CODE2JSON: CAN A ZERO-SHOT LLM EXTRACT CODE FEATURES FOR CODE RAG?

Aryan Singhal[*] **, Rajat Ghosh**[*] **, Ria Mundra, Harshil Dadlani, Debojyoti Dutta**
Nutanix
`{aryan.singhal, rajat.ghosh, debo.dutta}@nutanix.com`

ABSTRACT

A code retrieval-augmented generation (RAG) framework that accepts natural language (NL) queries and generates responses from relevant code contexts is crucial for enhancing developer productivity. However, building a code RAG system is inherently challenging due to the hierarchical structure and complex semantics of source code, especially with resource-constrained infrastructures. To address this, we introduce CODE2JSON, a zero-shot technique that leverages LLMs for extracting NL representations from code via semantic parsing. CODE2JSON serves as a programming language (PL)-agnostic feature extractor. We evaluate CODE2JSON on six programming languages—Python, Ruby, C++, Go, Java, and JavaScript—using approximately 125K records from eight widely used benchmark datasets, including HumanEval-X, MBPP, COIR, DS-1000, CSN, and ODEX. We examine the performance of CODE2JSON in different RAG setups for code retrieval and code generation tasks from NL queries. We explore nine retrieval models, encompassing sparse retrieval (e.g., BM25), text embeddings (e.g., BGE-Large), and code embeddings (e.g., CodeBERT), along with three LLMs: DeepSeekCoder-7B, Llama-3-8B, and Phi-2. Our findings indicate that CODE2JSON-assisted RAG outperforms the baseline approach in more than 50% of code retrieval and code generation tasks.

1 INTRODUCTION

The unique challenge in code RAG (Lu et al., 2022) is that it requires both knowledge-intensive retrieval of a standard RAG (Lewis et al., 2020b) and feature extraction, which generates NL features from a code document (Code $\longrightarrow$ NL Features). The feature extraction is essential because of the relatively low representation of code data in LLM pre-training datasets— resulting in many out-of-vocabulary (OOV) tokens (Tao et al., 2024) and limited context length of LLMs—inadequate for addressing hierarchical structures and complex semantics in codebases (Liu et al., 2021; Sadowski et al., 2015). One of the traditional code retrieval approaches designs domain-specific models trained on code comments, documentation, and discussions (Di Grazia & Pradel, 2023). However, this approach lacks generalizability across different codebases. The second approach, which uses abstract syntax tree (AST) (Lazar et al., 2017; Cui et al., 2010; Duracik et al., 2020)), is limited by the lack of semantic context (Hu et al., 2018), sensitivity (Cambronero et al., 2019), and complexity in traversal and matching (Allamanis et al., 2018).

Recent advancements in large language models (LLMs) have demonstrated remarkable proficiency in the $\{\text{text} \rightarrow \text{code}\}$ transformation task, wherein natural language descriptions are translated into executable code (Chen et al., 2021a; Nijkamp et al., 2022). These models exhibit robust multilingual capabilities and can generate syntactically and semantically correct code across various programming languages, often generalizing beyond their training data (Chen et al., 2021d). However, the inverse operation, $\{\text{code} \rightarrow \text{text}\}$, for example code summarization/documentation generation, remains a significant challenge (Haldar & Hockenmaier, 2024). Tasks such as code summarization, explanation, and intent prediction often yield inconsistent results, largely due to the inherent ambiguity and context dependence of programming logic (Zhang et al., 2023b). While fine-tuning

[*]equal contribution

and retrieval-augmented techniques have shown some promise, current models still struggle to produce consistently high-quality natural language descriptions of code, highlighting a key limitation in bidirectional text-code understanding.

LLMs have shown promise over previous symbolic AI (Smolensky, 1987) and deep reinforcement learning techniques (Van Hasselt et al., 2016) due to their rich priors (Requeima et al., 2024), inference time scalability (Kwon et al., 2023), and ability to generalize through natural language (Yuan et al., 2023). They are used in both zero-shot (Huang et al., 2022; Kojima et al., 2022) and few-shot settings (Brown & Mann, 2020)—supporting both reasoning (Shinn et al., 2024; Wei et al., 2022), planning (Yao et al., 2023; Chen et al., 2023; Huang et al., 2024), and of late tool usage (Patil et al., 2024; Li et al., 2023).

Building upon recent advancements with LLM, we introduce CODE2JSON, a zero-shot LLM framework designed for automated code feature extraction via semantic parsing. This study investigates the effectiveness of CODE2JSON in handling multiple programming languages, assessing its performance for both code retrieval and code generation from NL prompts. While our primary focus is on benchmark datasets, Code2JSON is also applicable to unstructured codebases where the mapping between prompts and code bodies is less defined. The application of Code2JSON to unstructured codebases is discussed in the Appendix (Section A.2).

2 METHOD

2.1 CODE2JSON

Figure 1 shows a schematic architecture for the CODE2JSON-assisted code RAG pipeline. Given a code chunk (c_i), CODE2JSON generates a semantic feature set d_i, consisting of four fields {*Summary*, *Function Description*, *Class Description*, *Data Description*}. Figure 2 shows the zero-shot prompt template used in CODE2JSON with a focus on composability and flexibility, conceptually similar to Meta Prompting (Zhang et al., 2024b). It is composed of three units. The first unit (highlighted in red) on the top is designed to extract a general representation of the code body. The second unit (highlighted in brown) is designed to specify different artifacts (e.g., function, class, data) in the code body. Finally, the third unit (highlighted in green) is designed to extract granular code features. It includes abstract reference fields such as LANGUAGE (for defining PL choices) and OBJECT (for defining artifacts such as function, etc.).

The features generated from CODE2JSON are stacked up vertically to produce a feature store in the JSON format. On the upstream side, CODE2JSON is capable of handling both structured and unstructured datasets. For a structured dataset, such as MBPP (Austin et al., 2021), it takes the field with code chunks (e.g., "code" in MBPP) and index them for the downstream code RAG. An index is a structured representation of a document collection that enables efficient retrieval of relevant information. It organizes and preprocesses text data to facilitate fast and accurate search. The corresponding text field (e.g., "text" in MBPP) act as the source of queries. The examples for CODE2JSON outputs of different types are shown in Table 4.

2.2 CODE RAG PIPELINE

We explore code RAG that uses a NL input sequence q to retrieve code documents d from a collection D and use them as an additional context when generating the target sequence y. Our code RAG pipeline follows the traditional RAG system in leveraging two components: (i) a pre-trained retriever model, $\mathcal{R}(D|q)$ that returns d (top-K documents) from D given a query q and (ii) a pre-trained generator, $\mathcal{G}(y_i|q, d, y_{1:i-1})$ that generates a current token based on a context of the previous $i-1$ tokens $y_{1:i-1}$, the original input q, and retrieved documents d. The sole distinction of our system compared to the traditional RAG system (Lewis et al., 2020a) is that we use Code2JSON to transform every code document, d_i into a NL feature set, d'_i.

2.2.1 RETRIEVAL PROCESS

We use a dense encoder $E_P(\cdot)$ to index for all the code documents offline. At run-time, we apply encoder $E_Q(\cdot)$ that maps an input question to a dense vector and retrieves k documents which are similar to the query vector.

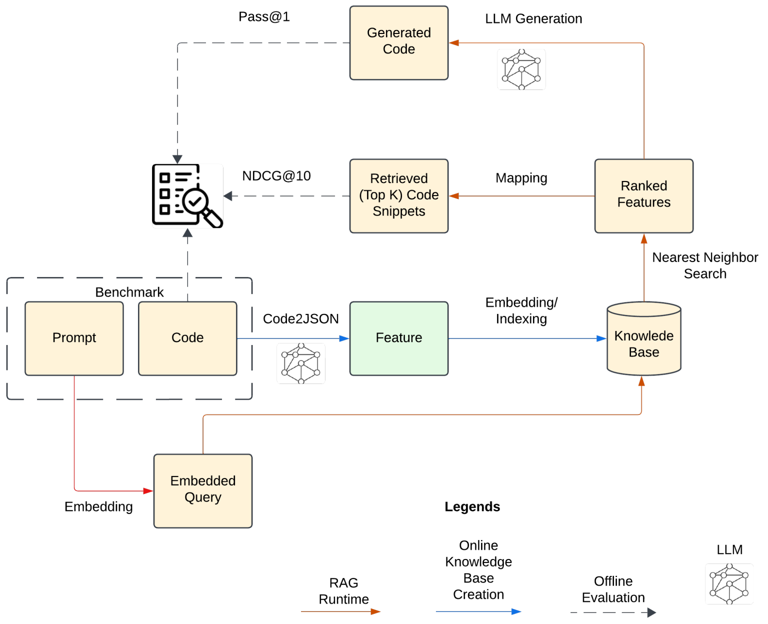


Figure 1: Schematic architecture of the CODE2JSON-assisted code RAG pipeline on benchmark datasets. The pipeline consists of three workflows. First, offline knowledge base creation: it processes all code snippets from a benchmark dataset, converting them into natural language (NL) summary features using the CODE2JSON prompt template, as illustrated in Figure 2. These NL features are subsequently embedded and indexed into a knowledge base, making them readily accessible during the code RAG runtime. Second, code RAG runtime: Each prompt from the benchmark dataset is processed sequentially and matched with the Top-K relevant features in the knowledge base. The ranked features serve two key functions: (1) retrieving the corresponding Top-K code snippets for retrieval evaluation, and (2) facilitating the generation of a new code snippet by serving as context. Finally evaluation for each query: we measure retrieval accuracy using NDCG@10(Normalized Discounted Cumulative Gain) Yining et al. (2013) and generation accuracy using pass@1, considering the corresponding code snippet as the ground truth. The code fields of the benchmark dataset remain inaccessible during the RAG runtime, mitigating the risk of data leakage.

$$\mathcal{R}(D \mid q) = \{d_i \in D \mid E_Q(q)^\top E_P(d_i) \text{ is among the top } k \text{ scores}\} \tag{1}$$

where $E_Q(q)^\top E_P(d_i)$ quantifies the similarity. This is essentially a maximum inner product search (MIPS) problem (Bruch et al., 2024) which has been widely studied (Johnson et al., 2019).

