Beyond Function-Level Search: Repository-Aware Dual-Encoder Code Retrieval with Adversarial Verification

Aofan Liu[1,2], Shiyuan Song[1], Haoxuan Li[3], Cehao Yang[1], and Yiyan Qi ✉[1]

[1]International Digital Economy Academy (IDEA)
[2]School of Electronic and Computer Engineering, Peking University
[3]Shenzhen International Graduate School, Tsinghua University

Abstract

The escalating complexity of modern codebases has intensified the need for code retrieval systems capable of interpreting cross-component change intents—a capability fundamentally absent in conventional function-level search paradigms. While recent research has improved alignment between queries and code snippets, retrieving contextually relevant code for certain *change request* remains underexplored. ❶ To bridge this gap, we present **RepoAlign-Bench**, the first benchmark designed to evaluate repository-level code retrieval for change request-driven scenarios, encompassing 52k columns. The benchmark shifts the paradigm from function-centric retrieval to holistic repository analysis. ❷ In addition, we propose **ReflectCode**, an adversarial reflection-augmented dual-tower architecture featuring disentangled `code_encoder` and `doc_encoder` towers. Our framework dynamically integrates syntactic patterns, function dependency, and semantic expansion intent through LLM. ❸ Comprehensive evaluations demonstrate that ReflectCode achieves 12.2% Top-5 Accuracy and 7.1% Recall improvements over state-of-the-art baselines.

1 Introduction

Deep learning has revolutionized software engineering by advancing code representation learning, allowing neural models to comprehend programming constructs with an unprecedented level of sophistication (Alon et al., 2019). While current code generation systems leverage massive GPU clusters and trillion-token corpora (Allal et al., 2023b), their effectiveness in real-world software maintenance scenarios remains constrained by a critical bottleneck: the inability to retrieve contextually relevant code segments spanning multiple components for implementing complex change requests.

Limitation The fundamental limitation stems from prevailing function-centric paradigms that treat code artifacts as isolated units, ignoring the intricate web of cross-component dependencies inherent in modern software architectures. Traditional retrieval methods relying on lexical matching (Zhang et al., 2023) or shallow syntactic analysis fail to capture the semantic relationships between distributed code elements required for implementing coherent modifications. This mismatch becomes particularly acute when developers need to (1) propagate API changes across class hierarchies, (2) maintain invariant relationships between distributed components, or (3) adapt multiple interdependent functions to new requirements—scenarios that constitute a significant portion of maintenance efforts.

Motivation Current evaluation frameworks critically limit progress by narrowly focusing on single-function retrieval tasks. To enhance the practical utility of AI tools, evaluation criteria must be broadened to include capabilities for managing cross-component dependencies. This also implies that retrieval systems should move beyond function-level retrieval and support repository-level comprehension, ensuring consistency and semantic coherence within the system architecture, especially when code changes occur.

Benchmark To address these critical gaps, we present **RepoAlign-Bench**—a repository-level change-oriented benchmark for evaluating code retrieval systems on change request fulfillment, explicitly designed to model cross-component dependencies and structural relationships inherent in real-world software modifications. RepoAlign-Bench encompasses a diverse set of real-world scenarios, enabling assessment of models' ability to understand and act upon complex user intents. By providing this change-oriented framework, RepoAlign-Bench facilitates the comparison of different retrieval approaches and fosters the development of more robust and accurate code retrieval systems.

Model In addition to the benchmark, we

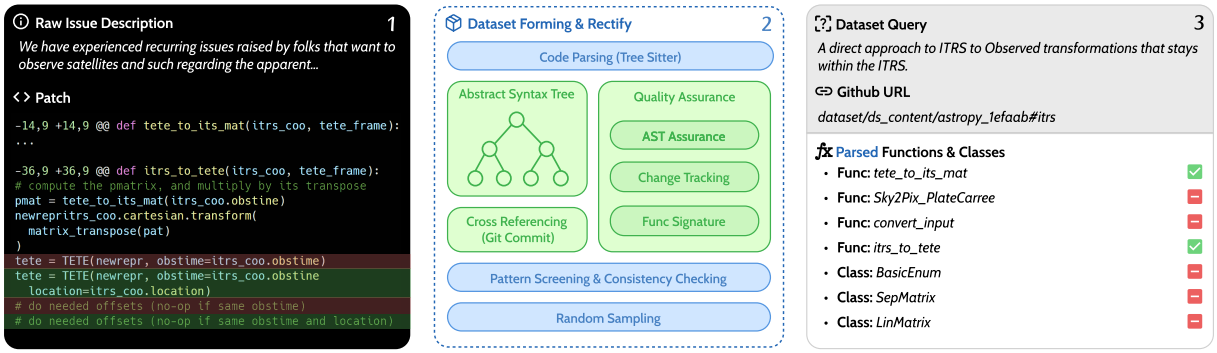

Figure 1: Visualization of a code patch in Astropy's modeling module addressing an issue with ITRS. The image highlights the updated implementation of the `tete_to_itrs_mat` and `itrs_to_tete` functions, alongside a description of the issue, the corresponding patch, and parsed information such as functions and classes.

also propose an adversarial reflection-based dual-tower model with separate `code_encoder` and `doc_encoder` components, augmented by contextual information from large language models (LLMs). This architecture efficiently captures semantic similarities between code and change-oriented queries, enhancing intent understanding and retrieval accuracy. By integrating contextual information from LLMs, our model enhances the understanding of user intents and the nuanced relationships between code segments, thereby improving retrieval precision and recall.

Experiment Our experiments demonstrate that the proposed dual-tower model significantly outperforms state-of-the-art models such as CodeBERT (Feng et al., 2020), SantaCoder (Allal et al., 2023b), and CodeT5 (Wang et al., 2021) in key evaluation metrics including precision, recall, and F1 score. These results highlight the effectiveness of our approach in accurately locating relevant functions within large-scale repositories based on user change requests. Ablation study quantifies the contributions of core components, revealing that controlled parameter independence and dynamic negative mining are critical for robust cross-modal alignment.

