CoRTL: Combination Utility Guided Context Retrieval for Repository-Level RTL Code Completion

Anonymous authors
Paper under double-blind review

Abstract

Completing register-transfer level (RTL) code within repositories often requires declarations distributed across multiple chunks, making context selection sensitive to interactions among retrieved chunks. We show that selecting chunks by individual utility frequently misses higher-utility combinations. We introduce CoRTL, a fill-in-the-middle (FIM) completion framework that learns from context combinations. Our method, combination utility search, uses a fixed generator to score combinations by their reduction in mean per-token cross-entropy on the reference target code relative to in-file context alone. The selected combinations provide candidate-level ranking supervision for a context reranker and cross-file context for fine-tuning a completion generator to predict the reference target code. We also construct 47,647 FIM training examples and introduce RealBench-FIM, a repository-level RTL FIM benchmark with testbench-based functional correctness evaluation. With a 0.6B reranker and a 7B completion generator, CoRTL achieves 75.31% func@1 on RealBench-FIM and 66.78% exact match on RTL-Repo, outperforming RLCoder and RepoShapley and exceeding GPT-5.4 by 6.34% and 4.17%, respectively, without using a larger teacher model for supervision construction.

1 Introduction

Large language models (LLMs) have been applied to register-transfer level (RTL) code generation and completion (Thakur et al., 2024; Liu et al., 2024a; Zhao et al., 2025; Wu et al., 2025). Repository-level RTL completion often requires context from other files, such as module interfaces, parameter declarations, and macro definitions. Existing approaches therefore retrieve relevant code chunks and provide them to the completion model together with the in-file context (Wu et al., 2025).

Retrieval usually returns multiple candidate chunks, whose usefulness can depend on the other chunks selected with them (Li et al., 2025; Huo et al., 2026). Prior work has explored using downstream completion performance to guide retrieval and context selection (Wu et al., 2024; Li et al., 2025). RLCoder (Wang et al., 2025b) trains its retriever using single-chunk utility, which measures how an individual candidate chunk affects target-code prediction when added to the in-file context. RepoShapley (Huo et al., 2026) further models chunk interactions with a nonlinear surrogate built from single-chunk utilities, using Shapley values to propose combinations for generator verification. However, chunks that are useful individually do not necessarily form a high-utility combination.

This issue is particularly relevant to repository-level RTL completion, where referenced declarations can span multiple chunks. In repositories sampled from our RTL corpus, 14.87% of nonblank, noncomment source lines contain resolved cross-file references, compared with 6.09% in RepoCoder's (Zhang et al., 2023) Python repositories (Figure 1(a)). Among code segments with resolved cross-file references, 26.34% of RTL segments require multiple chunks to cover the referenced declarations, versus 16.56% for Python (Figure 1(b)). These results motivate considering multiple chunks jointly for RTL completion.

These multi-chunk dependencies raise a practical question: do the individually most useful chunks form the most useful combination? To measure usefulness, we define context utility as the reduction in a fixed generator's mean target-code cross-entropy relative to using only in-file context.

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.

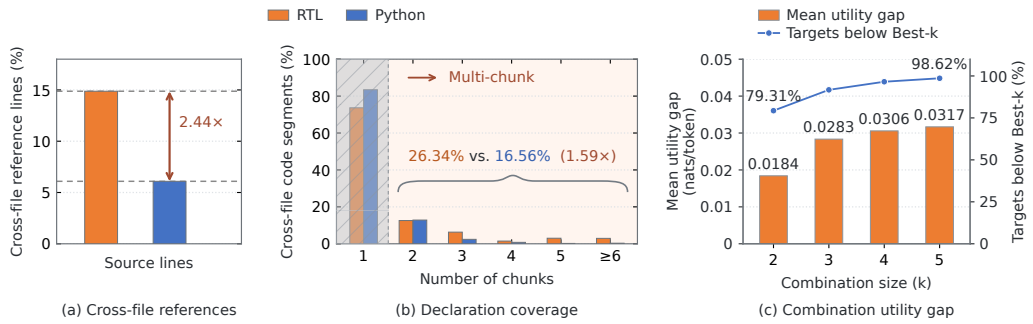

Figure 1: Cross-file references and combination utility gaps. (a) Percentage of nonblank, noncomment source lines with resolved cross-file references. (b) Minimum chunks covering referenced declarations per segment with resolved cross-file references. (c) Mean Best-k minus Top-k utility (bars) and percentage of targets with a positive gap (line), over 145 RealBench-FIM targets.

Higher utility means that the retrieved context makes the reference target code more likely under the fixed generator. For each of the 145 RealBench-FIM (our completion benchmark derived from RealBench (Jin et al., 2025)) targets, we exhaustively evaluate combinations of size $k \in \{2, 3, 4, 5\}$ drawn from the 10 chunks with the highest single-chunk utilities. We compare the best size-k combination in this pool (Best-k) with the Top-k combination selected by single-chunk utility. Figure 1(c) shows their mean utility gap (Best-k minus Top-k) and the proportion of targets where Top-k falls short of Best-k. As k increases from 2 to 5, the proportion of targets where Top-k falls short of Best-k rises from 79.31% to 98.62%, and the mean combination utility gap grows from 0.0184 to 0.0317. The growing average gap motivates directly evaluating combinations instead of evaluating single chunks separately. Appendices A.2 and A.3 provide details of both analyses.

To address this limitation, we introduce CoRTL, a repository-level RTL code completion framework that learns from high-utility context combinations. The framework consists of two components: a context reranker that ranks retrieved chunks and a completion generator that predicts the missing code using the top-ranked chunks and the in-file context. The central idea is to use directly evaluated combination utility to derive training supervision for both components. To this end, we develop combination utility search, an incremental multi-path search over candidate chunks. A fixed scoring generator evaluates each proposed combination using all of its chunks together as context, and the resulting combination utility determines which extensions are accepted and which combinations are retained. The selected high-utility combinations yield candidate-level ranking labels for training the reranker and serve as retrieved contexts for fine-tuning the completion generator to predict the reference target code. The reranker and completion generator are trained separately.

We adopt the fill-in-the-middle (FIM) setting (Bavarian et al., 2022; Fried et al., 2022) and construct 47,647 RTL FIM examples covering line, instantiation-and-call, and structured-span completion. We also introduce RealBench-FIM, to our knowledge the first repository-level RTL FIM benchmark with testbench-based functional evaluation in real-world IP projects.

With a 0.6B reranker and a 7B completion generator, CoRTL achieves 75.31% overall func@1 on RealBench-FIM and 66.78% exact match (EM) on RTL-Repo (Allam & Shalan, 2024), outperforming RLCoder (Wang et al., 2025b) and RepoShapley (Huo et al., 2026), two closely related repository-level frameworks that also use downstream completion behavior to guide context selection. It also exceeds GPT-5.4 by 6.34% and 4.17% on these two metrics, respectively, without using a larger teacher model for supervision construction.