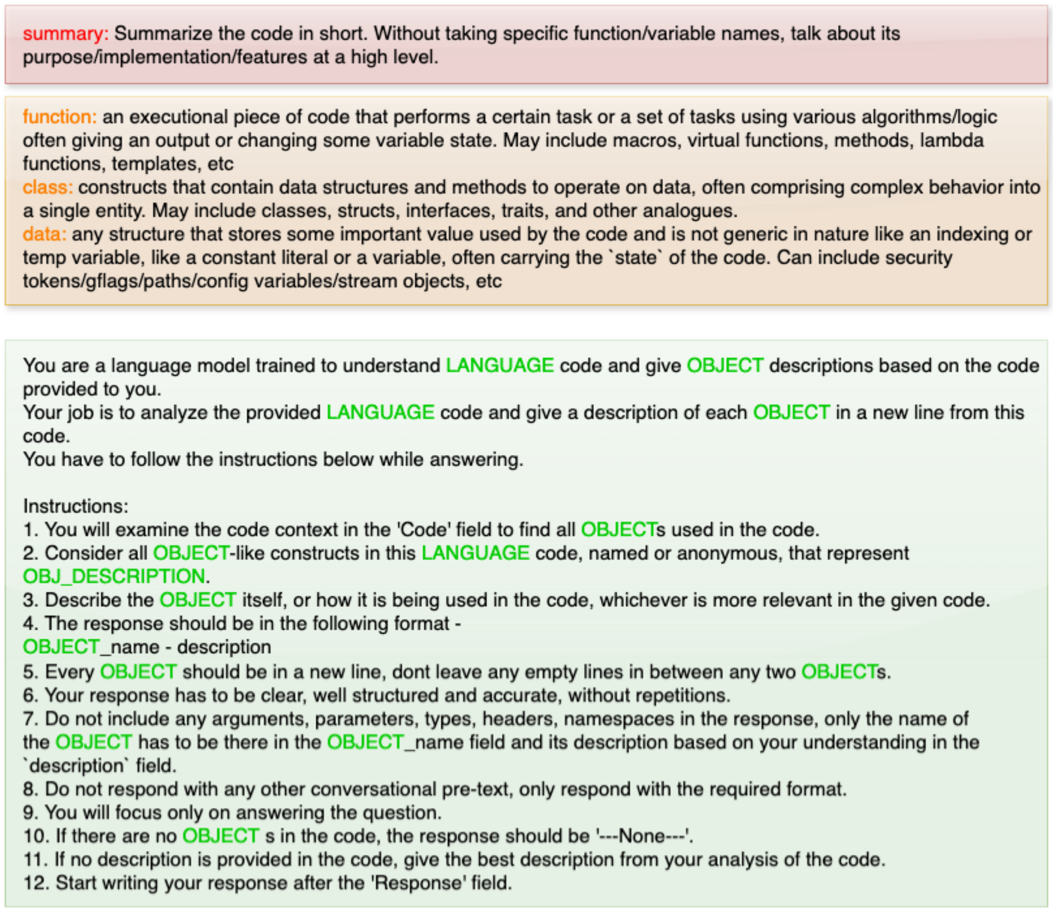
2.2.2 GENERATION PROCESS

The retrieved code documents $R(D \mid q)$ are then concatenated with the query q to form the model input for the generation model to produce the response.

$$p(y|x) = \prod_{i}^{N} \sum_{d \in \text{top-}k(p(\cdot|x))} \mathcal{R}(z|x)\mathcal{G}(y_i|q, d, y_{1:i-1}) \tag{2}$$

3 EXPERIMENT DESIGN

The primary research objective was to evaluate the effectiveness of CODE2JSON in code retieval and code generation from NL prompts.



summary: Summarize the code in short. Without taking specific function/variable names, talk about its purpose/implementation/features at a high level.

function: an executional piece of code that performs a certain task or a set of tasks using various algorithms/logic often giving an output or changing some variable state. May include macros, virtual functions, methods, lambda functions, templates, etc
class: constructs that contain data structures and methods to operate on data, often comprising complex behavior into a single entity. May include classes, structs, interfaces, traits, and other analogues.
data: any structure that stores some important value used by the code and is not generic in nature like an indexing or temp variable, like a constant literal or a variable, often carrying the `state` of the code. Can include security tokens/gflags/paths/config variables/stream objects, etc

You are a language model trained to understand LANGUAGE code and give OBJECT descriptions based on the code provided to you.
Your job is to analyze the provided LANGUAGE code and give a description of each OBJECT in a new line from this code.
You have to follow the instructions below while answering.

Instructions:
1. You will examine the code context in the 'Code' field to find all OBJECTs used in the code.
2. Consider all OBJECT-like constructs in this LANGUAGE code, named or anonymous, that represent OBJ_DESCRIPTION.
3. Describe the OBJECT itself, or how it is being used in the code, whichever is more relevant in the given code.
4. The response should be in the following format -
OBJECT_name - description
5. Every OBJECT should be in a new line, dont leave any empty lines in between any two OBJECTs.
6. Your response has to be clear, well structured and accurate, without repetitions.
7. Do not include any arguments, parameters, types, headers, namespaces in the response, only the name of the OBJECT has to be there in the OBJECT_name field and its description based on your understanding in the `description` field.
8. Do not respond with any other conversational pre-text, only respond with the required format.
9. You will focus only on answering the question.
10. If there are no OBJECT s in the code, the response should be '---None---'.
11. If no description is provided in the code, give the best description from your analysis of the code.
12. Start writing your response after the 'Response' field.

Figure 2: Prompt template used for zero-shot learning in CODE2JSON. It is composed of three units. The top unit gives the prompt used to obtain the summary for the whole code chunk. The middle unit denotes the abstract definitions used for function, class, and data. The bottom unit drives the extraction with the specific instantiations of a LANGUAGE (PL) and an OBJECT which is an programming artifact such as a function. Its abstract nature makes CODE2JSON PL agnostic.

3.1 CODE2JSON-ASSISTED RAG SETUP

Most code benchmarks consist of pairs of $\{\text{prompt}, \text{code}\}$, where the *prompt* represents a query or question, and the corresponding *code* contains the relevant implementation. To extract NL features, we apply the CODE2JSON prompt template to the *code* fields. These extracted NL features are then embedded and indexed into a knowledge base for efficient retrieval. Note that since these code snippets are typically standalone function definitions or smaller functional units, we omit granular features for functions, classes, and data descriptions for these datasets, which is needed for larger files and complete programs as done for code repositories (Section A.2).

During the retrieval-augmented generation (RAG) runtime, the embedded prompt is matched against the indexed feature set, and the Top-k NL features are retrieved. These retrieved features are subsequently used in two ways: (1) they are mapped to their corresponding code snippets for retrieval evaluation, and (2) they are leveraged to generate a new code snippet. The overall workflow is illustrated in Figure 1.

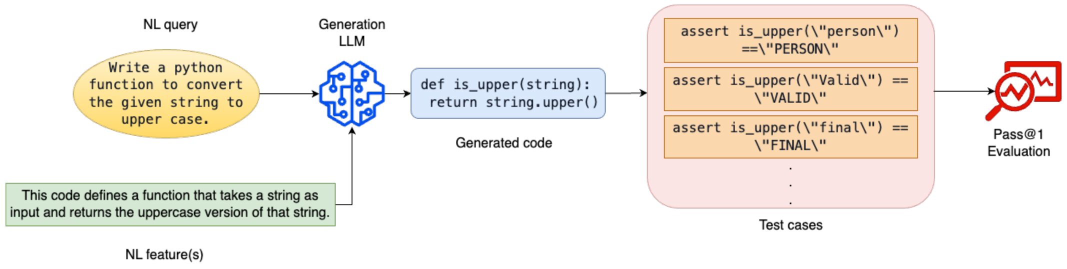

NL query
Write a python function to convert the given string to upper case.
Generation LLM
def is_upper(string):
  return string.upper()
Generated code
assert is_upper(\"person\") ==\"PERSON\"
assert is_upper(\"Valid\") == \"VALID\"
assert is_upper(\"final\") == \"FINAL\"
Test cases
Pass@1 Evaluation
This code defines a function that takes a string as input and returns the uppercase version of that string.
NL feature(s)

Figure 3: Example usage of CODE2JSON for code generation evaluation using pass@1 metric

For evaluation, we assess the quality of the retrieved code snippets using the NDCG@10 metric and the generated code snippet using the pass@1 metric. Figure 3 shows how pass@1 metric is computed for an example case.

The baseline approach (building on work done in CodeRAG-Bench (Wang et al., 2024)) directly indexes the code fields from a benchmark dataset, structured as $\{\text{prompt}, \text{code}\}$, into a knowledge base. The knowledge base is then utilized within the standard RAG workflow, as illustrated by the yellow boxes in Figure 1. The effectiveness of Code2JSON is measured as the relative improvement in code retrieval (measured in NDCG@10) and code generation (measured in pass@1) from NL prompts compared to the baseline approach.

