BRACE: AN R BENCHMARK FOR RIGOROUS BIOMEDICAL DATA SCIENCE CODE SEARCH

Anonymous authors
Paper under double-blind review

ABSTRACT

Effective code retrieval is critical for biomedical data scientists, who navigate thousands of R packages on CRAN and Bioconductor and a growing body of open-source analysis pipelines. Existing code search benchmarks, however, focus on Python and rarely stress-test robustness beyond superficial lexical cues. To address this gap, we adapt an automated benchmark-construction pipeline to R and present BRACE (Biomedical R Anonymized Code rEtrieval), a benchmark built from real-world biomedical repositories and packages. BRACE contains 1,208 query-code pairs for evaluation and 5,310 pairs for training. Queries are LLM-generated natural-language descriptions validated through scoring by domain experts and hypothesis testing. The pipeline first ensures that every snippet byte-compiles and resolves all of its dependencies inside a pinned R/Bioconductor environment. It then categorizes snippets by dependency complexity, distinguishing functions that use only base R, functions that rely on custom S3/S4/R6 or Bioconductor container classes, and functions that invoke user-defined helpers. BRACE further stress-tests retrieval robustness through identifier anonymization and through representation shift, in which functions are lowered to R bytecode or serialized as S-expression parse trees. Under these conditions, our evaluation of six retrieval models reveals consistent drops when identifiers are anonymized, NDCG falls by 3-25 points on base-R functions and by up to 56 points on container-class functions, and far larger drops on lowered representations, where even the strongest model ranks the correct function first for fewer than a quarter of bytecode queries. The results indicate that current models still rely on lexical features rather than code semantics, and that in R this reliance extends to string-typed data accessors.

1 INTRODUCTION

Biomedical data science is conducted largely in R. Bioconductor hosts more than two thousand software packages for genomics, transcriptomics, proteomics, and clinical data analysis (Gentleman et al., 2004; Huber et al., 2015), CRAN provides tens of thousands more (R Core Team et al., 2024), and laboratories publish analysis pipelines for single-cell, bulk RNA-seq, GWAS, and clinical studies on GitHub. Locating an existing implementation, a library-size normalization, a quality-control filter, a differential-expression wrapper, is a recurring task for analysts, and successful reuse directly supports reproducibility, a long-standing concern in computational biology (Peng, 2011; Sandve et al., 2013). Although large language models (LLMs) and code language models now write biomedical analysis code with considerable skill, running such heavy models over repository-scale collections is computationally prohibitive; searching across extensive codebases demands compact feature storage and efficient candidate ranking (Di Grazia & Pradel, 2023). Embedding-based retrieval therefore remains the standard for scalable navigation and is a core component of retrieval-augmented generation (RAG) pipelines (Lewis et al., 2020), including the scientific coding agents now being evaluated on computational-biology tasks (Chen et al., 2025; Mitchener et al., 2025).

Existing code search benchmarks have three limitations that restrict their usefulness in this setting. First, they prioritize Python. CodeSearchNet's six languages do not include R (Husain et al., 2019); CoSQA, DS-1000, and ARCADE are Python-only (Huang et al., 2021; Lai et al., 2023; Yin et al., 2023); and COIR is predominantly Python (Li et al., 2025). R appears in code generation benchmarks such as MultiPL-E (Cassano et al., 2023), but not in a text-to-code retrieval benchmark drawn from real-world biomedical projects; BioCoder, the closest domain-specific effort, targets genera-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

tion in Python and Java (Tang et al., 2024). Second, snippets in current benchmarks are often not executable: `library()` calls are missing, helper functions are undefined, and functions operate on container objects whose class definitions are absent. This contrasts with professional practice, where inspecting the S4 class or helper behind a call is essential to understanding what a function does. Third, benchmarks rarely evaluate robustness to textual perturbation such as identifier renaming (Yefet et al., 2020; Qu et al., 2024). R code is unusually rich in lexical cues: descriptive identifiers, explicit namespace prefixes such as `DESeq2::` (which indicates differential gene expression analysis), and column-name string literals such as `"padj"` (which specifies adjusted p-values) reveal a function's computational intent before a single line of control logic is executed. Whether high benchmark scores reflect true semantic understanding or mere lexical pattern matching therefore remains unresolved. This distinction is critical in practice: CRAN and Bioconductor packages are byte-compiled by default, real-world laboratory pipelines routinely rely on generic variable names (`tmp2`, `df_final_v3`), and sensitivity to identifier perturbations remains a known bottleneck for modern code models.

To address these limitations we introduce BRACE (Biomedical R Anonymized Code rEtrieval), a dataset of 6,518 query-code pairs (1,208 for evaluation and 5,310 for training). BRACE is built by an automated pipeline that extracts functions from recent biomedical repositories and packages to mitigate knowledge contamination, generates queries with LLMs, and validates them through expert assessment and hypothesis testing. Every snippet byte-compiles and resolves its dependencies in a pinned environment, and snippets are categorized by dependency complexity so that the effect of contextual information can be measured. BRACE additionally provides settings that anonymize identifiers or lower functions to R bytecode and S-expression parse trees. These representations isolate the contribution of surface lexical shortcuts and evaluate retrieval robustness across levels of structural abstraction.

On BRACE we evaluate six retrieval methods: two black-box systems (OpenAI-text-embedding-3-large, Voyage-code-3), a lightweight encoder (CodeT5+), a robustness-oriented model trained with augmented data (OASIS), a model adapted from a large code LM (Nomic-emb-code), and a lexical baseline (BM25). Because few code embedding models are trained on R, the evaluation also measures cross-language transfer. Our experiments reveal a consistent decline in retrieval metrics when identifiers are anonymized, showing that state-of-the-art models still rely on lexical features rather than code semantics.In R, this vulnerability is driven significantly by string-typed data accessors, which account for nearly half of the accuracy loss on container-class functions. We also observe a severe drop on bytecode and parse-tree representations, and a 38-40-point NDCG deficit for the one model whose pre-training excludes R, quantifying the cost of language coverage.

In summary, our main contributions are:

- Introducing BRACE, a fully executable R benchmark for biomedical code search that evaluates retrieval robustness under varying dependency complexity and settings;
- Adapting an automated, hypothesis-test-validated pipeline to R, including R-native lowering to bytecode and S-expression parse trees, enabling scalable creation of evaluation resources for an under-served language and domain; and
- Revealing the models' overreliance on lexical features in R, amplified by string-typed accessors, and their limited ability to generalize to lowered representations.

2 Related Works

Code search models. Code retrieval has become a critical component of software engineering, facilitating efficient development and improving code quality (Li et al., 2025). Following the paradigm of general dense retrieval (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. Notably, the training corpora behind these models rarely include R: CoRNStack covers six languages without R, and CodeT5+'s pre-training data covers nine. BRACE therefore probes transfer to a language that is central to data science practice but marginal in code-retrieval training.

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 & 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 neglect R 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 & 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 fail to address the influence of superficial textual features. Most closely related is CLARC (Wang et al., 2026), which introduced a pipeline for compilable, dependency-categorized C/C++ retrieval with identifier anonymization and compilation to Assembly and WebAssembly. None targets R text-to-code retrieval from real-world projects.

Benchmarks for scientific and biomedical code. Data-science code generation has been benchmarked with DS-1000 (Lai et al., 2023) and ARCADE (Yin et al., 2023), bioinformatics generation with BioCoder (Tang et al., 2024), and agentic scientific workflows with ScienceAgentBench (Chen et al., 2025) and BixBench (Mitchener et al., 2025). These evaluate generation or task completion rather than retrieval, and predominantly in Python. BRACE complements them by providing, in R, the retrieval component that RAG-based scientific assistants depend on.

LLMs for benchmark construction. 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 BRACE, 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 BRACE framework. Tab. 1 summarizes the evaluation set. § 3.1 describes construction, code collection, categorization, robustness settings, and LLM-based query formation, and § 3.2 describes the hypothesis-testing procedure used to validate query quality.

Table 1: **Statistics of BRACE's evaluation set.** LOC: lines of code; CC: cyclomatic complexity (computed with `cyclocomp`); R-Source: original R source; S-ByteC: serialized R bytecode; AST: S-expression parse tree. All code statistics are averages within the category. Training-set statistics are in Appx. C.1.

Category	# Pairs	# Tokens in Query	# Tokens R-Source	# Tokens S-ByteC	# Tokens AST	LOC R-Source	LOC S-ByteC	LOC AST	CC R-Source
Group 1	493	86.9	98.4	688.2	421.7	9.8	74.3	51.6	2.4
Group 2	462	85.1	121.5	874.9	538.0	11.6	91.8	66.2	2.6
Group 3	253	79.8	612.3	3,148.6	2,411.9	58.1	334.7	298.4	4.9
Total	1,208	84.7	214.9	1,274.9	883.0	20.6	135.5	108.9	3.0

3.1 DATASET CONSTRUCTION

Construction follows a four-stage pipeline. *Data Collection* mines functions from biomedical R repositories and packages and filters them for executability. *Categorization* organizes functions by their reliance on custom classes and helper functions. *Different Settings* transform snippets into

anonymized or lowered forms to test semantic understanding beyond lexical matching. *Query Formation* uses LLMs to write natural-language descriptions that serve as ground-truth queries.

