CLARC: C/C++ BENCHMARK FOR ROBUST CODE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Effective retrieval of code snippets from natural language queries is essential for code reuse and developer productivity. However, current benchmarks are limited: they predominantly focus on Python, lack support for industry-focused languages like C/C++, miss structured categorization, and are susceptible to models that exploit superficial lexical features instead of code semantics. To address these limitations, we introduce CLARC (**C/C++ LA**nguage **R**etrieval with Anonymized **C**ode), a benchmark of 1,245 C/C++ query-code pairs that is fully compilable, configurable, and extensible. CLARC systematically categorizes snippets into three groups based on dependency complexity, allowing for a nuanced evaluation of retrieval performance under varying levels of code complexity. CLARC also provides configurable settings, including anonymized identifiers and low-level representations, to evaluate model robustness across different levels of code context and abstraction. Evaluation of six state-of-the-art code search methods shows dramatic performance drops under identifier anonymization, exposing existing models' persistent reliance on superficial cues. Their poor performance on low-level languages such as Assembly and WebAssembly further reveals limited effectiveness beyond high-level programming languages. We also introduce an automated pipeline for scalable benchmark generation, validated through hypothesis tests, enabling the efficient creation of high-quality code search datasets that can be reused by other dataset builders. Our dataset is publicly available at `https://huggingface.co/datasets/ClarcTeam/CLARC`.

1 INTRODUCTION

As the computer science community expands rapidly, its reliance on automated systems for code analysis also grows. Efficiently understanding, categorizing, and retrieving code is becoming indispensable due to the increasing size and complexity of public codebases (Shekhar, 2024). Code search aims to retrieve the code snippet that best aligns with a natural language query, thus fostering code reuse and improving developers' efficiency (Di Grazia and Pradel, 2023; Sun et al., 2024). The current code search models achieve this by projecting the query and code snippet into the same vector space, followed by similarity metrics to rank the candidate snippets.

Although recent auto-regressive Large Language Models (LLMs) and Code Language Models (CLMs) show strong reasoning capabilities for tasks such as code completion and vulnerability detection (Jiang et al., 2024; Rozière et al., 2024; Hui et al., 2024; Chen et al., 2021), their direct application to large-scale code search is challenging. Searching extensive repositories with thousands or millions of candidates (Potvin and Levenberg, 2016; Benram, 2024; Howell et al., 2023) demands methods for compact feature storage and efficient reranking (Di Grazia and Pradel, 2023; Liu et al., 2021). Consequently, embedding-based retrieval models, which map code snippets to the dense vector space for efficient storage and similarity calculation, are more practical. Furthermore, effective code retrieval can also improve LLM's performance in generation tasks through Retrieval-Augmented Generation (RAG) (Chen et al., 2024a; Wang et al., 2025; Zhao et al., 2024).

Various benchmarks and datasets have been developed to evaluate their efficacy (Husain et al., 2020; Khan et al., 2024; Huang et al., 2021; Lu et al., 2021; Liu et al., 2024a; Li et al., 2025; 2024; Yao et al., 2018; Heyman and Cutsem, 2020; Yin et al., 2018). However, existing code search benchmarks suffer from limitations that undermine their practical utility. First, most current datasets (Huang et al., 2021; Husain et al., 2020; Li et al., 2025; Lu et al., 2021; Li et al., 2024; Yao et al., 2018; Heyman and Cutsem, 2020; Yin et al., 2018) prioritize research-favored languages like Python, while systematically neglecting text-to-code tasks of industrially prevalent languages such as C/C++ (Twist et al., 2025) or failing to collect samples from real-world projects (Khan et al., 2024). This imbalance restricts the application of research findings to real-world software development scenarios. Second, many code snippets in current benchmarks lack compilability, often due to missing include/import statements or necessary helper functions/classes (Cao et al., 2025). This contrasts with the professional human developer practice, where inspecting helper functions and dependencies is crucial for validating a piece of code against query requirements. Furthermore, existing benchmarks, even those that contain C/C++ text-to-code tasks (Liu et al., 2024a; Khan et al., 2024), fail to evaluate the impact of superficial textual features (such as variable and function names) on models (Chen et al., 2024b; Qu et al., 2024). Consequently, it is unclear whether high benchmark scores are based on genuine code comprehension or superficial pattern recognition of textual features.

To address the identified limitations of current code-search benchmarks, we introduce CLARC (**C/C++ LA**nguage **R**etrieval with Anonymized **C**ode), a comprehensive benchmark comprising 1,245 query-code snippet pairs in C/C++. These code snippets, sourced from popular GitHub repositories, are all compilable within a standardized environment. The snippets are then categorized based on their dependency, allowing for a nuanced evaluation of how models handle varying levels of code complexity and contextual information. Moreover, CLARC provides distinct settings that anonymize code identifiers or use snippets compiled into low-level languages such as Assembly or WebAssembly (Wasm). These settings are specifically designed to analyze how textual identifiers impact retrieval accuracy and to assess a model's adaptability in understanding and retrieving code across different levels of abstraction, including low-level languages.

On CLARC, we evaluated six diverse code search methods, including two black-box systems, a lightweight encoder model, a robustness-focused model fine-tuned with augmented data, and a model adapted from large-scale CLMs. In experiments, we find that the retrieval metrics drop on all models under the settings when original identifiers are replaced by less meaningful names. This suggests that the state-of-the-art code search models still rely on superficial features within the code snippets rather than code semantics. We also observe that models perform poorly when code is compiled to Assembly or WebAssembly, indicating their limited capability on low-level language representations.

Besides the evaluation results, to enable scalable benchmark generation and minimize the influence of knowledge contamination, we also propose an innovative pipeline for the automated generation of code search benchmarks. The pipeline systematically extracts code snippets from various sources and then utilizes LLMs to generate corresponding natural language descriptions, which serve as queries. The quality of LLM-generated queries is ensured through statistical validation. Our automated approach facilitates the scalable and cost-effective expansion of benchmark datasets, paving the way for more extensive and varied evaluations of code search models.

In summary, our main contributions of this work are:

- introducing CLARC, a C/C++ benchmark of 1,245 fully compilable query-snippet pairs with various settings, to rigorously evaluate retrieval performance and model robustness across varying levels of complexity, context, and abstraction;
- providing empirical evidence that current code search models' overreliance on non-functional features and the large performance disparity between high and low-level programming languages; and
- designing an automated pipeline for scalable benchmark generation, validated through rigorous hypothesis testing, enabling efficient creation of diverse, high-quality evaluation resources that can be reused by other dataset builders.

2 Related Works

Code Search Models. Code retrieval has become a critical component of software engineering in terms of efficient development and code quality improvement (Li et al., 2025). Like general dense retrieval models (Karpukhin et al., 2020; Izacard et al., 2022; Wang et al., 2024a; Li et al., 2023; Xiao et al., 2024; Bai et al., 2024; Wang et al., 2024b), modern code search models encode the code and queries as embeddings and calculate their similarities. Popular code models, such as

CodeBERT (Feng et al., 2020), UniXcoder (Guo et al., 2022), and CodeT5+ (Wang et al., 2023b), have demonstrated significant utility in code search tasks. Subsequently, recent studies have improved the quality of code embedding for retrieval in several directions (Liu et al., 2024b; Gao et al., 2025; Gurioli et al., 2025; Zhang et al., 2024; Nomic Team, 2025; Voyage AI, 2024; OpenAI, 2024). CodeXEmbed (Liu et al., 2024b) proposes a generalizable training approach for code embedding that converts multiple code-related tasks into retrieval tasks. OASIS (Gao et al., 2025) leverages order-based similarity labels to capture semantic nuances. Nomic-emb-code (Nomic Team, 2025) utilizes the CoRNStack dataset (Suresh et al., 2025) and a curriculum-based hard negative mining strategy to boost the model's performance. Closed-source code search models, such as voyage-code-3 (Voyage AI, 2024) and Open-AI-text-embedding (OpenAI, 2024), also show outstanding results on code retrieval tasks.

