Available at: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/. Last accessed May 2024.

[10] 2024. rust-analyzer. Available at: https://rust-analyzer.github.io/. Last accessed May 2024.

[11] 2024. Rust Programming Language. Available at: https://www.rust-lang.org. Last accessed May 2024.

[12] 2024. TIOBE Index. Available at: https://www.tiobe.com/tiobe-index/. Last accessed May 2024.

[13] Lakshya A Agrawal, Aditya Kanade, Navin Goyal, Shuvendu K. Lahiri, and Sriram K. Rajamani. 2023. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 1401, 29 pages.

[14] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732* (2021).

[15] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen Technical Report. *arXiv preprint arXiv:2309.16609* (2023).

[16] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2023. MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation. *IEEE Trans. Softw. Eng.* 49, 7 (July 2023), 3675–3691. https://doi.org/10.1109/TSE.2023.3267446

[17] 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* (2021).

[18] Gordon V. Cormack, Charles L A Clarke, and Stefan Buettcher. 2009. Reciprocal Rank Fusion outperforms Condorcet and Individual Rank Learning Methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval* (Boston, MA, USA) *(SIGIR '09)*. Association for Computing Machinery, New York, NY, USA, 758–759. https://doi.org/10.1145/1571941.1572114

[19] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. CROSSCODEEVAL: A Diverse and Multilingual Benchmark for Cross-File Code Completion. In *Proceedings of the 37th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '23)*. Curran Associates Inc., Red Hook, NY, USA, Article 2023, 23 pages.

[20] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 81, 13 pages. https://doi.org/10.1145/3597503.3639219

[21] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv preprint arXiv:2009.08366* (2020).

[23] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (2024).

[24] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The Curious Case of Neural Text Degeneration. *arXiv preprint arXiv:1904.09751* (2019).

[25] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv preprint arXiv:1909.09436* (2019).

[26] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping Language to Code in Programmatic Context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 1643–1652. https://doi.org/10.18653/v1/D18-1192

[27] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[28] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 611–626. https://doi.org/10.1145/3600006.3613165

[29] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C.H. Hoi. 2022. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1549, 15 pages.

[30] Jia Li, Ge Li, Xuanming Zhang, Yunfei Zhao, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. 2024. EvoCodeBench: An Evolving Code Generation Benchmark with Domain-Specific Evaluations. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 57619–57641. https://doi.org/10.52202/079017-1837

[31] 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* (2023).

[32] Tianyang Liu, Canwen Xu, and Julian McAuley. 2024. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. In *International Conference on Representation Learning*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 47832–47850.

[33] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *arXiv preprint arXiv:2402.19173* (2024).

[34] 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)*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, Dublin, Ireland, 6227–6240. https://doi.org/10.18653/v1/2022.acl-long.431

[35] Yi Luan, Jacob Eisenstein, Kristina Toutanova, and Michael Collins. 2021. Sparse, Dense, and Attentional Representations for Text Retrieval. *Transactions of the Association for Computational Linguistics* 9 (2021), 329–345. https://doi.org/10.1162/tacl_a_00369

[36] 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* (2022).

[37] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2402.19173* (2020).

[38] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. 1995. Okapi at TREC-3. *Nist Special Publication Sp* 109 (1995), 109.

[39] Baptiste Roziere, 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* (2023).

[40] 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* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1314, 23 pages.

[41] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. MPNet: Masked and Permuted Pre-training for Language Understanding. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1414, 11 pages.

[42] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288* (2023).

[43] Andrew Trotman, Xiangfei Jia, and Matt Crane. 2012. Towards an Efficient and Effective Search Engine. In *OSIR@ SIGIR*. 40–47.

[44] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit Test Case Generation with Transformers and Focal Context. *arXiv preprint arXiv:2009.05617* (2020).

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

[46] Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 1069–1088. https://doi.org/10.18653/v1/2023.emnlp-main.68

[47] 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*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. https://doi.org/10.18653/v1/2021.emnlp-main.685

[48] Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 53270–53290.

[49] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-Trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering* (Melbourne, Victoria, Australia) *(ICSE '23)*. IEEE Press, 1482–1494. https://doi.org/10.1109/ICSE48619.2023.00129

[50] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 37, 12 pages. https://doi.org/10.1145/3597503.3623316

[51] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2022. When Language Model Meets Private Library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (Eds.). Association for Computational Linguistics, Abu Dhabi, United Arab Emirates, 277–288. https://doi.org/10.18653/v1/2022.findings-emnlp.21

[52] Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. 2024. CoderUJB: An Executable and Unified Java Benchmark for Practical Programming Scenarios. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 124–136. https://doi.org/10.1145/3650212.3652115

[53] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, Houda Bouamor, Juan Pino, and Kalika Bali (Eds.). Association for Computational Linguistics, Singapore, 2471–2484. https://doi.org/10.18653/v1/2023.emnlp-main.151

Received 8 January 2025; Revised 28 October 2025; Accepted 13 November 2025