3.1.1 DATA COLLECTION

We mined 144 R repositories: 58 Bioconductor software packages from release 3.22 and 86 GitHub repositories matching biomedical topics (e.g., `single-cell`, `rna-seq`, `gwas`, `clinical-trials`, `epidemiology`) with at least 20 stars, an OSI-approved license, and commits after 1 January 2025, the latter to mitigate contamination.[1] Repositories were split into an evaluation set (45 repositories) and a training set (99 repositories). To ensure reproducibility, we built a pinned execution environment (R 4.5.1, Bioconductor 3.22, snapshotted with `renv` (Ushey, 2023)) with a whitelist of installable packages: base and recommended packages plus 42 infrastructure and domain packages (e.g., `SummarizedExperiment`, `SingleCellExperiment`, `GenomicRanges`, `Seurat`, `dplyr`).

We parsed every R source file, located function definitions (closure assignments, S4 `setMethod` bodies, and R6 methods), and extracted each function together with its full dependency context: the intra-repository call graph obtained with `codetools` (Tierney, 2011), class definitions (`setClass`, `R6Class`), and referenced constants. A function is retained only if (i) it parses, (ii) it byte-compiles with `compiler::cmpfun` (Tierney, 2012), and (iii) every free symbol resolves either to a whitelisted namespace or to a definition within the repository. Near-duplicate functions, common in R repositories, where utility functions are frequently copied between projects, were removed by hashing identifier-normalized parse trees. The retained functions were then categorized into three groups.

3.1.2 CATEGORIZATION

Group 1 consists of functions that depend only on base and recommended packages and built-in data structures (vectors, lists, matrices, data frames). **Group 2** contains functions that use base functionality but operate on custom classes: S3, S4, or R6 classes defined in the repository, or domain container classes imported from whitelisted packages such as `SummarizedExperiment`, `SingleCellExperiment` (Amezquita et al., 2020), or `Seurat` (Hao et al., 2021b). **Group 3** encompasses all other functions, namely those that invoke user-defined helper functions from the same repository. Fig. 1 shows brief examples.

To investigate the influence of helper functions, we designed two variants of Group 3. In *Group 3 Short*, the primary function and its helpers are separate retrievable snippets. In *Group 3 Long*, the primary function is merged with its helpers into one contiguous snippet, so that the main logic and its immediate dependencies fit in the model's context window during retrieval. Details are in Appx. C.3.

3.1.3 DIFFERENT SETTINGS

Beyond the standard task, BRACE evaluates whether models comprehend code functionality from semantics rather than from non-functional lexical features (function, variable, argument, class, and slot names). Implementation details appear in Appx. C.4.

Neutralized. Identifiers are replaced with generic placeholders such as `func_a`, `var_b`, `arg_c`, `class_d`, and `slot_e`, reducing non-functional information while preserving each identifier's structural role. Names exported by whitelisted packages are preserved. String literals that act as data accessors (e.g., $x padj$, `x[["counts"]]`, tidy-evaluation column references) are treated as identifiers because, in R, they carry the same lexical signal. **Randomized.** Identifiers are replaced with random strings, eliminating all lexical information. Randomization is repeated ten times and means are reported. **Bytecode.** Functions are compiled with `compiler::cmpfun` and disassembled with `compiler::disassemble`; the opcode sequence and constant pool are serialized as text. Constant-pool symbols that refer to user-defined identifiers are anonymized consistently, mirroring symbol stripping in compiled binaries, while references to whitelisted functions remain. Because installed R packages are byte-compiled by default, this is the representation an indexer of installed libraries encounters. Abstract Syntax Tree **(AST.)** Functions are parsed and serialized as nested S-expressions that mirror R's internal representation (Wickham, 2015), with comments and source references removed and identifiers anonymized as in the Neutralized setting. The setting is the analogue of a textual intermediate representation and tests whether models recover functionality from structure alone.

[1] Licensing details are provided in Appx. A.

```r
cpm <- function(counts) {
  t(t(counts) / colSums(counts)) *
      1e6
}
```

(a) Group 1

```r
drop_low_counts <- function(se,
   min_count = 10, min_samples =
       3) {
  keep <- rowSums(assay(se, "
      counts") >=
          min_count) >= min_
              samples
  se[keep, ]
}
```

(b) Group 2

```r
# Helper function:
.mito_fraction <- function(counts,
    is_mito) {
  colSums(counts[is_mito, , drop =
      FALSE]) /
    colSums(counts)
}

# Primary function:
flag_high_mito <- function(counts,
    is_mito,
                       max_frac = 0.2)
                             {
  frac <- .mito_fraction(counts, is_
      mito)
  names(frac)[frac > max_frac]
}
```

(c) Group 3

Figure 1: **Example functions of BRACE.** Full examples with their queries are in Appx. L.

3.1.4 Query Formation

Obtaining high-quality descriptions to serve as queries is the most labor-intensive step in building natural-language-to-code benchmarks. We used two LLMs (`o3-mini` and `grok-4`) to generate descriptions of the extracted functions; prompts are given in Appx. K, and quality is validated in § 3.2. Three manually authored function, description pairs served as few-shot examples for each group. For Groups 2 and 3, the prompt included the functional dependencies, class definitions or slot signatures, helper functions, and, when available, their roxygen documentation, so that the LLM could reason about container semantics and helper behavior. Because BRACE targets code semantics, the LLM was instructed to describe what the function computes in domain terms (e.g., "scales each sample's counts by library size") and to avoid identifier names, package names, and column-name literals.

3.2 Hypothesis Testing

To validate LLM-generated descriptions against expert-written ones, we adapted the hypothesis-testing framework of Wang et al. (2023a). We sampled 125 functions per group and obtained two descriptions for each: one from the LLM and one written by bioinformaticians with at least five years of R experience in biomedical research. Three PhD students in computational biology evaluated the pairs. A shared set of 50 functions per group was rated by all three to measure inter-annotator agreement; the remaining 75 were split into disjoint sets of 25, giving each annotator 75 evaluations per group.

Double-blind scoring. Annotators did not know which description was LLM-generated. They first checked factual correctness and then compared quality: a description containing a hallucination or technical error received -1.0; if both were correct, the clearly better one received $+1.0$ and the other $+0.5$, or both received $+0.5$ if of equal quality.

Statistical analysis and results. We compared aggregate LLM and human scores by bootstrap with 10,000 iterations and measured agreement with Krippendorff's α. Following Wang et al. (2023a), the p-value is the proportion of bootstrap iterations in which the total LLM score equaled or exceeded the total human score, testing the null hypothesis that LLM descriptions are comparable to or better than human descriptions. As Tab. 2 shows, p-values are high for Groups 1 and 2 and moderate for Group 3 (66.1%), where annotators most often penalized LLM descriptions for overstating what helper functions do. The 95% confidence intervals for LLM scores are comparable to or above those for human scores in every group, and α between 0.64 and 0.71 indicates acceptable agreement. LLM descriptions therefore match expert quality, justifying their use as queries, with the Group 3 prompt the natural target for further iteration.

Table 2: **Hypothesis testing results.** LLM-generated descriptions are comparable or superior in quality to those written by domain experts in all three groups.

Group	LLM Evaluation Score	LLM 95% CI	Human Evaluation Score	Human 95% CI	p-value (%)	Krippendorff's α (avg.)
Group 1	83.5	(78.0, 88.5)	64.5	(57.5, 71.0)	99.7	66.4
Group 2	75.0	(70.5, 79.5)	70.5	(66.0, 75.0)	82.9	71.7
Group 3	72.5	(68.5, 76.5)	71.0	(66.5, 75.5)	66.1	64.1

4 Experiment Setup

Models. Few embedding models advertise support for R, let alone R bytecode or parse trees, reflecting the Python focus of code search research. We evaluated the following models; details are in Appx. D. **BM25** (Trotman et al., 2014): a classical TF-IDF-based algorithm with length normalization. It depends on identifier names, serves as our baseline, and is evaluated only in the standard setting. **CodeT5+ (110M)** (Wang et al., 2023b): an encoder, decoder Transformer trained on code and text; its encoder produces the embeddings. **OASIS (1.5B)** (Gao et al., 2025): a code embedding model trained with order-based similarity labels on generated hard negatives. **Nomic-emb-code (7B)** (Nomic Team, 2025): a large code embedding model fine-tuned on CoRNStack (Suresh et al., 2025) with curriculum-based hard-negative mining. **OpenAI-text-embedding-3-large** (OpenAI, 2024): a closed-source general-purpose embedding model. **Voyage-code-3** (Voyage AI, 2024): a closed-source embedding model optimized for code retrieval.

Metrics. We report standard retrieval metrics: (Normalized Discounted Cumulative Gain) **NDCG**↑ (ranked-list quality), (Mean Reciprocal Rank) **MRR**↑ (how quickly the first relevant item appears), (Mean Average Precision) **MAP**↑ (overall ranking quality across queries), and (Recall@k) **R@k**↑ (the proportion of relevant items within the top k). Formal definitions are given in Appx. E.

5 Evaluation

We evaluate across the standard setting (§ 5.1), the neutralized and randomized settings (§ 5.2), and the lowered representations (§ 5.3); § 5.4 examines whether supervised fine-tuning closes the gaps. We pre-registered four hypotheses. **H1**: anonymizing identifiers degrades all models, and the randomized setting degrades more than the neutralized one. **H2**: Group 2 depends most on class and slot names, so its ordering relative to Group 1 reverses under anonymization. **H3**: helper-function context (Group 3 Long) matters more once names are removed. **H4**: performance on bytecode and AST representations falls well below the anonymized settings. All four are supported by the results below.

