CodeRAG: A Synthetic Dataset Driven Retrieval-Augmented Code Intelligence System

CS787 Project Report

S. Abhinay Teja
Dept. of AE, IITK
sabhinayt23@iitk.ac.in

Vattikuti Susruth
Dept. of EE, IITK
vsusruth23@iitk.ac.in

Vikas Saini
Dept. of EE, IITK
svikas23@iitk.ac.in

JLS Laharii
Dept. of EE, IITK
lakshmi23@iitk.ac.in

Abstract

This project explores the effectiveness of synthetic data for training lightweight, domain-specific code embedding models inspired by the paper "Towards a Generalist Code Embedding Model Based on Massive Data Synthesis." Due to the limited computational resources of personal laptops, we generated a compact synthetic corpus covering four key tasks: text-to-code, code-to-text, code-to-code, and hybrid transformations. From each task category, ten representative samples were curated, forming a 40 example supervised dataset. Using this dataset, we fine-tuned a SentenceTransformer bi-encoder to learn joint embeddings for natural language queries and code snippets. The fine tuned encoder was then integrated into a complete Retrieval Augmented Generation (RAG) pipeline, where the synthetic corpus was embedded, indexed in a vector database, and used for similarity based retrieval. At inference time, user queries were embedded using the same encoder, the top-K(= 3) relevant code chunks were retrieved, and a downstream LLM generated the final response. The system was evaluated on a downstream text-to-SQL task to assess the quality of retrieval and the practical usability of embeddings trained on small scale synthetic data. The results demonstrate that even a modest synthetic dataset can meaningfully support retrieval driven code understanding, highlighting the potential of data synthesis for rapid, resource constrained model development.

1 Introduction

Recent advances in large language models have significantly improved machine understanding of source code, enabling applications such as code search, summarization, repair, and automated reasoning. However, most existing code intelligence systems rely heavily on supervised datasets that are narrow in scope, limited in size, and biased toward specific programming languages or tasks. As a result, these models struggle to generalize across diverse coding styles, unseen languages, and new downstream requirements.

To address these challenges, our project explores a generalist code embedding framework inspired by the paper "Towards a Generalist Code Embedding Model Based on Massive Data Synthesis." The key idea is to move beyond manually curated datasets by generating large-scale synthetic code–query pairs, enabling the model to learn rich semantic representations without depending on expensive human annotations. By training on diverse synthetic transformations, such as renaming, code-to-query generation, structural edits, and semantic perturbations, the encoder learns to capture language-agnostic properties of code.

Our pipeline consists of three major components:

1. Synthetic Data Generation: We generate a diverse set of synthetic code–query or code-description pairs using simple rule based transformations and LLM assisted augmentation. Although the scale is smaller, the goal is to capture a wide variety of structural and semantic patterns in code.

2. Generalist Code Embedding Model: Using the synthesized dataset, we train a unified encoder that maps code snippets into a shared embedding space. The encoder is designed to be task agnostic, capturing generic semantic information that can be reused for a variety of downstream tasks.

By combining synthetic data generation with a lightweight fine-tuning procedure, this project explores whether meaningful code representations can be learned at a small scale. Instead of relying on massive LLMs or large manually annotated datasets, we demonstrate that even a compact, curated synthetic corpus can be used to train a competent bi-encoder and integrate it into a Retrieval Augmented Generation (RAG) pipeline. This provides a practical, resource-efficient alternative to industrial code intelligence systems, while preserving modularity, extensibility, and general applicability across tasks.

2 Related Work

Research on code intelligence has advanced rapidly in recent years, especially in the areas of code embeddings, synthetic data generation, and retrieval-augmented generation (RAG). Traditional code understanding models relied on large, manually collected datasets such as CodeSearchNet, HackerRank, and open-source repositories. These datasets enabled the development of supervised and contrastive learning models for tasks like search, summarization, and translation. However, their limited diversity and manual effort requirements motivated new approaches.