Our main contributions are as follows:

- We identify a limitation of context selection based on single-chunk utility in repository-level RTL completion. Our analyses characterize dependencies spanning multiple chunks and show that selecting the individually most useful chunks often misses higher-utility combinations within the same candidate pool.

- We introduce CoRTL, a repository-level RTL FIM completion framework that uses high-utility context combinations to supervise a context reranker and fine-tune a completion generator. We develop combination utility search to identify these combinations guided by generator-evaluated combination utility at each expansion.
- We construct a repository-level RTL training corpus with 47,647 FIM examples and introduce RealBench-FIM, to our knowledge the first benchmark for repository-level RTL FIM code completion with testbench-based functional correctness evaluation. Both cover line, instantiation-and-call, and structured-span targets, with RealBench-FIM evaluating the functional correctness of completed modules within real-world IP projects.
- With a 0.6B reranker and a 7B completion generator, CoRTL outperforms the evaluated RTL-domain models and repository-level frameworks on both benchmarks, with absolute margins over GPT-5.4 of 6.34% in overall func@1 on RealBench-FIM and 4.17% in EM on RTL-Repo.

2 Related Work

Repository-Level Code Completion. Repository-level completion exploits cross-file context through repository-aware prompting, retrieval, and context-conditioned training (Shrivastava et al., 2023b; Zhang et al., 2023; Shrivastava et al., 2023a), with later work improving context selection and organization through selective retrieval, context fusion, and graph-based retrieval (Wu et al., 2024; Liang et al., 2024; Liu et al., 2024c; Phan et al., 2025). More recent methods use downstream completion behavior to guide context selection: RLCoder (Wang et al., 2025b) trains its retriever with single-chunk utility, while RepoShapley (Huo et al., 2026) uses a surrogate built from single-chunk utilities and Shapley values to propose combinations before generator-based verification. In the RTL domain, RTLRepoCoder (Wu et al., 2025) combines domain fine-tuning with retrieval for next-line completion. In contrast, CoRTL evaluates context combinations with a fixed generator during search, uses combination utility to retain search paths, and uses the resulting combinations to supervise a reranker and fine-tune a separate completion generator.

LLMs for RTL Code Development. VeriGen (Thakur et al., 2024) and RTLCoder (Liu et al., 2024a) study domain data and fine-tuning. CodeV (Zhao et al., 2025) constructs descriptions through multi-level summarization, CraftRTL (Liu et al., 2025a) generates synthetic training data, and DeepRTL (Liu et al., 2025b) combines Verilog understanding and generation. BetterV (Pei et al., 2024) uses discriminative guidance, CodeV-R1 (Zhu et al., 2026) combines reasoning distillation with reinforcement learning using verifiable rewards, and AutoVCoder (Gao et al., 2024) integrates data construction, fine-tuning, and domain retrieval. For tool-assisted generation and repair, AutoChip (Thakur et al., 2023) uses compilation and simulation feedback, OriGen (Cui et al., 2024) combines code augmentation with compiler-guided repair, and VerilogCoder (Ho et al., 2025) uses graph-based planning and waveform tracing. CoRTL studies how repository context jointly supports completion within existing implementations, pairing combination utility supervision for retrieval with generator fine-tuning on retrieved context and in-file prefixes and suffixes.

Datasets and Benchmarks for RTL Generation and Completion. VerilogEval (Liu et al., 2023) and RTLLM (Lu et al., 2024) evaluate module generation, while OpenLLM-RTL (Liu et al., 2024b) provides RTL generation data and benchmarks. RTL-Repo (Allam & Shalan, 2024) evaluates single-line repository-level completion with EM and ES. RealBench (Jin et al., 2025) functionally evaluates real-world IP generation, and CktEvo (Shi et al., 2026) evaluates functionality-preserving repository edits for improved power, performance, and area. Our training corpus supports repository-level line, instantiation-and-call, and structured-span FIM completion, retaining in-file prefixes, suffixes, and repository associations for retrieval. We also introduce RealBench-FIM, converting RealBench generation tasks into FIM completion at these three granularities while preserving dependencies and testbenches. Unlike text matching in RTL-Repo, it checks functional correctness by simulating completed modules with their dependencies and comparing outputs against reference implementations under testbench inputs.

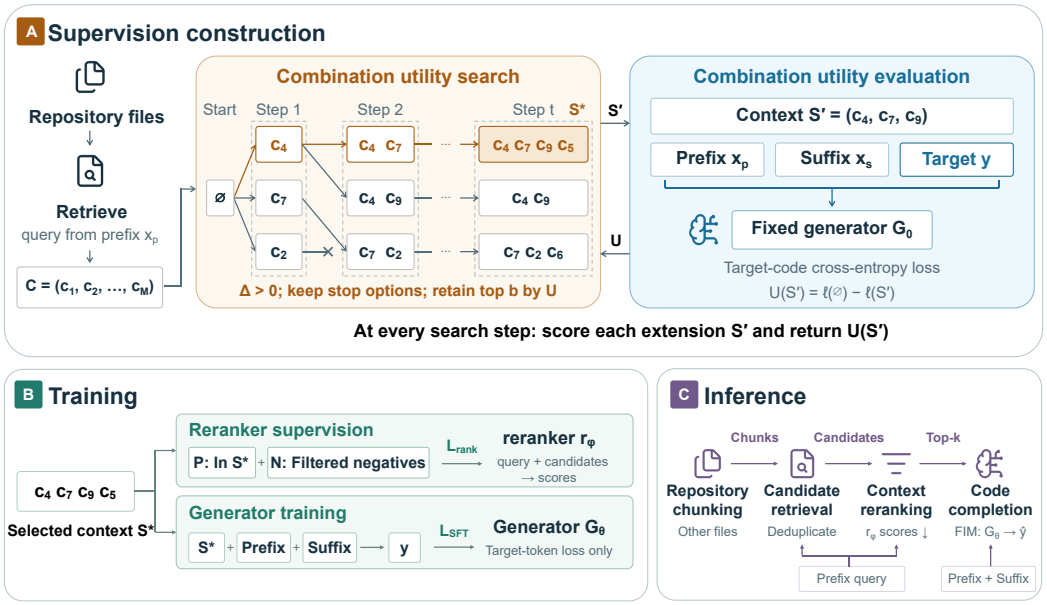

Figure 2: CoRTL: (A) supervision construction through combination utility search; (B) separate reranker training and generator SFT using searched combinations; (C) reranking and completion at inference.