5.1 Standard Setting

Tab. 3 shows model performance on the standard setting. BM25 remains far behind every neural model, and CodeT5+, whose pre-training corpus excludes R, trails the R-inclusive models by 38-40 NDCG points in Group 1, quantifying the cost of language coverage. Among recent models, Voyage-code-3 leads on most metrics, with OASIS and Nomic-emb-code within 3 points on Groups 1 and 2. The recent models score higher on Group 2 than on Group 1 (e.g., Voyage 92.1 vs. 87.6 NDCG), suggesting that they exploit container-class semantics, container-class *names*. Every model except CodeT5+ performs better on Group 3 Long than on Group 3 Short, so richer helper-function context helps retrieval; CodeT5+'s 512-token window truncates most merged snippets, and its Group 3 Long NDCG collapses to 15.3. Results on the expert-written subsets (Appx. F.2) are within about 2 points of the full dataset for Groups 1 and 2, confirming the quality of the queries.

5.2 Neutralized and Randomized Settings

Tab. 4 reveals a substantial and universal decline relative to Tab. 3, supporting H1. In Group 1, NDCG falls by 3-16 points in the neutralized setting and 8-25 points in the randomized setting; in Group 2 the randomized losses reach 22 points for Voyage-code-3 and 56 for OpenAI-text-embedding-3-large. Higher baseline effectiveness therefore does not imply robustness. Randomization hurts more than neutralization for every model, consistent with placeholders such as `func_a` leaking structural role. Among the open models, OASIS degrades least (8.9 NDCG points in Group 1 randomized versus 15.8 for Nomic-emb-code), in line with its robustness-oriented training. The ordering of Groups 1 and 2 reverses for all models (H2): once class names, slot names, and accessor

Table 3: **Evaluation results on the standard setting. Bold** entries mark the maximum value for each metric within a category. OpenAI stands for OpenAI-text-embedding-3-large; Voyage stands for Voyage-code-3.

Model	NDCG	MRR	MAP	R@1	R@5	R@10	R@20
Group 1							
BM25	14.2	11.3	12.6	6.7	16.4	22.9	30.8
CodeT5+	47.8	41.6	42.3	30.4	55.2	66.1	76.9
Nomic	85.9	83.1	83.3	76.3	91.7	94.5	96.3
OASIS	86.7	83.9	84.1	76.9	92.5	95.3	**98.0**
OpenAI	82.4	78.8	79.0	70.2	89.5	92.9	95.5
Voyage	**87.6**	**85.2**	**85.5**	**78.7**	**93.1**	**95.7**	97.6
Group 2							
BM25	21.5	17.9	19.6	12.1	24.3	32.7	44.6
CodeT5+	39.6	33.9	34.8	24.5	44.8	57.4	69.7
Nomic	90.8	88.4	88.5	82.0	96.5	98.3	99.1
OASIS	89.5	86.6	86.7	79.4	96.1	98.5	99.4
OpenAI	84.3	80.1	80.2	70.6	93.1	97.2	98.7
Voyage	**92.1**	**89.9**	**90.0**	**83.5**	**98.1**	**99.1**	**99.8**
Group 3 Short							
BM25	11.8	12.6	8.9	3.1	8.7	12.9	17.2
CodeT5+	30.2	33.5	21.4	9.6	22.8	32.7	41.9
Nomic	62.7	77.9	46.2	23.8	47.1	54.6	**63.0**
OASIS	60.9	71.8	45.1	23.5	46.3	54.0	60.2
OpenAI	60.4	72.3	45.4	24.1	45.9	52.8	59.9
Voyage	**64.3**	**78.4**	**48.7**	**25.9**	**48.8**	**55.1**	62.3
Group 3 Long							
BM25	22.7	18.9	20.6	12.6	27.3	34.0	45.1
CodeT5+	15.3	12.4	14.1	8.7	18.6	24.1	39.5
Nomic	66.8	62.1	62.9	52.6	74.3	81.0	88.1
OASIS	65.4	60.3	60.9	50.2	75.5	82.2	85.4
OpenAI	81.2	76.0	76.1	63.6	92.5	98.0	99.6
Voyage	**87.0**	**83.1**	**83.2**	**71.9**	**97.6**	**99.6**	**100.0**

literals are masked, the remaining logic of container-manipulating functions is insufficient for retrieval.

Group 3 suffers most, and the gap between Short and Long widens (Voyage: 22.7 NDCG points in the standard setting versus 34.9 neutralized), indicating that helper definitions become essential once descriptive names are gone (H3). The accessor-literal ablation (Tab. 8, Appx. C.4) attributes roughly half of the Group 2 neutralized loss to string-typed accessors alone, an R-specific vulnerability.

5.3 Bytecode and AST Settings

Tab. 5 reports performance on the lowered representations. Because helpers must be compiled together with the primary function, Group 3 is a single configuration here. Performance drops far below the anonymized settings for every model (H4): the best model, Voyage-code-3, ranks the correct function first for only 23.5% of Group 1 bytecode queries, and CodeT5+ is near the floor because its 512-token window truncates most bytecode (688 tokens on average in Group 1) (Appx. F.3). The AST representation is consistently easier than bytecode (Voyage 61.5 vs. 38.9 NDCG in Group 1): S-expressions preserve call structure and the names of whitelisted functions in reading order, whereas the bytecode's stack-oriented opcode sequences separate symbols in the constant pool from their use sites. Groups 1 and 2 are comparable at this level, suggesting that custom classes add little difficulty once lowered, while Group 3 is markedly weaker as its dependencies expand into instruction sequences about 4.6× longer than Group 1's. Voyage-code-3 outperforms the other models on every metric, indicating a stronger capacity to read program logic when lexical cues are absent.

5.4 Model Finetuning

To test whether the robustness gaps can be closed by supervised adaptation, we fine-tuned CodeT5+ and OASIS on the BRACE training set with InfoNCE loss and in-batch negatives (Oord et al., 2018; Gao et al., 2021) for five epochs. Tab. 6 reports MRR. CodeT5+ improves across all settings

Table 4: **Evaluation results on the Neutralized (Neu) and Randomized (Ran) settings. Bold** entries mark the maximum value within a category. Randomized results are averages over 10 trials; standard errors are in Appx. F.1.

Model	NDCG		MRR		MAP		R@1		R@5	
	Neu	Ran	Neu	Ran	Neu	Ran	Neu	Ran	Neu	Ran
Group 1										
CodeT5+	31.9	22.4	26.3	17.5	27.4	18.9	17.2	10.8	37.5	25.6
Nomic	82.6	70.1	78.8	65.2	78.9	65.7	70.3	54.9	89.2	78.6
OASIS	83.4	77.8	79.7	73.6	79.9	73.9	71.5	64.2	89.6	85.1
OpenAI	70.6	60.3	65.4	54.2	65.9	54.8	54.6	42.3	79.8	69.7
Voyage	**84.1**	**79.5**	**80.4**	**75.7**	**80.6**	**75.9**	**71.8**	**66.8**	**91.1**	**86.9**
Group 2										
CodeT5+	14.8	10.6	11.7	8.1	13.2	9.4	7.5	4.3	17.1	12.0
Nomic	67.9	49.2	61.8	42.0	62.4	43.1	48.7	29.6	79.5	60.8
OASIS	70.3	62.1	64.1	54.9	64.6	55.4	51.2	41.3	81.3	73.6
OpenAI	39.6	28.7	32.8	24.1	34.2	25.6	21.3	16.2	47.9	34.5
Voyage	**76.8**	**70.4**	**72.3**	**64.2**	**72.7**	**64.6**	**62.4**	**51.7**	**85.0**	**81.2**
Group 3 Short										
CodeT5+	5.4	4.6	4.5	4.4	3.8	3.4	1.1	1.0	4.0	3.3
Nomic	22.1	17.0	24.6	18.9	15.5	11.8	9.0	6.4	16.9	12.5
OASIS	24.9	23.3	26.8	26.5	17.4	15.7	**10.4**	9.1	**19.5**	18.0
OpenAI	17.6	14.2	19.4	16.5	12.3	9.1	7.3	4.9	13.1	10.2
Voyage	**25.8**	**28.1**	**29.3**	**32.6**	**17.6**	**19.0**	10.2	**11.6**	19.4	**21.1**
Group 3 Long										
CodeT5+	5.9	5.6	4.2	4.0	5.8	5.5	1.2	1.6	8.3	7.1
Nomic	35.2	27.4	31.0	23.5	32.5	25.1	24.1	17.0	40.6	31.2
OASIS	36.4	31.9	33.2	27.8	34.8	29.4	26.5	20.8	42.0	36.1
OpenAI	32.1	30.6	27.0	26.2	28.4	27.5	18.2	18.4	39.3	36.5
Voyage	**60.7**	**62.9**	**55.4**	**57.8**	**56.3**	**58.6**	**45.5**	**47.3**	**69.6**	**71.5**

Table 5: **Evaluation results on the Bytecode (Byc) and AST settings. Bold** entries mark the maximum value within a category.