3.2 BENCHMARKS

Due to its widespread use, ample public data, and popularity in both research and industry, Python enjoys abundant, easily accessible resources and is frequently leveraged in training code-focused language models, placing it squarely in the “high-resource” category (Cassano et al., 2024). By contrast, languages such as Ruby and Go are typically regarded as “low-resource” because they lack similarly extensive datasets. To ensure comprehensive coverage, robust performance, and broad generalizability, we therefore evaluated CODE2JSON on both Python and non-Python benchmarks.

- For Python dataset, we use 72,655 records from HumanEval (Chen et al., 2021c), MBPP (Austin et al., 2021), DS-1000 (Lai et al., 2023), ODEX (Wang et al., 2023c), COIR (Li et al., 2024b) CodeFeedback-MT dataset, and COIR (Li et al., 2024b) CodeTrans-Contest. The dataset is dominated by DS-1000 and ODEX.
- For non-Python dataset, we use 53,656 records from HumanEval-X (C++, Go, Java, JavaScript) (Zheng et al., 2023), and CodeSearchNet (Ruby) (Husain et al., 2019). The dataset is dominated by CodeSearchNet (Ruby). Overall, the evaluation datasets are rich in complexity and diversity.

3.3 LLMS IN CODE2JSON

Given the resource constraints of our setup, we use Llama-3-70B-Instruct-AWQ-4bit (ext) with an 8,000 context length for feature extraction, along with the prompt template shown in Figure 2. In the ablation study presented in the Appendix (Section A.4), we compare the CODE2JSON results obtained from Llama-3-70B-Instruct-AWQ-4bit with those from Llama-3-8B-Instruct. Both models achieve similar levels of performance.

3.4 INDEXING STRATEGIES FOR CODE RETRIEVAL

The comparative study has been conducted with nine different indexing strategies, including keyword matching (BM25), text embedding (BGE-Base, BGE-Large (Luo et al., 2024a), GIST-Base, GIST-Large (Solatorio, 2024), MPNET (Song et al., 2020)), and code embedding (Instructor (Su et al., 2023), CodeBert (Feng et al., 2020a), CodeT5-small (Wang et al., 2021b)).

3.5 LLMs for Code Generation

In the downstream NL2code generation phase, we compare the results from DeepSeekCoder-7B-Instruct (Guo et al., 2024), Llama-3-8b-Instruct ((lla)), and Phi-2 ((phi)). Phi-2 is a smaller model compared to the rest, having 2.7B parameters in floating-point 16 precision, and 2048 context length. All models are open-source with permissible licenses (gen). We use minimal prompting, since the data is already in a $query - response$ format. A simple one line instruction directing the LLM to generate the code based on given context is provided.

4 Results

We evaluate Code2JSON for $\sim$125K records, collected both from Python and non-Python benchmark datasets. We assess Code2JSON's performance following both retrieval and generation stages in our Code RAG pipeline.

4.1 Code2JSON Evaluation for Retrieval

We evaluate retrieval performance using NDCG@10 for both Python and non-Python datasets, as described in Section 3. Table 1 presents the results across six Python benchmarks (columns) and nine different indexing techniques (rows). Each value in the table indicates the percentage improvement of Code2JSON over the baseline. The last row shows Win Percentage of Code2JSON, defined as the number of times Code2JSON is winning over the baseline. The weighted average of the Win Percentage across all benchmarks is equal to 55.7% and the weighted standard deviation is equal 6.55%. We include HumanEval as a benchmark despite its primary design for code-to-code retrieval rather than NL2code, since it is a widely used dataset in code-specific tasks (Zhang et al., 2025; Qian et al., 2025; Luo et al., 2024b). In HumanEval, Code2JSON is winning only once, with keyword-matching, due to its code-to-code retrieval nature rather than NL2code. But, for the six out of remaining eight indexing models the NDCG@10 drop in Code@JSON is less than 1%. For BM25, NDCG@10 drops for more complex datasets such as DS-1000. Text Embedding models like MPNET do well for Python code becgentuse Python is highly readable and structured, making it easier for embeddings to capture meaning. In addition, the modular nature of Python (functions, classes, modules) fits well with Transformer-style tokenization and embeddings. CodeFeedback-MT has a blend of code snippets interspersed with descriptive text, which could explain its poorer performance with keyword and code embedding-based methods. A closer scrutiny of Table 1 suggests relative superiority of Code2JSON for smaller embedding models such as BGE-base and GIST-base. This indicates potential for greater resource efficiency.

Table 2 presents the results for five non-Python benchmarks (columns) across four indexing techniques (rows). The weighted average Win Percentage across all benchmarks is 50.1%, with a weighted standard deviation of 1.95%.For the non-Python datasets, it is observed that both keyword-matching and code-embedding models, such as Code2JSON, perform comparably to or even outperform the baseline. Notably, CodeT5-small (Wang et al., 2021b) demonstrates strong performance, even on non-Python datasets. This can be attributed to its encoder-decoder architecture and training on data that includes non-Python programming languages. Interestingly, BM25 performs significantly better on non-Python datasets compared to Python datasets, likely due to the more verbose nature of non-Python languages, which facilitates better keyword matching between queries and code snippets. A comprehensive analysis, including text embedding results, is presented in Table 6 in Appendix.

4.2 Code2JSON Evaluation after Generation

To fully capture the effectiveness of Code2JSON on the RAG task, we evaluate how it affects the generation of subsequent responses. We compare the relative quality of the final code produced by Code2JSON compared to the baseline. Due to the limited budget, we focus on evaluation only on the Python benchmarks such as HumanEval, MBPP, DS-1000, and ODEX. We use DeepSeekCoder-7B-Instruct (Guo et al., 2024), Llama-3-8b-Instruct (lla), and Phi-2 (phi), as the evaluator model. We use the pass@1 metric for evaluating code quality. The results are shown in Table 3

Table 1: Relative improvement in **NDCG@10** from using CODE2JSON feature extraction on Python datasets. Each cell shows the percentage change (positive or negative) relative to indexing raw code. The final row indicates how often the CODE2JSON-based approach outperforms direct code indexing.

	HumanEval (164)	**MBPP** (964)	**DS-1000** (34K)	**ODEX** (34K)	**CodeFeedback-MT** (3319)	**CodeTrans-Contest** (1008)
			Keyword Matching			
BM25	+22.6%	+1204.4%	-53.5%	-22.4%	-23.4%	-62.2%
			Text Embedding			
BGE-base	-0.68%	+8.2%	+0.54%	+11.4%	+0.7%	+1.2%
BGE-large	0.0%	+6.4%	+6.7%	+17.5%	-0.13%	+3.5%
GIST-base	-0.67%	+7.6%	+6.7%	-0.07%	+2.25%	+5.3%
GIST-large	-0.45%	+8.0%	-1.92%	-2.0%	+0.08%	+5.1%
MPNET	-0.22%	+3.8%	+17.7%	+11.3%	-7.7%	+5.1%
			Code Embedding			
Instructor	-0.22%	+7.6%	+25.7%	+31.1%	-0.14%	+1.8%
CodeBert	-41.7%	+43.7%	0.0%	0.0%	+62.5%	0.0%
CodeT5-small	-69.5%	-20.0%	-100.0%	+43.9%	-72.9%	-15.3%
Win Percentage	11.1%	88.9%	55.6%	55.6%	33.3%	66.7%

Table 2: Relative improvement in **NDCG@10** from using CODE2JSON feature extraction on non-Python datasets. Each cell shows the percentage change (positive or negative) relative to indexing raw code. The final row indicates how often the CODE2JSON-based approach outperforms direct code indexing.

	CSN(Ruby) (53K)	**HumanEval-X(cpp)** (164)	**HumanEval-X(go)** (164)	**HumanEval-X(java)** (164)	**HumanEval-X(js)** (164)
			Keyword Matching		
BM25	+213.7%	-17.3%	+14.6%	+29.9%	+7.2%
			Code Embedding		
Instructor	-6.98%	-7.59%	-10.7%	-10.9%	-6.7%
CodeBert	-75.0%	+25.7%	-7.88%	+50.6%	+4.9%
CodeT5-small	+32.4%	+9.5%	+8.9%	+2.4%	+94.6%
Win Percentage	50.0%	50.0%	50.0%	75.0%	75.0%

We see that barring MBPP, using featurized code is atleast as good as indexing raw code, if not better. HumanEval and DS-1000 shows very positive results with majority cases benefiting from code-features. For ODEX, there are many instances of CODE2JSON performing on par with the baseline. Diving into LLM-specific analysis, we see that DeepSeekCoder-7B-Instruct and Llama-3-8B-Instruct perform better than Phi-2, largely because of its smaller size of 2.7B parameters. These results highlight a strong evidence for using featurized descriptions as proxies for code in RAG and other related tasks.

No-retrieval generation (or the zero-shot setting) results are also provided in Appendix (Figure 7) along with exact pass@1 values, to compare with the RAG performance.

5 RELATED WORK

The retrieval-augmented generation (RAG) paradigm has gained significant traction in natural language processing (NLP) and code intelligence. Traditional RAG models augment language mod-

Table 3: Relative improvement in code generation (**pass@1**) scores. Each cell shows the percentage increase (or decrease) in performance when using CODE2JSON (descriptions) over the baseline (raw code). The final row indicates the proportion of instances where CODE2JSON outperforms the regular RAG workflow.