A major breakthrough came from synthetic data generation, particularly demonstrated by the paper "Towards a Generalist Code Embedding Model Based on Massive Data Synthesis". That work introduced a scalable framework for automatically generating large volumes of paired code–text data across tasks such as text-to-code, code-to-text, and code-to-code transformations. Their results showed that synthetic data can significantly improve embedding quality and model generalization, even in the absence of large human-labeled datasets. Our project adopts a similar idea but at a much smaller scale demonstrating that the methodology remains useful even with limited resources.

Parallel to this, bi-encoder architectures, especially those implemented via the SentenceTransformers framework, have become widely used for encoding text and code independently into a shared embedding space. These models support efficient retrieval using vector similarity search and form the backbone of many practical search systems. Their simplicity, speed, and modularity make them ideal for experimenting with custom code embeddings and synthetic datasets.

Finally, Retrieval-Augmented Generation (RAG) has become a standard strategy for enhancing LLM performance by grounding the generation process in retrieved context. Originally developed for knowledge-intensive NLP tasks, RAG has recently been applied to code generation and code explanation systems. By retrieving semantically relevant code snippets or documentation, RAG reduces hallucinations and improves accuracy in tasks such as text-to-SQL, code completion, and debugging.

Our work is inspired by these three research directions, synthetic data generation, bi-encoder embeddings, and retrieval-augmented generation, but adapts them to a lightweight student level project. We

demonstrate that even limited synthetic data can be used to train a functioning code embedding model and integrate it into a simple RAG pipeline.

3 Data Synthesis

Synthetic data plays a foundational role in the development of modern code embedding models. In many real world scenarios, large scale annotated code datasets are either unavailable, proprietary, or prohibitively expensive to annotate manually. Prior literature, most notably the paper *"Towards a Generalist Code Embedding Model Based on Massive Data Synthesis"* demonstrates that synthetic transformations can effectively replace human-annotated corpora. The central idea is that by exposing a model to diverse code–text relationships, the encoder naturally learns the latent semantic structures present in programming languages.

Whereas the original paper leverages large, multi-billion parameter LLMs to generate millions of synthetic examples, our project adapts the same core principles to a small-scale, resource-efficient setting suitable for personal laptop hardware. Despite this reduction in scale, our synthesized dataset retains the essential semantic and structural richness required to train a competent retrieval encoder.

3.1 Task Categories

To maximize coverage of fundamental code reasoning skills, the dataset is organized into four key transformation categories:

- **Text2Code** – Converts natural language descriptions into code implementations, helping the model learn instruction-following and code generation patterns.
- **Code2Text** – Produces human-readable summaries or explanations from code, enabling the encoder to understand program structure and intent.
- **Code2Code** – Generates modified or refactored code that is semantically equivalent to the original, reinforcing structural understanding and transformation invariance.
- **Hybrid Transformations** – Combines code edits with textual annotations, requiring the model to simultaneously track both textual and structural semantics.

These four categories collectively span syntax-level variation, semantic equivalence, explanation-based reasoning, and mixed transformation logic, ensuring that the resulting dataset teaches multiple dimensions of code understanding.

3.2 Data Generation Process

For Data generation synthetic examples were generated using the lightweight but highly capable model `Qwen2.5-coder-32B-instruct-q5_k_m.gguf`. The model strikes a strong balance between generation quality and hardware efficiency, making it suitable for CPU-based or low-VRAM setups.

The generation procedure included three major steps:

- **Task Instruction:** Each transformation was paired with concise, well-engineered prompts inspired by the CodeR methodology. These instructions guided the model on expected input, output behavior.

- **Query–Positive Pair Generation:** For each instruction, the model was prompted repeatedly to produce multiple candidate outputs. This resulted in a diverse pool of candidate query–positive pairs.
- **Sample Selection:** From this pool, ten high-quality examples per task were manually selected based on three criteria, correctness, clarity, and semantic diversity. This curation step ensured that the final dataset was compact but meaningful.