3 Method

3.1 Task Definition

We study repository-level RTL completion in the fill-in-the-middle (FIM) setting (Bavarian et al., 2022; Fried et al., 2022). Given an RTL repository and a target file, each example (x_p, x_s, y) consists of the in-file prefix x_p, suffix x_s, and reference target code y. The goal is to generate the missing target code between the prefix and suffix using the in-file context together with relevant cross-file context from the same repository.

3.2 Framework Overview

Figure 2 presents the framework of CoRTL. It contains two trainable components: a context reranker r_ϕ that selects cross-file context and a completion generator G_θ that generates the target code using the selected context together with the in-file prefix and suffix.

To obtain cross-file context, we first divide source files other than the target file into overlapping chunks using a sliding window. We construct a query q from the in-file prefix x_p and retrieve chunks using both lexical and dense retrieval. The retrieved results are merged and deduplicated to form a candidate pool of M chunks, $\mathcal{C} = (c_1, \ldots, c_M)$. Repository chunking and candidate retrieval require no task-specific training, and their configurations are provided in Appendix A.4.

At inference time, the trained reranker takes q and $\mathcal{C}$ as input, assigns each candidate chunk c_i a score $s_{\phi,i}$, and selects the top k candidates in descending order to form the context combination S. We denote by $F(x_p, x_s, S)$ the resulting FIM prompt, which combines the selected cross-file context with the in-file prefix and suffix. The completion generator G_θ then generates the target code from this prompt. Figure 4 in the appendix shows a real prompt example. Inference requires neither the reference target code y nor combination utility search.

The reranker and completion generator are trained using context combinations selected by combination utility search. We next describe how these combinations are constructed and used to train the two models.

Algorithm 1 combination utility search

Input: Prefix x_p, suffix x_s, reference target code y, candidate pool $\mathcal{C}$, fixed generator G_0, Beam width b, expansion limit m, maximum combination length $K_{\max}$
Output: Searched context combination S^*

$\ell_\emptyset \leftarrow \ell(\emptyset)$ using G_0 and y ▷ Eq. (1)
Compute $u_i \leftarrow U((c_i))$ for each $c_i \in \mathcal{C}$ ▷ Eq. (2)
$\mathcal{B} \leftarrow \{(\emptyset, 1)\}$; $S^* \leftarrow \emptyset$
for $t = 1, \ldots, K_{\max}$ **do**
  $\mathcal{H} \leftarrow \{(S, 0) : (S, a) \in \mathcal{B}\}$ ▷ Keep stopping options
  for all $(S, 1) \in \mathcal{B}$ **do**
    $\mathcal{A} \leftarrow$ up to m chunks in $\mathcal{C} \setminus S$ with highest u_i
    for all $c_i \in \mathcal{A}$ **do**
      $S' \leftarrow S \oplus c_i$
      Evaluate $U(S')$ using G_0 and y ▷ Evaluate combination utility
      if $U(S') > U(S)$ **then**
        $\mathcal{H} \leftarrow \mathcal{H} \cup \{(S', 1)\}$
      end if
    end for
  end for
  $\mathcal{B} \leftarrow$ top b states in $\mathcal{H}$ by $U(S)$
  $\widehat{S} \leftarrow$ highest-utility combination in $\mathcal{B}$
  if $U(\widehat{S}) > U(S^*)$ **then**
    $S^* \leftarrow \widehat{S}$
  end if
  if all states in $\mathcal{B}$ have $a = 0$ **then**
    break
  end if
end for
return S^*

3.3 Supervision Construction

For each training example, combination utility search uses the reference target code y to find a context combination S^* from the candidate pool $\mathcal{C}$. We first define combination utility and then describe the search procedure.

Combination utility. Combination utility measures how much the chunks in S jointly improve prediction of the reference target code. Given a combination S, a fixed scoring generator G_0 evaluates the reference target code $y = (y_1, \ldots, y_T)$ under the prompt $F(x_p, x_s, S)$. The corresponding mean per-token cross-entropy loss is

$$\ell(S) = -\frac{1}{T} \sum_{t=1}^{T} \log p_{G_0}\big(y_t \mid F(x_p, x_s, S), y_{<t}\big). \tag{1}$$

where p_{G_0} denotes the next-token distribution of G_0 and $y_{<t}$ denotes the preceding reference tokens. Only target tokens contribute to the loss. Using the in-file context alone as the baseline, we define combination utility as

$$U(S) = \ell(\emptyset) - \ell(S). \tag{2}$$

Thus, $U(\emptyset) = 0$, and positive utility indicates improved target prediction when the chunks in S are used together. For a single chunk c_i, we refer to $U((c_i))$ as its single-chunk utility.

Search procedure. We implement combination utility search with beam search, starting from the empty combination. For each active combination S, we consider up to m unselected chunks with the highest single-chunk utilities, append each candidate separately, and evaluate the resulting combination with G_0. Thus, single-chunk utility determines which chunks are proposed for expansion, while combination utility determines which extensions are accepted and which search paths are retained.

We define the utility gain from appending c_i to S as

$$\Delta(c_i \mid S) = U(S \oplus c_i) - U(S) = \ell(S) - \ell(S \oplus c_i). \tag{3}$$

Only extensions with $\Delta(c_i \mid S) > 0$ are eligible for further search, while the current combination is also retained as a stopping option. The beam keeps up to b eligible extensions or stopped combinations with the highest combination utility. Search terminates when all retained paths stop, no candidates remain for expansion, or the maximum combination length $K_{\max}$ is reached, and returns the highest-utility combination S^* found during search. Algorithm 1 summarizes the procedure using states (S, a), where $a = 1$ denotes an active combination and $a = 0$ a stopped combination.

3.4 Training

The resulting combination S^* provides candidate-level supervision for the reranker and cross-file context for completion-generator fine-tuning. The two components are trained separately.

Reranker training. Members of S^* form the positive set $P = \{i : c_i \in S^*\}$. Because an unselected chunk may still be useful, we label it negative only when it increases target loss both in isolation and when appended to S^*:

$$N = \{i \notin P : U((c_i)) < 0 \text{ and } \Delta(c_i \mid S^*) < 0\}. \tag{4}$$

After search, we evaluate $\Delta(c_i \mid S^*)$ for each unselected chunk to construct N without modifying S^*; candidates outside $P \cup N$ are excluded from the ranking loss.

For examples with $P \neq \emptyset$, we optimize the contrastive ranking loss

$$\mathcal{L}_{\text{rank}} = -\frac{1}{|P|} \sum_{i \in P} \log \frac{\exp(s_{\phi,i}/\tau)}{\exp(s_{\phi,i}/\tau) + \sum_{j \in N} \exp(s_{\phi,j}/\tau)}, \tag{5}$$

where τ is the temperature. The loss compares each selected chunk with the negatives in N without treating co-selected chunks as negatives.