Method	**HumanEval**	**MBPP**	**DS-1000**	**ODEX**
DeepSeekCoder-7B-Instruct				
BM25	+8.15%	-0.31%	+65.41%	+6.78%
GIST-Large	-16.35%	-16.72%	+46.48%	+5.77%
MPNET	+13.82%	-16.88%	+33.49%	+10.31%
Llama-3-8B-Instruct				
BM25	+4.63%	+42.36%	+24.00%	0.00%
GIST-Large	-15.17%	+50.70%	+46.43%	0.00%
MPNET	-28.41%	+41.50%	+22.73%	-4.60%
Phi-2				
BM25	+77.36%	-96.77%	0.00%	0.00%
GIST-Large	+68.93%	-99.40%	+11.11%	0.00%
MPNET	+67.18%	-99.69%	+15.56%	-100.00%
Win percentage	66.7%	33.3%	88.9%	33.3%

els by retrieving relevant documents from an external corpus to improve contextual awareness and response quality. While RAG is well-explored for textual queries, its application in code-based retrieval and generation poses additional challenges due to the hierarchical structure and syntactic complexity of code (Lu et al., 2022).

Early approaches to code retrieval relied on information retrieval techniques such as keyword-based search and classical retrieval models like BM25. However, these approaches often fail to capture the deep semantics of code. More recent works leverage embeddings-based retrieval using models such as CodeBERT (Feng et al., 2020b), GraphCodeBERT (Guo et al., 2021), and CodeT5 (Wang et al., 2021a), which train on large-scale code datasets to improve contextual retrieval.

Feature extraction from code has also been explored in different ways. Abstract Syntax Trees (ASTs) (Allamanis et al., 2018) and static analysis-based methods (Hu et al., 2018) extract structured representations from code, but they often lack semantic generalizability across different programming languages. Large language models (LLMs) have shown strong capabilities in zero-shot and few-shot learning for code tasks (Chen et al., 2021d; Gao et al., 2023), making them ideal candidates for code feature extraction. Studies such as (Su et al., 2024; Zhang et al., 2023a) explore LLM-assisted code retrieval and summarization, but they typically require fine-tuning or prompting strategies optimized for specific datasets.

Recent research highlights the advantages of using LLMs for code intelligence tasks. Models like Codex (Chen et al., 2021d), CodeLlama (Roziere et al., 2023), and DeepSeekCoder (Guo et al., 2024) demonstrate strong code generation and reasoning abilities, enabling applications such as automatic documentation, debugging, and code summarization. However, the effectiveness of zero-shot feature extraction for RAG-based code retrieval remains underexplored.

Our work introduces CODE2JSON, for feature extraction from code, designed to be programming language-agnostic. Unlike prior work that relies on supervised training of embedding models or AST-based representations, etc., CODE2JSON aims to bridge the gap between structured feature extraction and effective RAG-based retrieval. We demonstrate its efficacy across diverse programming languages and retrieval models, highlighting its potential for improving code search and retrieval in resource-constrained settings.

6 Conclusion, Limitations, and Future Work

We presented Code2JSON, a zero-shot LLM-based framework for generating natural language summaries and structured code representations. Our experiments on $\sim$125K code snippets across multiple programming languages demonstrate that indexing Code2JSON-generated features, rather than raw code, can improve or at least match retrieval and code generation quality in many scenarios, even under resource constraints. Below, we discuss some key limitations and future research directions:

- **Prompt Sensitivity:** Operating in a zero-shot setting, Code2JSON's performance is strongly influenced by the LLM's general-purpose capabilities. In some cases, it may generate incomplete or incorrect descriptions due to hallucinations. Advanced prompt optimization techniques such as "LLM as optimizers" (Yang et al., 2024) or self-consistency (Wang et al., 2023b) could mitigate this.
- **Data Leakage:** A critical challenge in evaluating LLM-based code retrieval and generation models is data leakage—where test queries may have direct or indirect overlap with pre-training data (Li et al., 2024a). This can result in inflated performance metrics that do not reflect real-world generalization and possible security threats (Zeng et al., 2024). Future work should rigorously audit benchmark datasets to detect such leakages and explore deduplication strategies (Lee et al., 2022) to ensure fair evaluations.
- **Lack of Ground Truth Annotations:** While our experiments were conducted on widely used benchmark datasets, real-world adoption requires further validation on diverse and unstructured repositories. Expanding the evaluation to additional datasets and introducing human-in-the-loop assessments could improve robustness.
- **Context Length Constraints:** For large codebases or long snippets, the LLM's context window poses a limitation. Splitting code into smaller chunks for processing can lead to loss of contextual dependencies, necessitating hierarchical retrieval methods or chunk-aware generation strategies (Wang et al., 2023a).
- **Computational Cost:** Indexing large codebases using LLMs can be computationally intensive, especially for real-time retrieval systems. Exploring efficient distillation techniques (Hinton et al., 2015) or lightweight retrieval augmentation (Borgeaud et al., 2022) may offer more scalable alternatives.

Despite these challenges, Code2JSON demonstrates a promising step toward structured feature extraction for code retrieval-augmented generation (code RAG) systems. We recommend future research should focus on hybrid approaches that integrate symbolic reasoning with LLM-based methods, aiming to improve efficiency, generalizability, and interpretability in automated code understanding.

REFERENCES

https://github.com/huggingface/diffusers.

https://huggingface.co/casperhansen/llama-3-70b-instruct-awq.

https://huggingface.co/meta-llama/Meta-Llama-3-8B/blob/main/LICENSE.

Introducing meta llama 3: The most capable openly available llm to date. https://ai.meta.com/blog/meta-llama-3/.

https://github.com/huggingface/peft.

Phi-2: The surprising power of small language models. https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/.

https://github.com/huggingface/transformers.

https://tree-sitter.github.io/tree-sitter/.

https://github.com/huggingface/trl.

Simone Alghisi, Massimo Rizzoli, Gabriel Roccabruna, Mahed S. Mousavi, and Giuseppe Riccardi. Should we fine-tune or rag? evaluating different techniques to adapt llms for dialogue. In *Proceedings of the 17th International Natural Language Generation Conference*, pp. 180–197, 06 2024. doi: 10.48550/arXiv.2406.06399.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

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

Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, Bogdan Damoc, Simon Osindero, Karen Simonyan, Jack W Rae, et al. Improving language models by retrieving from trillions of tokens. *arXiv preprint arXiv:2112.04426*, 2022.

Tom Brown and et al. Mann. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020.

Sebastian Bruch, Franco Maria Nardini, Amir Ingber, and Edo Liberty. Bridging dense and sparse maximum inner product search. *ACM Transactions on Information Systems*, 42(6):1–38, 2024.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. When deep learning met code search. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 964–974, 2019.

Federico Cassano, John Gouwar, Francesca Lucchetti, Claire Schlesinger, Anders Freeman, Carolyn Jane Anderson, Molly Q Feldman, Michael Greenberg, Abhinav Jangda, and Arjun Guha. Knowledge transfer from high-resource to low-resource programming languages for code llms. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):677–708, 2024.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021a.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021b.

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. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021c.

Mark Chen, Jerry Tworek, Henry Jun, Qian Yuan, Henrique de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Chen, William Thompson, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021d.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL `https://openreview.net/forum?id=YfZ4ZPt8zd`.

Baojiang Cui, Jiansong Li, Tao Guo, Jianxin Wang, and Ding Ma. Code comparison system based on abstract syntax tree. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pp. 668–673, 2010. doi: 10.1109/ICBNMT.2010.5705174.

Luca Di Grazia and Michael Pradel. Code search: A survey of techniques for finding code. *ACM Computing Surveys*, 55(11):1–31, 2023.

Michal Duracik, Patrik Hrkut, Emil Krsak, and Stefan Toth. Abstract syntax tree based source code antiplagiarism system for large projects set. *IEEE Access*, 8:175347–175359, 2020. doi: 10.1109/ACCESS.2020.3026422.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547, Online, November 2020a. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL `https://aclanthology.org/2020.findings-emnlp.139/`.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020b. URL `https://arxiv.org/abs/2002.08155`.

Tian Gao et al. Palcoder: Enhancing code generation with program-aided language models, 2023. arXiv preprint, available at `https://arxiv.org/abs/230X.XXXXX`.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021. URL `https://arxiv.org/abs/2009.08366`.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Aman Gupta, Anup Shirgaonkar, Angels de Luis Balaguer, Bruno Silva, Daniel Holstein, Dawei Li, Jennifer Marsman, Leonardo O Nunes, Mahsa Rouzbahman, Morris Sharp, et al. Rag vs fine-tuning: Pipelines, tradeoffs, and a case study on agriculture. *arXiv preprint arXiv:2401.08406*, 2024.

Rajarshi Haldar and Julia Hockenmaier. Analyzing the performance of large language models on code summarization. In *Proceedings of the 13th Language Resources and Evaluation Conference*, pp. 995–1008, 2024.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL `https://openreview.net/forum?id=nZeVKeeFYf9`.

Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *Proceedings of the 26th conference on program comprehension*, pp. 200–210, 2018.

Wenlong Huang, Pieter Abbeel, Deepak Pathak, and Igor Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International conference on machine learning*, pp. 9118–9147. PMLR, 2022.

Xu Huang, Weiwen Liu, Xiaolong Chen, Xingmei Wang, Hao Wang, Defu Lian, Yasheng Wang, Ruiming Tang, and Enhong Chen. Understanding the planning of llm agents: A survey. *arXiv preprint arXiv:2402.02716*, 2024.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*, 2019.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Philippe Laban, Alexander R Fabbri, Caiming Xiong, and Chien-Sheng Wu. Summary of a haystack: A challenge to long-context llms and rag systems. *arXiv preprint arXiv:2407.01370*, 2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pp. 18319–18345. PMLR, 2023.