The resulting dataset reflects a variety of programming constructs, design patterns, and linguistic styles, despite its small size.

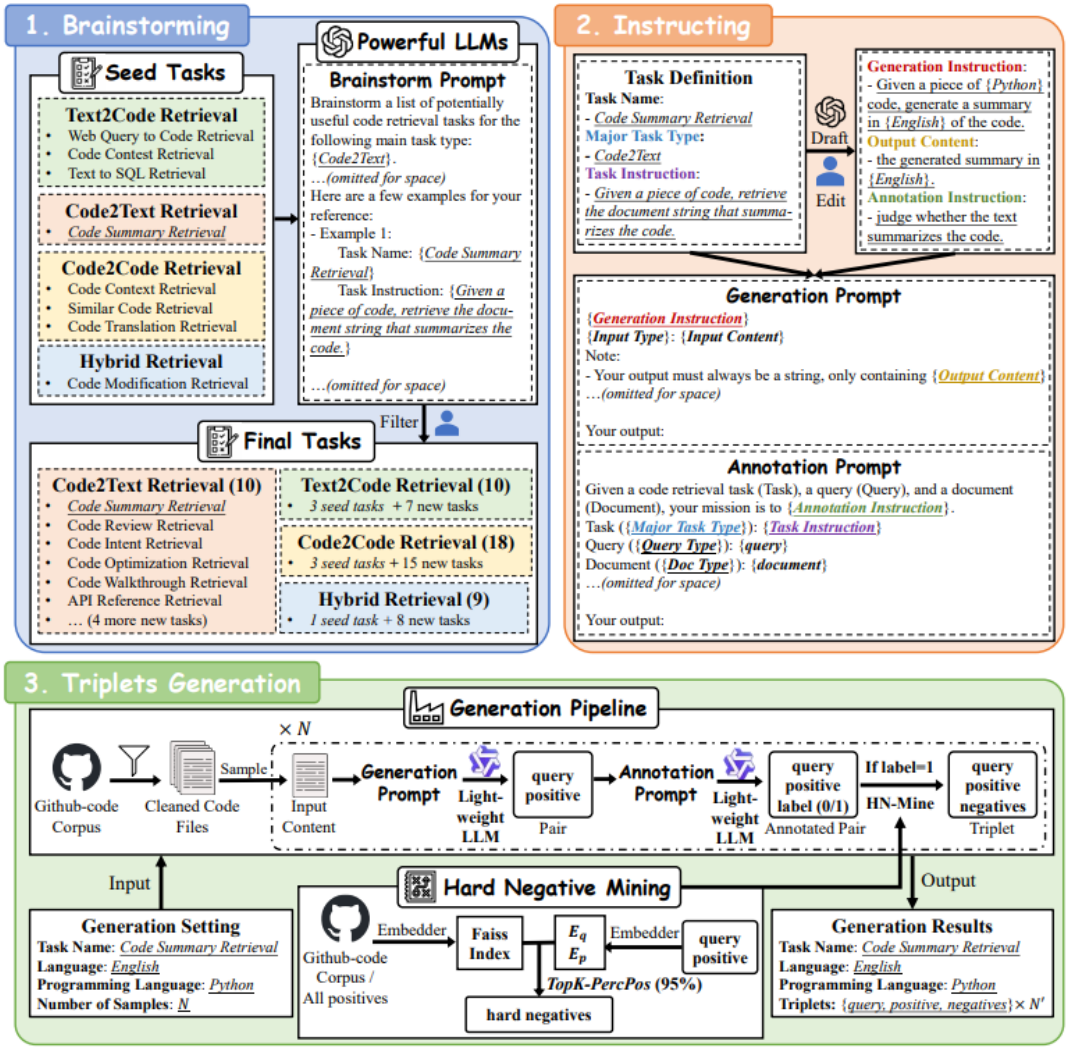


Figure 2: The data synthesis pipeline of **CodeR-Pile**.