Model	NDCG		MRR		MAP		R@1		R@5	
	Byc	AST	Byc	AST	Byc	AST	Byc	AST	Byc	AST
Group 1										
CodeT5+	7.9	13.6	5.7	10.2	6.8	11.5	2.6	5.1	8.9	15.3
Nomic	29.4	47.8	24.6	42.5	25.7	43.0	15.8	32.1	34.4	54.2
OASIS	32.7	51.9	27.5	46.9	28.6	47.3	18.1	36.2	38.0	58.5
OpenAI	17.6	45.3	13.8	39.8	15.2	40.4	7.9	28.7	20.5	52.0
Voyage	**38.9**	**61.5**	**33.6**	**56.4**	**34.7**	**56.8**	**23.5**	**45.9**	**45.2**	**68.3**
Group 2										
CodeT5+	6.1	9.7	4.4	7.1	5.5	8.3	1.9	3.4	6.8	10.6
Nomic	26.2	38.9	21.3	33.1	22.5	33.9	13.4	22.7	30.1	45.8
OASIS	30.5	47.4	25.1	41.6	26.3	42.2	16.2	30.5	35.9	54.3
OpenAI	13.2	33.8	10.0	28.4	11.5	29.5	5.6	18.9	15.7	40.6
Voyage	**37.1**	**56.2**	**31.4**	**50.5**	**32.6**	**51.0**	**21.0**	**39.4**	**43.5**	**63.1**
Group 3										
CodeT5+	3.2	4.5	2.1	3.1	2.9	4.0	0.8	1.2	3.6	4.8
Nomic	13.6	24.1	10.2	19.5	11.8	21.0	5.9	12.6	15.0	27.7
OASIS	16.0	27.9	12.4	23.0	13.9	24.6	7.1	15.0	18.6	32.4
OpenAI	8.3	22.6	5.9	18.1	7.2	19.4	2.8	11.5	9.5	25.3
Voyage	**21.4**	**42.7**	**17.1**	**37.5**	**18.8**	**38.4**	**11.1**	**26.9**	**25.7**	**49.8**

(Group 1 standard MRR 41.6 vs. 68.9), confirming that the pre-trained model was poorly adapted to R. OASIS, already competent on R, gains little (83.9 vs. 85.1) and over-specializes: fine-tuning on standard data lowers its randomized MRR by 4.2 points in Group 1 and 10.1 in Group 2, and fine-tuning on randomized data degrades the standard setting. Crucially, the gap between the standard

and anonymized settings persists under every training configuration, 17 MRR points for CodeT5+ (Group 1, ft. on Ran.) and 15.7 for OASIS (ft. on Std.), so standard supervised fine-tuning does not resolve the robustness failures the benchmark exposes. These findings underscore the need for benchmarks that target robust code search, and we hope the training set facilitates further work.

Table 6: **MRR of the fine-tuned models.** "ft. on X" indicates fine-tuning on setting X; "W/o ft." is the released checkpoint. Group 3 refers to Group 3 Long.

	Group 1			Group 2			Group 3		
FT Setting	Std	Neu	Ran	Std	Neu	Ran	Std	Neu	Ran
CodeT5+ (110M)									
W/o ft.	41.6	26.3	17.5	33.9	11.7	8.1	12.4	4.2	4.0
ft. on Std.	68.9	58.7	39.2	60.1	27.3	19.8	35.6	12.1	7.4
ft. on Neu.	66.8	62.5	41.0	54.9	33.8	20.6	31.9	17.5	8.7
ft. on Ran.	67.5	58.1	50.6	53.4	29.2	28.4	33.2	14.6	10.1
OASIS (1.5B)									
W/o ft.	83.9	79.7	73.6	86.6	64.1	54.9	60.3	33.2	27.8
ft. on Std.	85.1	78.2	69.4	82.7	55.3	44.8	58.9	24.6	15.7
ft. on Neu.	84.6	81.8	74.2	79.5	58.7	47.3	56.2	36.0	21.9
ft. on Ran.	81.3	77.5	75.9	73.4	50.1	49.6	52.8	30.7	17.4

6 Discussion

Our results show that current retrieval models can often rank the correct function when identifiers are intact but fail to preserve that ability under semantics-preserving reformulations. The consistent NDCG drops under anonymization (3–25 points on base-R functions, up to 56 points on container-class functions), together with the near-floor performance on bytecode and parse-tree representations, suggest that embedding models exploit surface lexical cues more readily than they learn program logic. In R this reliance is amplified by string-typed data accessors, which alone account for roughly half of the Group 2 degradation, and by container-class names, whose removal reverses the Group 1/Group 2 ordering. Our fine-tuning study further shows that standard contrastive adaptation can improve accuracy (CodeT5+ MRR 41.6 - 68.9) without removing sensitivity to identifier perturbation: training on one setting degrades the others, and the standard–anonymized gap persists under every configuration. These findings have implications for both evaluation and deployment: aggregate scores on intact code overestimate semantic understanding, and retrieval components in RAG-based scientific assistants may fail precisely on the container-manipulating, helper-dependent code typical of biomedical pipelines, or on byte-compiled installed packages.

These results highlight a gap in current code-retrieval training: models need mechanisms that map lexically distinct but functionally equivalent code to the same representation and preserve it across lowering. Future work should explore perturbation-invariant and representation-aware objectives, R- and Bioconductor-specific corpora that include lowered forms, and curricula that explicitly separate lexical shortcuts from structural and logical understanding, together with cross-dialect and cross-language retrieval between equivalent analyses.

7 Conclusion

This paper introduces BRACE, a benchmark and automated pipeline for evaluating and improving code search robustness in R for biomedical data science. Our evaluation shows that current models perform poorly once textual features are obfuscated or removed by lowering, exposing a heavy reliance on superficial cues; in R this reliance extends to string-typed data accessors, which alone account for about half of the degradation on container-class functions. Future work includes cross-dialect retrieval (base R, tidyverse (Wickham et al., 2019), data.table), cross-language retrieval between R and Python implementations of the same analysis (e.g., Seurat and Scanpy workflows), practitioner-style queries that name packages, and using the pipeline to generate large-scale training data for lowered representations. We encourage the community to build more executable datasets whose provenance and pinned environments allow diverse evaluation and training.

Ethics Statement

BRACE is constructed from publicly available R packages and repositories. We record license information for every source and provide license-based splits (Appx. A). Because biomedical repositories may contain patient-level data, we include only source code, exclude data files, and scan snippets for embedded identifiers such as dates of birth or record numbers, removing any matches. No human-subjects research was conducted; the annotation study involved expert reviewers assessing code descriptions. We recognize a limited dual-use concern: improved robustness to identifier anonymization could aid analysis of obfuscated code. We release BRACE for research and evaluation and document its intended scope. LLM use is described in the AI Use Statement.

Reproducibility Statement

The pipeline pins the R and Bioconductor versions and snapshots, the package environment with `renv` (§ 3.1.1); settings are specified in Appx. F.5, prompts in Appx. K, fine-tuning details in Appx. F.4, and compute in Appx. B. Code and data will be released upon publication.

AI Use Statement

LLMs were used to generate the natural-language queries in the dataset; their quality was validated by the hypothesis test in § 3.2 and by evaluation on expert-written subsets (Appx. F.2). LLMs were also used for writing assistance (proofreading, grammar, and clarity) on human-authored drafts.

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, pp. 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.

Robert A Amezquita, Aaron TL Lun, Etienne Becht, Vince J Carey, Lindsay N Carpp, Ludwig Geistlinger, Federico Marini, Kevin Rue-Albrecht, Davide Risso, Charlotte Soneson, et al. Orchestrating single-cell analysis with bioconductor. *Nature methods*, 17(2):137–145, 2020.

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 (eds.), *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 10846–10857, Torino, Italia, May 2024. ELRA and ICCL. URL https://aclanthology.org/2024.lrec-main.947/.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023.

Ziru Chen, Shijie Chen, Yuting Ning, Qianheng Zhang, Boshi Wang, Botao Yu, Yifei Li, Zeyi Liao, Chen Wei, Zitong Lu, et al. Scienceagentbench: Toward rigorous assessment of language agents for data-driven scientific discovery. In *International Conference on Learning Representations*, volume 2025, pp. 96934–96990, 2025.

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.

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 2020. Association for Computational Linguistics. doi: 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139/.

Tianyu Gao, Xingcheng Yao, and Danqi Chen. Simcse: Simple contrastive learning of sentence embeddings. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pp. 6894–6910, 2021.

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, 2025. URL https://arxiv.org/abs/2503.08161.

Robert C Gentleman, Vincent J Carey, Douglas M Bates, Ben Bolstad, Marcel Dettling, Sandrine Dudoit, Byron Ellis, Laurent Gautier, Yongchao Ge, Jeff Gentry, et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology*, 5(10): R80, 2004.

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 (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 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.

Yuhan Hao, Stephanie Hao, Erica Andersen-Nissen, William M Mauck, Shiwei Zheng, Andrew Butler, Maddie J Lee, Aaron J Wilk, Charlotte Darby, Michael Zager, et al. Integrated analysis of multimodal single-cell data. *Cell*, 184(13):3573–3587, 2021a.

Yuhan Hao, Stephanie Hao, Erica Andersen-Nissen, William M Mauck, Shiwei Zheng, Andrew Butler, Maddie J Lee, Aaron J Wilk, Charlotte Darby, Michael Zager, et al. Integrated analysis of multimodal single-cell data. *Cell*, 184(13):3573–3587, 2021b.

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.

Max Hort, Linas Vidziunas, and Leon Moonen. Semantic-preserving transformations as mutation operators: A study on their effectiveness in defect detection. In *2025 IEEE international conference on software testing, verification and validation workshops (ICSTW)*, pp. 337–346. IEEE, 2025.

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. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 5690–5700, 2021.

Wolfgang Huber, Vincent J Carey, Robert Gentleman, Simon Anders, Marc Carlson, Benilton S Carvalho, Hector Corrada Bravo, Sean Davis, Laurent Gatto, Thomas Girke, et al. Orchestrating high-throughput genomic analysis with bioconductor. *Nature methods*, 12(2):115–121, 2015.

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.

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.

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 (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 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 (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 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/.