Timotej Lazar, Martin Možina, and Ivan Bratko. Automatic extraction of ast patterns for debugging student programs. In *Artificial Intelligence in Education: 18th International Conference, AIED 2017, Wuhan, China, June 28–July 1, 2017, Proceedings 18*, pp. 162–174. Springer, 2017.

S. Lee, B. Zoph, A. Kolesnikov, A. Joulin, D. Yarats, J-B. Alayrac, M. Malinowski, and S. Kornblith. Deduplicating training data makes language models better. *arXiv preprint arXiv:2210.03381*, 2022.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33: 9459–9474, 2020a.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33: 9459–9474, 2020b.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL `https://openreview.net/forum?id=o2HBfgY20b`.

Qinbin Li, Junyuan Hong, Chulin Xie, Jeffrey Tan, Rachel Xin, Junyi Hou, Xavier Yin, Zhun Wang, Dan Hendrycks, Zhangyang Wang, et al. Llm-pbe: Assessing data privacy in large language models. *arXiv preprint arXiv:2408.12787*, 2024a.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Yichun Yin, Hao Zhang, Yong Liu, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models. *arXiv preprint arXiv:2407.02883*, 2024b.

Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. Opportunities and challenges in code search tools. *ACM Computing Surveys (CSUR)*, 54(9):1–40, 2021.

Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. ReACC: A retrieval-augmented code completion framework. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6227–6240, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.431. URL https://aclanthology.org/2022.acl-long.431/.

Kun Luo, Zheng Liu, Shitao Xiao, Tong Zhou, Yubo Chen, Jun Zhao, and Kang Liu. Landmark embedding: A chunking-free embedding method for retrieval augmented long-context large language models. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 3268–3281, Bangkok, Thailand, August 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.180. URL https://aclanthology.org/2024.acl-long.180/.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations*, 2024b. URL https://openreview.net/forum?id=UnUwSIgK5W.

Micah Musser. A cost analysis of generative language models and influence operations. *arXiv preprint arXiv:2308.03740*, 2023.

Erik Nijkamp, Bo Pang Lee, Shang-Wen Tu, Quoc Le, Ruslan Salakhutdinov, and Shuangfei Zhai. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive APIs. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tBRNC6YemY.

Anupam Purwar et al. Evaluating the efficacy of open-source llms in enterprise-specific rag systems: A comparative study of performance and scalability. *arXiv preprint arXiv:2406.11424*, 2024.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Scaling large language model-based multi-agent collaboration. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=K3n5jPkrU6.

Keshav Rangan and Yiqiao Yin. A fine-tuning enhanced rag system with quantized influence measure as ai judge. *Scientific Reports*, 14, 11 2024. doi: 10.1038/s41598-024-79110-x.

James Requeima, John F Bronskill, Dami Choi, Richard E. Turner, and David Duvenaud. LLM processes: Numerical predictive distributions conditioned on natural language. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=HShs7q1Njh.

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

Caitlin Sadowski, Kathryn T Stolee, and Sebastian Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pp. 191–201, 2015.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Paul Smolensky. Connectionist ai, symbolic ai, and the brain. *Artificial Intelligence Review*, 1(2): 95–109, 1987.

Aivin V Solatorio. Gistembed: Guided in-sample selection of training negatives for text embedding fine-tuning. *arXiv preprint arXiv:2402.16829*, 2024.

Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems*, 33: 16857–16867, 2020.

Hongjin Su, Weijia Shi, Jungo Kasai, Yizhong Wang, Yushi Hu, Mari Ostendorf, Wen-tau Yih, Noah A. Smith, Luke Zettlemoyer, and Tao Yu. One embedder, any task: Instruction-finetuned text embeddings. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 1102–1121, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-acl.71. URL `https://aclanthology.org/2023.findings-acl.71/`.

Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. Arks: Active retrieval in knowledge soup for code generation. *arXiv preprint arXiv:2402.12317*, 2024.

Chaofan Tao, Qian Liu, Longxu Dou, Niklas Muennighoff, Zhongwei Wan, Ping Luo, Min Lin, and Ngai Wong. Scaling laws with vocabulary: Larger models deserve larger vocabularies. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL `https://openreview.net/forum?id=sKCKPr8cRL`.

Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, 2016.

Xingyao Wang, Tianyu Lan, Jie Yao, Di He, and Tie-Yan Liu. Chunking and hierarchical retrieval for long document understanding. *arXiv preprint arXiv:2304.03287*, 2023a.

Xuezhi Wang, Yao Kordi, Swaroop Mishra, Denny Yang, Matthew Lamm, Hongrae Wang, Sharan Patel, Jiahui Zhou, Sameer Singh, and Oren Etzioni. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2023b.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021a. URL `https://arxiv.org/abs/2109.00859`.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 8696–8708, Online and Punta Cana, Dominican Republic, November 2021b. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.685. URL `https://aclanthology.org/2021.emnlp-main.685/`.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023c. URL `https://openreview.net/forum?id=wKqdk1sOMY`.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. Coderag-bench: Can retrieval augment code generation? *arXiv preprint arXiv:2406.14497*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers, 2024. URL `https://arxiv.org/abs/2309.03409`.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=WE_vluYUL-X`.

Wang Yining, Wang Liwei, Li Yuanzhi, He Di, Chen Wei, and Liu Tie-Yan. A theoretical analysis of ndcg ranking measures. In *Proceedings of the 26th annual conference on learning theory*, 2013.

Jiayi Yuan, Ruixiang Tang, Xiaoqian Jiang, and Xia Hu. Llm for patient-trial matching: Privacy-aware data augmentation towards better performance and generalizability. In *American Medical Informatics Association (AMIA) Annual Symposium*, 2023.

Shenglai Zeng, Jiankun Zhang, Pengfei He, Yue Xing, Yiding Liu, Han Xu, Jie Ren, Shuaiqiang Wang, Dawei Yin, Yi Chang, et al. The good and the bad: Exploring privacy issues in retrieval-augmented generation (rag). *arXiv preprint arXiv:2402.16893*, 2024.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023a. URL `https://openreview.net/forum?id=q09vTY1Cqh`.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=z5uVAKwmjf`.

Tianjun Zhang, Shishir G Patil, Naman Jain, Sheng Shen, Matei Zaharia, Ion Stoica, and Joseph E. Gonzalez. RAFT: Adapting language model to domain specific RAG. In *First Conference on Language Modeling*, 2024a. URL `https://openreview.net/forum?id=rzQGHXNReU`.

Yifan Zhang, Yang Yuan, and Andrew Chi-Chih Yao. Meta prompting for ai systems, 2024b. URL `https://arxiv.org/abs/2311.11482`.

Ziyin Zhang, Chaoyu Chen, Bingchang Liu, Cong Liao, Zi Gong, Hang Yu, Jianguo Li, and Rui Wang. Unifying the perspectives of nlp and software engineering: A survey on language models for code. *arXiv preprint arXiv:2311.07989*, 2023b.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 5673–5684, 2023.

A Appendix

A.1 Example of Code2JSON

Another example is provided in Table 5, where we show the full feature set extracted by Code2JSON, including the summary, function, class, and data descriptions.

A.2 Results for Unstructured Code Data from Git Repositories

Figure 4 shows schematic architecture of the Code2JSON-assisted code RAG pipeline on Git repositories. Algorithm 1 shows the chunking strategy used.

The real-world raw codebases (e.g., HuggingFace's Transformers repository), however, are not structured with no distinct code or text fields. To resolve this, we use Algorithm 1 which converts an unstructured raw code body into manageable code chunks. Algorithm 1 leverages AST with the tree-sitter library (tre) and depth-first tree traversal.

For the evaluation of Code2JSON on unstructured code data sources, we use Transformers `v4.42.3` library (tra), `commit#` 2033 for the Peft library (pef), `commit#` 9330 of the Diffusers

Table 4: Examples outputs from CODE2JSON on code samples from MBPP dataset with Llama-3-70B as the extraction LLM.

Raw Code	**Natual Language Feature Set**
`def is_upper(string):` `  return string.upper()`	This code defines a function that takes a string as input and returns the uppercase version of that string.
`def check_permutation(str1, str2):` `  n1=len(str1)` `  n2=len(str2)` `  if(n1!=n2):` `    return False` `  a=sorted(str1)` `  str1=\" \".join(a)` `  b=sorted(str2)` `  str2=\" \".join(b)` `  for i in range(0, n1, 1):` `    if(str1[i] != str2[i]):` `      return False` `  return True`	The code defines a function to check if two input strings are permutations of each other. It first checks if the strings are of equal length, and if not, returns False. If they are of equal length, it sorts the characters in each string and then compares the sorted strings character by character. If all characters match, it returns True, indicating the strings are permutations of each other, otherwise, it returns False.
`import re` `def match_num(string):` `  text = re.compile(r\"^5\")` `  if text.match(string):` `    return True` `  else:` `    return False`	This code defines a function that checks if a given string starts with the digit "5" using regular expression. The function takes a string as input, compiles a regular expression pattern that matches the digit "5" at the start of the string, and returns True if the string matches the pattern, and False otherwise.