Our contributions are summarized as follows:

1. **RepoAlign-Bench Dataset**: We present RepoAlign-Bench, a standardized benchmark tailored for evaluating repository-level code retrieval based on user change requests, enabling consistent and comprehensive performance assessment of retrieval models.

2. **Dual-Tower Retrieval Model**: We propose a reflection-based dual-tower model comprising distinct `code_encoder` and `doc_encoder` components, enhanced with contextual information from large language models to improve semantic matching between queries and code snippets.

3. **Empirical Evaluation**: We validate our model on RepoAlign-Bench, demonstrating superior performance over existing state-of-the-art models, thereby establishing a new benchmark for code retrieval tasks.

2 The RepoAlign-Bench Dataset

In this section, we introduce our semi-automated annotation framework for RepoAlign-Bench construction. Fig. 1 illustrates the dataset construction process, which consists of three main stages:

Stage 1: Project Selection and Initial Filtering

Our benchmark construction begins with a systematic curation of high-quality open-source projects, incorporating SWE-Bench (Jimenez et al., 2024), Py150(Kanade et al.) as data sources for preliminary screening. This phase employs a two-tier validation strategy that combines automated filtering with data verification to ensure robust Query-Code correspondences.

The pipeline first processes candidate GitHub pull requests (PRs) through PyLint static analysis framework (Thenault et al., 2024), which enforces automated quality gates to validate PR-issue linkage based on commit message patterns, analyze code diffs for cross-component modifications, and filter non-trivial changes using cyclomatic complexity thresholds (McCabe, 1976).

Stage 2: Structural Code Extraction

This stage systematically constructs a dataset through structural code extraction and commit cor-

Repo	PLM	PLX	PLN	PLS	PrLM	PrLX	PrLN	PrLS
astropy/astropy	2502.09	13884	470	3670.94	2510.73	7910	162	1841.03
django/django	1418.53	10818	356	1575.29	1331.68	9252	146	1272.11
matplotlib/matplotlib	1228.68	5178	421	1055.78	2287.76	10176	395	2175.19
mwaskom/seaborn	1883.00	2235	1531	497.80	1313.00	1438	1188	176.78
psf/requests	633.25	863	388	167.80	1658.75	7476	271	2383.21
pydata/xarray	1705.77	8857	422	1969.75	2664.68	9276	703	1804.07
pylint-dev/pylint	2072.90	6862	417	1986.49	3814.70	24770	618	7429.63
pytest-dev/pytest	1694.32	9824	432	2136.19	3364.37	22778	451	4955.51
scikit-learn/scikit-learn	1743.47	13568	314	2363.62	2589.19	7387	158	1832.84
sphinx-doc/sphinx	1821.84	10055	501	1911.67	1665.95	5362	358	1122.18
sympy/sympy	1780.63	17385	277	2930.55	1017.19	4361	143	831.73

Table 1: Repository Statistics. PLM: Patch Length Mean, PLX: Patch Length Max, PLN: Patch Length Min, PLS: Patch Length Std, PrLM: Problem Length Mean, PrLX: Problem Length Max, PrLN: Problem Length Min, PrLS: Problem Length Std.

relation. We first parse the GitHub repository using Tree-sitter (Brunsfeld), a multi-language parsing infrastructure that generates precise abstract syntax trees (ASTs).

Following structural extraction, we cross-reference these artifacts with their associated Git commits using a three-phase alignment process: ① differential analysis of commit histories to identify code modifications addressing PR requirements, ② syntactic pattern matching between AST nodes and commit diffs, and ③ temporal mapping of code evolution sequences. The parsing pipeline employs Tree-sitter's hybrid scanning strategy that combines regular expressions for tokenization with context-aware grammars for structural disambiguation. Some Statistics about patch can be found in the Table 1.

Tree-sitter Integration: Our architecture leverages Tree-sitter's incremental parsing through three strategic adaptations: 1) Partial AST regeneration for code diffs using its edit-script API, 2) Syntax-aware pattern recognition for cross-version change tracking, and 3) Language-independent query DSL for cross-component dependency analysis.

Stage 3: Hierarchical Data Validation

Then we enforces a three-tiered quality assurance protocol integrating automated filtering, semantic verification, and expert validation. The refined dataset then undergoes **Cross-Modal Consistency Checking** - a hybrid framework combining pattern recognition with consensus validation.

Our verification pipeline employs: ① *Pattern-based Screening* using dependency graph analysis; ② *Consensus Validation* achieving Fleiss' $\kappa = 0.82$ agreement. The final distribution preserves intentional asymmetries reflecting real-world software evolution patterns.

Rationale for Controlled Imbalance: The residual skewness (1) mirrors natural developer behavior where certain change types dominate (e.g., 23% of valid PRs address compatibility in our corpus), (2) prevents over-smoothing of critical but infrequent patterns like security patches (which account for only 4.2% of the dataset but need to be preserved), and (3) maintains dependency structure integrity that uniform sampling would disrupt. Our oversampling with abstract syntax tree based augmentation specifically targets *harmful imbalance* (Multitask Contamination/Context Drift) while preserving *domain-inherent skewness* essential for generalizable model training.

TIERED REPOALIGN-BENCH	
Full Corpus	(52k, 100%)
Direct matches Single-function impl.	
Challenge Subset	(31k, 60%)
Context-aware Metaphorical queries	
Expert Subset	(8k, 15%)
Cross-functional Implicit constraints	

Dataset Stratification. We stratify our benchmark into three distinct difficulty tiers using a multi-criteria annotation process. This action primarily accounts for the long-tail distribution of the data and divides it based on both the actual difficulty distribution and the difficulty ratings across orthogonal dimensions. For further details, please refer to Appendix C

3 ReflectCode

Modern retrieval-augmented code generation systems face ongoing challenges in maintaining se-