Code Search Benchmarks. Numerous benchmarks have been developed to evaluate code search models (Husain et al., 2020; Khan et al., 2024; Huang et al., 2021; Lu et al., 2021; Liu et al., 2024a; Li et al., 2025; 2024; Yao et al., 2018; Heyman and Cutsem, 2020; Yin et al., 2018). CodeSearchNet challenge (Husain et al., 2020) established an extensive multilingual dataset for semantic code search, while XCodeEval (Khan et al., 2024) built a large executable multilingual benchmark. CoSQA (Huang et al., 2021) and CodeXGLUE (Lu et al., 2021) incorporated real-world user queries, RepoQA (Liu et al., 2024a) focused on understanding long-context code, and COIR (Li et al., 2025) introduced more diverse retrieval tasks and domains. However, these benchmarks have limitations regarding the C/C++ code search. Some neglect C/C++ samples for text-to-code retrieval (Huang et al., 2021; Husain et al., 2020; Li et al., 2025; Lu et al., 2021; Li et al., 2024; Yao et al., 2018; Heyman and Cutsem, 2020; Yin et al., 2018), and others, like XCodeEval (Khan et al., 2024), do not use samples from real-world projects. Furthermore, several benchmarks with C/C++ datasets, such as XCodeEval (Khan et al., 2024) and RepoQA (Liu et al., 2024a), fail to address the influence of superficial textual features. In contrast, CLARC constructs a compilable and extendable C/C++ code search benchmark from real-world GitHub repositories and more deeply evaluates code search models through code anonymization, filling a gap in existing studies.

LLMs for Benchmarks With the rapid advancement of LLMs and their remarkable capabilities, researchers have increasingly utilized these models to help build benchmarks. LLMs help constructing critical evaluation components, including natural language instructions (Zhu et al., 2024), code solutions (Ahmad et al., 2025), and test cases (Schäfer et al., 2024; Alshahwan et al., 2024). They are also applied to support the description generation (Dilgren et al., 2025) and annotation (Sghaier et al., 2025; Liu et al., 2024a; Li et al., 2024; Wang et al., 2023a) of existing datasets. In CLARC, we similarly harness LLMs' ability in code summarization to generate queries for code candidates with hypothesis testing as the validation mechanism, significantly reducing the manual effort required in the benchmark construction process and enhancing the scalability.

3 Dataset

This section details the construction of CLARC. First, C/C++ functions and their corresponding call graphs are extracted from popular GitHub repositories. These functions are then categorized into groups based on their dependencies (Sections 3.1 and 3.2). We then generate detailed descriptions for each function using LLMs (Section 3.3). These descriptions serve as the queries within the dataset, and their quality is validated through hypothesis tests (Section 3.4). Finally, we introduce different settings beyond the standard task, facilitating comprehensive evaluations on code search models' robustness (Section 3.5).

3.1 Dataset Summary

Table 1 presents the statistics for CLARC. Functions within CLARC were classified into three distinct categories based on their dependencies: Group 1 consists of functions that solely depend on whitelisted standard library functions and types; Group 2 contains functions that rely on standard library functions, but utilize custom-defined variable types; and Group 3 encompasses all functions that involve other helper functions. Figure 1 illustrates brief example functions from each group. The full examples with their corresponding queries can be found in Appendix H.

Table 1: Statistics of Datasets in CLARC Benchmark. LOC stands for lines of code; CC stands for the Cyclomatic Complexity; Src stands for the original code; Asm stands for the Assembly Code, and Wasm stands for the WebAssembly code in `.wat` format. All Code Statistics reported in the table are the average values in the corresponding category.

Category	# of Pairs	# of Tokens in Query	Code Statistics						
			# of Tokens			LOC			CC
			Src	Asm	Wasm	Src	Asm	Wasm	Src
Group 1	526	88.3	119.2	753.7	665.5	12.8	80.7	96.2	2.9
Group 2	469	84.7	137.7	831.3	947.1	13.3	84.4	134.4	2.8
Group 3	250	77.4	706.9	2272.6	967.8	71.5	212.3	138.3	5.4
Total	1245	84.8	244.2	1092.7	811.4	24.8	108.9	116.1	3.4

```
bool IsDigit(const char d) {
    return ('0' <= d)&&(d <= '9');
}
```

(a) Group 1

```
int is_set_opt_anc_info
                    (OptAnc* to, int anc) {
    if ((to->left&anc)!=0)
        return 1;
    return ((to->right&anc)!=0?1:0);
}
```

(b) Group 2

```
typedef unsigned char* string;
int scmp(string s1, string s2 ) {
    // Helper function
}
void simplesort(string a[], int n, int b) {
    int i, j; string tmp;
    for (i = 1; i < n; i++)
        for (j = i; j > 0 && scmp(a[j-1]+b,
     ↪a[j]+b) > 0; j--) {
            tmp = a[j];
            a[j] = a[j-1];
            a[j-1] = tmp;
        }
}
```

(c) Group 3

Figure 1: Example Functions of CLARC

To investigate the influence of helper functions on the Code Search Task, we designed two distinct variants: Group 3 Short and Group 3 Long. In the Group 3 Short variant, the main function and its associated helper functions are treated as separate relevant functions for retrieval. In contrast, the Group 3 Long variant merges the main function and its helper functions into a single contiguous code snippet, allowing us to evaluate retrieval performance when the main logic and its immediate functional dependencies are presented as a unified whole.

3.2 Data Collection

We constructed the CLARC dataset by crawling 45 popular C/C++ repositories on GitHub.[1] First, we established a compilation environment by creating a whitelist of all standard libraries used across these repositories. We then extracted each function along with its dependencies, including its call graph and necessary definitions. To ensure the quality of code snippets in CLARC, we only retained the functions that successfully compiled within this predefined environment. Finally, these filtered functions were categorized into three groups based on their dependencies, as detailed in Section 3.1.

3.3 Query Formation

A significant challenge in developing natural language to programming language code search benchmarks is obtaining high-quality code descriptions to serve as queries. To address this, our approach utilized LLM (`gpt-4o` and `grok-4`) to automatically generate descriptions for extracted C/C++ functions. The prompts for description generation are provided in Appendix G. The quality of these LLM-generated descriptions was subsequently validated through the hypothesis tests detailed in Section 3.4.

To enhance the LLM's comprehension of functions in Group 2 and Group 3, we incorporated the functional dependencies, including the definitions of the custom-defined variables and helper

[1] Licensing information is provided in Appendix B

Table 2: Hypothesis Testing Results. The LLM-generated descriptions for functions in all 3 groups are comparable or superior in quality to those written by human annotators.

	LLM Score	**LLM 95% CI**	**Human Score**	**Human 95% CI**	**p-value (%)**	**Avg. Krippendorff's α**
Group 1	86.0	(80.5, 91.0)	60.0	(52.5, 67.0)	99.99	68.41
Group 2	76.5	(72.5, 80.5)	72.0	(67.5, 76.5)	76.32	74.77
Group 3	75.5	(72.0, 79.5)	71.5	(67.0, 76.0)	84.92	65.51

functions, into the prompts. Additionally, three manually authored function-description pairs were provided for all three groups as few-shot examples to guide the desired format and style of the generated queries. As CLARC aims to assess the ability of code search models on code semantics, we explicitly instructed the LLM to avoid including identifier names and generate descriptions based on the code's purpose.