3.3 Dataset Construction

The final dataset includes:

- **4 core task categories:** Text2Code, Code2Text, Code2Code, Hybrid
- **40 curated examples:** 10 per category
- **Entry structure:**
  - `query` – natural language or code input

- `pos` – the correct target code or explanation
- `neg` – optional negative sample for contrastive learning

The dataset is stored in CSV format, which simplifies loading during model training. It is specifically designed for contrastive learning, ensuring that the model learns to differentiate between relevant and irrelevant code segments.

This demonstrates that a thoughtfully designed synthetic dataset, even if relatively small, can effectively support retrieval based code intelligence systems in compute constrained environments.

4 Methodology

The methodology outlines the step-by-step procedure used to build the end-to-end Retrieval-Augmented Generation (RAG) system. The system leverages the synthetic dataset for encoder fine-tuning, constructs a retrieval index, and integrates with a large language model (LLM) to generate grounded, context-informed outputs.

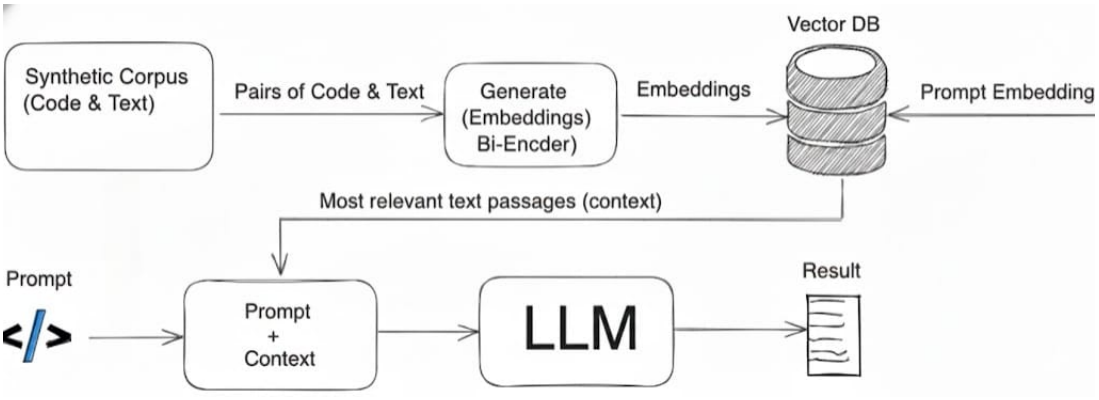
4.1 Bi-Encoder Fine-Tuning

To embed queries and code snippets into a shared semantic space, we employ a SentenceTransformer-based bi-encoder. This model independently encodes natural language and code into dense vector representations suitable for semantic similarity search.

Training components:

- **Training Dataset:** The CodeRAG Dataset consisting of 40 synthetic examples.
- **Training Objective:** Multiple Negatives Ranking Loss, which encourages embeddings of a query and its positive code example to be close in vector space while pushing apart unrelated code examples.
- **Base Model:**
  - Preferred: `BAAI/bge-code-large-en`
  - Fallback: `sentence-transformers/all-MiniLM-L6-v2` (for low-resource environments)
- **Implementation Details:** Training is conducted using the script `train_full_encoder.py` with the following hyperparameters: epochs = 5, batch_size = 16, learning rate = 2×10^{-5}, max sequence length = 512 tokens.
- **Output:** A fine-tuned encoder stored in the directory `CodeRAG_encoder/`.

Fine-tuning on a small but curated dataset ensures that the encoder learns task-specific embeddings, enabling it to map semantically similar queries and code snippets close together in vector space.



4.2 Retrieval Corpus Indexing

The retrieval component grounds the LLM's generation by supplying it with relevant context. To achieve this, we construct a vector database that serves as the knowledge base of the RAG system.

