

ICL4Decomp: A Step Towards Re-executable Decompile

Xiaohan Wang

Vanderbilt University
Nashville, TN, USA

Yuxin Hu

Vanderbilt University
Nashville, TN, USA

Kevin Leach*

Vanderbilt University
Nashville, TN, USA

Abstract

Binary decompilation, which aims to recover source code from compiled binaries, can be viewed as a low-resource and high-constraint form of neural machine translation. However, existing decompilation techniques often fail to produce source code that can be successfully recompiled and re-executed, particularly for binaries that have undergone compiler optimizations. Recent advances in large language models (LLMs) have enabled neural approaches to decompilation, but the generated code is typically only semantically plausible rather than truly executable, limiting their practical usability. These shortcomings arise from compiler optimizations and the loss of semantic cues in compiled code, which LLMs struggle to recover without contextual guidance. To address this challenge, we propose ICL4Decomp, a hybrid decompilation framework that leverages in-context learning (ICL) to guide LLMs toward generating re-executable source code in a single inference pass. Notably, our approach is model-agnostic and requires no fine-tuning, enabling plug-and-play use with off-the-shelf LLMs. We evaluate ICL4Decomp across HumanEval-Decompile, ExeBench, and real-world GitHub2025 functions, spanning multiple optimization levels and compilers. ICL4Decomp demonstrates around 30% improvement in re-executability over state-of-the-art neural decompilation methods while maintaining robustness and practical inference cost.

1 Introduction

Binary decompilation can be conceptualized as a specialized instance of neural machine translation (NMT): translating low-level binaries into high-level source code. It plays a critical role in security auditing, vulnerability analysis, and malware reverse engineering, particularly when the original source code is unavailable (Cifuentes

et al., 2001; Yakdan et al., 2016). Despite decades of research, producing readable and executable source code from optimized binaries remains a challenging problem.

Unlike traditional natural language translation, the fundamental difficulty of binary translation lies in the fact that compilation is a lossy process (Wilhelm et al., 2013). During compilation, high-level semantic information, such as variable names, control structures, and type annotations, is discarded or transformed (Cao and Leach, 2023; Yang et al., 2025a). This loss is exacerbated by aggressive compiler optimizations, which further obscure data flow and control structure, making accurate decompilation increasingly difficult.

Traditional decompilers rely on heuristic rules and control-flow analysis to reconstruct source-level structures (Hex-Rays; National Security Agency, 2025). While effective for unoptimized binaries, these tools often struggle under higher optimization levels, frequently producing output that is incomplete, misleading, or not recompileable. As a result, the generated pseudo-code may fail to preserve the original program semantics or support downstream reuse.

Recently, neural and LLM-based decompilation approaches (e.g., Dire (Lacomis et al.), DeGPT (Hu et al.), LLM4Decompile (Tan et al., 2024)) generate readable code by learning syntactic patterns from large-scale pre-training. However, these methods primarily rely on statistical correlations between assembly and source code and optimize for textual similarity metrics such as BLEU (Papineni et al., 2002) and ROUGE (Lin, 2004), rather than explicitly enforcing semantic correctness or executability. As a result, they often fail to account for compiler optimization semantics, leading to errors in types, boundary conditions, and control logic, especially for optimized binaries.

These limitations highlight two key challenges in executable decompilation. **Challenge 1: Recover-**

*Corresponding author.

ing Lost Semantics. Stripped binaries lack explicit semantic cues such as variable names and high-level control structures. Existing approaches struggle to robustly infer this information, particularly under aggressive optimization. **Challenge 2: Re-compilability and Re-executability are second-class.** Even when decompiled code appears plausible, it is rarely recompilable or behaviorally equivalent to the original binary, limiting its practical usefulness in real-world scenarios. For reverse engineering workflows, practical usefulness also depends on reproducibility, latency, and deployment cost: a decompiler that requires specialized retraining or many rounds of feedback may be difficult to apply at scale.

To address these challenges, we present **ICL4Decomp**, an in-context learning framework (Dong et al.; Wies et al., 2023) for executable decompilation. Rather than relying on model retraining, dedicated deployment infrastructure, or purely heuristic reconstruction, ICL4Decomp conditions a pretrained language model on carefully-designed contextual information in a single inference pass. The framework integrates retrieved assembly–source exemplars and optimization-aware semantic guidance to improve both structural recovery and behavioral correctness of decompiled code.

We evaluate ICL4Decomp on multiple datasets across different compilers (GCC and Clang) and optimization levels (O0–O3), using re-executability rate as the primary metric, which measures whether the generated source code compiles and produces behavior consistent with the original binary on a held-out test suite. We further evaluate on GitHub2025 functions from Decompile-Bench to test generalization beyond algorithmic and benchmark-contained programs. Experimental results demonstrate substantial improvements over state-of-the-art baselines, achieving an average increase of approximately 30% in re-executability. The gains are particularly pronounced at higher optimization levels, where compiler transformations introduce greater semantic ambiguity, while ICL4Decomp maintains robust performance across all optimization settings. Profiling shows that retrieval contributes little to end-to-end latency relative to LLM generation, making the approach practical for batch decompilation.

Our main contributions are as follows:

- We introduce ICL4Decomp, an in-context

learning framework that improves executable decompilation without model retraining.

- We demonstrate substantial gains in recompilability and semantic correctness across datasets, compilers, optimization levels, and real-world GitHub functions.
- We provide empirical analysis showing that contextual guidance improves robustness under aggressive compiler optimizations, and we characterize the latency and cost of the inference-time pipeline.

2 Related Work

2.1 Foundations and Traditional Binary Analysis

Wilhelm and colleagues characterize compilation as a sequence of transformations from high-level programs to lower-level representations (Wilhelm et al., 2013). Cifuentes and colleagues demonstrate the value of decompilation and high-level debugging for security analysis (Cifuentes et al., 2001). Yakdan and colleagues study a usability-oriented decompiler in the context of malware analysis (Yakdan et al., 2016). IDA Pro is a commercial platform for disassembly and decompilation (Hex-Rays). Ghidra is an open-source reverse-engineering framework with decompilation support (National Security Agency, 2025). ByteWeight learns to identify function boundaries in binary code (Bao et al., 2014). The rev-ng framework recovers control-flow graphs and function boundaries within a unified binary-analysis framework (Di Federico et al., 2017).