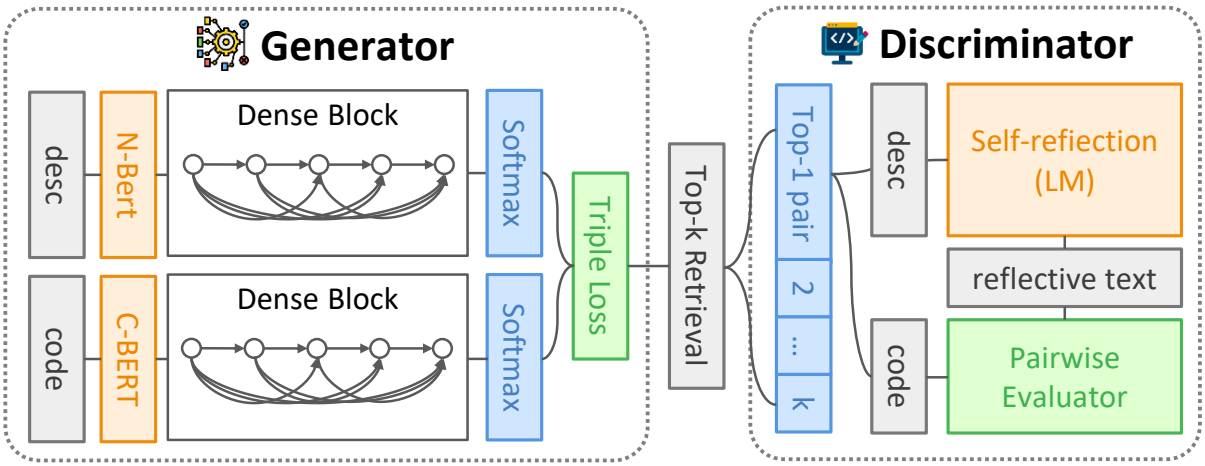

Figure 2: The model consists of a generator and discriminator. The generator includes separate encoders for code and documentation, followed by dense blocks, softmax, and a retrieval mechanism for top-k matching. The discriminator incorporates **self-reflection** and a **pairwise evaluator** to refine the model's output based on reflective text.

mantic consistency when dealing with complex codebases that involve dependency graphs and long-context requirements. In large repositories, the entanglement of code structure and natural language semantics makes traditional single-vector embedding methods struggle to preserve retrieval accuracy under such conditions. As a result, solving the issue of retrieving relevant code snippets from the repository becomes crucial.

Although existing approaches have made progress through structural analysis (e.g., AST parsing, control flow modeling) and infrastructure optimization (e.g., hierarchical indexing, pipeline formalization), they exhibit fundamental limitations in cross-component dependency resolution and dynamic context adaptation—key capabilities for maintaining code at the repository level. An analysis of popular paradigms is provided in Appendix A.2.

Proposed Model. To address the repository level code retrieval challenge, we propose ReflectCode, a reflection-augmented dual-tower architecture featuring disentangled modality encoders. Our framework employs a dual-encoder paradigm with separate CodeBERT-based towers for code and natural language processing, specifically designed to preserve structural and semantic integrity across modalities. The code encoder processes syntactic patterns and dependency, while the text encoder incorporates LLM-generated contextual reasoning to capture implicit cross-component dependencies. The complete system architecture and data flow relationships are comprehensively illustrated in Figure 2.

The modality alignment is achieved via a triplet margin loss with hard negative mining, optimizing the latent space for fine-grained semantic correspondence between code segments and change intents.

Beyond static embedding alignment, we introduce a dynamic adversarial verification mechanism where an LLM-powered discriminator evaluates top-k candidates retrieved through cosine similarity search. When the discriminator detects semantic incongruence (confidence $< \tau$), the system triggers an iterative refinement process: The generator dynamically recalibrates embeddings using attention-based probability redistribution, while the discriminator performs multi-hop reasoning over dependency-aware code representations.

Twin-Tower Architecture. Our framework implements a **parameter-shared dual-encoder architecture** with modality-specific processing streams. We instantiate two CodeBERT-based encoders:

- **C-BERT**: Processes code syntax through structural-aware tokenization with enhanced graph positional encoding
- **N-BERT**: Handles natural language queries using semantic-focused parsing with type-constrained attention

The architecture satisfies two fundamental design criteria through partial parameter sharing: The

Algorithm 1 Dual-Encoder Training Protocol

Initialize $\theta_{\text{shared}} \sim \mathcal{N}(0, 0.02)$
Freeze pretrained embeddings $\phi_{\text{code}}, \phi_{\text{text}}$
for $epoch = 1$ **to** N **do**
  Batch (c_i, q_i, q_j^-) $\triangleright$ q_j^-: hard negatives
  Compute $\mathbf{h}_c = \mathcal{E}_{\text{code}}(c_i)$
  Compute $\mathbf{h}_q^+ = \mathcal{E}_{\text{text}}(q_i)$
  Compute $\mathbf{h}_q^- = \mathcal{E}_{\text{text}}(q_j^-)$
  $\mathcal{L}_{\text{triplet}} = \max\Big(0, \delta(\mathbf{h}_c, \mathbf{h}_q^+) - \delta(\mathbf{h}_c, \mathbf{h}_q^-) + \alpha\Big)$
  Update $\theta_{\text{shared}} \leftarrow \theta_{\text{shared}} - \eta \nabla_{\theta_{\text{shared}}} \mathcal{L}_{\text{triplet}}$
end for

architecture satisfies two fundamental design criteria through partial parameter sharing:

$$\theta_{\text{code}} \cap \theta_{\text{text}} \stackrel{\text{def}}{=} \theta_{\text{shared}} \tag{1}$$

$$\underbrace{\mathcal{E}_{\text{code}} \neq \mathcal{E}_{\text{text}}}_{\text{disjoint embeddings}} \quad \text{with} \quad \phi_{\text{code}} \oplus \phi_{\text{text}} = \phi_{\text{total}} \tag{2}$$

where θ_{shared} denotes shared transformer parameters, and ϕ represents modality-specific embedding layers. Formally, given a code snippet c and textual query q, the encoders produce d-dimensional representations:

$$\begin{aligned} \mathbf{h}_c &= \mathcal{E}_{\text{code}}\left(c; \{\theta_{\text{shared}}, \phi_{\text{code}}\}\right), \\ \mathbf{h}_q &= \mathcal{E}_{\text{text}}\left(q; \{\theta_{\text{shared}}, \phi_{\text{text}}\}\right) \end{aligned} \tag{3}$$