Completion generator training. The completion generator G_θ is initialized from the same pretrained checkpoint as the fixed generator G_0. While G_0 remains fixed during supervision construction, G_θ is fine-tuned using the combination S^* selected by search.

For each example e in the training corpus $\mathcal{D}$, we pair $F(x_{p,e}, x_{s,e}, S_e^*)$ with the reference target code y_e and minimize

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{\sum_{e \in \mathcal{D}} T_e} \sum_{e \in \mathcal{D}} \sum_{t=1}^{T_e} \log p_\theta\big(y_{e,t} \mid F(x_{p,e}, x_{s,e}, S_e^*), y_{e,<t}\big). \tag{6}$$

Here, T_e is the number of tokens in y_e, and only these target tokens contribute to the loss.

4 Experiments

We evaluate CoRTL at three levels. We first evaluate the overall performance of the complete framework, then isolate the effectiveness of context retrieval and reranking with a fixed generator, and finally analyze how combination utility improves context selection and reranker supervision.

4.1 Experimental Setup

Training. We construct 47,647 FIM examples from public Verilog and SystemVerilog repositories. We use Qwen2.5-Coder-7B-Instruct (Hui et al., 2024) for combination utility evaluation and initialize the completion generator from the same checkpoint. The reranker is initialized from jina-reranker-v3 (Wang et al., 2025a). We train the reranker for 1 epoch with effective batch size 32 and learning rate 6×10^{-6}. We fine-tune all generator parameters for 2 epochs with an effective batch size of 256 and a learning rate of 10^{-5}. Appendices A.1 and A.4 provide data construction and full training configurations, respectively.

Table 1: Main results on RealBench-FIM and RTL-Repo, with all scores reported as percentages. **Bold** and underlined values denote the best and second-best results in each column, including ties. †: result reported in the original paper under its prefix-only setting.

Model	RealBench-FIM (%)								RTL-Repo (%)	
	Line		Inst. & Call		Span		Overall		EM	ES
	f@1	f@5	f@1	f@5	f@1	f@5	f@1	f@5		
Commercial models										
GPT-5.4	65.00	–	64.29	–	75.44	–	68.97	–	62.61	86.02
Claude Opus 4.6	**83.33**	–	**78.57**	–	**80.70**	–	**81.38**	–	**70.87**	82.18
DeepSeek-V3.2	66.67	–	57.14	–	71.93	–	66.90	–	52.04	72.40
GLM-5.2	60.00	–	46.43	–	70.18	–	61.38	–	59.71	82.76
RTL-domain models										
RTLCoder-6.7B	20.50	23.50	10.18	12.31	16.49	21.17	16.93	20.42	42.33	79.33
CodeV-All-QC-7B	35.33	39.86	22.50	29.10	44.56	51.87	36.48	42.51	53.83	81.25
CodeV-R1-7B	16.50	20.30	27.32	38.41	23.51	29.24	21.34	27.31	15.67	40.21
Repository-level frameworks										
RLCoder-7B	49.75	57.41	31.79	43.29	45.18	53.99	44.48	53.34	62.10	86.66
RepoShapley-7B	66.83	73.08	54.46	59.59	60.53	67.02	61.97	68.09	61.41	88.79
RTLRepoCoder-6.7B†	–	–	–	–	–	–	–	–	55.80	84.30
CoRTL-7B	76.25	**76.67**	74.11	**75.00**	74.91	**76.70**	75.31	**76.36**	66.78	**89.03**

Benchmarks and Metrics. (1) RealBench-FIM. We construct this benchmark from RealBench (Jin et al., 2025), with 60 line, 28 instantiation-and-call, and 57 structured-span targets (Inst. & Call and Span in the tables). Completions are evaluated in their original modules using project dependencies and existing testbenches. Following RealBench, we report func@k, the probability that at least one of k sampled completions passes functional evaluation. We average over targets within each type and across all 145 targets for Overall. We show construction details in Appendix A.1. (2) RTL-Repo. We evaluate completion using exact match (EM) as the primary metric and edit similarity (ES) as secondary (Allam & Shalan, 2024). This benchmark contains 1,174 single-line targets.

Baseline Methods. We compare with 3 groups of baselines: (1) commercial models, including GPT-5.4, Claude Opus 4.6, DeepSeek-V3.2, and GLM-5.2; (2) RTL-domain models, including RTLCoder (Liu et al., 2024a), CodeV-All (Zhao et al., 2025), and CodeV-R1 (Zhu et al., 2026); and (3) repository-level frameworks, including RLCoder (Wang et al., 2025b), RepoShapley (Huo et al., 2026), and RTLRepoCoder (Wu et al., 2025). We reproduce RLCoder and RepoShapley on our RTL training corpus. As no public implementation of RTLRepoCoder is available, we report its published results, using prefix context only. Model sizes in Table 1 refer to completion generators.

Evaluation Protocol. Following RealBench, we estimate func@1 and func@5 from 20 samples per target on RealBench-FIM. Commercial models use 1 sample per target and report only func@1. On RTL-Repo, we generate 1 completion per target and use the official postprocessing and evaluator. We construct a hybrid candidate pool using BM25 (Robertson & Zaragoza, 2009) and UniXcoder (Guo et al., 2022). RLCoder and CoRTL use the full pool, while RepoShapley follows the original paper's 10-candidate limit to keep computation manageable. At inference time, CoRTL supplies the top 10 reranked chunks to the completion generator. Appendices A.4 and A.5 provide retrieval, search, and evaluation configurations. In all tables, f@n abbreviates func@n.

4.2 Overall Performance

CoRTL outperforms the evaluated RTL-domain models and repository-level frameworks on both benchmarks. In Table 1, CoRTL achieves 75.31% overall func@1 on RealBench-FIM and 66.78% EM on RTL-Repo. The absolute margins over the strongest baselines in these two groups are

Table 2: Retrieval and reranking results with the raw generator, reported as percentages.