2.2 Neural and LLM-based Decompilation

Sequence-to-sequence learning provides the general translation framework that motivates neural decompilation (Sutskever et al., 2014). Katz and colleagues explore recurrent neural networks for translating binary code into C source code (Katz et al., 2018). Katz and colleagues further investigate neural approaches to end-to-end decompilation (Katz et al., 2019). Hosseini and Dolan-Gavitt study neural-machine-translation decompilation that can target languages beyond C (Hosseini and Dolan-Gavitt, 2022). Coda is an end-to-end neural framework for program decompilation (Fu et al., 2019). DIRE addresses the complementary problem of restoring informative identifier names in decompiled code (Lacomis et al.). LLM4Decompile

uses large language models to generate source code from binaries (Tan et al., 2024). SLaDe develops a portable small language-model decompiler for optimized assembly (Armengol-Estapé et al., 2024). DeGPT improves decompiler output through language-model reasoning and compiler feedback (Hu et al.). D-LIFT improves an LLM-based decompiler backend through code-quality-driven fine-tuning (?). ReF Decompile enriches decompilation with relabeling and function-call information (Feng et al.). SK2Decompile separates LLM-based decompilation into skeleton recovery and detail recovery stages (Tan et al., 2026).

2.3 Binary-code Representation and LLM Analysis

NOVA learns representations of assembly code with hierarchical attention and contrastive learning (Jiang et al., 2023). Jin and colleagues benchmark large language models on binary-code summarization (Jin et al., 2023). KTrans learns knowledge-aware embeddings for binary code (Zhu et al., 2023). Recent work has further explored learned representations and language-model-based techniques for recovering semantic information from low-level programs (Wang and Leach, 2026). ReSym uses large language models to recover variables and data-structure symbols from stripped binaries (Xie et al., 2024). Su and colleagues show that source-code foundation models transfer useful knowledge to binary analysis (Su et al., 2024). BinMetric provides a comprehensive benchmark for evaluating large language models on binary-analysis tasks (Shang et al., 2025). Liu and colleagues analyze hallucinations in LLM-powered code generation and their effect on correctness (Liu et al., 2024). Cao and Leach reassess deep-learning methods for variable-type recovery (Cao and Leach, 2023). Yang and colleagues conduct a human study of automatically generated decompiler annotations (Yang et al., 2025a).

2.4 In-context Learning and Retrieval-augmented Code Generation

Brown and colleagues show that large language models can adapt to tasks through few-shot in-context examples (Brown et al., 2020). Dong and colleagues survey the mechanisms and applications of in-context learning (Dong et al.). Wies and colleagues study the conditions under which in-context learning is learnable (Wies et al., 2023). CodeGen2 reports lessons from training large lan-

guage models on programming and natural language (Nijkamp et al., 2023). Yang and colleagues empirically examine the opportunities and limitations of retrieval-augmented code generation (Yang et al., 2025b).

2.5 Datasets Benchmarks and Evaluation

Chen and colleagues introduce HumanEval for evaluating the functional correctness of code-generating language models (Chen et al., 2021). ExeBench provides a machine-learning-scale dataset of executable C functions (Armengol-Estapé et al., 2022). DecompileBench evaluates decompilers on real-world scenarios (Gao et al., 2025). The million-scale Decompile-Bench dataset supplies binary–source function pairs for real-world decompilation (Tan et al., 2025). BLEU measures n-gram overlap between generated and reference text (Papineni et al., 2002). ROUGE provides a family of reference-overlap metrics for generated text (Lin, 2004).

3 Approach

We propose ICL4Decomp, an in-context learning framework for binary decompilation that aims to recover high-level, recompilable source code from optimized assembly. The key insight is that large language models can be guided to better reconstruct program structure and semantics when conditioned on carefully designed contextual information at inference time, without any model retraining.

Given a target assembly function, ICL4Decomp constructs a context and conditions a frozen large language model to directly generate corresponding source code. The framework incorporates two complementary forms of contextual guidance: (i) retrieved assembly–source exemplars that expose concrete instruction-to-structure correspondences, and (ii) optimization-aware natural language rules that encode compiler transformation semantics. Figure 1 provides an overview of the framework.

3.1 Problem Setup

Let A denote a compiled assembly function and S its corresponding high-level source implementation. The goal of decompilation is to recover a source program $\hat{S}$ such that, when recompiled, it exhibits behavior equivalent to the original binary. In this work, we focus on *re-executable decompilation*, where success requires both syntactic correctness (the generated code compiles)

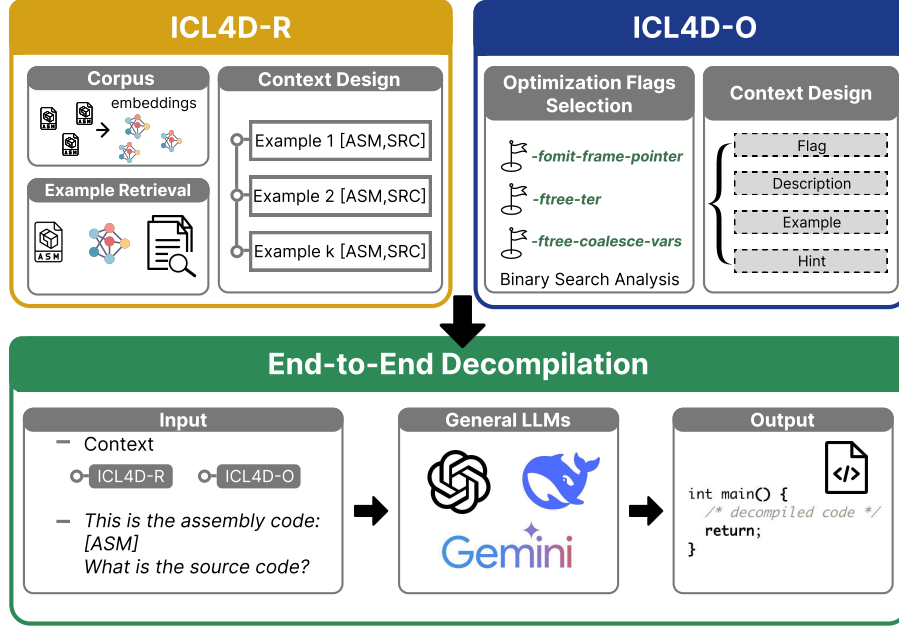


Figure 1: System overview for in-context decompilation.

and semantic equivalence under a held-out test suite. Our input is function-level assembly. Function boundary recovery, code–data separation, and disassembly are treated as upstream preprocessing steps, following the standard protocol used by neural decompilation benchmarks.