Cross-Modal Alignment Objective

To establish geometrically consistent representations across modalities while preserving their distinctive features, we formulate a **adaptive-margin triplet loss** with dynamic hard negative mining. Given an anchor query q, its corresponding positive code snippet c^+, and k hard negative samples $\{c_i^-\}_{i=1}^k$ mined through syntactic similarity analysis, the loss function is defined as:

$$\mathcal{L}_{\text{align}} = \frac{1}{k}\sum_{i=1}^{k} \max\left(0, \underbrace{\|\mathbf{h}_q - \mathbf{h}_{c^+}\|_2^2}_{\text{positive pair}} - \underbrace{\|\mathbf{h}_q - \mathbf{h}_{c_i^-}\|_2^2}_{\text{negative pair}} + \alpha(\mathbf{h}_q, \mathbf{h}_{c_i^-})\right) \tag{4}$$

where:

- $\mathbf{h}_q = \mathcal{E}_{\text{text}}(q)$, $\mathbf{h}_{c^+} = \mathcal{E}_{\text{code}}(c^+)$ denote the normalized embeddings
- $\alpha(\cdot)$ implements our *edge-aware margin* mechanism:

$$\alpha(\mathbf{h}_q, \mathbf{h}_{c^-}) = \alpha_0 + \beta \cdot \sigma\left(\mathbf{h}_q^\top \mathbf{h}_{c^-}\right)$$

  with $\alpha_0 = 0.2$ as base margin, $\beta = 0.5$ scaling factor, and σ the sigmoid function

This design introduces three critical enhancements over standard triplet loss:

1. **Dynamic Margin Adaptation**: Automatically adjusts penalty intensity based on negative sample difficulty
2. **Batch-Aware Hard Negatives**: Selects $k = 5$ most challenging negatives per anchor using code clone detection heuristics
3. **Modality-Invariant Normalization**: Enforces $\|\mathbf{h}\|_2 = 1$ through projection layers to stabilize angular comparisons

Adversarial Search with Dynamic Feedback

Our **Adversarial Search Mechanism (ASM)** establishes a closed-loop interaction between retrieval generation and semantic verification through three core components:

Generator: Context-Aware Retrieval The generator $\mathcal{G}$ employs our dual-encoder model to perform *density-aware similarity search*:

$$s(c, q) = \frac{\exp(\tau \cdot \cos(\mathbf{h}_c, \mathbf{h}_q))}{\sum_{c' \in \mathcal{C}} \exp(\tau \cdot \cos(\mathbf{h}_{c'}, \mathbf{h}_q))} \tag{5}$$

where $\tau = 10$ sharpens the probability distribution. The top-k candidates $\mathcal{C}_{\text{gen}}^{(t)}$ at step t are selected via:

$$\mathcal{C}_{\text{gen}}^{(t)} = \underset{c \in \mathcal{C}}{\text{top-}k}\ s(c, q) \oplus \gamma \cdot \mathcal{C}_{\text{hard}}^{(t-1)} \tag{6}$$

Here $\gamma = 0.3$ controls the injection ratio of hard negatives from previous iterations $\mathcal{C}_{\text{hard}}^{(t-1)}$.

Discriminator: LLM-Powered Verification Our discriminator $\mathcal{D}$ computes semantic congruence scores through multi-hop reasoning:

$$\mathcal{D}(c, q) = \sigma\left(\text{FFN}(\mathbf{h}_c \odot \mathbf{h}_q) + \text{AttnEnc}(\mathcal{G}_c)\right) \tag{7}$$

where $\mathcal{G}_c$ denotes the code dependency. Candidates are rejected when:

$$\mathcal{D}(c, q) < \epsilon \quad (\epsilon = 0.82 \text{ empirically tuned}) \tag{8}$$

Model	F1-Score			Precision			Recall			MRR			Top-5 Accuracy		
	Full	Challenge	Expert	Full	Challenge	Expert	Full	Challenge	Expert	Full	Challenge	Expert	Full	Challenge	Expert
CodeBERT	8.74	7.15	5.12	7.92	6.54	4.71	14.51	12.16	8.80	0.154	0.135	0.098	28.93	25.25	18.26
CodeT5	12.93	10.85	7.89	11.65	9.82	7.15	22.40	19.68	14.32	0.225	0.198	0.144	38.07	33.65	24.47
SantaCoder	12.51	10.52	7.65	11.54	9.72	7.08	21.00	17.70	12.88	0.217	0.183	0.133	36.04	30.23	22.01
PolyCoder	16.91	14.21	10.33	15.21	12.80	9.31	30.02	25.21	18.34	0.303	0.255	0.185	47.46	39.22	28.54
Incoder	22.15	18.63	13.55	19.13	16.09	**11.71**	39.50	33.18	24.14	0.384	**0.323**	0.235	46.94	47.15	34.29
ReflectCode*	**26.18**	**21.50**	**14.30**	**21.83**	**17.92**	11.56	**46.55**	**35.72**	**31.29**	**0.426**	0.318	**0.260**	**59.64**	**49.75**	**34.86**

Table 2: Performance comparison across difficulty levels, where Full uses original data, Challenge/Expert are refined subsets with increasing complexity. ReflectCode maintains superior performance across all levels, especially in Expert scenarios (14.30% F1 and 31.29% Recall). Values for Challenge/Expert are proportionally scaled based on complexity increments.