Method	RealBench-FIM (%)								RTL-Repo (%)	
	Line		Inst. & Call		Span		Overall		EM	ES
	f@1	f@5	f@1	f@5	f@1	f@5	f@1	f@5		
No retrieval	44.75	51.41	21.79	31.76	44.12	55.66	40.07	49.29	55.03	84.50
BM25	47.42	60.05	31.61	43.13	40.18	52.26	41.52	53.72	63.20	87.19
UniXcoder	52.33	60.41	32.32	42.47	39.91	52.19	43.59	53.72	63.03	86.95
jina-reranker-v3	52.00	59.30	27.50	30.72	36.49	47.57	41.17	49.17	63.46	86.87
RLCoder	49.75	57.41	31.79	43.29	45.18	53.99	44.48	53.34	62.10	86.66
Ours	**54.75**	**61.33**	**39.82**	**50.31**	**45.70**	**57.37**	**48.31**	**57.65**	**64.14**	**87.41**

13.34% for func@1 (RepoShapley) and 4.68% for EM (RLCoder). Compared with RepoShapley, its func@1 gains span all 3 completion types, with larger margins on instantiation-and-call targets as well as structured-span targets than on line targets. Overall func@5 also reaches 76.36%, compared with RepoShapley's 68.09%. The gains thus extend from single-line exact matching to functional correctness across completion granularities.

CoRTL surpasses several commercial models with a 7B-parameter generator. It exceeds GPT-5.4, DeepSeek-V3.2, and GLM-5.2 on both overall func@1 and EM, with absolute margins of 6.34% and 4.17% over GPT-5.4. Among all evaluated systems, CoRTL ranks second on both primary metrics, behind Claude Opus 4.6, and achieves the highest ES at 89.03%. These results show that the complete framework brings a 7B-parameter completion generator to a competitive level with the evaluated commercial models on repository-level RTL completion.

4.3 Effectiveness of Context Retrieval and Reranking

To isolate the effect of context selection from generator fine-tuning, we fix the completion generator to the pretrained Qwen2.5-Coder-7B-Instruct checkpoint and vary only the retrieval or reranking method. We compare BM25 (Robertson & Zaragoza, 2009), UniXcoder (Guo et al., 2022), RLCoder (Wang et al., 2025b), the pretrained jina-reranker-v3 (Wang et al., 2025a), and our combination-supervised reranker, together with a no-retrieval baseline. The generator prompt, decoding settings, and evaluation protocol are fixed across methods. RLCoder and the two rerankers operate on the same hybrid candidate pool, while BM25 and UniXcoder use their directly retrieved results.

Our trained reranker achieves the best completion results among the evaluated retrieval and reranking methods. Table 2 reports 48.31% overall func@1 and 64.14% EM, with absolute margins of 7.14% and 0.68% over the pretrained reranker, and 3.83% and 2.04% over RLCoder. It also leads in func@1 and func@5 across all 3 completion types, reaching 57.65% overall func@5. Because the completion generator is identical across configurations, these gains directly reflect improved context selection rather than generator adaptation.

4.4 Analysis of Combination Utility

We analyze combination utility of retrieved chunks at two stages of CoRTL: context selection during supervision construction and reranker supervision during training.

Combination utility search selects more effective context than single-chunk utility selection. To test the effectiveness of the search within a fixed candidate set, we compare combination utility search with Single-Chunk Utility Selection on RealBench-FIM. Both methods use the same complete hybrid candidate pool for each target and select at most 10 chunks, with Qwen2.5-Coder-7B-Instruct held fixed for scoring. Single-Chunk Utility Selection tests whether combining individually useful chunks is sufficient. It selects up to 10 positive-utility chunks in descending single-chunk utility order, returning empty context if none has positive utility. Each method's selected context is supplied to the raw generator under the same prompt format and decoding protocol. Reference

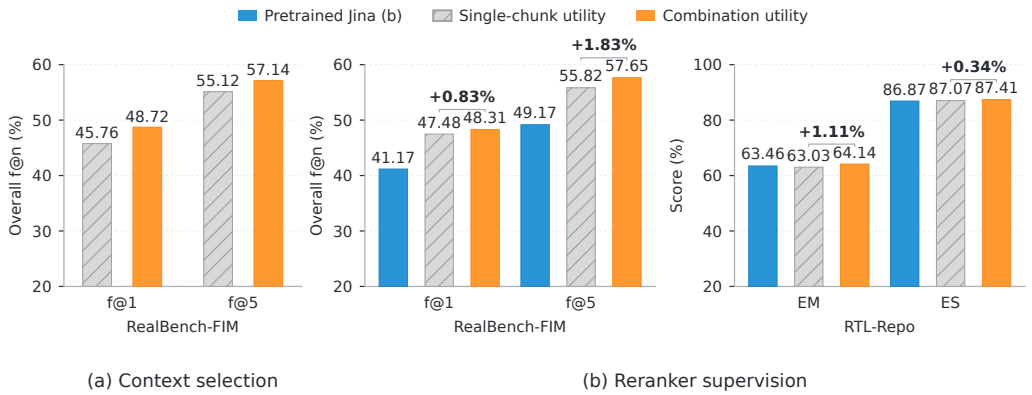

Figure 3: With the raw generator: (a) single-chunk utility selection versus combination utility search over the full hybrid pool, using reference targets; (b) pretrained Jina and rerankers trained with single-chunk or combination utility supervision. Scores are percentages, and f@n denotes overall func@n. Brackets show absolute gains from combination over single-chunk supervision. The left and right plots in (b) show RealBench-FIM and RTL-Repo, respectively.

targets are used for scoring but excluded from completion prompts. In Figure 3(a), combination utility search achieves 48.72% overall func@1 and 57.14% overall func@5, with absolute gains of 2.97% and 2.02% over single-chunk selection, respectively. These gains reflect more effective context selection without subsequent reranker or generator training.

Combination utility provides more effective supervision for reranker training than single-chunk utility. We compare rerankers trained with single-chunk and combination utility labels, and include the pretrained reranker as a reference. Single-chunk labels mark up to 10 positive-utility chunks as positives and negative-utility chunks as negatives. Combination utility labels follow Section 3.4. All three rerankers supply up to 10 chunks to the same raw generator. Compared with the pretrained reranker, combination utility supervision achieves absolute gains of 7.14% in overall func@1 and 0.68% in EM. In Figure 3(b), it also achieves 48.31% overall func@1 and 64.14% EM, compared with 47.48% and 63.03% under single-chunk utility supervision. The two trained rerankers share the jina-reranker-v3 initialization, RTL training examples, candidate pools, ranking loss, and optimizer configuration, with both types of utility evaluated by the same fixed generator. Combination utility supervision provides further gains over this baseline.

5 Conclusion

We presented CoRTL, a repository-level RTL FIM completion framework trained with combination utility supervision. Its combination utility search uses a fixed generator to evaluate context combinations and guide incremental expansion across multiple search paths. The resulting combinations supervise a context reranker and provide context for generator SFT, without requiring combination search at inference time. We also constructed an RTL FIM training corpus and introduced RealBench-FIM for testbench-based functional evaluation across line, instantiation-and-call, and structured-span completion tasks. With a 7B-parameter generator, CoRTL achieves 75.31% overall func@1 on RealBench-FIM and 66.78% EM on RTL-Repo. On both metrics, it outperforms the evaluated RTL-domain models and repository-level frameworks, as well as commercial models including GPT-5.4, DeepSeek-V3.2, and GLM-5.2. Further analyses show that combination utility improves context selection and provides more effective reranker supervision than single-chunk utility, supporting its use in learning repository context selection for RTL completion.