We formalize in-context decompilation as conditional generation with a frozen language model f_θ :

$$\hat{S} = f_\theta(A, C),$$

where C denotes contextual information provided at inference time. No parameter updates or fine-tuning are performed; all adaptation occurs purely through prompt conditioning.

3.2 Retrieved-Exemplar In-Context Decompilation (ICL4D-R)

The first variant of our framework, **ICL4D-R**, leverages retrieved assembly–source exemplars as in-context demonstrations. Given a target assembly function A_t , we retrieve a small set of semantically similar function pairs $\{(A_i, S_i)\}$ from a pre-constructed corpus and include them in the prompt prior to the target assembly.

Each exemplar provides an explicit mapping between low-level instructions and high-level program structure, allowing the model to observe how control flow, expressions, and variable usage are recovered under similar compilation patterns. By conditioning on multiple such demonstrations, the

language model implicitly adapts to the compiler style and optimization level of the target function.

The retrieved exemplars are ordered by semantic similarity and formatted as alternating assembly and source segments, followed by the target assembly and a decompilation instruction. Retrieval is performed over assembly embeddings rather than source symbol names, so exemplar selection does not require debug symbols or preserved identifiers. This structured prompting exposes the model to concrete instruction-to-structure correspondences before generation. Details of corpus construction, embedding, similarity computation, and retrieval are provided in Appendix A, while the prompt formatting and exemplar organization used for in-context decompilation are described in Appendix C.

3.3 Optimization-Aware In-Context Decompilation (ICL4D-O)

While retrieved exemplars are effective when local instruction–structure correspondences are preserved, aggressive compiler optimizations often introduce non-local transformations that obscure data flow and control structure. To address this challenge, we introduce **ICL4D-O**, a rule-based variant that augments the prompt with optimization-aware contextual guidance.

ICL4D-O encodes compiler optimization semantics as natural language descriptions that explain how transformations affect assembly. These rules

inform the model that certain source-level constructs, such as temporary variables or stack frames, may be absent due to semantics-preserving compiler optimizations, and should be reconstructed accordingly during decompilation.

Rather than attempting to infer optimization behavior implicitly, ICL4D-O explicitly conditions generation on relevant optimization rules, guiding the model’s reasoning about data flow, variable lifetimes, and control structure. This approach is particularly beneficial at higher optimization levels, where conventional decompilation and purely exemplar-based prompting tend to fail. The identification of influential optimization flags and the design of rule-based prompts are described in Appendix B and Appendix C, respectively.

3.4 End-to-End Decompilation Pipeline

Both ICL4D-R and ICL4D-O operate in an end-to-end manner at inference time. Given a target assembly function, the framework constructs a prompt by selecting either retrieved exemplars, optimization-aware rules, or both, and conditions a pretrained language model to generate the corresponding source code in a single forward pass. No symbolic execution, control-flow reconstruction, or post-hoc repair is performed.

This design combines the flexibility of in-context learning with compiler-aware semantic grounding, enabling the generation of source code that is not only readable but also recompilable and executable across diverse optimization levels.

4 Evaluation

In this section, we evaluate our in-context decompilation framework by addressing research questions:

- **RQ1 (Executability):** Can in-context learning improve the re-executable rate compared to baselines?
- **RQ2 (Error Mitigation):** Does in-context guidance mitigate specific compilation and runtime errors?
- **RQ3 (Robustness):** How robust is the framework across varying program complexities?

4.1 Experimental Setup

Datasets. We utilize two datasets covering system-level and algorithmic domains. Statistics are detailed in Table 2. (I) EXEBENCH (Armengol-Estapé et al., 2022): A machine-learning-scale

dataset of real-world C functions. We use the `test-real` subset ($N = 1,135$), containing functions with concrete auxiliary definitions suitable for I/O-driven synthesis. (II) HUMANEVAL-DECOMPILE: Adapted from HUMANEVAL (Chen et al., 2021), comprising 1,312 Python problems translated into executable C/C++ by LLM4Decompile (Tan et al., 2024). Both datasets cover optimization levels O0–O3. Samples are normalized (see §3.2) and validated by SHA-256 to ensure no overlap with the retrieval corpus.

Implementation & Baselines. We employ DEEPSEEK-V3.2 as the primary generation model and NOVA-1.3B (Jiang et al., 2023) as the embedding encoder (1024-d). We compare our approach against: (1) **DeepSeek-V3.2** (zero-shot baseline); (2) **LLM4Decompile-End** (Tan et al., 2024) (state-of-the-art learning-based baseline); (3) **Ghidra** (National Security Agency, 2025), an open-source traditional decompiler baseline chosen for reproducibility; and (4) **IDA (Hex-Rays)**, a commercial decompiler baseline reported for HumanEval-Decompile where IDA-generated assembly is available. We evaluate two variants of our framework: **ICL4D-R** (Retrieval-Augmented) and **ICL4D-O** (Rule-Guided). Generation uses temperature 0.1, exemplar count $k = 5$, and max tokens 10,000.

Metrics. We report the **Executable Success Rate (ESR)**, defined as the proportion of decompiled functions $\hat{S}_t$ that compile at `-O0` and pass all I/O test cases within a 5-second timeout, ensuring both syntactic and semantic correctness.