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.

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, 2020.

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 (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 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. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 22074–22091, 2025.

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.

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.

Michael I Love, Wolfgang Huber, and Simon Anders. Moderated estimation of fold change and dispersion for rna-seq data with deseq2. *Genome biology*, 15(12):550, 2014.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin 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, 2021. URL https://arxiv.org/abs/2102.04664.

Ludovico Mitchener, Jon M Laurent, Alex Andonian, Benjamin Tenmann, Siddharth Narayanan, Geemi P Wellawatte, Andrew White, Lorenzo Sani, and Samuel G Rodriques. Bixbench: a comprehensive benchmark for llm-based agents in computational biology. *arXiv preprint arXiv:2503.00096*, 2025.

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.

Aaron van den Oord, Yazhe Li, and Oriol Vinyals. Representation learning with contrastive predictive coding. *arXiv preprint arXiv:1807.03748*, 2018.

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

Roger D Peng. Reproducible research in computational science. *Science*, 334(6060):1226–1227, 2011.

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

R Consortium. R submissions working group: Pilot submissions to the U.S. food and drug administration. https://rconsortium.github.io/submissions-wg/, 2026. Accessed September 2026.

R R Core Team et al. R: A language and environment for statistical computing, 2024.

Matthew E Ritchie, Belinda Phipson, DI Wu, Yifang Hu, Charity W Law, Wei Shi, and Gordon K Smyth. limma powers differential expression analyses for rna-sequencing and microarray studies. *Nucleic acids research*, 43(7):e47–e47, 2015.

Mark D Robinson, Davis J McCarthy, and Gordon K Smyth. edger: a bioconductor package for differential expression analysis of digital gene expression data. *bioinformatics*, 26(1):139–140, 2010.

Pamela H Russell, Rachel L Johnson, Shreyas Ananthan, Benjamin Harnke, and Nichole E Carlson. A large-scale analysis of bioinformatics code on github. *PloS one*, 13(10):e0205898, 2018.

Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten simple rules for reproducible computational research. *PLoS computational biology*, 9(10):e1003285, 2013.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Trans. Softw. Eng.*, 50(1):85–105, January 2024. ISSN 0098-5589. doi: 10.1109/TSE.2023.3334955. URL https://doi.org/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.

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.

Xiangru Tang, Bill Qian, Rick Gao, Jiakang Chen, Xinyun Chen, and Mark B Gerstein. Biocoder: a benchmark for bioinformatics code generation with large language models. *Bioinformatics*, 40 (Supplement_1):i266–i276, 2024.

Luke Tierney. codetools: Code analysis tools for r. *R package version 0.2-8, URL http://CRAN. R-project. org/package= codetools*, 2011.

Luke Tierney. A byte-code compiler for r. *Manuscript, Department of Statistics and Actuarial Science, University of Iowa. URL www. stat. uiowa. edu/˜ luke/R/compiler/compiler. pdf*, 2012.

Ana Trisovic, Matthew K Lau, Thomas Pasquier, and Mercè Crosas. A large-scale study on research code quality and execution. *Scientific Data*, 9(1):60, 2022.

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, pp. 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.

Kevin Ushey. Wickham h. renv: Project environments, 2023. *R package version*, 1(1), 2023.

Voyage AI. voyage-code-3: more accurate code retrieval with lower dimensional, quantized embeddings, December 2024. URL https://blog.voyageai.com/2024/12/04/voyage-code-3/.

Jianyou Andre Wang, Kaicheng Wang, Xiaoyue Wang, Prudhviraj Naidu, Leon Bergen, and Ramamohan Paturi. Scientific document retrieval using multi-level aspect-based queries. *Advances in Neural Information Processing Systems*, 36:38404–38419, 2023a.

Kaicheng Wang, Liyan Huang, Weike Fang, and Weihang Wang. Clarc: C/c++ benchmark for robust code search. *arXiv preprint arXiv:2603.04484*, 2026.

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.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. Codet5+: Open code large language models for code understanding and generation, 2023b. URL https://arxiv.org/abs/2305.07922.

Hadley Wickham. *Advanced r*. CRC press, 2015.

Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, et al. Welcome to the tidyverse. *Journal of open source software*, 4(43):1686, 2019.

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, pp. 1693–1703. ACM Press, 2018. doi: 10.1145/3178876.3186081. URL http://dx.doi.org/10.1145/3178876.3186081.

Noam Yefet, Uri Alon, and Eran Yahav. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

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.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, et al. Natural language to code generation in interactive data science notebooks. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 126–173, 2023.

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.

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.

Appendix

Table of Content

A Licenses **18**

A.1 Data Source Licenses 18

A.2 Model Licenses 18

B Compute Resources **18**

C Supplementary Discussion of the Dataset **18**

C.1 Training Set Statistics 18

C.2 Data Filtering for Bytecode/AST 18

C.3 Categorization and Relevance 18

C.4 Dataset Settings 19

D Model Details **19**

E Metric Definitions **20**

F Evaluation **21**

F.1 Full Evaluation Results on the Randomized Setting 21

F.2 Evaluation Results on Expert-Written Subsets 21

F.3 Deeper Analysis on Lowered Representations 22

F.4 Finetuning Details 22

F.5 Correlation Analysis on Model Robustness 23

G Error Analysis **23**

G.1 C1: Skeleton Collision (Randomized, Group 1, Nomic-emb-code). 23

G.2 C2: Accessor-literal Dependence (Neutralized, Group 2, OpenAI-text-embedding-3-large). 24

G.3 C3: Query-term Trap (Standard, Group 1, OpenAI-text-embedding-3-large). 24

G.4 C4: Helper Absorption (Neutralized, Group 3 Short, Voyage-code-3). 25

G.5 C5: Lowered-representation dilution (Bytecode, Group 1, Voyage-code-3). 25

H Practical Relevance of the Benchmark Settings **25**

I Industrial Relevance of R for Biomedical Data Science **26**

J Generalize to Other Programming Languages **26**

K Query Generation Prompts **27**

K.1 Prompt for G1 27

K.2 Prompt for G2 27
K.3 Prompt for G3 28
K.4 Prompt for Style Alignment 29

L Examples **29**
L.1 Example from Group 1 29
L.2 Example from Group 2 29
L.3 Example from Group 3 30

A LICENSES

A.1 DATA SOURCE LICENSES

Bioconductor and CRAN packages carry declared licenses (predominantly Artistic-2.0, GPL-2/3, and MIT), and GitHub repositories were required to declare an OSI-approved license. Because some licenses are restrictive (the GPL family), each sample is tagged with its license and we provide separate permissive and restrictive splits to support compliant downstream use. 71% of evaluation pairs fall in the permissive split.

A.2 MODEL LICENSES

- CodeT5+: BSD 3-Clause License[2].
- OASIS: MIT License[3].
- Nomic-emb-code: Apache-2.0[4].
- OpenAI text-embedding-3-large: users own the embeddings they generate under OpenAI's policies[5].
- Voyage-code-3: unclear; we do not redistribute its embeddings.

B COMPUTE RESOURCES

Prompt engineering, hypothesis testing, and query generation cost approximately $45 in API calls. Evaluation ran on a server with two 32-core AMD EPYC CPUs, 512 GiB of RAM, and one NVIDIA A100 80 GB GPU, for roughly 6 GPU hours; fine-tuning added 9 GPU hours.

C SUPPLEMENTARY DISCUSSION OF THE DATASET

C.1 TRAINING SET STATISTICS

The training set was built with the same pipeline from 99 repositories disjoint from the evaluation repositories and includes the standard, neutralized, and randomized settings (Tab. 7).

Table 7: **Statistics of the BRACE training set (averages).**

Category	# Pairs	# Tokens in Query	# Tokens in Code	LOC	CC
Training Set	5,310	86.1	205.7	24.3	3.9

C.2 DATA FILTERING FOR BYTECODE/AST

We excluded 23 and 41 pairs from the Bytecode and AST settings, respectively, where serialization exceeded a cap of 8,192 tokens or failed (e.g., functions embedding raw byte vectors). These exclusions are a small fraction of the evaluation set and do not affect the executability claims.

C.3 CATEGORIZATION AND RELEVANCE

Labeling strategies. Groups 1, 2, and 3 Long use *binary relevance*: the function (or merged snippet) a query was written for receives label 1 and all other candidates label 0. Group 3 Short uses *graded relevance*: the primary function receives label 2, its isolated helper functions label 1, and all others label 0.

[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

Impact on metrics. NDCG compares the model's ranking against the ideal ranking, in which label-2 snippets precede label-1 snippets, which precede label-0 snippets. MRR is computed strictly on the position of the label-2 snippet; helpers are treated as non-targets. Recall counts any non-zero label as relevant, so both primary and helper functions contribute.

C.4 Dataset Settings

Neutralized. Identifiers are renamed by syntactic role using a parse-tree walk: user-defined functions become `func_a`, `func_b`, ...; local variables `var_a`, ...; formal arguments `arg_a`, ...; classes `class_a`, ...; and slots or fields `slot_a`, ... Symbols exported by whitelisted packages are preserved. Data-accessor literals (`$`, `[[` with a string, `@`, and column references inside tidy-evaluation verbs) are renamed with the same mapping. Tab. 8 reports the ablation in which accessor literals are left intact: for every model, intact literals recover roughly half of the Group 2 loss, confirming that string-typed accessors are a major lexical channel in R.

Table 8: **Accessor-literal ablation on Group 2, Neutralized setting: identifiers masked with accessor literals masked (default) versus left intact.** Δ is the difference (intact − masked).