AI USE STATEMENT

AI tools assisted literature search, writing, translation, LaTeX, and review. The authors are responsible for experimental records, references, and claims.

REPRODUCIBILITY STATEMENT

Section 3 describes the combination utility search algorithm and training objectives, and Section 4.1 specifies the benchmarks, metrics, and experimental setup. Appendices A.1, A.4, and A.5 provide further details on data construction, training configurations, and evaluation procedures, respectively.

REFERENCES

Ahmed Allam and Mohamed Shalan. Rtl-repo: A benchmark for evaluating llms on large-scale rtl design projects. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5. IEEE, 2024.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *arXiv preprint arXiv:2207.14255*, 2022.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, et al. Origen: Enhancing rtl code generation with code-to-code augmentation and self-reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2024.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. Autovcoder: A systematic framework for automated verilog code generation using llms. In *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, pp. 162–169. IEEE, 2024.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7212–7225, 2022.

Chia-Tung Ho, Haoxing Ren, and Brucek Khailany. Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 300–307, 2025.

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

Yu Huo, Kun Zeng, Siyu Zhang, Yuquan Lu, Cheng Yang, Yifu Guo, and Xiaoying Tang. Reposhapley: Shapley-enhanced context filtering for repository-level code completion. In *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 10390–10412, 2026.

Pengwei Jin, Di Huang, Chongxiao Li, Shuyao Cheng, Yang Zhao, Xinyao Zheng, Jiaguo Zhu, Shuyi Xing, Bohan Dou, Rui Zhang, et al. Realbench: Benchmarking verilog generation models with real-world ip designs. *arXiv preprint arXiv:2507.16200*, 2025.

Yanzhou Li, Shangqing Liu, Kangjie Chen, Tianwei Zhang, and Yang Liu. Impact-driven context filtering for cross-file code completion. *arXiv preprint arXiv:2508.05970*, 2025.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Hongwei Chen, Chengpeng Wang, Gang Fan, et al. Repofuse: Repository-level code completion with fused dual context. *arXiv preprint arXiv:2402.14323*, 2024.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 1–8. IEEE, 2023.

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair. In *International Conference on Learning Representations*, volume 2025, pp. 90377–90422, 2025a.

Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Outperforming gpt-3.5 in design rtl generation with our open-source dataset and lightweight solution. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pp. 1–5. IEEE, 2024a.

Shang Liu, Yao Lu, Wenji Fang, Mengming Li, and Zhiyao Xie. Openllm-rtl: Open dataset and benchmark for llm-aided design rtl generation. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9, 2024b.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. Graphcoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 570–581, 2024c.

Yi Liu, Changran Xu, Yunhao Zhou, Zeju Li, and Qiang Xu. Deeprtl: Bridging verilog understanding and generation with a unified representation model. *arXiv preprint arXiv:2502.15832*, 2025b.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 722–727. IEEE, 2024.

Zehua Pei, Hui-Ling Zhen, Mingxuan Yuan, Yu Huang, and Bei Yu. Betterv: Controlled verilog generation with discriminative guidance. *arXiv preprint arXiv:2402.03375*, 2024.

Huy N Phan, Hoang N Phan, Tien N Nguyen, and Nghi DQ Bui. Repohyper: Search-expand-refine on semantic graphs for repository-level code completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pp. 14–25. IEEE, 2025.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Foundations and trends® in information retrieval*, 4(1-2):1–174, 2009.

Zhengyuan Shi, Jingxin Wang, Tairan Cheng, Changran Xu, Weikang Qian, and Qiang Xu. Cktevo: Repository-level rtl code benchmark for design evolution. *arXiv preprint arXiv:2603.08718*, 2026.

Disha Shrivastava, Denis Kocetkov, Harm De Vries, Dzmitry Bahdanau, and Torsten Scholak. Repofusion: Training code models to understand your repository. *arXiv preprint arXiv:2306.10998*, 2023a.

Disha Shrivastava, Hugo Larochelle, and Daniel Tarlow. Repository-level prompt generation for large language models of code. In *International Conference on Machine Learning*, pp. 31693–31715. PMLR, 2023b.

Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. Autochip: Automating hdl generation using llm feedback. *arXiv preprint arXiv:2311.04887*, 2023.

Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems*, 29(3):1–31, 2024.

Feng Wang, Yuqing Li, and Han Xiao. jina-reranker-v3: Last but not late interaction for listwise document reranking. *arXiv preprint arXiv:2509.25085*, 2025a.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. Rlcoder: Reinforcement learning for repository-level code completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pp. 1140–1152. IEEE, 2025b.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. *arXiv preprint arXiv:2403.10059*, 2024.

Peiyang Wu, Nan Guo, Junliang Lv, Xiao Xiao, and Xiaochun Ye. Rtlrepocoder: Repository-level rtl code completion through the combination of fine-tuning and retrieval augmentation. *arXiv preprint arXiv:2504.08862*, 2025.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 2471–2484, 2023.

Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, et al. Codev: Empowering llms with hdl generation through multi-level summarization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2025.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Pengwei Jin, et al. Qimeng-codev-r1: Reasoning-enhanced verilog generation. *Advances in Neural Information Processing Systems*, 38:154266–154300, 2026.

A APPENDIX

We provide data construction details, protocols for the introductory analyses, retrieval and training configurations, evaluation settings, and a concrete completion prompt.

A.1 TRAINING CORPUS AND BENCHMARK CONSTRUCTION

Source repositories. We collect public GitHub repositories containing Verilog and SystemVerilog source files and record their available license metadata. Repository selection considers both GitHub star counts and engineering quality, assessed through project provenance, source organization, and maintenance evidence. Preprocessing distinguishes RTL source and headers from testbenches, generated files, and vendored code, and restricts completion targets to eligible RTL files.

Deduplication and benchmark overlap. The pipeline represents files with comment-stripped token sequences and five-token shingles. MinHash and locality-sensitive hashing retrieve possible duplicate files, which are verified using shingle Jaccard similarity at a threshold of 0.80. Repository deduplication uses one-to-one matching of eligible files, treating two repositories as duplicates when matching covers at least 80% of the eligible files in both directions. Benchmark-overlap screening compares repository source and headers against complete code from RealBench and RTL-Repo using normalized token containment and verified shingle similarity. A matching repository is quarantined as a whole. Selection manifests record the retained repositories and target-eligible files.