3.4 Hypothesis Testing

To statistically compare the quality of function descriptions generated by an LLM against those from human experts, we adapted the hypothesis testing procedure from Wang et al. (2023a). First, both the LLM and a group of expert software engineers (5+ years of experience) created descriptions for 125 sampled functions in each category. These descriptions were then evaluated by three Computer Science PhD students. To measure inter-annotator agreement, a shared set of 50 functions was rated by all three students, while the remaining 75 functions were divided equally among them, with each student rating a unique set of 25. This design resulted in a total workload of 75 evaluations per student. Finally, we applied bootstrapping to the complete set of scores to compare the quality distributions and calculate a p-value. This entire hypothesis test was conducted independently for three distinct function groups to account for varying task complexity.

Double-blind scoring was a crucial step in the hypothesis test. The annotators first checked for errors. If both descriptions for a function were correct, they then judged their relative quality. Incorrect descriptions scored -1. A correct description versus an incorrect one scored +1. Two correct and equally good descriptions each received +0.5. Otherwise, if one description was better, it scored +1 and the other +0.5.

We conducted a statistical comparison between the scores of human and LLM generated descriptions using a bootstrap analysis, with the results presented in Table 2. We measured inter-annotator agreement using Krippendorff's α to establish the reliability of the human annotation. The average α values indicated a consistent and reliable level of agreement among the three annotators.

Our analysis tested the null hypothesis that the quality of LLM-generated descriptions is greater than or equal to that of human-generated descriptions. The p-value was defined as the proportion of bootstrap iterations where the total LLM score equaled or surpassed the total human score. For all experimental groups, the p-values were insufficient to reject the null hypothesis. Moreover, the 95% confidence intervals for the LLM scores were comparable to, or higher than, those for the human scores. Collectively, these results indicate that the LLM-generated descriptions achieve the quality on par with human-generated descriptions, validating their use as queries in our task. This validation serves as a strong foundation for our automated pipeline, ensuring that benchmark construction or extension can scale without requiring human expert annotation.

3.5 Different Settings

Beyond the standard code search task, CLARC was also designed to evaluate models' ability to comprehend code functionality based on its semantics, rather than relying solely on non-functional lexical features (e.g., function, variable, class names). To facilitate this evaluation under different conditions, we introduce several different settings of CLARC. The settings were detailed in Appendix D.2.

- **Neutralized**: Identifiers in the code snippets were replaced with generic, neutral placeholders like `func_a`, `var_b`, `MACRO_c`, or `class_d`, to reduce non-functional information while preserving the structural role of each identifier.

- **Randomized**: Identifiers in the code snippets were replaced with random names to eliminate all lexical information in the identifiers.
- **Assembly**: The C/C++ code is compiled to x86 assembly using the `g++` compiler. Most identifiers were eliminated when compiled to assembly, while for function names, we removed the symbols by post-processing the assembly using `objcopy -strip-all`.
- **WebAssembly (Wasm)**: We used Emscripten (Emscripten Team, 2024) for compilation in WebAssembly with default settings, ensuring no identifiers are preserved in the Wasm version.

4 Experiment Setup

Models A small number of embedding models support C/C++, Assembly, and Wasm, due to the focus on Python in existing code search research. We evaluated the following models on CLARC across its standard, neutralized, and randomized settings, unless noted otherwise. The details of the models can be found in Appendix E.

- **BM25 (Trotman et al., 2014)** A classical TF-IDF based retrieval algorithm using term frequency, inverse document frequency, and length normalization. It relies on lexical features (e.g., identifier names) and serves as our baseline, and is evaluated only on the standard setting.
- **CodeT5+(110M) (Wang et al., 2023b)** An encoder-decoder Transformer trained on code and text. Its encoder half is used to generate the embeddings for code search.
- **OASIS(1.5B) (Gao et al., 2025)** A code embedding model using an Order-Augmented Strategy with generated hard negatives and order-based similarity labels to learn finer code semantic distinctions.
- **Nomic-emb-code(7B) (Nomic Team, 2025)** A large code embedding model trained on CoRNStack (Suresh et al., 2025) using curriculum-based hard negative mining.
- **OpenAI-text-embedding-large (OpenAI, 2024)** A large, closed-source, general-purpose text embedding model. Despite not being code-specific, its broad training enables effective semantic representation of code. This model is evaluated on all settings.
- **Voyage-code-3 (Voyage AI, 2024)** A closed-source embedding model optimized for code retrieval, trained on a diverse corpus including extensive code data. It claims state-of-the-art performance on code benchmarks. This model is evaluated on all settings.

Metrics We evaluated model performance using standard information retrieval metrics: **NDCG** (Normalized Discounted Cumulative Gain) to assess the quality of ranked lists, **MRR** (Mean Reciprocal Rank) to measure how quickly the first relevant item is found, **MAP** (Mean Average Precision) to gauge overall ranking quality across queries, and **Recall@k** (R@k) to determine the proportion of relevant items retrieved within the top k results.

5 Evaluation

This section evaluates the code search models' performance across three settings: standard (Section 5.1), neutralized/randomized (Section 5.2), and low-level languages (Section 5.3). A comparison between the standard and neutralized/randomized settings reveals a dramatic performance drop when identifier names are anonymized, indicating that current models rely heavily on lexical information rather than pure code semantics. Furthermore, the poor performance of general-purpose embedding models like OpenAI-text-embedding-large and Voyage-code-3 on low-level languages underscores the need for specialized solutions for retrieval tasks involving Assembly or Wasm.

5.1 Standard Setting

Table 3 shows model performance on the CLARC standard setting. The limitations of simple text similarity (BM25) and older models like CodeT5+ (early 2023) become clear when compared to newer releases. Models such as OpenAI-text-embedding-large (early 2024), Voyage-code-3 (late 2024), and Nomic-emb-code and OASIS (2025), demonstrate substantially higher effectiveness. The dominance of the latest models underscores the rapid evolution of code search technology.

Beyond general performance differences, Table 3 also reveals how model performance varies in different CLARC categories. First, the latest models–Nomic-emb-code, OASIS, OpenAI-text-embedding-large, and Voyage-code-3–achieve higher retrieval scores in Group 2 over Group 1,

Table 3: Evaluation Results on the Standard Setting. **Bold entries** stand for the maximum values for the metrics in the category. OpenAI stands for OpenAI-text-embedding-large. Voyage stands for Voyage-code-3.