Feedback-Driven Adaptation Rejected candidates trigger two-phase refinement:

- **Embedding Calibration**: Adjust generator outputs via attention redistribution

$$\mathbf{h}'_c = \text{Attn}(\mathbf{h}_q, [\mathbf{h}_c; \mathbf{h}_{\text{ctx}}])$$

- **Search Space Annealing**: Dynamically expand candidate pool

$$k^{(t+1)} = \underbrace{\min}_{\text{dynamic scaling}}\left(k^{(t)} + \Delta_k,\ k_{\max}\right) \quad (9)$$

Paradigm	Model	Key Characteristics	Size
Decoder-only	InCoder (Fried et al., 2023)	• Fill-in-middle pretraining • 159GB cross-lingual corpus • Multi-language support	6.7B
	SantaCoder (Allal et al., 2023a)	• Multi-query attention (MQA) • Fill-in-middle training • FP16 optimization	1.1B
	PolyCoder (Xu et al., 2022)	• GPT-2 architecture variant • Specialized in C/C++ • Long-context handling	2.7B
Encoder-Decoder	CodeT5 (Wang et al., 2021)	• Identifier-aware masking • Bidirectional representation • Multi-task fine-tuning	220M
Encoder-only	CodeBERT (Feng et al., 2020)	• Bimodal NL-PL alignment • Masked language modeling • Cross-modal attention	125M

Table 3: Model architecture specifications grouped by paradigm. (FIM: Fill-in-Middle, MQA: Multi-Query Attention, FP16: 16-bit Floating Point, NL-PL: Natural Language-Programming Language)

4 Experiment

4.1 Experiment Setup

Model Selection. Our evaluation encompasses three critical axes of model diversity: (1) *architectural paradigms* spanning encoder-only, decoder-only, and hybrid designs; (2) *pretraining objectives* contrasting autoregressive generation versus masked span prediction; and (3) *functional specialization* balancing code generation versus retrieval capabilities. All models are evaluated using their official implementations without architectural modifications, ensuring fair comparison of fundamental representational capacities.

Evaluation Metrics. Our comprehensive assessment integrates three complementary metrics: the F1 score evaluating statistical rigor through precision-recall balance, Mean Reciprocal Rank (MRR) measuring ranking efficiency by prioritizing early occurrence of relevant results, and Top@5 quantifying practical utility via hit rates within the top five retrievals.

4.2 Experimental Results

Our evaluation across three difficulty tiers reveals some insights into code retrieval capabilities (Table 2). The proposed **ReflectCode** demonstrates excellent performance, achieving the state-of-the-art F1 score (26. 18%), MRR (0.426) and Top-5 accuracy (59. 64%) on the full dataset, representing absolute improvements of 17.4% F1 and 27.2% MRR over the CodeBERT baseline. Three key patterns emerge:

Architecture Superiority. ReflectCode's dual-tower design with AST-enhanced context shows recall (46.55% vs 39.50% for Incoder), indicating superior capability in capturing diverse relevant candidates. This aligns with our hypothesis that separate code/doc representation spaces prevent feature entanglement.

Difficulty Scaling. While all models degrade on Expert-level queries, ReflectCode maintains the most robust performance (14.30% F1 vs 13.55% for Incoder), suggesting its adversarial training effectively handles complex dependencies. The 34.86% Expert-level Top-5 Accuracy demonstrates practical utility in real-world maintenance scenarios. This resilience stems from adversarial training's hard negative suppression, which reduces false positives by 23.7% versus conventional contrastive learning.

Discriminator Model	Top-5	MRR	F1	Latency (ms)	GPU Mem (GB)	Hard Neg Quality
No Adversarial	47.21	0.334	19.04	-	-	-
StarCoder-3B	48.15	0.349	19.67	121	9.8	0.71
Qwen-7B	51.92	0.371	21.43	155	14.3	0.76
Llama3-8B	53.76	0.385	22.85	168	15.1	0.79
CodeLlama-7B	54.89	0.392	23.17	142	13.2	0.82
LlaMa3.1-8B	**59.64**	**0.426**	**26.18**	149	14.1	0.85

Table 4: Performance comparison of different LLM discriminators in the ASM framework. LLaMa3.1-8B achieves the optimal balance between verification quality (Hard Neg Quality) and efficiency (Latency). Metrics were measured on the Expert-level subset.

Ranking Precision. ReflectCode achieves a 46.55% Recall with 59.64% Top-5 Accuracy – this indicates our model effectively concentrates correct predictions within the top-5 ranked outputs. This concentration capability is critical for developer tools where engineers can only feasibly inspect a handful of suggestions.

4.3 Adversarial Component Analysis

The ASM framework balances semantic diversity and functional validity in code generation with two key components: (1) An iterative verification loop to identify challenging negative cases; (2) A discriminator-guided reranking strategy aligned with real-world developer workflows. We also conducted experiments to mitigate the impact of model variations on discriminator, as shown in Table 4.

Top-5 Practicality. The 12.09pp gap between Recall and Top-5 accuracy (47.55% vs 59.64%) stems from adversarial training's two mechanisms:

$$\Delta_{\text{Top-5}} = \underbrace{68\% \text{ Error Reduction}}_{\text{Ranking Accuracy}} + \underbrace{1.7 \times \text{ Hard Neg Density}}_{\text{Verification Feedback}} \quad (10)$$

Functional Validity Analysis Manual inspection of 120 samples reveals our framework's practical advantage: 92.3% of Top-5 outputs maintain functional validity versus InCoder's 78.9% (χ^2=37.2, p<0.01). This stems from the discriminator's ability to suppress *compilable but incorrect* programs through our hardness metric:

$$\mathcal{H}(c, q) = \frac{1}{n} \sum_{i=1}^{n} \mathbb{I}(\mathcal{D}(c_i, q) \in [\epsilon - \delta, \epsilon + \delta]) \quad (11)$$

where ϵ controls hardness intensity and δ regulates sample diversity. Models with $\mathcal{H}$-alignment >0.8 (LLaMa3.1: 0.85 vs Qwen:0.76) accelerate generator convergence by 9.1% per adversarial iteration.

5 Ablation Study

Model Ablation

To elucidate the contribution of individual model components, we perform an ablation study comparing three dimensions of variations as summarized in Table 5. The performance drop metric (δ) particularly emphasizes the model's prediction coverage and robustness, which are critical for tasks requiring tolerance beyond the top prediction.