Table 3: Tree-sitter node types used to construct completion targets. Span groups follow the extractor's categories.

Target / group	Node types
Inst. & Call	`module_instantiation`, `udp_instantiation`, `interface_instantiation`, `checker_instantiation`, `program_instantiation`, `subroutine_call`, `system_tf_call`
Span: structural	`loop_generate_construct`, `if_generate_construct`, `case_generate_construct`, `generate_region`, `function_declaration`, `task_declaration`, `interface_declaration`, `package_declaration`
Span: functional	`always_construct`, `always_ff_construct`, `always_comb_construct`, `always_latch_construct`, `conditional_statement`, `case_statement`, `loop_statement`, `seq_block`, `par_block`, `net_declaration`, `continuous_assign`, `blocking_assignment`, `nonblocking_assignment`
Span: verification	`assert_property_statement`, `property_declaration`, `covergroup_declaration`, `initial_construct`

Training corpus construction. We parse the retained source with Tree-sitter[1]. Line targets are sampled from eligible nonempty code lines. Instantiation-and-call targets are identified from instantiation and call nodes, including module instantiations and function or task calls. Structured-span targets are identified from declaration, procedural, and generate nodes. We use the source-line range of a selected node to define the missing region, retaining the preceding and following code as the in-file prefix and suffix. Instantiation-and-call targets as well as structured-span targets are limited to 30 lines. Target comments are removed, and targets that become empty are discarded. Each example retains its repository identifier and file path so that retrieval can access other files from the same project.

Table 3 lists the node types used by the extractor to construct instantiation-and-call candidates as well as structured-span candidates.

RealBench-FIM construction. We start from the AES, E203, and SDC designs in RealBench and parse their original RTL modules with Tree-sitter. Line targets are randomly sampled from eligible

[1] https://tree-sitter.github.io/tree-sitter/

Table 4: Training and validation examples retained after supervision construction.

Split	Repositories	Line	Inst. & Call	Span	Total
Training	2,272	15,141	15,122	15,139	45,402
Validation	106	733	734	734	2,201

Table 5: The 14 RTL repositories sampled for the cross-file reference analysis.

RTL repositories	
0ctobyte/procyon	eisl-nctu/falco
BenBergman/AlteraDE2Labs_Verilog	five-elephants/nux
Elphel/eddr3	hypernyan/eth_vlg
SpiNNakerManchester/spio	intel/aib-phy-hardware
ZipCPU/s6soc	intel/pfr-wilson-city
aap/fpdpga	kdurant/axi-ddr3
aswaterman/trainwreck	lawrie/ulx3s_68k

code lines, while instantiation-and-call targets as well as structured-span targets follow the node-type groups described above. The extracted region serves as the reference target, and the surrounding code forms the in-file prefix and suffix. Each task remains linked to its source module, project dependencies, and existing testbench. The resulting benchmark contains 60 line, 28 instantiation-and-call, and 57 structured-span targets. During evaluation, we reconstruct the module by inserting the model's completion between the prefix and suffix. The original RealBench evaluator simulates the reconstructed module with its dependencies and compares its outputs against reference outputs under the existing testbench inputs.

Dataset splits. The constructed corpus contains 47,647 examples. Supervision construction filters out 44 examples that exceed the generator's context limit after adding retrieved context, leaving 45,402 training examples and 2,201 validation examples. The splits have no shared repositories or task identifiers. Table 4 reports the counts by completion type.

A.2 Cross-file References in Source Code

Source cohort. We randomly sample 14 RTL repositories from the source-repository pool collected during corpus construction. Eligible repositories contain at least 30 target-language files and 5,000 nonblank, noncomment source lines. The RTL sample contains 1,830 Verilog/SystemVerilog files, with repositories listed in Table 5. We use the 14 Python repositories from RepoCoder (Zhang et al., 2023), comprising 2,345 files. The RTL and Python samples contain 326,378 and 340,113 nonblank, noncomment source lines, respectively. Line-level statistics pool these source lines within each language.

Reference resolution. We count explicit references that resolve uniquely to declarations in another file of the same repository. Each reference is assigned to the source line containing its identifier, and repeated references to the same declaration on a line are counted once.

Code segments. We use Tree-sitter to partition source code in both languages into non-overlapping segments based on syntactic structure. Simple statements and declarations remain intact, while compound constructs are recursively divided into inner segments and meaningful headers. Each source fragment belongs to at most 1 segment. Eligible segments have unambiguous reference ownership and do not intersect syntax-error or missing-node spans. For each segment, we collect the distinct starting locations of its referenced external declarations.

Chunk coverage and metrics. We divide declaration files into 20-physical-line chunks with a stride of 10, matching repository chunking. For each source line or eligible code segment, we compute the minimum number of chunks covering its distinct referenced declaration starting locations in each destination file, then sum across files. Line-level percentages use all nonblank, noncomment

Table 6: Best-combination versus Top-k utility on RealBench-FIM. The gap is the best utility minus the Top-k utility.

k	Combinations per target	Targets with a better combination	Mean gap
2	45	115 / 145 (79.31%)	0.0184
3	120	133 / 145 (91.72%)	0.0283
4	210	140 / 145 (96.55%)	0.0306
5	252	143 / 145 (98.62%)	0.0317

source lines as the denominator and count lines with at least 1 resolved cross-file reference or requiring at least 2 chunks. The segment-level distribution uses eligible code segments with resolved cross-file references as the denominator.

RTL contains 48,545 lines with resolved cross-file references and 1,796 lines requiring multiple chunks, accounting for 14.87% and 0.55% of source lines, respectively. Python contains 20,717 and 1,295 such lines, accounting for 6.09% and 0.38%. Among eligible code segments with resolved cross-file references, 1,451 of 5,509 RTL segments (26.34%) and 2,281 of 13,775 Python segments (16.56%) require multiple chunks to cover the referenced declaration locations.

A.3 Single-Chunk and Combination Utility Analysis

For each of the 145 targets in RealBench-FIM, we score all chunks in the retrieved candidate pool with the fixed generator and FIM prompt construction described in Section 3. We retain the 10 chunks with the highest single-chunk utilities and exhaustively evaluate combinations of size $k \in \{2, 3, 4, 5\}$, giving 45, 120, 210, and 252 combinations per target, respectively. Within each combination, chunks are presented in descending single-chunk utility, with ties broken by candidate index. The same ordering rule applies to the Top-k combination and every alternative combination. For each k, the utility gap is the highest utility among combinations of size k minus the utility of the single-chunk-ranked Top-k combination. A target has a better combination when this gap exceeds 10^{-6}. Table 6 reports the target counts and mean gaps. Figure 1(c) plots the fraction of targets with a better combination and the mean gap over all 145 targets, including targets with tied utilities in the mean.