Model	NDCG	MRR	MAP	R@1	R@5	R@10	R@20
Group 1							
BM25	10.50	8.20	9.33	4.75	12.55	18.06	23.00
CodeT5+	64.54	58.84	59.57	47.34	74.14	82.51	89.54
Nomic	88.61	86.23	86.41	80.04	**94.11**	95.82	96.96
OASIS	89.08	86.54	86.71	79.85	**94.11**	**96.77**	**98.48**
OpenAI	83.57	80.16	80.45	71.67	91.06	93.92	96.01
Voyage	**88.99**	**86.93**	**87.18**	**80.99**	**94.11**	95.06	97.53
Group 2							
BM25	17.83	14.64	16.42	9.81	20.47	28.36	40.72
CodeT5+	52.97	46.67	47.80	35.82	60.77	73.35	83.16
Nomic	93.61	91.61	91.63	**86.14**	98.72	99.57	99.57
OASIS	91.11	88.30	88.33	81.02	98.29	99.57	**100.00**
OpenAI	85.87	81.66	81.73	71.86	95.52	98.72	99.57
Voyage	**94.06**	**92.10**	**92.11**	85.93	**99.57**	**99.79**	**100.00**
Group 3 Short							
BM25	10.50	11.52	7.94	2.35	7.98	11.51	15.45
CodeT5+	43.55	47.82	31.24	14.68	32.44	44.83	53.93
Nomic	65.39	**80.58**	48.81	25.33	49.99	**57.22**	**65.78**
OASIS	63.15	73.70	47.35	25.22	48.58	56.87	62.43
OpenAI	62.97	74.54	47.50	25.80	48.33	54.87	62.65
Voyage	**66.66**	80.53	**50.93**	**27.28**	**51.01**	57.04	64.67
Group 3 Long							
BM25	19.09	15.82	17.47	10.40	23.60	29.60	40.40
CodeT5+	21.12	17.78	19.97	12.80	26.00	32.00	50.40
Nomic	69.46	64.93	65.66	55.20	77.20	83.60	90.00
OASIS	68.59	63.53	64.04	53.20	78.40	84.40	87.20
OpenAI	83.80	78.76	78.83	66.40	94.00	99.20	**100.00**
Voyage	**89.13**	**85.43**	**85.43**	**74.40**	**98.80**	**100.0**	**100.00**

suggesting these recent models can effectively utilize custom-defined types for the retrieval task. Additionally, with the exception of CodeT5+, all other models perform better on most retrieval metrics in Group 3 Long than in Group 3 Short. This implies that the richer contextual information from helper functions in longer code snippets generally enhances code search performance for these models. On the other hand, CodeT5+ displays a contrasting pattern, indicating that CodeT5+ is less effective when dealing with these more complex code features.

5.2 Neutralized and Randomized Settings

Table 4 presents the results of the model evaluation in the neutralized and randomized settings of CLARC. A comparison with the standard setting (Table 3) reveals a universal decline in performance across all models, especially for the randomized setting. The extent of this degradation varies: CodeT5+ experiences the most significant drop, followed by OpenAI. In contrast, Nomic-emb-code, OASIS, and Voyage-code-3 have smaller performance decreases. This disparity suggests that CodeT5+ and OpenAI are more vulnerable, whereas the other three models demonstrate relatively stronger robustness. Nevertheless, the performance drop observed in code search models reveals their dependence on lexical information in the identifiers.

In particular, the model performance degrades more severely in the randomized setting. We hypothesize that the higher performance metrics observed in the neutralized setting are attributable to the residual lexical cues within the identifier, such as their classification as variables or functions. In contrast, performance in the randomized setting reflects the models' comprehension of code semantics without these cues. Among the open-box models, OASIS has the smallest performance reduction,

Table 4: Evaluation Results on the Neutralized and Randomized Settings. Neu stands for Neutralized and Ran stands for Randomized. **Bold entries** stand for the maximum values for the metrics in the category. The evaluation results on the Randomized Setting are the average after ten trials, and results with standard errors could be found in Appendix F.

Model	NDCG		MRR		MAP		R@1		R@5	
	Neu	Ran	Neu	Ran	Neu	Ran	Neu	Ran	Neu	Ran
Group 1										
CodeT5+	46.44	34.96	40.18	29.52	41.48	31.03	29.66	20.57	53.42	41.52
Nomic	87.46	77.05	84.03	72.78	84.15	73.26	76.43	63.35	93.54	85.21
OASIS	87.13	82.33	83.66	78.74	83.78	79.02	**76.62**	70.11	91.44	89.62
OpenAI	74.82	66.60	70.13	60.75	70.62	61.40	59.89	48.90	84.22	76.41
Voyage	**87.56**	**83.85**	**84.22**	**80.68**	**84.33**	**81.00**	76.05	**72.66**	**94.87**	**90.53**
Group 2										
CodeT5+	19.15	14.42	15.67	11.27	17.63	12.79	10.66	6.50	22.60	16.91
Nomic	73.37	55.27	67.65	48.23	68.14	49.24	54.80	34.75	84.65	66.74
OASIS	74.79	67.20	68.91	60.19	69.30	60.77	56.50	46.63	85.29	78.29
OpenAI	44.20	32.45	37.14	27.55	38.53	29.11	24.95	19.21	52.88	38.51
Voyage	**81.09**	**75.22**	**77.18**	**69.43**	**77.52**	**69.82**	**68.23**	**56.84**	**88.27**	**85.97**
Group 3 Short										
CodeT5+	6.52	5.73	5.37	5.59	4.56	4.28	1.33	1.40	4.82	4.13
Nomic	24.40	19.13	27.24	21.21	17.24	13.44	10.23	7.55	18.83	14.36
OASIS	27.14	25.71	29.08	29.14	**19.18**	17.48	**11.68**	10.24	**21.28**	19.96
OpenAI	19.46	15.95	21.37	18.42	13.69	10.38	8.30	5.63	14.55	11.58
Voyage	**27.65**	**30.54**	**31.40**	**35.28**	18.91	**20.72**	11.14	**12.85**	20.94	**23.00**
Group 3 Long										
CodeT5+	7.28	7.11	5.21	5.18	7.15	6.87	1.60	2.40	10.00	8.52
Nomic	38.70	30.30	34.22	26.06	35.73	27.86	26.80	19.04	44.40	34.60
OASIS	39.35	34.69	36.08	30.51	37.65	32.15	29.20	22.96	45.20	39.00
OpenAI	34.80	33.28	29.44	28.64	30.83	30.03	20.00	20.16	42.40	39.64
Voyage	**63.90**	**66.40**	**58.58**	**61.15**	**59.45**	**61.95**	**48.40**	**50.48**	**72.80**	**75.04**

indicating that its training methodology, which incorporates enhanced data for robustness, is also beneficial in the neutralized and randomized environment in CLARC.

The retrieval metrics across different groups in the neutralized and randomized setting also diverge from those observed in the standard setting. Specifically, all models now achieve higher performance on Group 1 than on Group 2. The reversal suggests that the models' comprehension of custom-defined types might be more closely tied to the type or variable names themselves, rather than the underlying logic of these types, which becomes obscured in the neutralized and randomized settings. Additionally, the performance drop in neutralized and randomized settings is more dramatic in Group 3. The larger performance drop suggests models rely more heavily on textual cues to understand program functionality when the code snippets are more complex. Meanwhile, the performance gap between Group 3 Short and Group 3 Long widens for most models from the standard setting to the neutralized and randomized settings. An analysis of the reranking results shows that the code search models often fail to retrieve the main function rather than the helper functions in the neutralized and randomized setting. This is likely because the loss of descriptive helper function names due to neutralization increases the difficulty in understanding the main function's overall purpose.

5.3 Assembly & Wasm Settings

Table 5 presents the performance of models on the Assembly and Wasm settings of CLARC. As noted in Section 4, only two general-purpose embedding models, OpenAI and Voyage-code-3, were evaluated due to the incompatibility of other models with Assembly and Wasm. Also, when compiled to low-level languages, the helper functions have to be compiled with the main function. Thus, there is only one variant for Group 3.

Table 5: Evaluation Results on the Assembly and Wasm Settings.