Algorithm 1 Depth-First Search (DFS)-based Chunking on AST

```
1: Input: Raw unstructured code, $\mathcal{C}$, and chunk size threshold, $\xi$
2: Output: Code chunks, $c_i$
3: Build Abstract Syntax Tree (AST) from Raw Code, $\mathcal{C}$
4: Traverse the AST using DFS to identify semantic components (e.g., classes)
5: for each node in the AST do
6:     Identify code segments that belong together based on structure and hierarchy.
7:     Aggregate related nodes to form chunks, $c_i$, of size $\leq \xi$
8: end for
```

(dif) library, and commit# 1952 of the TRL (trl) library. Files with extensions .csv, .tsv, .png, and .ipynb were ignored. We use Tree-sitter library (tre) for chunking using Algorithm 1, BM25 for retrieval, and Llama3-8B-Instruct with temperature 0.1 for generation. For the baseline, we directly indexed the code chunked generated by the Tree-sitter library.

For evaluating CODE2JSON on unstructured code data, we curated a dataset of 486 questions: {255 from Transformers, 100 from Diffusers, 98 from TRL , 33 from Peft}, with the help of authoritative domain experts. We examine the RAG performance with CODE2JSON and without CODE2JSON, and also with zero shot inference. Figure 5 shows how CODE2JSON stacks up against a vanilla approach without CODE2JSON and zero-shot learning. It is remarkable that even with BM25-based indexing and Llama3-8B-Instruct (BF16), CODE2JSON performs on par with the default approach. In popular libraries such as Transformers and Diffusers, zero shot learning with Llama3-8B-Instruct (BF16) outperforms both RAG approaches. This can be attributed to training data leakage (Li et al.,

Table 5: Example outputs from CODE2JSON on code samples from the Transformers dataset with Llama-3-70B as the extraction LLM.

Feature	**Details**

Raw Code	`from ..utils import DummyObject,`
	`requires_backends`
	
	`SLOW_TO_FAST_CONVERTERS = None`
	
	`def convert_slow_tokenizer(*args, **kwargs):`
	`    requires_backends(convert_slow_tokenizer,`
	`    ["sentencepiece", "tokenizers"])`

Summary Feature	This code appears to be part of a larger project and seems to be auto-generated. It imports some utility functions and defines a converter function for a specific type of tokenizer, which requires certain back-end dependencies to be present.
Function Description	`convert_slow_tokenizer` - Converts a slow tokenizer to a fast tokenizer using SentencePiece and Tokenizers backends.
Class Description	`DummyObject` - A utility class used to create dummy objects, likely used as placeholders or for testing purposes.
Data Description	`SLOW_TO_FAST_CONVERTERS` - A converter for slow tokenizers, currently set to `None`. `DummyObject` - A dummy object used in the code. `requires_backends` - A function to check if required backends are available.

2024a). We expect CODE2JSON to perform better with Text Embedding models such as MPNET or Code Embedding models such as CodeT5-small.

A.3 DETAILED RAG RESULTS

The full results for retrieval using CODE2JSON on different datasets using different retrieval strategies is provided in Table 6. We present these results as an ordered pair 'baseline/CODE2JSON', with green cells indicating CODE2JSON's superiority and red cells indicating the baseline's superiority.

Table 6: Comparative retrieval performance (NDCG@10) between CODE2JSON and a baseline (CodeRAG-Bench pipeline). The NDCG@10 scores are displayed as 'baseline/CODE2JSON', with green cells indicating CODE2JSON's superiority and red cells indicating the baseline's superiority.

	Python						**Non-Python**				
	HumanEval	**MBPP**	**DS-1000**	**ODEX**	**CodeFeedback-MT**	**CodeTrans-Contest**	**CSN(Ruby)**	**HumanEval-X(cpp)**	**HumanEval-X(go)**	**HumanEval-X(java)**	**HumanEval-X(js)**
	(164)	(964)	(34K)	(34K)	(3319)	(1008)	(53K)	(164)	(164)	(164)	(164)
						Keyword Matching					
BM25	100.0 / 81.39	3.67 / 47.89	5.21 / 2.42	6.7 / 5.2	73.69 / 56.40	30.59 / 11.57	5.56 / 17.45	69.77 / 57.72	46.22 / 52.94	45.31 / 58.88	53.65 / 57.49
						Text Embedding					
BGE-base	99.78 / 99.1	76.45 / 82.74	25.99 / 26.13	22.06 / 24.57	58.84 / 59.25	64.28 / 65.06	64.63 / 59.24	97.8 / 90.12	95.33 / 89.01	97.92 / 91.85	97.82 / 89.96
BGE-large	100.0 / 100.0	78.32 / 83.31	25.28 / 26.98	21.87 / 25.69	61.70 / 61.62	69.07 / 71.47	65.61 / 62.19	97.82 / 91.97	97.98 / 91.15	97.59 / 90.08	98.65 / 92.91
GIST-base	100.0 / 99.33	78.37 / 84.35	27.29 / 29.13	26.8 / 26.78	60.78 / 62.15	67.19 / 70.77	64.81 / 60.05	97.82 / 91.04	96.33 / 91.88	98.22 / 95.36	96.94 / 91.07
GIST-large	100.0 / 99.55	79.08 / 85.44	30.17 / 29.59	28.08 / 27.51	61.75 / 61.80	69.30 / 72.83	65.19 / 62.33	97.86 / 92.68	97.85 / 92.91	98.27 / 94.35	96.33 / 93.37
MPNET	100.0 / 99.78	78.72 / 81.75	24.36 / 28.69	21.83 / 24.30	62.73 / 57.9	70.57 / 74.15	52.24 / 49.38	94.44 / 90.19	98.35 / 93.45	97.51 / 96.61	93.04 / 94.37
						Code Embedding					
Instructor	100.0 / 99.78	73.73 / 79.35	16.23 / 20.4	16.94 / 22.22	56.02 / 55.94	69.30 / 70.55	54.21 / 50.42	97.33 / 89.92	96.98 / 86.63	96.06 / 85.5	93.02 / 86.83
CodeBert	9.6 / 5.59	0.96 / 1.38	0.0 / 0.0	0.0 / 0.0	0.16 / 0.26	0.15 / 0.15	0.04 / 0.01	3.42 / 4.3	5.2 / 4.79	3.26 / 4.91	3.82 / 4.01
CodeT5-small	30.47 / 9.3	8.12 / 6.49	0.06 / 0.0	1.64 / 2.36	0.85 / 0.23	2.88 / 2.44	0.74 / 0.98	7.01 / 7.68	6.61 / 7.2	6.26 / 6.41	5.5 / 10.7

We largely see a superior performance for CODE2JSON on Python datasets. However, when using text-embedding models, the performance takes a hit for non-Python datasets. While still compara-

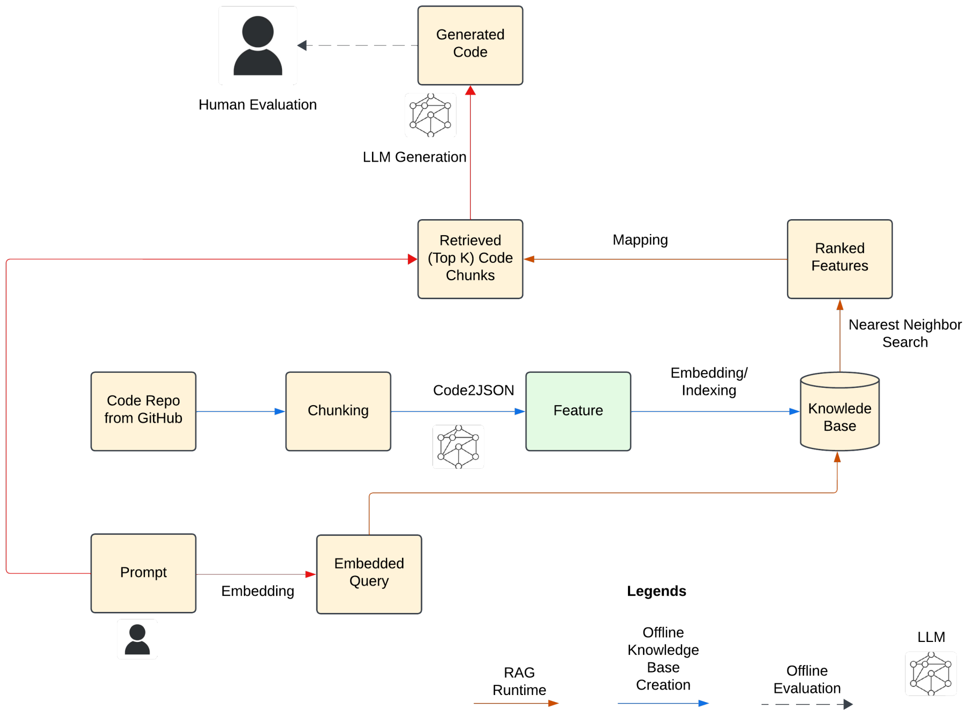


Figure 4: Schematic architecture of the CODE2JSON-assisted code RAG pipeline on Git repositories. The pipeline consists of three workflows. First, offline knowledge base creation: it processes the code repo by chunking it to small fragments and then converting them into natural language (NL) features using the CODE2JSON prompt template, as illustrated in Figure 2. These NL features are subsequently embedded and indexed into a knowledge base, making them readily accessible during the code RAG runtime. Second, code RAG runtime: Each NL prompt from a user is embedded and matched with the Top-K relevant features in the knowledge base (summary, function, class, data descriptions). The ranked features serve as the context in RAG pipeline for generation. Finally evaluation for each prompt: we measure the accuracy of the generated code as a response to the NL prompt using a human-in-the-loop.