A.4 Retrieval and Training Configuration

Retrieval and search. We split repository files using a sliding window that covers 20 consecutive source lines and advances by 10 lines to form the next chunk. The overlap preserves local context around chunk boundaries. BM25 performs lexical retrieval by matching terms between the query and code, while UniXcoder performs dense retrieval by comparing their vector representations. We merge the top 20 chunks each from BM25 and UniXcoder, retaining at most 40 after deduplication. Search uses a beam width of $b = 3$ and considers up to $m = 10$ candidate extensions per active path at each step, with a maximum combination length of $K_{\max} = 10$ chunks.

For utility scoring during supervision construction, we reserve 512 tokens for the reference target code within a 16,384-token sequence budget. We discard a training example if its target exceeds 512 tokens or a scoring prompt cannot fit the full in-file context and retrieved chunks within the remaining budget.

Reranking and completion. The reranker query uses the last 20 prefix lines, capped at 512 tokens. Each candidate is limited to 2,048 tokens, with an aggregate prompt limit of 131,072 tokens. The trained reranker takes the query and candidate pool together as input and scores all candidates in one forward pass. The top k chunks, or all candidates if fewer are available, form the context combination S in descending score order. The fine-tuned completion generator receives S together with the in-file prefix and suffix through the FIM prompt $F(x_p, x_s, S)$. Figure 4 shows a real prompt example.

Table 7: Training configurations for the reranker and completion generator. Each GPU has 40 GB of memory.

Setting	Reranker	Completion generator
Epochs	1	2
Effective batch size	32	256
Learning rate	6×10^{-6}	10^{-5}
Warmup ratio	0.03	0.03
Precision	BF16	BF16
DeepSpeed	ZeRO-2	ZeRO-3
Hardware	4 A100 SXM4 GPUs	8 A100 PCIe GPUs

Training configuration. Table 7 summarizes the training configuration. The reranker uses ranking temperature $\tau = 0.1$, weight decay 0.01, and seed 42. Generator training uses full-parameter fine-tuning, a cosine learning-rate schedule, and a maximum context length of 16,384 tokens. Section 3.4 defines the training objectives, and Appendix A.6 illustrates the completion prompt.

A.5 Evaluation Details

Baseline configurations. Commercial models receive the task description and in-file prefix and suffix without retrieved repository context. Following their original prompting conventions, we use prefix-continuation, native FIM, and Qwen chat formats for RTLCoder-DeepSeek-v1.1, CodeV-All-QC-7B, and CodeV-R1-Qwen-7B, respectively. These RTL-domain models are evaluated without retrieved repository context. We use context limits of 2,048 tokens for RTLCoder and CodeV-All-QC-7B and 16,384 tokens for CodeV-R1. We train RLCoder and RepoShapley on our corpus. RLCoder uses the fixed Qwen2.5-Coder-7B-Instruct completion generator, while RepoShapley initializes its context-selection and completion model from that checkpoint and trains it using its own objectives. RLCoder and CoRTL share the hybrid pool constructed with BM25 and UniXcoder. RLCoder retains its retriever training objective and selects at most 10 chunks using its learned STOP rule. For RepoShapley, we follow its default candidate budget of 10 to balance context coverage and computational cost. Its exact Shapley computation scales exponentially with the candidate count, while expanding the post-verification scope requires additional generator calls (Huo et al., 2026). We therefore use the first 10 chunks from the shared hybrid pool. CoRTL reranks the full pool. For RTLRepoCoder, Table 1 reports the published DeepSeek-based next-line completion result under its prefix-only setting, marked with †.

Model sizes in Table 1 refer to completion generators. RLCoder and RTLRepoCoder use separate 126M- and 137M-parameter retrievers, respectively, while CoRTL uses a separate 0.6B-parameter reranker. RepoShapley shares its 7B model between context selection and completion.

In the retrieval and reranking comparison in Section 4.3, BM25, UniXcoder, and both rerankers supply 10 chunks when available, while RLCoder selects at most 10 using its learned stopping rule.

Decoding settings. Commercial models use provider-default API sampling settings. All other models use temperature 0.2 and top-p 0.95, with output limits of 2,048 tokens on RealBench-FIM and 256 tokens on RTL-Repo.

Completion output processing. For RealBench-FIM, a shared procedure processes completions using the generated text, supplied suffix, and termination metadata. We first apply the recorded stop boundary or configured chat and Markdown stop markers. We then truncate completions at detected repetitions of the supplied in-file suffix. For outputs recorded as reaching the generation limit, we additionally remove a comment-only tail after a complete code line, or a new code section introduced after complete code by a punctuation-only comment divider, and retain at most 30 physical lines. Naturally stopped outputs use only the stop-boundary and copied-suffix rules. For RTL-Repo, we use the official postprocessing: the first nonempty, noncomment line, with the original fallback for all-comment outputs.

A.6 COMPLETION PROMPT

CoRTL Completion Prompt

```
<|fim_prefix|>
// Here are some relevant code fragments from other files of the repo:
// file path: sd_fifo_tx_filler.v
                                        ...
// sd_tx_fifo Tx_Fifo (
//     .d(din),
//     .wr(wr_tx),
//     .wclk(clk),
//     .q(dat_o),
//     .rd(rd),
//     .full(fe),
//     .empty(empty),
//     .mem_empt(mem_empt),
//     .rclk(sd_clk),
//     .rst(rst | reset_tx_fifo)
// );
                                        ...
// file path: sd_rx_fifo.v
// `include "sd_defines.v"
//
// module sd_rx_fifo
//   (
//    input [4-1:0] d,
//    input wr,
//    input wclk,
//    output [32-1:0] q,
//    input rd,
//    output full,
//    output empty,
//    output [1:0] mem_empt,
//    input rclk,
//    input rst
//    );
                                        ...

// Based on above, complete the following code:
// file path: sd_fifo_rx_filler.v
                                        ...
    input sd_clk,
    input [`SD_BUS_W-1:0] dat_i,
    input wr,
    output full,
    output empty
);
                                        ...
    wire [31:0] dat_o;
    reg rd;
    reg reset_rx_fifo;
                                        ...
<|fim_suffix|>
                                        ...
    assign m_wb_adr_o = adr + offset;
                                        ...
<|fim_middle|>
```

Generated Completion

```
sd_rx_fifo Rx_Fifo (
    .d(dat_i), .wr(wr), .wclk(sd_clk), .q(dat_o),
    .rd(rd), .full(full), .empty(empty), .rclk(clk),
    .rst(rst | reset_rx_fifo)
);
```

Figure 4: A prompt and generated completion for an SDC module instantiation in RealBench-FIM. Ellipses indicate omitted context.