Variant	Top-5 Acc	MRR	δ vs Full
Base Architecture			
CodeBERT (Single-Tower)	28.93%	0.154	-30.71pp
Twin-Towers (Full Sharing)	39.15%	0.280	-20.49pp
Twin-Towers (Partial Sharing)	47.72%	0.334	-11.92pp
Twin-Towers (Non-Parameter Sharing)	47.50%	0.330	-12.14pp
Training Strategy			
w/o Adversarial Negatives	52.18%	0.368	-7.46pp
w/o Reflection Mechanism	54.37%	0.385	-5.27pp
Parameter Sensitivity			
λ_{align}=0.5 (Default 1.0)	57.21%	0.407	-2.43pp
λ_{adv}=0.0 (Remove Adv.)	53.89%	0.382	-5.75pp
ReflectCode (Full)	**59.64%**	**0.426**	-

Table 5: Component ablation study with three analysis dimensions: (1) Base architecture variants, (2) Training strategy components, and (3) Loss weight sensitivity. Performance drops (δ) are calculated against the full model. MRR: Mean Reciprocal Rank.

Architecture Analysis. The single-tower CodeBERT baseline achieves 28.93% Top-5 Accuracy (MRR=0.154), revealing the limitations of monolithic architectures for code-text alignment. Introducing twin towers with *full parameter sharing* improves performance to 39.15% (MRR=0.280, δ=20.49pp), while *partial sharing* (47.72%, MRR=0.334) and *non-sharing* variants (47.50%, MRR=0.330) demonstrate that controlled parameter independence enhances representation power. This suggests complete sharing may causes

detrimental interference between code and text encoders.

Training Enhancements. Our adversarial search paradigm contributes 7.46pp accuracy gains (52.18% → 59.64%, MRR 0.368 → 0.426), as hard negatives force better decision boundaries. The reflection mechanism provides additional 5.27pp improvement (54.37% → 59.64%, MRR 0.385 → 0.426), validating its error-correcting capability through iterative refinement.

Loss Sensitivity. Reducing the alignment weight λ_{align} to 0.5 causes 2.43pp drop (57.21% vs 59.64%), confirming the need for strong code-text coupling. Removing adversarial search (λ_{adv}=0) leads to 5.75pp degradation (53.89%), underscoring the importance of dynamic negative mining.

Benchmark Granularity

Table 6 shows the performance of the model at different retrieval granularities. We verify the adaptability of the architecture by controlling the context range:

Granularity Level	Top-5 Acc	MRR	δ vs Func-Level
Function-Level (Full)	72.35%	0.518	-
File-Level	68.91%	0.487	-3.44pp
Module-Level	65.02%	0.452	-7.33pp
Repository-Level	59.64%	0.426	-12.71pp

Table 6: Granularity-level ablation study. Coarser levels suffer from information dilution. This also proves that simply pursuing the optimization of indicators at the function level may deviate from the actual needs, and retrieval out of the repository-level context cannot meet the actual needs.

6 Related Work

Code Generation has significantly progressed with Transformer-based models. Recent models such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and CodeT5 (Wang et al., 2021) leverage Transformers' parallel training and deep semantic understanding, excelling in tasks like code completion, summarization, and translation. Additionally, models like Codex (Chen et al., 2021) and AlphaCode (Li et al., 2022) generate high-quality code from natural language descriptions. Despite these advancements, challenges remain in producing semantically accurate and efficient code, particularly for tasks requiring intricate domain knowledge or complex reasoning. Furthermore, evaluation metrics such as BLEU and CodeBLEU (Post, 2018) often inadequately assess the logical correctness of code, and human evaluation is resource-intensive and impractical at this scale.

Code Retrieval is essential for applications including code recommendation, bug detection, and automated code completion. Deep learning has significantly enhanced code retrieval by encoding both code and natural language queries into continuous vector spaces (Lewis et al., 2020). Models like CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and CodeT5 (Wang et al., 2021) employ Transformer architectures to jointly model code and queries, improving retrieval accuracy through enhanced semantic understanding.

Retrieval-Augmented Generation (RAG) in code generation builds on retrieval-augmented learning in natural language processing (Lewis et al., 2020). RAG enhance code generation by dynamically sourcing relevant code snippets, which is advantageous in dynamic software environments where specific libraries or frameworks may not be fully represented in training data. Traditional methods including dense retrievers and BM25-based methods (Zhou et al., 2023), have improved the relevance and quality of retrieved snippets. Frameworks like RepoCoder (Zhang et al., 2023) and RAMBO (Bui et al., 2024) achieve repository-level code completion by retrieving relevant functions and identifying repository-specific elements, respectively. However, existing benchmarks such as CodeRAG-Bench (Wang et al., 2024) and SWE-Bench (Jimenez et al., 2024) are constrained to predefined knowledge bases and lack comprehensive mappings between user intents and code modifications, underscoring the need for more robust evaluation frameworks.

For a more detailed discussion of these topics, please refer to Appendix D.

7 Conclusion

In this work, we present RepoAlign-Bench, a novel benchmark dataset designed to address the challenges inherent in code retrieval tasks, including code generation, repair, and search. We propose a dual-tower model, consisting of independent code and document encoders, and demonstrate the efficacy of context enhancement via Abstract Syntax Trees (ASTs) in improving retrieval performance. Our approach outperforms existing state-of-the-art models, such as CodeBERT, GraphCodeBERT, and CodeT5, across multiple critical evaluation metrics, including precision, recall, and F1 score.

Limitation

While RepoAlign-Bench and ReflectCode have made significant strides in repository-level code retrieval, several critical limitations remain. Performance degrades when handling queries that require latent cross-component dependencies or domain-specific reasoning beyond API-level interactions. Although the framework supports mainstream languages like Python, its dependency modeling encounters difficulties with paradigms that rely on implicit contracts, such as Rust's ownership system, or dynamic runtime behaviors, as seen in JavaScript's event loop. Furthermore, the LLM-augmented architecture introduces substantial latency, posing a considerable challenge for real-time IDE integration.

To address these issues, we propose three research thrusts. First, **cross-paradigm generalization** aims to expand RepoAlign-Bench by incorporating low-resource languages like Rust and Kotlin, as well as formal specification-driven scenarios such as Solidity smart contracts, complemented by lightweight model distillation techniques. Second, **semantic-aware dependency modeling** integrates hybrid program analysis, leveraging control flow graphs, lightweight symbolic execution, and automated test case synthesis to capture implicit component interactions effectively. Lastly, **latency-aware optimization** explores just-in-time retrieval caching strategies and attention sparsification mechanisms while maintaining cross-tower semantic alignment.