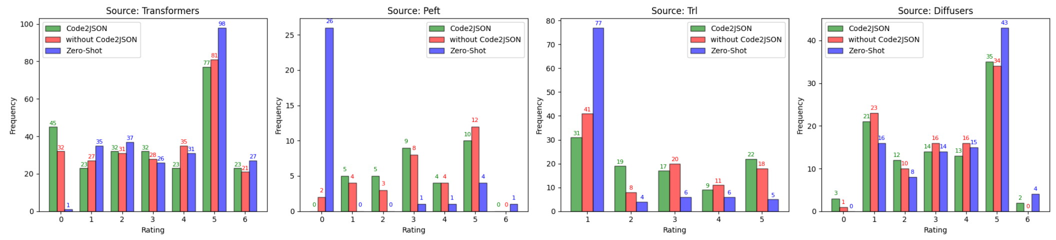


Figure 5: Distribution of ratings (0–6) across different sources: Transformers (N=255), PEFT (N=33), TRL (N=98), and Diffusers (N=100). The figure illustrates the frequency of ratings for code RAG with CODE2JSON (green), code RAG without CODE2JSON (red), and zero-shot inference (blue) for each source.

ble, this highlights the inability of text-embedding models to capture rich semantic information in code data. This pattern could be attributed to the relatively lower representation of programming languages other than Python in the training data used for LLMs.

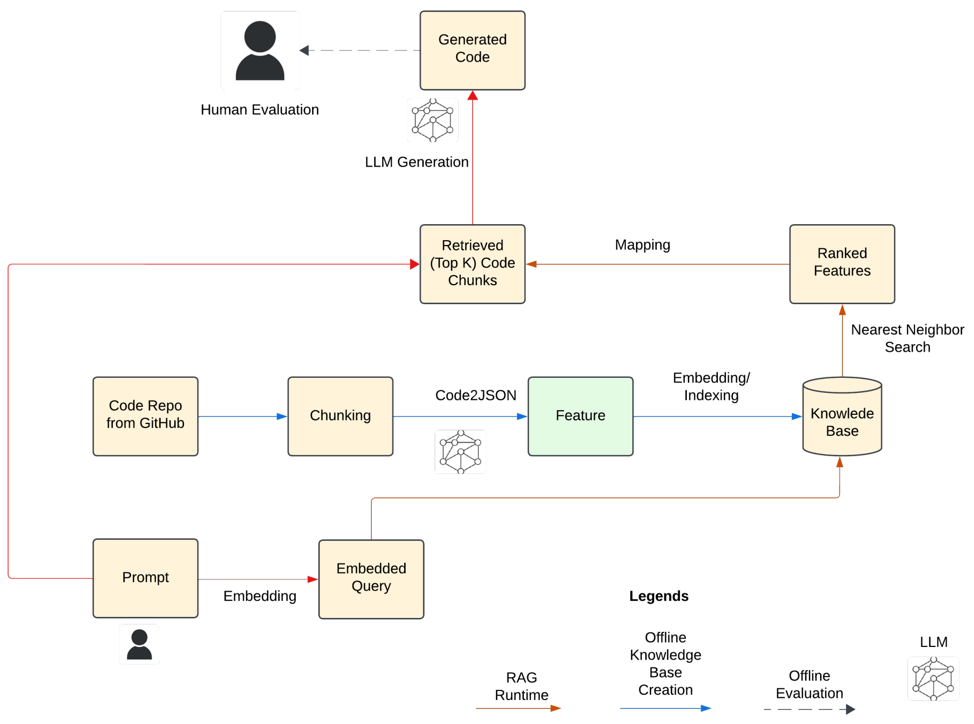


Figure 6: Schematic architecture of the CODE2JSON-assisted code RAG pipeline on Git repositories. The pipeline consists of three workflows. First, offline knowledge base creation: it processes the code repo by chunking it to small fragments and then converting them into natural language (NL) features using the CODE2JSON prompt template, as illustrated in Figure 2. These NL features are subsequently embedded and indexed into a knowledge base, making them readily accessible during the code RAG runtime. Second, code RAG runtime: Each NL prompt from a user is embedded and matched with the Top-K relevant features in the knowledge base. The ranked features retrieve the corresponding Top-K code chunks which serve as the context in RAG pipeline. The retrieved context together with prompt generate response with a zero shot prompt as shown in Figure 8.

Detailed results for code generation after retrieval are also presented in Figure 7. We show the results as an ordered pair 'baseline/CODE2JSON'. For code generation, we use minimal prompting, since the data is already in a *query-response* format. A simple one line instruction directing the LLM to generate the code based on given context is used.

A.4 ABLATION STUDY USING DIFFERENT EXTRACTION LLMS ON PYTHON BENCHMARK DATASETS

The retrieval performance (NDCG@10) of CODE2JSON, when evaluated with two distinct feature extraction LLMs—Llama-3-70b-instruct-awq-4bit and Llama-3-8B-Instruct, both with an 8K context length—demonstrates comparable effectiveness, as shown in Figure 7. This suggests that increasing the model size at the expense of numerical precision does not necessarily translate into better performance.

A.5 PROMPT TEMPLATE FOR CODE GENERATION

Figure 8 shows the prompt template used for the generation of code for hand-made queries for unstructured code repositories.

Table 7: Generation performance (**pass@1**). The zero-shot performance (highlighted in yellow) is obtained by prompting the LLM with just the question without any retrieval. The pass@1 scores are displayed as 'baseline/CODE2JSON', with green cells indicating CODE2JSON's superiority and red cells indicating the baseline's superiority. Best retrieval performance for each dataset are in bold.

Method	HumanEval	MBPP	DS-1000	ODEX
DeepSeekCoder-7B-Instruct				
Zero-shot	25.00	62.8	0.0	**41.46**
BM25	59.76 / 64.63	63.6 / 63.4	7.3 / 21.1	33.49 / 35.76
GIST-Large	63.41 / 53.05	**80.2** / 66.8	11.4 / 21.3	35.54 / 37.59
MPNET	61.59 / **70.12**	77.0 / 64.0	14.3 / **21.5**	35.31 / 38.95
Llama-3-8B-Instruct				
Zero-shot	48.78	40.0	0.0	4.56
BM25	65.85 / **68.90**	28.8 / 41.0	1.9 / 2.5	**5.01** / **5.01**
GIST-Large	68.29 / 57.93	28.4 / **42.8**	1.5 /**2.8**	4.78 / 4.78
MPNET	66.46 / 47.56	29.4 / 41.6	1.7 / 2.2	4.78 / 4.56
Phi-2				
Zero-shot	41.46	0.4	0.0	0.0
BM25	11.59 / **51.22**	12.4 / 0.4	3.8 / 3.8	0.0 / 0.0
GIST-Large	14.02 / 45.12	**66.2** / 0.4	3.2 / 3.6	0.0 / 0.0
MPNET	13.41 / 40.85	64.6 / 0.2	3.8 / **4.5**	**0.23** / 0.0
Win percentage	66.7%	33.3%	88.9%	33.3%

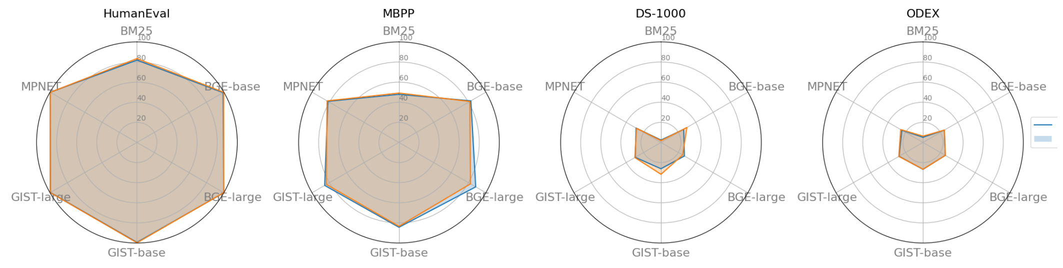
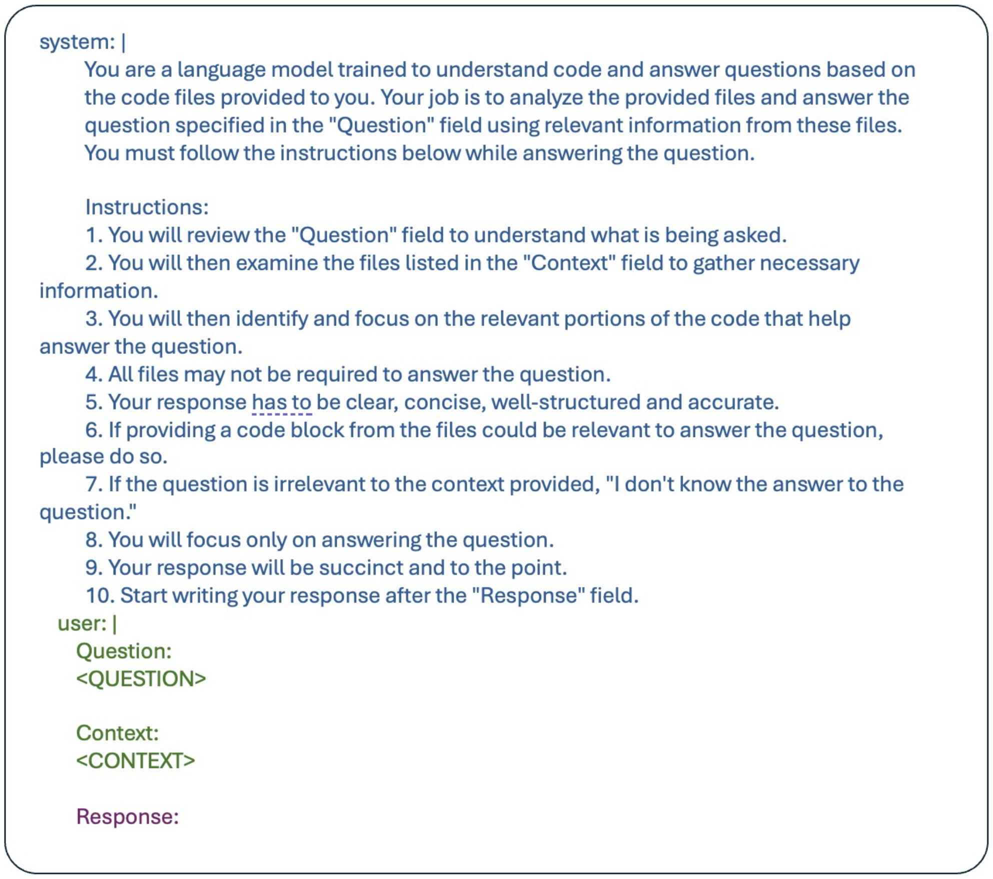


Figure 7: Ablation study comparing extraction models Llama-3-70b-instruct-awq-4bit and Llama-3-8B-Instruct across four datasets: HumanEval, MBPP, DS-1000, and ODEX. Each radar plot displays NDCG@10 values across six indexing strategies—BM25, BGE-Base, BGE-Large, GIST-Base, GIST-Large, and MPNET (clockwise). Results demonstrate comparable performance between the LLMs.