4.2 RQ1: Improvement in Executability

We first assess the overall executability improvements provided by our in-context strategies.

Performance of ICL4D-R. As shown in Table 1, **ICL4D-R** consistently outperforms all baselines across both datasets and all optimization levels. On *HumanEval-Decompile*, it achieves **54.3%** accuracy at O0, surpassing DeepSeek-V3.2 (46.7%) and LLM4Decompile (26.8%). On the more complex *ExeBench*, ICL4D-R demonstrates up to a **30% improvement** over learning-based baselines. This confirms that retrieval-based exemplars effectively help the model generalize across diverse compiler transformations.

Table 1: Re-executability rate (%) comparison on HumanEval-Decompile and ExeBench. Ghidra is the reproducible open-source decompiler baseline; IDA is reported where dataset-provided IDA assembly is available.

Method	HumanEval-Decompile					ExeBench				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
Ghidra	12.50	17.07	12.50	11.28	13.34	16.37	18.94	16.66	17.64	17.40
IDA	18.60	19.82	17.68	16.77	18.22	N/A	N/A	N/A	N/A	N/A
LLM4Decompile-1.3B	26.78	11.53	13.22	11.53	15.77	15.10	12.83	13.22	11.53	13.17
DeepSeek-V3.2	46.65	32.32	33.23	35.06	36.82	26.17	19.26	19.79	16.13	20.34
ICL4D-R	54.27	42.38	40.24	42.07	44.74	34.39	35.68	36.16	33.74	34.99

Table 2: Evaluation dataset statistics (mean $\pm$ std) per function. EB: ExeBench; HE-D: HumanEval-Decompile.

Dataset	#N	LOC	Cycl.	Blocks
EB	1135	13.6 $\pm$ 16.1	3.4 $\pm$ 15.3	3.5 $\pm$ 15.3
HE-D	1312	13.3 $\pm$ 8.4	4.9 $\pm$ 3.1	5.1 $\pm$ 3.0

Table 3: Re-executability rate (%) after applying rule-based prompts (ICL4D-O) for different optimization options across two datasets. Values in parentheses indicate baseline performance.

Option	ExeBench	HumanEval
O1 (baseline: ExeBench 11.18%, HumanEval 14.86%)		
-fomit-frame-pointer	9.34	15.32
-ftree-ter	8.17	15.77
-ftree-coalesce-vars	11.67	12.61
-fipa-pure-count	8.95	11.26
O2 (baseline: ExeBench 7.48%, HumanEval 15.98%)		
-fomit-frame-pointer	9.52	12.41
-ftree-ter	7.87	16.89
-ftree-coalesce-vars	10.63	17.35
-fipa-pure-count	6.72	10.50
-fcrossjumping	7.91	5.05
O3 (baseline: ExeBench 3.18%, HumanEval 15.02%)		
-fomit-frame-pointer	8.18	14.08
-ftree-ter	5.48	15.49
-ftree-coalesce-vars	10.96	11.27
-fipa-pure-count	6.40	7.51
-fcrossjumping	5.45	12.21

Performance of ICL4D-O. Since decompilation failures increase at higher optimization levels, we apply the rule-based **ICL4D-O** to recover samples that failed in the first round (O1–O3). Table 3 shows that while less stable than retrieval, ICL4D-O yields selective gains. For instance, prompting with `-ftree-coalesce-vars` at O2 improves executability to 17.35% on HumanEval. While rule-based prompting can be rigid, manual inspection suggests it produces more localized, repairable errors, which we analyze further in RQ2.

Answer to RQ1: In-context learning dramatically improves executability. **ICL4D-R** provides robust gains across all settings, while **ICL4D-O** offers targeted improvements for high-optimization failures.

4.3 RQ2: Error Analysis and Mitigation

To understand *how* in-context learning improves performance, we analyze the shift in failure modes using a taxonomy derived from compiler `stderr` diagnostics (Table 4).

Error Distribution Shift. Figure 2 illustrates the transition of error categories from the baseline to ICL4D-R.

- **HumanEval (Fig. 2):** We observe a substantial reduction in *Syntax*, *Runtime*, and *Declaration* errors converting into successes. This indicates that exemplars aid in reconstructing valid syntactic structures and resolving undefined symbols.
- **ExeBench (Fig. 2):** Improvements are concentrated in *Type* and *Syntax* categories. Given ExeBench’s structural complexity, this suggests ICL effectively enhances semantic reasoning and type inference.

Rule-Guided Error Localization. For ICL4D-O, Table 5 reveals a trade-off: while it reduces *Declaration* and *Type* errors, it often increases *Syntax* errors. This shift is beneficial: global structural failures (like missing declarations) are converted into localized syntax errors, which are generally easier to repair.

Qualitative Analysis. Figure 3 presents a case study from ExeBench. The ground truth involves parsing digits into an integer. **DeepSeek-V3.2** hallucinates a constant boolean return, and

Table 4: Error taxonomy for `stderr` diagnostics.

Category	Example cause / message
Assert	Assertion failure or output mismatch (<i>assertion failed</i>)
Syntax	Token or structure error (<i>expected ';' , unterminated string</i>)
Return	Invalid return or argument count (<i>void value not ignored</i>)
Type	Incompatible or invalid type (<i>invalid conversion</i>)
Declaration	Missing or conflicting symbol (<i>undefined reference</i>)
Runtime/Link	Crash or linking error (<i>segmentation fault</i>)
Other	Uncategorized message

Table 5: HumanEval (ICL4D-O) rule-wise error distribution (%), aggregated over O1–O3. Baseline is averaged over O1–O3. Arrows ($\downarrow/\uparrow/\approx$) indicate change relative to baseline. Rule abbreviations: FC (`-ftree-coalesce-vars`), FP (`-fipa-pure-const`), FT (`-ftree-ter`), FFP (`-fomit-frame-pointer`), FJ (`-fcrossjumping`).