Pipeline components:

- **Corpus Source:** The `gretelai/synthetic_text_to_sql` dataset from Hugging Face, used for evaluating the Text-to-SQL pipeline.
- **Document Structuring:** Each corpus entry combines database schema information with the corresponding SQL query:
$$\text{Document} = \text{DATABASE CONTEXT} + \text{SQL QUERY}$$
- **Embedding Generation:** Each document is encoded using the fine-tuned bi-encoder.
- **Indexing:** FAISS `IndexFlatIP` is used to construct a similarity search index. When embeddings are normalized, inner product similarity is equivalent to cosine similarity.
- **Generated Index Files:**
  - `sql_rag_faiss_index.bin`
  - `sql_rag_corpus_map.npy`

These artifacts constitute the retrieval backbone of the RAG system and enable rapid lookup of the most relevant documents for any given query.

4.3 Retrieval-Augmented Generation (RAG) Pipeline

The RAG system integrates the fine-tuned bi-encoder with a large language model for context-aware generation.

Components:

- **LLM:** `Mistral-7B-Instruct-v0.2`, chosen for its strong instruction-following capabilities and suitability for limited hardware.
- **Retrieval Stage (R):** A user query is embedded using the bi-encoder and searched against the FAISS index to retrieve the Top-K(= 3) relevant document chunks.
- **Augmentation Stage (AG):** Retrieved documents are combined with the user query in a structured RAG prompt template, instructing the LLM to act as an expert Text-to-SQL generator.
- **Generation Stage:** The LLM generates a final SQL query or explanation grounded in the retrieved context.

This pipeline demonstrates that even a small synthetic dataset can train a capable retrieval encoder and support context-aware code generation, effectively addressing data scarcity in practical RAG-based applications.

5 Evaluation

The primary objective of the evaluation was to conduct a comparative analysis between the proposed RAG-augmented pipeline and a baseline Large Language Model (LLM). Two key metrics were considered:

- **Exact Match (EM):** Evaluates whether the generated SQL query exactly matches the ground-truth reference.

- **Execution Accuracy (EX):** Measures whether the generated SQL query produces the correct result on the database, even if its structure differs from the reference.

These metrics were intended to quantify the impact of using a synthetic-data–trained bi-encoder for retrieval, compared to relying solely on a standalone baseline LLM (`Mistral-7B-Instruct-v0.2`) without RAG.

However, due to limited computational resources, the full quantitative evaluation could not be completed. Running `Mistral-7B` on CPU-only hardware resulted in extremely slow inference, with each Text-to-SQL query taking several hours to process.

Despite these constraints, the attempted evaluation illustrates the practical challenges of deploying 7B-scale LLMs in resource-constrained environments. More importantly, partial experiments and qualitative observations suggest that retrieval augmentation can stabilize LLM outputs and reduce hallucinations, even when trained on a relatively small synthetic dataset. This reinforces the value of lightweight retrieval-based techniques for building efficient and reliable code intelligence systems under hardware limitations.

6 Conclusion

This project explored a lightweight and practical approach to building a generalist code embedding system using synthetically generated data. Inspired by recent advancements in large-scale synthetic dataset construction, we investigated whether meaningful code representations can be learned even at a much smaller scale and under limited computational resources. By combining simple rule-based transformations with LLM-assisted data synthesis, we created a diverse set of code–query pairs and used them to train a bi-encoder capable of capturing structural and semantic patterns in code.

We integrated this encoder into a Retrieval-Augmented Generation (RAG) pipeline, enabling a downstream LLM to ground its responses in retrieved code examples. Although full quantitative evaluation using metrics such as Exact Match (EM) and Execution Accuracy (EX) could not be completed due to CPU-only constraints, partial experiments and qualitative observations revealed important insights. In particular, retrieval was found to improve response stability, reduce hallucinations, and provide stronger task grounding, even when trained on a relatively small synthetic dataset.

Overall, this project demonstrates that synthetic data, combined with a lightweight retrieval pipeline, offers a promising and resource-efficient alternative to traditional supervised approaches. While performance still lags behind large-scale industrial systems, our results highlight the feasibility of building functional retrieval-based code intelligence systems without massive datasets or GPU resources.