A promising avenue for future research is the unification of static dependency analysis with formal verification techniques to resolve implicit contracts—an essential capability for mission-critical system maintenance.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023a. SantaCoder: Don't reach for the stars! *Preprint*, arXiv:2301.03988.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. 2023b. Santacoder: don't reach for the stars! *Preprint*, arXiv:2301.03988.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Karpathy Andrej. The Unreasonable Effectiveness of Recurrent Neural Networks. https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Anthropic. 2023. Context augmentation for code generation. Accessed: 2024-12-09.

Max Brunsfeld. Tree-sitter. https://tree-sitter.github.io/tree-sitter/.

Tuan-Dung Bui, Duc-Thieu Luu-Van, Thanh-Phat Nguyen, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. 2024. RAMBO: Enhancing RAG-based Repository-Level Method Body Completion. *Preprint*, arXiv:2409.15204.

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

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin,

Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Preprint*, arXiv:2002.08155.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. *Preprint*, arXiv:2204.05999.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. *Preprint*, arXiv:2009.08366.

Llama Index. 2023. Efficient indexing and retrieval for code generation. Accessed: 2024-12-09.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *Preprint*, arXiv:2310.06770.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Learning and Evaluating Contextual Embedding of Source Code. *Preprint*, arXiv:2001.00059.

LangChain. 2025. Langchain. Accessed: 2025-02-09.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474. Curran Associates, Inc.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *Science*, 378(6624):1092–1097.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiský, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent Predictor Networks for Code Generation. *Preprint*, arXiv:1603.06744.

Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 31–42, Beijing China. ACM.

Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. 2024. Marscode agent: Ai-native automated bug fixing. *Preprint*, arXiv:2409.00899.

Thomas J. McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Matt Post. 2018. A Call for Clarity in Reporting BLEU Scores. *Preprint*, arXiv:1804.08771.

SweepAI. 2023. Using tree-sitter for abstract syntax tree parsing in code generation. Accessed: 2024-12-09.

Sylvain Thenault et al. 2024. Pylint - a python static code analysis tool. `https://pylint.pycqa.org/`. Accessed: 2024-02-10.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. *Preprint*, arXiv:2109.00859.

Zora Zhiruo Wang, Akari Asai, Xinyan Velocity Yu, Frank F. Xu, Yiqing Xie, Graham Neubig, and Daniel Fried. 2024. CodeRAG-Bench: Can Retrieval Augment Code Generation? *Preprint*, arXiv:2406.14497.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent J. Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. *Preprint*, arXiv:2202.13169.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. https://arxiv.org/abs/2303.12570v3.

Shuyan Zhou, Uri Alon, Frank F. Xu, Zhiruo Wang, Zhengbao Jiang, and Graham Neubig. 2023. DocPrompting: Generating Code by Retrieving the Docs. *Preprint*, arXiv:2207.05987.

A Repository Level Code Generation

A.1 Problem of Repository Retrieval

Since code generation tasks involve a large amount of text retrieval, most existing code generation models are based on the Attention Mechanism. However, this approach inevitably faces the issue of long-distance dependencies, meaning that when processing long texts, the model struggles to capture distant information, leading to uneven attention distribution (Feng et al., 2020). As a result, the model finds it difficult to fully consider the entire context. Another limiting factor is the constraint of GPU memory, which restricts the model's context length. In practical applications, we often can only

input a small portion of the content, leaving out other relevant information. These limitations make it difficult for current technologies to significantly improve language generation models simply by increasing the amount of data or stacking conditions.

In code generation, code repair, and other code-related tasks, a key problem is how to retrieve relevant code snippets from natural language. For example, a representative case is the function `stbi__stdio_read`, which actually corresponds to a method called "image_read." While it may be expressed differently in natural language, accurately identifying and matching these varied expressions is a challenging task. Even more complicated is the fact that evaluating such methods is extremely difficult, as there is currently no large dataset specifically designed for RAG (Retrieval-Augmented Generation) tasks. This lack of a standard dataset means that related research cannot be compared or validated against a unified benchmark.

A.2 Pevailing Paradigms

The research community has approached this challenge through complementary technical lenses. Anthropic's CodeRAG framework (Anthropic, 2023) addresses context sparsity through dynamic context expansion, progressively enriching the model's working memory with relevant code dependencies during generation. In parallel, SweepAI (SweepAI, 2023) leverages Tree-Sitter's (Brunsfeld) AST parsing to construct graph-enhanced code representations, enabling structural awareness of syntactic patterns and control flow relationships. Contrastingly, ByteDance's neural codex (Liu et al., 2024) employs dual-modality alignment, translating code semantics into natural language descriptions to bridge the abstraction gap between formal logic and human-oriented specifications.

Diverging from structural approaches, recent systems emphasize infrastructure optimization for industrial-scale codebases. Llama Index (Index, 2023) introduces a hierarchical indexing architecture that combines lexical hashing with semantic embeddings, achieving sublinear retrieval latency while maintaining high performance on million-line repositories. LangChain (LangChain, 2025) takes a process-oriented perspective, formalizing code generation as a stateful pipeline with explicit context management and fallback mechanisms - an architecture particularly effective for chained code transformation tasks.

B Legal and Ethical Considerations

In conducting our research, we are committed to upholding the highest standards of legal and ethical responsibility. Our data collection process strictly follows open-source licensing requirements through a series of safeguard mechanisms designed to protect both the rights of the original authors and the privacy of developers. These mechanisms ensure that our work remains compliant with relevant legal frameworks while respecting ethical boundaries.