Model	NDCG		MRR		MAP		R@1		R@5	
	Asm	Wasm	Asm	Wasm	Asm	Wasm	Asm	Wasm	Asm	Wasm
Group 1										
OpenAI	11.50	8.89	8.61	6.60	10.12	8.29	4.25	3.22	13.51	10.30
Voyage	34.12	31.40	29.02	27.19	30.21	28.81	19.88	20.17	40.15	37.12
Group 2										
OpenAI	6.86	10.85	5.15	8.14	6.70	10.11	2.40	4.20	9.15	13.09
Voyage	35.28	30.56	28.77	24.36	30.19	25.96	17.43	14.81	44.01	39.26
Group 3										
OpenAI	4.79	8.90	3.46	5.86	5.04	8.14	1.60	2.63	5.20	8.77
Voyage	18.77	23.17	15.20	19.26	17.02	21.20	9.20	13.16	22.80	26.32

When comparing the models' performance in Assembly and Wasm settings to their results in Table 3 and Table 4, we see a more substantial performance drop. The significant performance drops in both OpenAI-text-embedding-large and Voyage-code-3 demonstrate their limited proficiency in understanding these low-level languages. Also, the direct comparison between the two models in these challenging low-level language settings reveals that Voyage-code-3 consistently outperforms OpenAI-text-embedding-large. Note that both Assembly and Wasm environments inherently remove superficial identifier information, and the performance of Voyage-code-3, under these two low-level language settings, shows its capacity to understand the program logic to some extent.

When comparing the models' performance across different categories within these low-level language settings, Group 1 and Group 2 exhibit broadly comparable results. The similarity suggests that custom-defined types do not introduce substantial retrieval challenges in the low-level language setting. In contrast, the models' performance on Group 3 is generally weaker across most metrics. We hypothesize that this disparity arises because functions in Group 3 often involve more dependencies. While such dependencies may not substantially increase complexity in a high-level language or standard setting, compiling them into a low-level language can result in more intricate instruction sequences, consequently making the retrieval task more challenging.

6 Conclusion & Future Works

This paper introduces CLARC, a new benchmark designed to evaluate the robustness of code search models. We also present an automated pipeline for augmenting this benchmark, which produces data of a quality comparable to human experts and mitigates potential knowledge contamination. Our evaluation reveals that while existing models perform decently under standard conditions, their effectiveness substantially degrades when superficial textual features are obfuscated or when code is compiled into a low-level language. These findings demonstrate that current code search models lack a robust understanding of code semantics.

The performance degradation observed across CLARC settings highlights the need for in-depth research into the robustness of code search models. Current models are too reliant on lexical variations, making them unreliable in real-world scenarios involving varied code styles or deliberate obfuscation by malicious attackers. Future investigations can explore methods to enhance model resilience against such perturbations. CLARC and its automatic augmenting pipeline provide a good starting point for retrieving high-quality training data. Furthermore, since C/C++ code can be translated into low-level languages, another natural future direction involves leveraging our proposed pipeline to generate data to train/test code search models that target Assembly or Wasm.

We hope that our findings can also encourage the research community to expand its focus beyond Python by developing code search benchmarks for diverse programming languages. Furthermore, we emphasize the importance of dataset quality, particularly with respect to compilable code snippets. As our experiments demonstrate, compilable code offers inherent versatility, facilitating straightforward conversion into diverse formats suitable for evaluation and potentially for training purposes.

Reproducibility Statement

The code and data required to reproduce the experimental results presented in Section 5 are publicly available. The codebase is hosted on GitHub at `https://github.com/ClarcTeam/CLARC`, and the dataset is available on Hugging Face at `https://huggingface.co/datasets/ClarcTeam/CLARC`. All results were verified to be reproducible with our implementation as of the submission date (September 22, 2025). We note the specific date as certain experimental results rely on API calls (OpenAI-text-embedding-large, Voyage-code-3).

References

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. Opencodeinstruct: A large-scale instruction tuning dataset for code llms, 2025. URL https://arxiv.org/abs/2504.04030.

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, page 185–196, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400706585. doi: 10.1145/3663529.3663839. URL https://doi.org/10.1145/3663529.3663839.

Yang Bai, Anthony Colas, Christan Grant, and Zhe Wang. M3: A multi-task mixed-objective learning framework for open-domain multi-hop dense sentence retrieval. In Nicoletta Calzolari, Min-Yen Kan, Veronique Hoste, Alessandro Lenci, Sakriani Sakti, and Nianwen Xue, editors, *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 10846–10857, Torino, Italia, May 2024. ELRA and ICCL. URL https://aclanthology.org/2024.lrec-main.947/.

Gad Benram. Understanding the cost of large language models (llms). https://www.tensorops.ai/post/understanding-the-cost-of-large-language-models-llms, February 2024. TensorOps AI Blog. Updated March 5, 2024. Accessed May 1, 2025.

Jialun Cao, Yuk-Kit Chan, Zixuan Ling, Wenxuan Wang, Shuqing Li, Mingwei Liu, Chaozheng Wang, Boxi Yu, Pinjia He, Shuai Wang, et al. How should i build a benchmark? *arXiv preprint arXiv:2501.10711*, 2025.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024a. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639085. URL https://doi.org/10.1145/3597503.3639085.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Yuchen Chen, Weisong Sun, Chunrong Fang, Zhenpeng Chen, Yifei Ge, Tingxu Han, Quanjun Zhang, Yang Liu, Zhenyu Chen, and Baowen Xu. Security of language models for code: A systematic literature review. *arXiv preprint arXiv:2410.15631*, 2024b.

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

Connor Dilgren, Purva Chiniya, Luke Griffith, Yu Ding, and Yizheng Chen. Secrepobench: Benchmarking llms for secure code generation in real-world repositories, 2025. URL https://arxiv.org/abs/2504.21205.

Emscripten Team. Emscripten: a complete open source LLVM-based compiler toolchain for WebAssembly. https://emscripten.org, 2024.

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, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/.

Zuchen Gao, Zizheng Zhan, Xianming Li, Erxin Yu, Ziqi Zhan, Haotian Zhang, Bin Chen, Yuqun Zhang, and Jing Li. Oasis: Order-augmented strategy for improved code search. *arXiv preprint arXiv:2503.08161*, 2025.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, May 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.499. URL https://aclanthology.org/2022.acl-long.499/.

Andrea Gurioli, Federico Pennino, João Monteiro, and Maurizio Gabbrielli. One model to train them all: Hierarchical self-distillation for enhanced early layer embeddings, 2025. URL https://arxiv.org/abs/2503.03008.

Geert Heyman and Tom Van Cutsem. Neural code search revisited: Enhancing code snippet retrieval through natural language intent, 2020. URL https://arxiv.org/abs/2008.12193.

Kristen Howell, Gwen Christian, Pavel Fomitchov, Gitit Kehat, Julianne Marzulla, Leanne Rolston, Jadin Tredup, Ilana Zimmerman, Ethan Selfridge, and Joseph Bradley. The economic trade-offs of large language models: A case study. *arXiv preprint arXiv:2306.07402*, 2023.

Junjie Huang, Duyu Tang, Linjun Shou, Ming Gong, Ke Xu, Daxin Jiang, Ming Zhou, and Nan Duan. Cosqa: 20,000+ web queries for code search and question answering, 2021. URL https://arxiv.org/abs/2105.13239.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. Codesearchnet challenge: Evaluating the state of semantic code search, 2020. URL https://arxiv.org/abs/1909.09436.

Gautier Izacard, Mathilde Caron, Lucas Hosseini, Sebastian Riedel, Piotr Bojanowski, Armand Joulin, and Edouard Grave. Unsupervised dense information retrieval with contrastive learning, 2022. URL https://arxiv.org/abs/2112.09118.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024. URL https://arxiv.org/abs/2406.00515.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu, editors, *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6769–6781, Online, November 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.emnlp-main.550. URL https://aclanthology.org/2020.emnlp-main.550/.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.367. URL https://aclanthology.org/2024.acl-long.367/.