Category	Base	FC	FP	FT	FFP	FJ
Syntax	15.3	17.2 $\uparrow$	15.4 $\approx$	19.8 $\uparrow$	16.0 $\uparrow$	25.8 $\uparrow$
Declaration	27.3	12.3 $\downarrow$	30.9 $\uparrow$	23.3 $\downarrow$	23.9 $\downarrow$	16.4 $\downarrow$
Type	6.0	4.6 $\downarrow$	8.8 $\uparrow$	8.1 $\uparrow$	5.6 $\approx$	8.3 $\uparrow$
Return	0.7	1.1 $\approx$	0.2 $\downarrow$	1.6 $\uparrow$	0.7 $\approx$	0.5 $\approx$
Assert	41.5	49.6 $\uparrow$	40.3 $\downarrow$	39.0 $\downarrow$	42.9 $\uparrow$	33.3 $\downarrow$
Other	9.1	15.3 $\uparrow$	4.4 $\downarrow$	8.1 $\downarrow$	10.8 $\uparrow$	15.6 $\uparrow$

LLM4Decompile introduces spurious bitwise operations. In contrast, **ICL4D-R** correctly recovers the loop bounds and arithmetic logic, demonstrating that retrieval contexts prevent control-flow collapse and semantic hallucination.

Answer to RQ2. In-context learning mitigates structural errors in decompilation. **ICL4D-R** effectively reduces syntax and declaration failures, improving structural and symbolic consistency, while **ICL4D-O** produces more localized and repairable errors.

4.4 RQ3: Robustness and Ablation

Robustness (RQ3). We stratify performance by cyclomatic complexity (Figure 4); the complementary analysis by lines of code is reported in Appendix D. While all models degrade as complexity increases, ICL4D-R (Ours) exhibits a slower rate of decay, particularly in the mid-complexity range (5–10 branches). This indicates stronger generalization to intricate control flows compared to the baseline.

Table 6: Ablation on random retrieval control. The random-retrieval variant replaces semantically similar examples with random samples while preserving context length.

HumanEval-Decompile				
Model	O0	O1	O2	O3
DeepSeek-V3.2	46.65	32.32	33.23	35.06
ICL4D-R	54.27	42.38	40.24	42.07
ICL4D-R (Ablation)	42.25	40.49	41.34	0.35
ExeBench				
Model	O0	O1	O2	O3
DeepSeek-V3.2	26.17	19.26	19.79	16.13
ICL4D-R	34.39	35.68	36.16	33.74
ICL4D-R (Ablation)	25.50	21.62	19.45	15.72

Answer to RQ3. ICL4D-R demonstrates strong robustness as cyclomatic complexity increases, consistently outperforming the baseline and maintaining higher re-execution success rates.

Ablation Study. To disentangle the effect of *semantic retrieval* from that of merely providing *additional few-shot context*, we construct a controlled baseline termed **Random Retrieval** (Table 6). This variant preserves the *exact prompt structure*, *exemplar count* ($k = 5$), and *context length*, but replaces semantically matched assembly–source pairs with randomly sampled ones.

Performance degrades substantially and consistently across datasets and optimization levels. For example, on HumanEval-Decompile at O3, executability drops from 42.07% to 0.35%. Similar degradation is observed on ExeBench (e.g., O3: 33.74% $\rightarrow$ 15.72%).

These results demonstrate that improvements are not attributable to extended context alone. Instead, semantic alignment of retrieved exemplars is essential for guiding structural and behavioral reconstruction during decompilation.

4.5 Generalization to Real-World GitHub Functions

HumanEval-Decompile and ExeBench provide executable tests, but they are still dominated by small, self-contained functions. To evaluate generalization on more realistic code, we additionally sample functions from GitHub2025 in Decompile-Bench (Tan et al., 2025). The evaluation set contains 600 unique functions, and the retrieval database contains 2,400 disjoint functions. Both

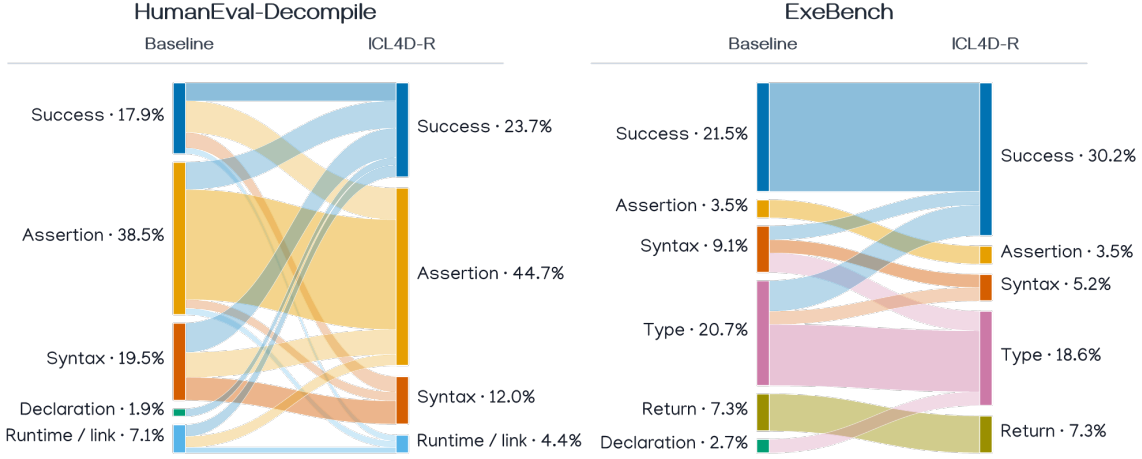


Figure 2: Major error-category transitions from the baseline to ICL4D-R. Each band represents a listed transition and its width encodes the percentage of all evaluated samples. To preserve readability, the figure shows the top 15 transitions for HumanEval-Decompile and the 10 transitions available for ExeBench; colours encode the post-ICL4D-R category.

Table 7: Edit similarity on GitHub2025 functions from Decompile-Bench.

Method	O0	O1	O2	O3
DeepSeek-V3.2	0.0806	0.0759	0.0658	0.0707
LLM4Decomp-DC	0.1635	0.1315	0.1312	0.1248
ICL4D-R	0.2369	0.2468	0.2145	0.1826

Table 8: LLM-as-a-Judge scores on GitHub2025 functions from Decompile-Bench.