A.6 SUPERVISED FINE TUNING

SFT (Supervised Fine-tuning) is used to adjust a model's parameters to improve its performance on specific downstream tasks. Recently, there has been work towards comparing ((Gupta et al., 2024; Alghisi et al., 2024)) and even combining ((Zhang et al., 2024a; Rangan & Yin, 2024)) RAG and fine-tuning approaches for similar tasks and evaluating their usefulness and efficiency.

To further validate the quality of representations generated by LLMs, we perform experiments by finetuning smaller LLMs (<10B parameters) in a supervised manner. Our focus here is to study the impact when we use LLM generated features to fine tune another LLM, in isolation to RAG. Concretely, we compare two fine tuning approaches, (i) Fine-tuning with the raw question-answer pairs, and (ii) Fine-tuning with LLM generated descriptions as additional context for the question-answer pair. For (i), we simply fine tune the LLM to generate the answer code snippet in response to the question provided in the dataset. For (ii), the LLM-generated description of the code snippet is used as additional context in addition to the question. During inference, in both the cases, we only provide the question without the context and evaluate the quality of the generated output using pass@1 ((Chen et al., 2021b)) score, to test the code on functional effectiveness. To optimize for



```
system: |
        You are a language model trained to understand code and answer questions based on
        the code files provided to you. Your job is to analyze the provided files and answer the
        question specified in the "Question" field using relevant information from these files.
        You must follow the instructions below while answering the question.

        Instructions:
        1. You will review the "Question" field to understand what is being asked.
        2. You will then examine the files listed in the "Context" field to gather necessary
information.
        3. You will then identify and focus on the relevant portions of the code that help
answer the question.
        4. All files may not be required to answer the question.
        5. Your response has to be clear, concise, well-structured and accurate.
        6. If providing a code block from the files could be relevant to answer the question,
please do so.
        7. If the question is irrelevant to the context provided, "I don't know the answer to the
question."
        8. You will focus only on answering the question.
        9. Your response will be succinct and to the point.
        10. Start writing your response after the "Response" field.
   user: |
     Question:
     <QUESTION>

     Context:
     <CONTEXT>

     Response:
```

Figure 8: Prompt template (in yaml) used for downstream code generation.

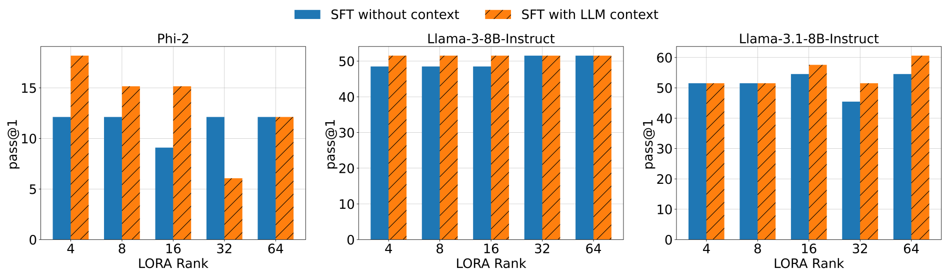
GPU resources, and considering the size of the datasets used for our SFT, we use LoRA ((Hu et al., 2022)), and perform ablations with different values of the $rank$ variable.

The fine tuning experiments were done on an NVIDIA-A100 GPU (Ubuntu 22.04) with 40GB GPU RAM and Intel(R) Xeon(R) Gold 6354 CPU @ 3.00GHz chips with 125GB GPU RAM. The results are shown in Figure 9. The $alpha$ value used was 16. We also used a lora dropout of 0.05, learning rate 5e-5, weight decay 5e-2, and 3 epochs as hyperparamters. Detailed configuration settings can be found in our code.

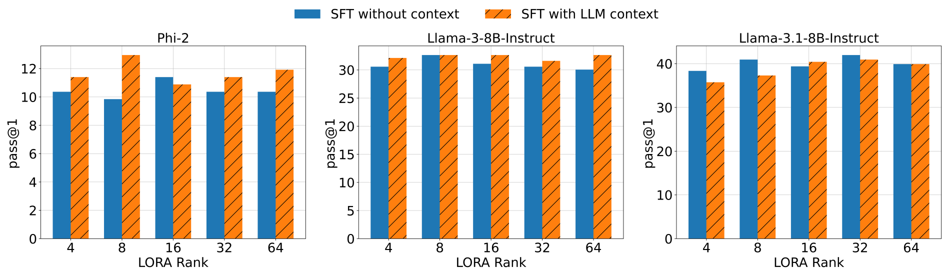
As highlighted in Figure 9, fine-tuning with additional LLM-generated context can help during inference, even when a similar context is not available for test data. Although the increase in pass@1 score is not as significant for all cases, and even drops occasionally, we argue that this experiment shows the potential of LLM-generated context in improving fine-tuning procedures and perhaps alignment as well.

A.7 Cost analysis

This section provides a detailed analysis of the costs associated with various experiments for feature extraction using an LLM. We do not discuss costs associated with retrieval, generation, or fine-tuning here, since we believe those are widely discussed in literature ((Purwar et al., 2024; Musser, 2023; Laban et al., 2024)). Instead our focus is to provide estimate costs for processsing a dataset with an LLM, to make it easier for researchers and developers to plan and budget any work they may wish to reproduce with our methodology.



(a) Humaneval



(b) MBPP

Figure 9: pass@1 score after SFT with LoRA on different datasets.

Our first analysis is done for Llama3-8B-Instruct as the LLM used for feature extraction. We hosted the model on a g5.4xlarge instance with 24GB GPU RAM, with the help of a vllm server, and batching for upto 4 sequences. On the same machine which calls the LLM server and stores the results, we use multiprocessing to retrieve results in parallel. Concretely, we use 16 jobs (number of CPUs) for best results. We compare 3 datasets, namely Humaneval (Python version), MBPP, and ODEX for which we used an open retrieval corpus (Wang et al., 2024). The summary for the experiments is given in Table 8

We consider the storage and GPU cost for determining the overall cost for each experiment. The network cost is negligible since our requests do not exceed the IOPS threshold (3000) of the instance, for any experiments. We provision 1000GB EBS gp3 volume which costs $80/month. The GPU cost for g5.4xlarge at the time of the experiments was $0.974/hr. We consider the wall time to account towards the volume and GPU costs. We summarize the costs in Table 9.

Table 8: Llama3-8B-Instruct feature extraction logs

Dataset (size)	**Time(s)**	**Total input tokens**	**Total generated tokens**
HumanEval (164)	86.1	35.3K	12.9K
MBPP (964)	470.1	85.7K	69.6K
ODEX (34003)	23.7K	11.4M	3.0M

Table 9: Llama3-8B-Instruct feature extraction costs

Dataset (size)	**Total cost($)**	**Cost (1M tokens)($)**
HumanEval (164)	0.03	0.54
MBPP (964)	0.14	0.91
ODEX (34003)	7.14	0.5

We also give costs for running a Llama3-70B-Instruct-awq model hosted on a g5.12xlarge instance with 96GB GPU RAM, with the help of a vllm server, batching for upto 4 sequences, and max GPU memory utilization of 0.8. We compare Humaneval, MBPP, and CodeSearchNet (ruby split) for these costs. The summary for these costs is given in Table 10.

For per token cost, we consider the prompt + generation token count. The higher per-token cost for MBPP can be attributed to higher generation tokens to prompt tokens ratio compared to other datasets.

Table 10: Llama3-70B-Instruct-awq feature costs

Dataset (size)	**Time(s)**	**Input / Output tokens**	**Total Cost($)**	**Cost (1M tokens)($)**
HumanEval (164)	86.11	35.3K / 12.8K	0.23	4.78
MBPP (964)	470.14	85.7K / 63.2K	0.93	6.24
CSN (53270)	72486.08	7.6M / 6.4M	69.66	5.70

Apart from computational complexity, we also analyze space requirements when creating an index on raw code data and featurized descriptions. In Table 11, it is clear that a feature index is more efficient that a raw index for HumanEval, DS-1000, and ODEX, when using BM25.

Table 11: BM25 index sizes for various datasets using raw code/features as documents

Dataset	**Raw Code Index size**	**Feature Index size**
HumanEval	248KB	176KB
MBPP	396KB	584KB
DS-1000	72MB	22MB
ODEX	72MB	22MB