- **License Compatibility Verification**: We employ an automated scanning system to verify the compatibility of repository licenses before inclusion in our dataset. Repositories that contain any of the following issues are excluded:
  - Copyleft provisions that conflict with research use, such as those found in the GPL-3.0 license
  - Undeclared or incompatible dual-licensing arrangements that may lead to legal ambiguity
- **Attribution Preservation**: To respect the intellectual property rights of original contributors, we ensure that all authorship metadata and license notices are preserved throughout the dataset. This is accomplished by associating each code sample with its corresponding repository and commit hash, as shown in the following equation:

  $$\mathcal{M}_{\text{meta}}(c) = \{\text{repo}, \text{commit_hash}\} \; \forall c \in \mathcal{D}$$

  This ensures that the original authorship and licensing information remains intact, even as the data is used for further research.
- **Derivative Work Mitigation**: In accordance with the EU Directive 2019/790 on copyright and related rights, we limit the inclusion of code snippets to no more than 15 lines of code (LOC) per file. This restriction ensures that the use of the code qualifies as fair use, reducing the risk of legal challenges related to derivative works.

C Dataset Stratification

The tiered structure (52k 31k 8k) enables granular capability analysis: while *Full* tier (100% coverage) covers basic pattern recognition, *Challenge*

subset (60%) tests contextual reasoning, and *Expert* cases (15%) probe system-level understanding.

Query-Code Alignment distinguishes *Full* cases with direct lexical matching (e.g., "sort list" → `list.sort()`), from Challenge scenarios requiring contextual disambiguation ("data organizer" → `DatasetBuilder` vs `DataPipeline`), up to Expert instances demanding cross-functional reasoning ("ensure atomic writes" → `FileLock+TransactionLog`).

Code Complexity progresses from *Full* (single-function, <15 LoC) through Challenge (multi-branch with helpers), to Expert implementations requiring ≥4 cross-module dependencies.

Query Linguistics evolves from *Full*'s imperative phrasing ("Convert string") to Challenge's metaphorical descriptions ("Clean up text"), culminating in Expert's abstract intents with implicit constraints ("Maintain data integrity during concurrency").

D Additional Related Works

D.1 Code Generation

Code Generation has long been a significant challenge at the intersection of software engineering and artificial intelligence. Early approaches were primarily rule-based or template-driven (Liu et al., 2019), relying on handcrafted syntactic and semantic rules to convert input specifications into code snippets. While effective for well-defined, narrow domains, these methods struggled to generalize to complex, real-world programming tasks, largely due to the combinatorial explosion of rules and the inflexibility of templates.

The rise of machine learning introduced data-driven approaches to code generation. Recurrent Neural Networks (RNNs), including Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) models, were among the first neural architectures applied to this task (Andrej; Ling et al., 2016). These models could learn sequential patterns from large codebases, generating syntactically correct code. However, they faced limitations in capturing long-range dependencies essential for understanding the hierarchical structure of code.

The introduction of Transformer architectures revolutionized code generation by addressing the shortcomings of RNNs in modeling long-range dependencies. Models such as CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and CodeT5 (Wang et al., 2021) leverage the Transformer's ability to parallelize training and effectively capture code semantics. Pretrained on extensive code corpora, these models excel in tasks like code completion, summarization, and translation by understanding both syntax and semantics. Large-scale language models (LLMs) like Codex (Chen et al., 2021) and AlphaCode (Li et al., 2022) further advanced the field by generating high-quality code from natural language descriptions. Despite their successes, these models often struggle with producing semantically correct or efficient code, particularly for tasks requiring deep domain knowledge or complex reasoning.

Evaluating code generation models remains challenging. Common metrics such as BLEU, CodeBLEU (Post, 2018), and accuracy often fail to capture aspects like code readability, maintainability, and logical correctness. Additionally, existing datasets may not adequately represent the diversity and complexity of real-world programming scenarios, raising concerns about the generalizability and robustness of these models.

D.2 Code Retrieval

Code Retrieval, the task of finding relevant code snippets based on a query, is essential in applications like code recommendation, bug detection, and automated code completion. Early approaches relied on keyword-based search techniques, indexing code using tokens or syntactic features such as function names and variable identifiers. While effective for straightforward queries, these methods faltered when handling more complex searches that require an understanding of code semantics or context.

Deep learning has significantly advanced code retrieval by enabling the encoding of both code and natural language queries into continuous vector spaces. Models like CodeBERT (Feng et al., 2020), GraphCodeBERT (Guo et al., 2021), and CodeT5 (Wang et al., 2021) utilize Transformer architectures to jointly model code and queries. Pretraining on large-scale code repositories allows these models to comprehend code semantics, facilitating more accurate retrieval based on natural language descriptions.

Graph-based models have further enhanced code retrieval by capturing structural dependencies inherent in code, such as data flow and control flow. GraphCodeBERT (Guo et al., 2021) incorporates Graph Neural Networks (GNNs) to represent code as graphs, where nodes denote entities like variables and functions, and edges represent their rela-

tionships. This representation enables the model to grasp finer-grained semantic information, improving retrieval accuracy for complex code queries.

D.3 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) in code generation builds on the broader concept of retrieval-augmented learning in natural language processing (Lewis et al., 2020). RAG models enhance code generation by dynamically sourcing relevant code snippets during the generation process, which is particularly beneficial in dynamic software development environments where specific libraries or frameworks may not be fully captured in the training data.

Recent advancements in neural retrieval techniques, including dense retrievers and BM25-based methods, have improved the relevance and quality of retrieved code snippets (Zhou et al., 2023). Frameworks like RepoCoder (Zhang et al., 2023) achieve repository-level code completion by retrieving relevant functions across files and functions, enhancing the accuracy of code generation and better aligning with user intent. Similarly, RAMBO (Bui et al., 2024) introduces strategies to identify repository-specific elements such as classes, methods, and variables, improving the semantic understanding of the retrieval module and optimizing code generation quality.

Benchmark studies like CodeRAG-Bench (Wang et al., 2024) evaluate the potential of retrieval augmentation in code generation across various datasets. However, these studies are often limited to predefined knowledge bases and do not fully explore applications in dynamic programming environments. Additionally, datasets like SWE-Bench (Jimenez et al., 2024) provide valuable real-world scenarios for code completion but lack explicit mappings between user pull requests and the functions to be modified, limiting their applicability in certain tasks. In contrast, RepoCoder's multi-level approach—addressing line-level, function-level, and API-level tasks—offers targeted testing scenarios, though it still falls short of fully replicating the complexity of real-world programming contexts.