Method	Avg.	O0	O1	O2	O3
DeepSeek	26.7	32.4	23.3	21.1	23.5
LLM4Decomp	10.2	12.9	8.8	7.8	8.7
ICL4D-R	35.6	40.2	36.0	34.6	30.5

sets are deduplicated and separated from the main evaluation benchmarks.

GitHub2025 does not provide executable test cases for each function, so we treat this experiment as supplementary evidence rather than a replacement for ESR. We report edit similarity and an LLM-as-a-Judge score, using a structured rubric with 70% semantic correctness, 20% compilability, and 10% readability. LLM-based judging has also been used in recent decompilation evaluations (Hu et al.; Tan et al., 2024; Feng et al.; Tan et al., 2026).

ICL4D-R outperforms both baselines across all optimization levels under both metrics. The high-score tail is stronger: 80 out of 600 ICL4D-R outputs score at least 80 under the judge, compared with 18 out of 600 for DeepSeek-V3.2. These re-

sults suggest that semantically aligned exemplars generalize beyond algorithmic benchmarks to more diverse real-world GitHub functions.

5 Discussion

5.1 Applicability

ICL4Decomp is model-agnostic and can run with either local open-source models or commercial APIs, depending on privacy and deployment constraints. Because it operates entirely at inference time without retraining, it can be integrated with existing IDA- or Ghidra-style workflows through plugin interfaces.

5.2 Cost and Runtime Analysis

The primary cost is LLM invocation over the target assembly, retrieved exemplars, and generated source; with *DeepSeek V3.2*, decompiling about 1,500 HumanEval-Decompile functions costs roughly \$3.

Table 9: Interactive single-function latency breakdown for ICL4D-R with a FAISS index and $k = 5$ retrieved exemplars.

Component	Mean latency (ms/function)
Embedding	290.70
FAISS retrieval	6.00
LLM generation	10579.29
End-to-end	10875.99

Table 9 shows that retrieval is not the bottleneck:

Source Code

```
unsigned int func0(char *string, unsigned int *addr) {
    unsigned int pos = 0;
    *addr = 0;
    while (string[pos] >= '0' && string[pos] <= '9'
        && pos < 9)
        *addr = *addr * 10 + (string[pos++] - '0');
    return pos;
}
```

ICL4Decomp-R

```
unsigned int func0(const char *s, unsigned int *u)
{
    unsigned int pos = 0, result = 0, c;
    *u = 0;
    while ((c = (unsigned int)(s[pos] - '0')) < 10) {
        result = result * 10 + c;
        pos++;
    }
    *u = result;
    return pos;
}
```

DeepSeek-V3.2

```
void func0(const char *str, int *result) {
    *result = 0;
    char c = *str;
    if (c >= '0' && c <= '9')
        *result = 1;
}
```

LLM4Decompile-End-1.3B

```
int func0(char *s, int *pi) {
    unsigned char c = *s - '0';
    if (c > 9) return get(&s);
    unsigned char x = 1, c0 = c;
    c &= ~0xff;
    c0 <= 6;
    c ^= c0;
    *pi = c;
    return x;
}
```

Figure 3: Qualitative comparison on an ExeBench function. Green highlights show the loop and decimal accumulation recovered by ICL4Decomp-R; red highlights show the spurious operations introduced by LLM4Decompile.

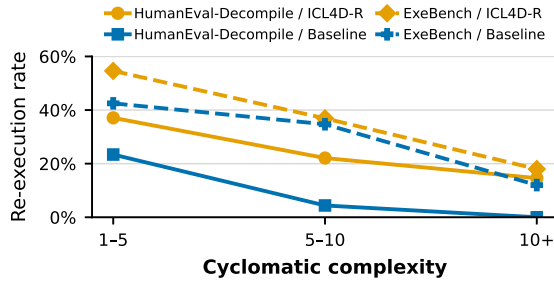


Figure 4: Re-execution success rate by cyclomatic complexity. Solid lines denote HumanEval-Decompile and dashed lines denote ExeBench; ICL4D-R consistently outperforms the baseline.

FAISS lookup adds about 6 ms per function, while LLM generation dominates latency. With 20 functions and 4 parallel workers, the pipeline reaches 0.166 functions/s, or about 6.0 s per function.

5.3 Relation to Agentic Refinement

Compiler-feedback agents are complementary, but they require multi-round LLM calls with different latency and cost, whereas this work evaluates single-call re-executability.

6 Conclusion

This paper presents **ICL4Decomp**, an in-context learning framework for executable decompilation with two mechanisms: **ICL4D-R**, which retrieves assembly–source exemplars, and **ICL4D-O**, which injects optimization semantics through

rule-based prompts. Across ExeBench and HumanEval-Decompile, ICL4Decomp improves re-executability, especially under O1–O3, and GitHub2025 results show that retrieval also improves real-world edit similarity and judge-based quality without model finetuning.

Future work includes hybrid retrieval/rule contexts, inference of compiler optimization patterns, and extension from single-function recovery to multi-function and repository-level decompilation.

Artifact Availability

The artifact containing the implementation and experimental materials is available at <https://github.com/hannahwang37/ICL4Decomp>.

Limitations

Our evaluation follows a controlled function-level protocol, which is standard for neural decompilation but does not cover full repository recovery, cross-function state propagation, or project-level type reconstruction. Although we evaluate O0–O3, the benchmark excludes link-time optimization, obfuscation, indirect calls, inline assembly, and complex exception handling. Finally, ICL4Decomp targets single-call re-executability and does not include an agentic compiler-feedback baseline, whose multi-round repair setting has different cost and latency.

Ethics Statement

Improved decompilation can be misused to recover proprietary logic or bypass software protections, so our intended use is defensive analysis, malware understanding, and maintenance of legacy binaries. ICL4Decomp relies on in-context learning over public datasets and does not require private codebases or user data for finetuning.