	NDCG			MRR		
Model	Masked	Intact	Δ	Masked	Intact	Δ
Group 2						
CodeT5+	14.8	24.9	+10.1	11.7	20.3	+8.6
Nomic	67.9	80.2	+12.3	61.8	75.6	+13.8
OASIS	70.3	81.6	+11.3	64.1	77.0	+12.9
OpenAI	39.6	63.7	+24.1	32.8	57.4	+24.6
Voyage	76.8	84.9	+8.1	72.3	81.5	+9.2

Randomized. The same positions are replaced by random alphanumeric strings of length 6-10; ten independent randomizations are generated and results averaged (Tab. 4). Complete results including standard errors are provided in Appx. F.1.

Bytecode. Functions are compiled with `compiler::cmpfun` at the default optimization level and disassembled with `compiler::disassemble`. We print one opcode per line with operand indices, followed by the constant pool; symbols in the pool that correspond to user identifiers are anonymized with the neutralized mapping.

Abstract Syntax Tree (AST). Functions are parsed with `parse(keep.source = FALSE)` and printed as nested S-expressions by a deterministic printer over the resulting language objects. Identifiers are anonymized with the neutralized mapping.

D 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 BRACE.

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.

E Metric Definitions

Let Q be the set of queries of a group and C the group's candidate pool, i.e., all snippets of that group in the evaluation set (for Group 3 Short, all primary and helper functions). For a query q, the model ranks C by cosine similarity between the query and code embeddings; $r_q(c)$ denotes the rank of snippet c and c_i the snippet at rank i. Relevance labels $\mathrm{rel}_q(c)$ follow Appx. C.3: binary (1 for the target snippet, 0 otherwise) in Groups 1, 2, and 3 Long, and graded in Group 3 Short (2 for the primary function, 1 for each of its helpers, 0 otherwise). $R_q = \{c : \mathrm{rel}_q(c) > 0\}$ is the set of relevant snippets and c_q^* the target snippet (the label-2 snippet in Group 3 Short). All metrics are averaged over Q and multiplied by 100; in the Randomized setting they are further averaged over the ten randomizations (Appx. F.1).

Normalized Discounted Cumulative Gain (NDCG). Discounted cumulative gain over the full ranking is

$$\mathrm{DCG}_q = \sum_{i=1}^{|C|} \frac{2^{\mathrm{rel}_q(c_i)} - 1}{\log_2(i+1)}, \qquad \mathrm{NDCG}_q = \frac{\mathrm{DCG}_q}{\mathrm{IDCG}_q}, \tag{1}$$

where IDCG_q is the DCG of the ideal ranking, which places label-2 snippets before label-1 snippets before label-0 snippets. With the gain $2^{\mathrm{rel}} - 1$, a helper contributes gain 1 and the primary function gain 3, so NDCG penalizes ranking a helper above its primary function.

Mean Reciprocal Rank (MRR). The reciprocal rank considers only the target snippet:

$$\mathrm{MRR} = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{r_q(c_q^*)}. \tag{2}$$

In Group 3 Short, helpers are treated as non-targets for this metric.

Mean Average Precision (MAP). Average precision is the mean of precision at the rank of each relevant snippet,

$$\mathrm{AP}_q = \frac{1}{|R_q|} \sum_{c \in R_q} \mathrm{P}_q@r_q(c), \qquad \mathrm{P}_q@k = \frac{|\{c \in R_q : r_q(c) \le k\}|}{k}, \tag{3}$$

and MAP is the mean of AP_q over Q. When $|R_q| = 1$, $\mathrm{AP}_q = 1/r_q(c_q^*)$, so MAP and MRR coincide; in Group 3 Short, R_q also contains the helpers, which is why MAP falls below MRR there.

Recall@k ($R@k$).

$$\mathrm{R}@k = \frac{1}{|Q|} \sum_{q \in Q} \frac{|\{c \in R_q : r_q(c) \le k\}|}{|R_q|}, \tag{4}$$

so that in Group 3 Short both the primary function and its helpers contribute to recall, whereas MRR counts the primary function alone.

F Evaluation

F.1 Full Evaluation Results on the Randomized Setting

As Tab. 9 shows, performance is stable across the ten randomizations, with standard errors below 2.0 for every metric.

Table 9: **Randomized setting: mean ± standard error over 10 trials. Bold** entries stand for the maximum values for the metrics in the category.

Model	NDCG	MRR	MAP	R@1	R@5
Group 1					
CodeT5+	22.4±1.05	17.5±1.12	18.9±1.08	10.8±1.30	25.6±1.71
Nomic	70.1±0.71	65.2±0.84	65.7±0.82	54.9±1.35	78.6±0.62
OASIS	77.8±0.31	73.6±0.39	73.9±0.38	64.2±0.67	85.1±0.55
OpenAI	60.3±0.78	54.2±0.97	54.8±0.95	42.3±1.42	69.7±0.66
Voyage	**79.5±0.48**	**75.7±0.61**	**75.9±0.60**	**66.8±1.09**	**86.9±0.57**
Group 2					
CodeT5+	10.6±0.49	8.1±0.46	9.4±0.47	4.3±0.44	12.0±0.98
Nomic	49.2±1.24	42.0±1.36	43.1±1.31	29.6±1.58	60.8±1.85
OASIS	62.1±0.80	54.9±0.88	55.4±0.86	41.3±1.37	73.6±1.46
OpenAI	28.7±0.69	24.1±0.81	25.6±0.79	16.2±1.10	34.5±1.29
Voyage	**70.4±0.59**	**64.2±0.70**	**64.6±0.69**	**51.7±1.21**	**81.2±0.97**
Group 3 Short					
CodeT5+	4.6±0.72	4.4±0.95	3.4±0.41	1.0±0.38	3.3±0.61
Nomic	17.0±0.74	18.9±1.03	11.8±0.49	6.4±0.58	12.5±0.56
OASIS	23.3±0.70	26.5±1.31	15.7±0.52	9.1±0.64	18.0±0.66
OpenAI	14.2±0.66	16.5±1.02	9.1±0.44	4.9±0.53	10.2±0.55
Voyage	**28.1±0.39**	**32.6±0.56**	**19.0±0.40**	**11.6±0.66**	**21.1±0.67**
Group 3 Long					
CodeT5+	5.6±0.80	4.0±0.71	5.5±0.69	1.6±0.77	7.1±1.49
Nomic	27.4±1.41	23.5±1.17	25.1±1.10	17.0±1.31	31.2±1.99
OASIS	31.9±0.70	27.8±0.81	29.4±0.78	20.8±1.23	36.1±0.88
OpenAI	30.6±0.77	26.2±1.04	27.5±1.09	18.4±1.58	36.5±1.75
Voyage	**62.9±0.53**	**57.8±0.75**	**58.6±0.77**	**47.3±1.60**	**71.5±0.98**

F.2 Evaluation Results on Expert-Written Subsets

The annotation study yielded 125 expert-written queries per group. This LLM-free subset lets researchers evaluate retrieval without any bias introduced by generative models. Tab. 10 compares performance on this subset with the full dataset. Voyage-code-3 is consistent across the two query

sources in all groups; CodeT5+ is slightly lower on the human set for Group 3 but aligns for Groups 1 and 2. Overall the expert-written and LLM-generated queries yield closely matching results.

Table 10: **Evaluation results on expert-written subsets versus the full dataset.**

Model	Queries	NDCG	MRR	R@1	R@5
Group 1					
CodeT5+	All queries	47.8	41.6	30.4	55.2
	Expert-written subset	48.6	40.2	28.8	53.6
Voyage	All queries	87.6	85.2	78.7	93.1
	Expert-written subset	89.4	87.0	80.8	94.4
Group 2					
CodeT5+	All queries	39.6	33.9	24.5	44.8
	Expert-written subset	40.5	35.1	26.4	44.0
Voyage	All queries	92.1	89.9	83.5	98.1
	Expert-written subset	91.8	89.4	82.4	98.4
Group 3 Short					
CodeT5+	All queries	30.2	33.5	9.6	22.8
	Expert-written subset	27.4	31.0	8.0	19.2
Voyage	All queries	64.3	78.4	25.9	48.8
	Expert-written subset	62.9	79.0	24.0	46.4
Group 3 Long					
CodeT5+	All queries	15.3	12.4	8.7	18.6
	Expert-written subset	10.9	8.6	5.6	13.6
Voyage	All queries	87.0	83.1	71.9	97.6
	Expert-written subset	86.7	82.8	70.4	99.2

F.3 Deeper Analysis on Lowered Representations

To understand degradation on lowered representations, we measured average lines, tokens, and unique opcodes or node types for the Standard, Bytecode, and AST forms of each group (Tab. 11). Lowering expands Group 1 functions from 98 to 688 tokens (bytecode) and 422 tokens (AST), a 7× and 4× increase, and Group 3 grows to 3,149 and 2,412 tokens. The number of unique opcodes or node types is three to four times the number of unique keywords in the source. The dense logic of a short R function is thus diluted across hundreds of low-level instructions, which plausibly overwhelms the models' ability to recognize semantic equivalence.