Rui Li, Qi Liu, Liyang He, Zheng Zhang, Hao Zhang, Shengyu Ye, Junyu Lu, and Zhenya Huang. Optimizing code retrieval: High-quality and scalable dataset annotation through large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 2053–2065, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.123. URL https://aclanthology.org/2024.emnlp-main.123/.

Xiangyang Li, Kuicai Dong, Yi Quan Lee, Wei Xia, Hao Zhang, Xinyi Dai, Yasheng Wang, and Ruiming Tang. Coir: A comprehensive benchmark for code information retrieval models, 2025. URL https://arxiv.org/abs/2407.02883.

Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. Towards general text embeddings with multi-stage contrastive learning, 2023. URL https://arxiv.org/abs/2308.03281.

Chao Liu, Xin Xia, David Lo, Cuiyun Gao, Xiaohu Yang, and John Grundy. Opportunities and challenges in code search tools. *ACM Comput. Surv.*, 54(9), October 2021. ISSN 0360-0300. doi: 10.1145/3480027. URL https://doi.org/10.1145/3480027.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. Repoqa: Evaluating long context code understanding, 2024a. URL https://arxiv.org/abs/2406.06025.

Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval, 2024b. URL https://arxiv.org/abs/2411.12644.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation. *CoRR*, abs/2102.04664, 2021.

Nomic Team. Nomic Embed Code: A State-of-the-Art Code Retriever. https://www.nomic.ai/blog/posts/introducing-state-of-the-art-nomic-embed-code, 2025. Nomic Blog; accessed May 5, 2025.

OpenAI. New embedding models and api updates, January 2024. URL https://openai.com/index/new-embedding-models-and-api-updates/.

OpenAI. Introducing openai o3-mini, January 2025. URL https://openai.com/index/openai-o3-mini/. Accessed: 2025-09-22.

Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 59:78–87, 2016. URL http://dl.acm.org/citation.cfm?id=2854146.

Yubin Qu, Song Huang, and Yongming Yao. A survey on robustness attacks for deep code models. *Automated Software Engineering*, 31(2):65, 2024.

Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. An empirical study of bugs in webassembly compilers. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, ASE '21, page 42–54. IEEE Press, 2022. ISBN 9781665403375. doi: 10.1109/ASE51524.2021.9678776. URL https://doi.org/10.1109/ASE51524.2021.9678776.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2024. URL https://arxiv.org/abs/2308.12950.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 50(1):85–105, 2024. doi: 10.1109/TSE.2023.3334955.

Oussama Ben Sghaier, Martin Weyssow, and Houari Sahraoui. Harnessing large language models for curated code reviews, 2025. URL https://arxiv.org/abs/2502.03425.

Gaurav Shekhar. The impact of ai and automation on software development: A deep dive. https://ieeechicago.org/the-impact-of-ai-and-automation-on-software-development-a-deep-dive/, November 2024. IEEE Chicago Section. Accessed 2025-05-01.

Weisong Sun, Chunrong Fang, Yifei Ge, Yuling Hu, Yuchen Chen, Quanjun Zhang, Xiuting Ge, Yang Liu, and Zhenyu Chen. A survey of source code search: A 3-dimensional perspective. *ACM Trans. Softw. Eng. Methodol.*, 33(6), June 2024. ISSN 1049-331X. doi: 10.1145/3656341. URL https://doi.org/10.1145/3656341.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. Cornstack: High-quality contrastive data for better code retrieval and reranking, 2025. URL https://arxiv.org/abs/2412.01007.

Andrew Trotman, Antti Puurula, and Blake Burgess. Improvements to bm25 and language models examined. In *Proceedings of the 19th Australasian Document Computing Symposium*, ADCS ’14, page 58–65, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330008. doi: 10.1145/2682862.2682863. URL https://doi.org/10.1145/2682862.2682863.

Lukas Twist, Jie M Zhang, Mark Harman, Don Syme, Joost Noppen, and Detlef Nauck. Llms love python: A study of llms’ bias for programming languages and libraries. *arXiv preprint arXiv:2503.17181*, 2025.

Voyage AI. voyage-code-3: more accurate code retrieval with lower dimensional, quantized embeddings, December 2024.

Jianyou Wang, Kaicheng Wang, Xiaoyue Wang, Prudhviraj Naidu, Leon Bergen, and Ramamohan Paturi. Doris-mae: scientific document retrieval using multi-level aspect-based queries. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS ’23, Red Hook, NY, USA, 2023a. Curran Associates Inc.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training, 2024a. URL https://arxiv.org/abs/2212.03533.

Liang Wang, Nan Yang, Xiaolong Huang, Linjun Yang, Rangan Majumder, and Furu Wei. Improving text embeddings with large language models, 2024b. URL https://arxiv.org/abs/2401.00368.

Yuchen Wang, Shangxin Guo, and Chee Wei Tan. From code generation to software testing: Ai copilot with context-based rag. *IEEE Software*, pages 1–9, 2025. doi: 10.1109/MS.2025.3549628.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023b.

WebAssembly Community. WABT: The WebAssembly Binary Toolkit. https://github.com/WebAssembly/wabt, 2025. Accessed: 2025-05-01.

xAI. Grok 4.
urlhttps://x.ai/news/grok-4, July 2025. Accessed: 2025-09-22.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. C-pack: Packed resources for general chinese embeddings, 2024. URL https://arxiv.org/abs/2309.07597.

Ziyu Yao, Daniel S. Weld, Wei-Peng Chen, and Huan Sun. Staqc: A systematically mined question-code dataset from stack overflow. In *Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18*, WWW '18, page 1693–1703. ACM Press, 2018. doi: 10.1145/3178876.3186081. URL http://dx.doi.org/10.1145/3178876.3186081.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. Learning to mine aligned code and natural language pairs from stack overflow, 2018. URL https://arxiv.org/abs/1805.08949.

Dejiao Zhang, Wasi Ahmad, Ming Tan, Hantian Ding, Ramesh Nallapati, Dan Roth, Xiaofei Ma, and Bing Xiang. Code representation learning at scale, 2024. URL https://arxiv.org/abs/2402.01935.

Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473*, 2024.

Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. Domaineval: An auto-constructed benchmark for multi-domain code generation, 2024. URL https://arxiv.org/abs/2408.13204.

A The Use of LLMs

In this work, the LLMs are used to generate the queries in the dataset, and the quality of the queries is validated by the hypothesis test in Section 3.4. We also use LLMs for post-writing assistance, including proofreading for typographical errors, fixing grammatical errors, enhancing the clarity of expression in human-authored drafts.

B Licenses

B.1 Data Source License

The GitHub repositories utilized by our dataset have various licensing schemes. While the majority use permissive licenses such as the MIT License, a small subset utilizes relatively restrictive licenses like the GPL. To address potential licensing concerns for users, we tag our dataset samples with corresponding license information and provide separate data splits based on license type, distinguishing between permissive and restrictive licenses.

B.2 Model Licenses

- **CodeT5+**: BSD 3-Clause License [2]
- **OASIS**: MIT License[3]
- **Nomic-emb-code**: Apache-2.0 [4]
- **OpenAI text-embedding-large**: Users own the embeddings generated by this model according to OpenAI's policies. The linked documentation provides guidance on sharing these embeddings.[5]
- **Voyage-code-3**: Unclear, but we do not include any embeddings from voyage-code-3 in our codebase.

C Compute Resource

For the query generation component of CLARC, we utilized OpenAI's o3-mini (OpenAI, 2025) and XAI's grok4 xAI (2025). The combined expense for the prompt engineering, hypothesis testing, and query generation phase was approximately $30.