References

- Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley De Souza Magalhães, and Michael F. P. O’Boyle. 2022. [Exebench: An ml-scale dataset of executable c functions](#). In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 50–59, San Diego CA USA. ACM.
- Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael FP O’Boyle. 2024. Slade: A portable small language model decompiler for optimized assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 67–80. IEEE.
- Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. {BYTEWEIGHT}: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 845–860.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, and 1 others. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.
- Kevin Cao and Kevin Leach. 2023. Revisiting deep learning for variable type recovery. In *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, pages 275–279. IEEE.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Cristina Cifuentes, Trent Waddington, and Mike Van Emmerik. 2001. Computer security analysis through decompilation and high-level debugging. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 375–380. IEEE.
- Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: A unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 131–141.
- Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Jingyuan Ma, Rui Li, Heming Xia, Jingjing Xu, Zhiyong Wu, Tianyu Liu, Baobao Chang, Xu Sun, Lei Li, and Zhi-fang Sui. [A survey on in-context learning](#). *Preprint*, arXiv:2301.00234.
- Yunlong Feng, Bohan Li, Xiaoming Shi, Qingfu Zhu, and Wanxiang Che. [Ref decompile: Relabeling and function call enhanced decompile](#). *Preprint*, arXiv:2502.12221.
- Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuan-dong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32.
- Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Zhang Bolun, and Chao Zhang. 2025. Decompilebench: A comprehensive benchmark for evaluating decompilers in real-world scenarios. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 23250–23267.
- Hex-Rays. [IDA Pro: Powerful disassembler, decompiler & debugger](#) [online].
- Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the c: Retargetable decompilation using neural machine translation. *arXiv preprint arXiv:2212.08950*.
- Peiwei Hu, Ruigang Liang, and Kai Chen. [Degpt: Optimizing decompiler output with llm](#). In *Proceedings 2024 Network and Distributed System Security Symposium*. Internet Society.
- Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. 2023. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. *arXiv preprint arXiv:2311.13721*.
- Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601*.
- Deborah S Katz, Jason Ruchti, and Eric Schulte. 2018. Using recurrent neural networks for decompilation. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, pages 346–356. IEEE.
- Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. 2019. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*.
- Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. [Dire: A neural approach to decompiled identifier naming](#). In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE.

- Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.
- Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*.
- National Security Agency. [Ghidra](#) [online]. 2025.
- Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.
- Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.
- Xiuwei Shang, Guoqiang Chen, Shaoyin Cheng, Benlong Wu, Li Hu, Gangyang Li, Weiming Zhang, and Nenghai Yu. 2025. Binmetric: A comprehensive binary analysis benchmark for large language models. *arXiv preprint arXiv:2505.07360*.
- Zian Su, Xiangzhe Xu, Ziyang Huang, Kaiyuan Zhang, and Xiangyu Zhang. 2024. Source code foundation models are transferable binary analysis knowledge bases. *Advances in Neural Information Processing Systems*, 37:112624–112655.
- Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27.
- Hanzhuo Tan, Weihao Li, Xiaolong Tian, Siyi Wang, Jiaming Liu, Jing Li, Yuqun Zhang, and 1 others. 2026. SK2Decompile: Llm-based two-phase binary decompilation from skeleton to skin. In *International Conference on Learning Representations*, volume 2026, pages 89731–89762.
- Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3473–3487.
- Hanzhuo Tan, Xiaolong Tian, Hanrui Qi, Jiaming Liu, Zuchen Gao, Siyi Wang, Qi Luo, Jing Li, and Yuqun Zhang. 2025. Decompile-bench: Million-scale binary-source function pairs for real-world binary decompilation. *arXiv preprint arXiv:2505.12668*.
- Xiaohan Wang and Kevin Leach. 2026. Still: Recovering lowered stl semantics for llm-assisted c++ decompilation. *arXiv preprint arXiv:2608.26408*.
- Noam Wies, Yoav Levine, and Amnon Shashua. 2023. The learnability of in-context learning. *Advances in Neural Information Processing Systems*, 36:36637–36651.
- Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. 2013. *Compiler design: syntactic and semantic analysis*. Springer Science & Business Media.
- Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4554–4568.
- Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177. IEEE.
- Yuwei Yang, Skyler Grandel, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, Claire Le Goues, and Kevin Leach. 2025a. A human study of automatically generated decompiler annotations. In *2025 55th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 129–142.
- Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025b. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology*.
- Wenyu Zhu, Hao Wang, Yuchen Zhou, Jiaming Wang, Zihan Sha, Zeyu Gao, and Chao Zhang. 2023. ktrans: Knowledge-aware transformer for binary code embedding. *arXiv preprint arXiv:2308.12659*.

A Retrieval Infrastructure and Corpus Construction

This appendix provides implementation details for the retrieval-based in-context learning variant (ICL4D-R), including corpus construction, normalization, embedding, similarity computation, and the retrieval procedure.

A.1 Retrieval Corpus Construction

To support retrieval-based in-context decompilation, we construct a corpus of paired assembly-source functions (A_i, S_i) drawn from two widely adopted datasets: MBPP (Mostly Basic Programming Problems) and ExeBench.

MBPP is commonly used to evaluate code generation models and provides algorithmic-level programs, while ExeBench contains system-level C functions collected from real-world GitHub projects. Combining these datasets yields a diverse corpus covering both algorithmic and system-oriented programming patterns. In total, the corpus contains approximately 10^5 unique function pairs.

To ensure consistency across compilers and architectures, all assembly functions are normalized following the preprocessing used by the NOVA foundation model. Corresponding source code is also normalized to a canonical format. Each function is further annotated with a functional category (algorithm, string, I/O, system, or math), inferred from library headers and API usage. Exact duplicate pairs are removed via content hashing.

All retrieval corpus functions are strictly disjoint from the evaluation datasets (ExeBench test-real and HumanEval-Decompile). Disjointness is verified using SHA-256 hashing to ensure that no function appears in more than one split.

A.2 Assembly and Source Normalization

Assembly code is normalized to remove superficial lexical variation while preserving instruction semantics. Specifically, we apply the following preprocessing steps:

- Removal of comments and instruction addresses.
- Stripping of register prefixes (e.g., `%rax` $\rightarrow$ `rax`).
- Normalization of whitespace and punctuation.
- Conversion of hexadecimal constants to decimal form.
- Replacement of instruction addresses with symbolic placeholders (e.g., `[INST-1]`).