Table 11: **Comparison between Standard, Bytecode, and AST representations.**

	Group 1			Group 2			Group 3		
Metric	Std	Byc	AST	Std	Byc	AST	Std	Byc	AST
LOC	9.8	74.3	51.6	11.6	91.8	66.2	58.1	334.7	298.4
# Tokens	98.4	688.2	421.7	121.5	874.9	538.0	612.3	3,148.6	2,411.9
# Unique keywords / opcodes / nodes	4.6	18.2	12.4	4.4	17.1	11.9	6.3	23.7	14.8

F.4 Finetuning Details

We follow the contrastive paradigm of Gao et al. (2021). A batch contains N query-code pairs $(q_i, c_i)_{i=1}^{N}$; for query q_i, c_i is the positive and the remaining $N-1$ snippets are in-batch negatives. The InfoNCE loss for pair (q_i, c_i) is

$$\mathcal{L}i = -\log \frac{\exp\left(\mathrm{sim}(h_{q_i}, h_{c_i})/\tau\right)}{\sum_{j=1}^{N} \exp\left(\mathrm{sim}(h_{q_i}, h_{c_j})/\tau\right)}, \tag{5}$$

where h_{q_i} and h_{c_i} are the query and code embeddings, $\mathrm{sim}(\cdot,\cdot)$ is cosine similarity, and τ is a temperature. We use $\tau = 0.05$, a batch size of 64, learning rates of 2×10^{-5} (CodeT5+) and 1×10^{-5} (OASIS), and five epochs for all settings.

F.5 Correlation Analysis on Model Robustness

To identify determinants of robustness, we measure the L_2 distance between the embedding of the original code and that of its neutralized or randomized variant, and correlate this shift with lines of code, cyclomatic complexity, and the *perturbation fraction* (the share of the snippet's characters occupied by modified identifiers). As Tab. 12 shows, structural features correlate weakly with embedding shift ($|r|$ mostly below 0.35), so code length and complexity are not primary drivers of instability. The perturbation fraction, by contrast, correlates positively and consistently across models and groups (0.22-0.76), supporting the hypothesis that embeddings are driven primarily by lexical overlap: as a larger share of characters changes, even with logic preserved, embeddings diverge substantially.

Table 12: **Correlation between embedding shift distance and code features.**

Model	Group 1		Group 2		Group 3	
	Neu	Ran	Neu	Ran	Neu	Ran
Lines of Code						
CodeT5+	−0.12	−0.14	−0.10	−0.19	−0.03	0.07
OASIS	−0.15	0.06	−0.25	−0.02	0.05	−0.27
Nomic	−0.18	−0.22	−0.29	−0.11	−0.08	−0.31
Voyage	−0.30	−0.32	−0.33	−0.37	−0.28	−0.26
Cyclomatic Complexity						
CodeT5+	0.05	−0.04	−0.11	−0.10	−0.14	−0.07
OASIS	−0.08	−0.11	−0.12	0.13	0.02	0.07
Nomic	−0.02	−0.21	−0.16	0.01	−0.11	−0.04
Voyage	−0.20	−0.18	−0.17	−0.19	−0.22	−0.18
Perturbation Fraction						
CodeT5+	0.36	0.53	0.44	0.55	0.39	0.22
OASIS	0.74	0.37	0.62	0.65	0.40	0.38
Nomic	0.47	0.63	0.66	0.70	0.54	0.55
Voyage	0.76	0.75	0.68	0.72	0.63	0.66

G Error Analysis

Protocol. For Voyage-code-3 and OASIS we sampled 50 failed queries per setting (gold snippet ranked below 5) from the standard, neutralized, randomized, and bytecode settings, 400 cases in total. Authors independently inspected each query, its gold snippet, and the top-three retrieved snippets, and assigned one primary failure category (Cohen's $\kappa = 0.71$); disagreements were resolved by discussion. Fig. 2 gives the distribution, and representative cases follow, shown with the identifiers the model actually saw.

G.1 C1: Skeleton Collision (Randomized, Group 1, Nomic-emb-code).

Query: *"The function takes a numeric matrix of raw read counts with features in rows and samples in columns, divides every entry by the total count of its sample, and scales the result by one million."* The gold function was ranked 3; rank 1 was a transcripts-per-million implementation from another repository.

```
# Gold (rank 3), as seen by the model
qz81 <- function(vk3) {
  t(t(vk3) / colSums(vk3)) * 1e6
}
# Retrieved at rank 1
hw2m <- function(vk3, p7d) {
  r4 <- vk3 / (p7d / 1e3)
  t(t(r4) / colSums(r4)) * 1e6
}
```

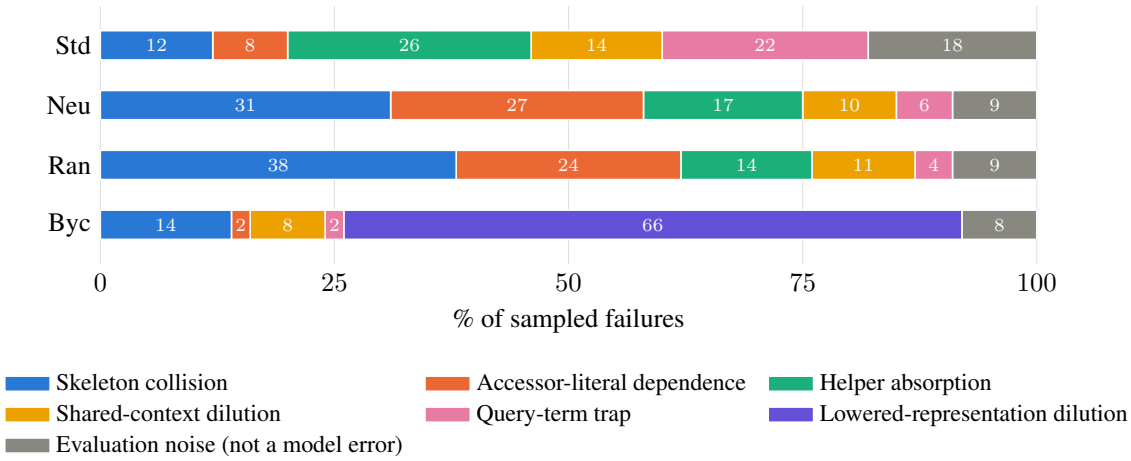


Figure 2: Distribution of primary failure categories among sampled failures, as a share of cases per setting. Segments below 5% are unlabeled.

With names removed, both snippets reduce to the same multiset of whitelisted calls (`t`, `colSums`, division, `1e6`); the only discriminating content is the length-normalization line, which the embedding does not weight enough to separate the two. In the standard setting the identifiers `cpm` and `tpm` resolve the tie, which is exactly the lexical shortcut the setting removes.

G.2 C2: Accessor-literal Dependence (Neutralized, Group 2, OpenAI-text-embedding-3-large).

Query: *"The function keeps the rows of a differential-expression result table whose adjusted p-value is known and falls below a supplied threshold."* Rank 1 was a filter on absolute log fold change; the gold function was ranked 4.

```
# Gold (rank 4)
func_a <- function(arg_a, arg_b = 0.05) {
  arg_a[!is.na(arg_a$var_a) & arg_a$var_a < arg_b, ]
}
# Retrieved at rank 1
func_a <- function(arg_a, arg_b = 1) {
  arg_a[!is.na(arg_a$var_a) & abs(arg_a$var_a) > arg_b, ]
}
```

Once the column literals (`padj` versus `log2FoldChange`) are masked, the snippets differ by one call to `abs` and the direction of a comparison. In the literals-intact ablation the gold snippet is ranked first, confirming that the column name was carrying the semantic distinction; the query's "adjusted p-value" has no other anchor in the code.

G.3 C3: Query-term Trap (Standard, Group 1, OpenAI-text-embedding-3-large).

Query: *"The function performs library-size normalization of a count matrix…"* Rank 1 was a setup function whose body consists of `library()` calls.

```
# Retrieved at rank 1
load_deps <- function() {
  suppressPackageStartupMessages({
    library(edgeR); library(limma); library(dplyr)
  })
  invisible(TRUE)
}
```

The general-purpose embedder matches the query token "library" against repeated `library` calls, an R-specific collision between a domain term and a language keyword. The case shows that lexical shortcuts operate even in the standard setting, and it motivates two pipeline changes we adopted after this analysis: setup functions with no data-dependent logic are filtered out, and the style-alignment prompt is instructed to prefer "sequencing-depth normalization" where a domain term coincides with an R keyword.

G.4 C4: Helper Absorption (Neutralized, Group 3 Short, Voyage-code-3).

Query: *"The function identifies cells whose reads come disproportionately from mitochondrial genes... and returns the names of the cells whose share exceeds the threshold."* The helper that computes the per-cell fraction was ranked 1 and the primary function 2.

```r
# Retrieved at rank 1 (helper, label 1)
func_b <- function(arg_a, arg_b) {
  colSums(arg_a[arg_b, , drop = FALSE]) / colSums(arg_a)
}
# Gold primary (rank 2, label 2)
func_a <- function(arg_a, arg_b, arg_c = 0.2) {
  var_a <- func_b(arg_a, arg_b)
  names(var_a)[var_a > arg_c]
}
```

The description's semantic content (mitochondrial fraction) lives almost entirely in the helper, while the primary function is a thin wrapper whose only distinctive step is thresholding. Graded relevance awards partial credit through NDCG, but MRR counts only the primary, which is why Group 3 Short MRR falls furthest under anonymization. In Group 3 Long the merged snippet is retrieved at rank 1.

G.5 C5: Lowered-representation dilution (Bytecode, Group 1, Voyage-code-3).