The evaluation environment for the computational experiments was an x86_64-based system running Ubuntu 22.04. This server was configured with two AMD 48-Core Processors and possessed 1.0 TiB of system RAM. An NVIDIA L40 GPU, featuring 46068 MiB of memory, was utilized for the relevant computational tasks; this GPU operated with NVIDIA driver version 550.54.15 and CUDA version 12.4. The aggregate time spent on evaluation across all experiments amounted to roughly 5 GPU hours.

D Supplementary Discussion of the Dataset

D.1 Dataset Statistics

We present the statistics of CLARC in Table 1. For the x86 assembly and WebAssembly code-query pairs, we excluded 20 and 227 samples from the total, respectively, due to technical limitations. The exclusions resulted from challenges in function name extraction from the assembly code, and the higher number of WebAssembly exclusions stemmed from differences in the compilation environment compared to the g++ compiler (for instance, some header files are unsupported for the Wasm compiler).

[2] https://github.com/salesforce/CodeT5?tab=BSD-3-Clause-1-ov-file

[3] https://huggingface.co/Kwaipilot/OASIS-code-embedding-1.5B

[4] https://huggingface.co/nomic-ai/nomic-embed-code

[5] https://platform.openai.com/docs/guides/embeddings#can-i-share-my-embeddings-online

As these exclusions represent only a relatively small fraction of our dataset and do not affect our compilability claims, we consider this acceptable.

D.2 DATASET SETTINGS

Neutralized: Identifiers in the code snippets are replaced with generic, neutral placeholders like `func_a`, `var_b`, `MACRO_c`, or `class_d`, to reduce non-functional information while preserving the structural role of each identifier.

Randomized: Identifiers in the code snippets are replaced with random strings. To ensure stability, we performed randomization **ten** times to create corresponding dataset versions, reporting mean results in Table 4. Complete results including standard errors are provided in Appendix F.

Assembly: Leveraging the fact that all functions in the benchmark are compilable C/C++ code, we provide the low-level assembly code generated by compiling the original functions. The objective is to directly assess a model's capability to interpret assembly language instructions and structure. The C/C++ code is compiled to x86 assembly using the g++ compiler. To achieve a complete anonymization, we remove the function symbols by post-processing the assembly using `objcopy -strip-all`.

WebAssembly (Wasm): Analogous to the Assembly setting, we first compile functions into WebAssembly binaries by Emscripten (Emscripten Team, 2024), the most widely used WebAssembly compiler (Romano et al., 2022). These binaries are subsequently converted to the WebAssembly Text Format (`.wat`) using the WABT toolkit (WebAssembly Community, 2025). This setting specifically tests a model's comprehension of WebAssembly code structure and semantics. Compared to the assembly setting, WebAssembly code features inherent anonymization, as Emscripten does not preserve the function names in the compiled code by default.

E MODEL DETAILS

BM25 (Trotman et al., 2014) BM25 calculates a relevance score for each function by considering the frequency of query terms within that function (Term Frequency), the inverse frequency of those query terms across the entire code collection (Inverse Document Frequency or IDF), and the function's length relative to the average function length. Since BM25 is based on the superficial features like the identifiers' name, we only use BM25 as the baseline for the standard setting.

CodeT5+(110M) (Wang et al., 2023b) CodeT5+ is an encoder-decoder transformer model pre-trained on a vast corpus of source code and associated natural language text. For code search, its encoder generates dense embedding to capture the meaning of both natural queries and functions in programming languages. CodeT5+ is evaluated on the standard, neutralized, and randomized settings.

OASIS(1.5B) (Gao et al., 2025) OASIS (Order-Augmented Strategy for Improved code Search) is a code embedding model designed to capture finer semantic distinctions than traditional contrastive learning approaches. It is trained on generated hard-negatives with assigned "order-based similarity labels" to provide a more granular training signal. OASIS learned to generate embeddings that encode a more nuanced understanding of code functionality, aiming to improve code search performance by better discriminating between semantically close but incorrect candidates. OASIS is evaluated in the standard, neutralized, and randomized settings.

Nomic-emb-code(7B) (Nomic Team, 2025) Nomic-emb-code is a large-scale embedding model optimized for code retrieval tasks. It utilized the CoRNStack dataset (Suresh et al., 2025) and a curriculum-based hard negative mining strategy, which progressively introduces more challenging negative examples to the model over time using softmax-based sampling during training. Nomic-emb-code has strong code search performance according to its reported state-of-the-art results on benchmarks like CodeSearchNet upon release. Nomic-emb-code is evaluated on the standard, neutralized, and randomized settings.

OpenAI-text-embedding-large (OpenAI, 2024) OpenAI's text-embedding-3-large is a large-scale, close-source embedding model accessible via API, widely regarded as a state-of-the-art model for generating general-purpose text representations. While not exclusively trained for code, its training on vast and diverse datasets allows it to produce high-dimensional embeddings that effectively capture semantic meaning for a wide range of inputs, including natural language queries and code snippets. Because of its general-purpose design, we evaluated OpenAI-text-embedding-large on all setting of CLARC.

Voyage-code-3 (Voyage AI, 2024) Voyage-code-3 is a specialized, proprietary embedding model explicitly optimized for code retrieval tasks. It is trained on a large, curated corpus combining general text, mathematical content, and extensive code-specific data to handle the nuances of code semantics. Voyage-code-3 demonstrates state-of-the-art performance on a wide suite of code retrieval benchmarks compared to strong generalist models. Similar to OpenAI-text-embedding-large, we also evaluate Voyage-code-3 on all settings of the benchmark.

F Evaluation

F.1 Full Evaluation Results on Randomized Settings

As shown in Table 6, the models' performance under the Randomized Setting is stable across trials, with standard errors below 1.0 for most metrics.

Table 6: Evaluation Results on Randomized Setting. **Bold entries** stand for the maximum values for the metrics in the category. Results shown as Mean ± Standard Error after 10 trials.

Model	NDCG	MRR	MAP	R@1	R@5
			Group 1		
CodeT5+	34.96±1.12	29.52±1.21	31.03±1.17	20.57±1.41	41.52±1.88
Nomic	77.05±0.63	72.78±0.78	73.26±0.77	63.35±1.32	85.21±0.51
OASIS	82.33±0.24	78.74±0.33	79.02±0.33	70.11±0.58	89.62±0.49
OpenAI	66.60±0.70	60.75±0.95	61.40±0.97	48.90±1.47	76.41±0.53
Voyage	**83.85±0.44**	**80.68±0.58**	**81.00±0.59**	**72.66±1.06**	**90.53±0.51**
			Group 2		
CodeT5+	14.42±0.54	11.27±0.49	12.79±0.49	6.50±0.52	16.91±1.13
Nomic	55.27±1.19	48.23±1.32	49.24±1.27	34.75±1.60	66.74±1.80
OASIS	67.20±0.73	60.19±0.82	60.77±0.80	46.63±1.33	78.29±1.42
OpenAI	32.45±0.65	27.55±0.79	29.11±0.77	19.21±1.14	38.51±1.32
Voyage	**75.22±0.54**	**69.43±0.64**	**69.82±0.63**	**56.84±1.18**	**85.97±0.92**
			Group 3 Short		
CodeT5+	5.73±0.78	5.59±1.04	4.28±0.46	1.40±0.43	4.13±0.69
Nomic	19.13±0.71	21.21±1.01	13.44±0.47	7.55±0.61	14.36±0.53
OASIS	25.71±0.68	29.14±1.29	17.48±0.50	10.24±0.67	19.96±0.62
OpenAI	15.95±0.63	18.42±1.00	10.38±0.42	5.63±0.56	11.58±0.52
Voyage	**30.54±0.36**	**35.28±0.52**	**20.72±0.37**	**12.85±0.63**	**23.00±0.63**
			Group 3 Long		
CodeT5+	7.11±0.78	5.18±0.69	6.87±0.67	2.40±0.75	8.52±1.53
Nomic	30.30±1.38	26.06±1.14	27.86±1.06	19.04±1.27	34.60±1.97
OASIS	34.69±0.67	30.51±0.78	32.15±0.75	22.96±1.20	39.00±0.85
OpenAI	33.28±0.74	28.64±1.01	30.03±1.07	20.16±1.56	39.64±1.72
Voyage	**66.40±0.50**	**61.15±0.72**	**61.95±0.74**	**50.48±1.56**	**75.04±0.95**