Query: *"The function centers every row of a numeric matrix on its mean and divides by its standard deviation, substituting one for zero standard deviations."* Rank 1 was a row-wise min–max scaler; the gold function was ranked 9. The two functions compile to 71 and 68 opcode lines with near-identical control flow (a `rowMeans` call, an `apply` over rows, a guard, and an arithmetic epilogue); they differ only in which symbols appear in the constant pool (`sd` versus `min` and `max`), three tokens among roughly seven hundred. The signal that separates the functions is present but diluted by two orders of magnitude relative to the source, matching the token expansion reported in Appx. F.3.

Evaluation noise. A minority of failures (8-18% depending on setting) are not model errors: the top-ranked snippet is a near-duplicate or a semantically equivalent implementation from another repository that survived parse-tree deduplication because its identifiers or argument order differ. We report these cases rather than relabeling them, since the query does describe both snippets; a semantic-duplicate audit of the corpus is left to future work, and the affected fraction bounds the noise in the absolute scores.

Implications. Categories 1, 2, and 5 are all instances of the same mechanism: when identifiers, accessor literals, or source-level structure are removed, the remaining discriminating information is a small number of tokens that current embeddings underweight relative to the shared skeleton. This suggests two directions for robust code search: training with anonymized views of the same function as positives, and mining hard negatives that share a function's whitelisted-call skeleton but differ in one operation, which is precisely the contrast these models fail to make.

H Practical Relevance of the Benchmark Settings

The anonymized settings are not artificial perturbations of otherwise well-named code; they model three situations a biomedical code search system meets routinely. First, the same analysis step is written against different tools' output conventions (`padj`, `FDR`, `adj.P.Val`), so a retriever that anchors on one literal misses equivalent implementations; masking accessor literals measures how

much retrieval depends on such anchors. Second, much biomedical R code is analysis code rather than library code: scripts shared alongside publications are frequently idiosyncratic and often do not even execute (Trisovic et al., 2022), identifiers are commonly placeholders (`df1`, `tmp`, `res2`) or written in the authors' first language, and a growing share is generated by graphical front ends and LLM assistants whose naming carries little information. Third, identifiers are the primary attack surface in adversarial attacks on code models (Yefet et al., 2020; Hort et al., 2025; Qu et al., 2024); the neutralized and randomized settings therefore double as a robustness stress test of whether semantic understanding survives when informative names are unavailable. The stakes are higher than in general-purpose code search because retrieval increasingly feeds retrieval-augmented coding assistants (Lewis et al., 2020; Chen et al., 2025): a retrieved function that matches on names but not on behavior becomes a wrong normalization or a wrong filter in a downstream analysis. Finally, the settings form a ladder of abstraction. Neutralized code preserves structure and syntactic role, randomized code preserves structure only, and the bytecode and AST settings remove the source-level syntax as well; bytecode in particular is not hypothetical, since every installed R package has contained byte-compiled closures since R 3.5 (Tierney, 2012).

I Industrial Relevance of R for Biomedical Data Science

While Python dominates general-purpose machine learning and JavaScript the web, R remains the working language of biomedical statistics and bioinformatics. Bioconductor alone ships more than two thousand software packages for genomic, transcriptomic, proteomic, and clinical data analysis; the most widely used tools for differential expression and single-cell analysis—DESeq2, edgeR, limma, and Seurat—are R packages with tens of thousands of citations each (Love et al., 2014; Robinson et al., 2010; Ritchie et al., 2015; Hao et al., 2021a); a large-scale analysis of bioinformatics code on GitHub found R among the most frequently used languages in the field (Russell et al., 2018); and R is now accepted in regulatory clinical-trial submissions, following the R Consortium's pilot submissions to the U.S. FDA (R Consortium, 2026). Despite this footprint, R is absent from every text-to-code retrieval benchmark we are aware of, from CodeSearchNet to COIR, a gap we highlight in the introduction of our paper.

R also offers properties that make it a particularly informative testbed for robustness. First, it exposes two native lowered representations of every function: byte-compiled code, which is what installed packages actually contain since R 3.5, and S-expression parse trees, which are R's internal representation of programs. These allow robustness studies across representations without a cross-language compiler, a capability not directly available for Python or JavaScript. Second, R's mixture of class systems (S3, S4, R6, and Bioconductor's container classes) and its string-typed data accessors create dependency-complexity and lexical axes that existing benchmarks do not cover: the same analysis step is written against different tools' output conventions (`padj`, `FDR`, `adj.P.Val`), so a retriever that keys on one literal misses equivalent implementations—a daily search problem for practitioners rather than an artificial perturbation.

Furthermore, the construction pipeline in our work, can serve as a template for other languages with strong domain footprints, such as Julia in scientific computing or SAS and Stata in clinical statistics. The port requires a parser and an executability check for the target language, followed by the same annotation and hypothesis-testing procedure; we quantify this effort in § 3.

J Generalize to Other Programming Languages

We focused on R for two main reasons. First, high-quality code search benchmarks for R are rare. Second, R code can be readily converted to other lowered representations, such as Bytecode and AST, which enables further research directions.

Meanwhile, our pipeline is fundamentally language-agnostic. Extending it to Python, Java, C#, C/C++, or other languages require only (1) sampling new, representative code snippets for the target language and (2) conducting the corresponding annotation and hypothesis testing.

K Query Generation Prompts

K.1 Prompt for G1

Prompt K.1: Group 1

```
You are an expert bioinformatician. Please write a summary for the
following R 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. You should also state the types
of the input and output objects (for example, a numeric matrix with
features in rows, a data frame, a logical vector) while avoiding
mentioning the variable names, argument names, or the function name.
Describe the computation in biomedical or statistical terms rather
than by naming R functions, and do not mention package names or any
string literals such as column names that appear in the code.
```

K.2 Prompt for G2

Prompt K.2: Group 2

```
You are an expert bioinformatician. Please analyze the following R
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 classes of the input and output objects (as
defined in the provided class definitions or slot signatures) without
mentioning any variable names, slot or field names, class names,
package names, or the function name. Describe a custom object by what
it holds (for example, "a container with a feature-by-sample count
matrix and per-sample metadata").
Here are the definitions of the custom classes used in the function
(S3 constructors, S4 setClass calls, R6 classes, or the slot
signatures and documentation of imported container classes):
{class_definitions}
Here is the function:
{function_text}
Instructions:
1. Read and understand the function step by step, using the class
definitions to work out what each object contains.
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 classes by their
contents, while avoiding any reference to variable names, slot names,
```

```
class names, package names, function names, or too many implementation
details.
```

K.3 PROMPT FOR G3

Prompt K.3: Group 3

```
You are an expert bioinformatician.  Generate a high-quality
description for the following R function based on the provided
guidelines.  Focus on summarizing the function's purpose and behavior
without reproducing the code or referencing internal variable or
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.  Do not describe the helpers separately
and do not name them.  This is a critical requirement.
  * Explicitly mention the input and output types of the function,
while avoiding specific variable names, function names, or too many
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 any function, variable, or argument names defined
in the snippet, nor any package names or string literals such as
column names.
  * You may name built-in R types (for example, numeric vector,
matrix, data frame, list).  Describe objects of custom or container
classes by what they hold rather than by their class name.
  * Describe the computation in biomedical or statistical terms rather
than by naming R functions.
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.
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:
{helper_functions}
Function to Summarize:
{function_text}
Provide the description as plain text, following the guidelines
strictly.  At last, output your summary on a new line starting with
"SUMMARY:".  Please note that in your final summary, you should not
```

```
consider the background of the function, and only focus on the
functionality.  You should also mention the types of the input and
output objects while avoiding the variable names in your final
summary.
Here are some examples of how the final summary should look like:
{few_shot_examples}
```

K.4 Prompt for Style Alignment

Prompt K.4: Style alignment

```
You are an expert bioinformatician.  Your task is to rewrite a summary
for a function in R so that the revised summary is in the same format
as the provided examples.  Remember, the revised summary should not
include specific function names, variable or argument names, class
names, package names, or string literals such as column names;
describe objects and data fields by what they contain instead.
Here are the example summaries:
Example Summary 1:
{few_shot_example_1}
Example Summary 2:
{few_shot_example_2}
Example Summary 3:
{few_shot_example_3}
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 is helpful to make the format of
your revised summary more similar to the examples, but only details
that are verifiable in the original function.
```

L Examples

L.1 Example from Group 1

Query.

The function takes a numeric matrix of raw read counts with features in rows and samples in columns. It divides every entry by the total count of its sample and scales the result by one million, producing a counts-per-million matrix of the same dimensions as the input.

Code Snippet.

```r
cpm <- function(counts) {
  t(t(counts) / colSums(counts)) * 1e6
}
```

L.2 Example from Group 2

Query.

The function accepts a container object holding a matrix of raw read counts together with a minimum count and a minimum number of samples. It keeps only those features whose count reaches the minimum in at least the required number of samples and returns the subsetted container with all associated row and column metadata preserved.

Code Snippet.

```r
drop_low_counts <- function(se, min_count = 10, min_samples = 3) {
```

```
keep <- rowSums(assay(se, "counts") >= min_count) >= min_samples
se[keep, ]
}
```

L.3 Example from Group 3

Query.

The function identifies cells whose reads come disproportionately from mitochondrial genes. Given a count matrix, a logical vector marking the mitochondrial features, and a fraction threshold, it computes each cell's share of counts attributable to the marked features and returns the names of the cells whose share exceeds the threshold.

Code Snippet.

```
.mito_fraction <- function(counts, is_mito) {
colSums(counts[is_mito, , drop = FALSE]) / colSums(counts)
}

flag_high_mito <- function(counts, is_mito, max_frac = 0.2) {
frac <- .mito_fraction(counts, is_mito)
names(frac)[frac > max_frac]
}
```