G Query Generation Prompts

G.1 Prompt for Group 1

Please refer to Figure 2 for the prompt.

```
Please write a summary for the following C/C++ function that focuses on its functionality without including
overly detailed discussions about the specific algorithm or process used. The goal is to ensure that
someone who treats the function as a black box can understand its functionality after reading your
summary.

Here is the function:
{function_text}

Please read and understand the function step by step. At last, generate your summary after "SUMMARY:".
Please note that in your final summary, you should not consider the background of the function, and only
focus on the functionality. Also, you should also mention the type of the input and output variables while
avoid mentioning the variable names in your final summary.
```

Figure 2: Prompt for Group 1

G.2 Prompt for Group 2

Please refer to Figure 3 for the prompt.

```
Please analyze the following function with name {function_name} and generate a concise summary of its
functionality. Your summary should:
- Focus solely on what the function does (its functionality) rather than detailing the specific algorithms or
processes used.
- Be written from the perspective of a black-box user; that is, someone using the function without
needing to know its internal workings.
- Not include any examples or discuss the function's background—only describe its behavior.
- Use high-level language where possible. If a high-level description isn't sufficient, include necessary
details.
- Explicitly state the types of the input and output variables (as defined in the provided type declarations)
without mentioning any variable names or the function name.

Here are the declaration(s) of the variable types used in the function:
{type_declaration}

Here is the function:
{function_text}

Instructions:
1. Read and understand the function step by step.
2. After your analysis, output your summary on a new line starting with "SUMMARY:"
3. In the final summary, describe only the functionality of the function, explicitly mention the input and
output types, while avoiding any reference to variable names, function names, or too much
implementation details.
```

Figure 3: Prompt for Group 2

G.3 PROMPT FOR GROUP 3

Please refer to Figure 4 for the prompt.

```
Generate a high-quality description for the following C/C++ function based on the provided guidelines. Focus on summarizing the function's purpose and behavior without reproducing the code or referencing internal variable/function names.

Please follow these guidelines strictly:

1. **Function Summarization**:
   - Do not reproduce the entire function or any code in the description.
   - Focus on summarizing the function's purpose and behavior at a high level.
   - If the snippet includes helper functions or other code, treat them as context to better understand the target function's behavior, but only describe the target function (after the label 'Function to Summarize:') in the summary. This a critical requirement.
   - Explicitly mention the input and output types of the function, while avoiding mentioning specific variable names, function names, or too much implementation details.

2. **Description Quality**:
   - Write clear, concise, and accurate descriptions that avoid unnecessary details.
   - Use high-level descriptions when possible, focusing on what the function does rather than how it does it.
   - If a high-level description is insufficient, include comprehensive details covering all necessary aspects of the function's behavior.
   - Be careful about the details and ensure the description correctly aligns with the function's behavior.

3. **Naming Conventions**:
   - Do not reference internal function or variable names defined within the function body.
   - You may reference names of types, classes, or structs if they are relevant to the description.

4. **Output Format**:
   - Provide the description as plain text.
   - Ensure the description is standalone and does not assume prior context beyond the provided snippet.

5. **Constraints**:
   - Avoid speculative details or assumptions about the code's broader context.
   - Focus only on the functionality implied by the provided snippet.
   - Do not mention any specific identifiers (e.g., variable or function names) unless they are types, classes, or structs.

Your goal is to produce a description that is precise, professional, and aligned with the provided guidelines, suitable for documentation purposes.

Here is the code snippet for description:

{function_text}

Provide the description as plain text, following the guidelines strictly. At last, generate your summary after "SUMMARY:\n". Please note that in your final summary, you should not consider the background of the function, and only focus on the functionality. Also, you should also mention the type of the input and output variables while avoid mentioning the variable names in your final summary.

Here are some examples of how the final summary should look like:

{few_shot_examples}
```

Figure 4: Prompt for Group 3

G.4 Prompt for Style Alignment

Please refer to Figure 5 for the prompt. The few-shot examples used in the prompt are sampled from the prompt engineering set.

```
Your task is to rewrite a summary for a function in C/C++ so that the revised summary is in the same
format as the provided examples. Remember, the revised function should not include specific function
name or variable names.

Here are the example summaries:

Example Summary 1:
{few_shot_example}

Example Summary 2:
{few_shot_example}

Example Summary 3:
{few_shot_example}

Here is the summary you should rewrite:
SUMMARY: {original_query}

For reference, here is the original function:
{function_text}

Please only output the revised summary. Feel free to include additional details if you think it's helpful to
make the format of your revised summary more similar to the examples.
```

Figure 5: Prompt for Style Alignment

H EXAMPLES

H.1 EXAMPLES FROM GROUP 1

Query Example

```
The function takes a single input of type char. It verifies whether the
input character is a numeric digit by checking if it lies between '0'
and '9' (inclusive). If the input character meets this condition, the
function returns true; otherwise, it returns false. The output of the
function is of type bool.
```

Code Snippet Example

```
static bool IsDigit(const char d) {
    return ('0' <= d) && (d <= '9');
}
```

H.2 EXAMPLES FROM GROUP 2

Query Example

```
The function accepts an input of type pointer to a structure (OptAnc)
containing two integers ("left" and "right") and a second input of type
int. It checks whether the int input is represented as a set bit in the
first integer element; if not, it then checks the second integer
element. The output is an int that indicates success (1) if the bit is
set in either of the integer fields, or failure (0) otherwise.
```

Code Snippet Example

```
static int is_set_opt_anc_info(OptAnc* to, int anc) {
    if ((to->left & anc) != 0) return 1;

    return ((to->right & anc) != 0 ? 1 : 0);
}
```

H.3 EXAMPLES FROM GROUP 3

Query Example

```
This function performs an in-place sort on an array of unsigned char
pointers, which represent strings, for a specified number of elements
starting from the beginning of the array. It orders the strings in
ascending lexicographical order based on the substrings beginning at a
given offset position in each string. The function takes an array of
unsigned char pointers, an integer specifying the number of elements to
sort, and an integer offset for comparisons, and returns void.
```

Code Snippet Example

```
typedef unsigned char* string;

int scmp( unsigned char *s1, unsigned char *s2 )
{
    while( *s1 != '\0' && *s1 == *s2 )
    {
        s1++;
```

```
        s2++;
    }
    return( *s1-*s2 );
}


static void simplesort(string a[], int n, int b)
{
   int i, j;
   string tmp;

   for (i = 1; i < n; i++)
      for (j = i; j > 0 && scmp(a[j-1]+b, a[j]+b) > 0; j--)
         { tmp = a[j]; a[j] = a[j-1]; a[j-1] = tmp; }
}
```