Source code is stripped of header inclusions and reformatted into a canonical style. These normalization steps reduce syntactic noise and ensure that retrieval emphasizes functional similarity rather than surface-level matching.

A.3 Assembly Embedding and Indexing

Each assembly function A_i is encoded into a dense vector representation $h(A_i) \in \mathbb{R}^d$ using the encoder component of NOVA (Jiang et al., 2023), a pretrained foundation model designed for assembly code understanding. NOVA employs functionality contrastive learning and optimization contrastive learning to encourage embeddings of functionally equivalent code and to organize representations across optimization levels.

The final representation for each function is obtained by taking the mean of all instruction token embeddings. All embeddings are precomputed and

indexed using FAISS, enabling efficient similarity search during retrieval.

A.4 Similarity Computation and Category-Aware Re-ranking

Given a target assembly function A_t with embedding $h(A_t)$, similarity to each corpus function is computed using Cross-domain Similarity Local Scaling (CSLS), which mitigates the hubness problem in high-dimensional embedding spaces.

To further promote semantic relevance, we apply a category-aware re-ranking strategy. If the functional category of a candidate exemplar does not match that of the target function, its similarity score is downweighted by a penalty factor α . This adjustment biases retrieval toward semantically related examples while allowing structurally similar cross-category exemplars to be selected.

The top- k exemplars with the highest adjusted similarity scores are selected to form the retrieval context.

A.5 Retrieval Procedure

Retrieval is performed deterministically for all experiments. Given a target assembly function, its embedding is computed, similarity scores are calculated using CSLS, category-aware penalties are applied, and the top- k assembly–source pairs are selected. The retrieved exemplars are ordered by decreasing similarity and inserted into the prompt as in-context demonstrations. Unless otherwise specified, we use $k = 5$.

B Optimization Flag Discovery

This appendix describes how compiler optimization flags used in the optimization-aware variant (ICL4D-O) are identified.

Modern compilers expose a large number of fine-grained optimization flags, many of which are interdependent and architecture-specific. To focus on optimizations that meaningfully affect emitted assembly in practice, we empirically identify which flags are active in our dataset.

Given a random subset of functions from the corpus, we extract a candidate list of optimization flags from GCC and Clang documentation. For each flag, we compile the same source code twice, once with the flag enabled and once with it disabled, while keeping all other compilation conditions fixed. The resulting assembly outputs are compared token-wise. A flag is considered active if enabling or dis-

abling it changes the instruction sequence, register allocation, or control-flow structure.

To reduce the search space, we apply a binary search strategy by grouping related flags and iteratively testing subsets. This process yields a ranked list of optimization flags based on how frequently they affect assembly generation across sampled functions.

Based on this analysis, we select the most frequently active flags for use in optimization-aware prompting, including `-fomit-frame-pointer`, `-ftree-ter`, `-fipa-pure-count`, `-fcrossjumping`, and `-ftree-coalesce-vars`.

C Prompt Design for In-Context Decompilation

This appendix describes the prompt design used in both variants of ICL4Decomp: retrieval-based in-context decompilation (ICL4D-R) and optimization-aware in-context decompilation (ICL4D-O).

C.1 Prompt Design for Retrieved-Exemplar In-Context Decompilation (ICL4D-R)

In ICL4D-R, the prompt is constructed by concatenating a small number of retrieved assembly-source exemplars followed by the target assembly function and an explicit decompilation instruction.

Each exemplar consists of an alternating pair of assembly code and its corresponding high-level source implementation. Retrieved exemplars are ordered by decreasing semantic similarity to the target assembly function. The prompt follows a consistent structured format:

Example Prompt

```
[Example 1]
Assembly: <A_1>
Source: <S_1>
[Example 2] ... <A_2, S_2>
[This is the assembly:] <A_t>
What is the source code?
```

This structured formatting exposes the language model to concrete instruction-to-structure correspondences before generation, allowing it to implicitly adapt to the compiler style and optimization level of the target function. Unless otherwise specified, we use $k = 5$ retrieved exemplars. All other generation settings remain fixed across experiments.

C.2 Prompt Design for Optimization-Aware In-Context Decompilation (ICL4D-O)

ICL4D-O augments the exemplar-based prompt with optimization-aware contextual guidance that encodes compiler transformation semantics in natural language.

Each compiler optimization flag is associated with a rule descriptor consisting of four components: (i) the optimization flag name, (ii) a natural language description of the transformation, (iii) an illustrative source-level example, and (iv) a decompilation hint explaining how to reason about the transformation during code reconstruction.

The prompt is extended with an *Optimization Context* section that precedes the target assembly code. This section informs the model that certain source-level constructs may be absent or transformed due to semantics-preserving compiler optimizations, and should be reconstructed accordingly during code reconstruction.

An example optimization-aware prompt follows:

Example Prompt for `-ftree-coalesce-vars`

Optimize options instructions

The binary you are decompiling may have been compiled with the GCC/Clang option `-ftree-coalesce-vars` (Tree SSA Variable Coalescing).

- This optimization merges multiple variables into a single register or memory location. It identifies variables that are copies of each other (e.g., `int b = a;`) or whose lifetimes do not overlap.
- The goal is to eliminate redundant `mov` instructions and reduce register pressure, resulting in smaller and faster code.
- As a result, intermediate variables from the source code may be completely absent in the final assembly; the logic is performed directly on the coalesced register.

Illustrative Source:

```
int coalesce_example(int x, int y) {
  int a = x + 5;
  int c = y - 2;
  int b = a;
  int d = b;
  int e = c;
  return d + e;
}
```

Decompilation Hint:

Trace data flow from inputs to the return expression; do not create redundant temporary variables for copies (b, d, e) since they have been optimized away.

```
[This is the assembly:] <A_t>
What is the source code?
```

When both retrieved exemplars and optimization rules are used, the optimization context is inserted before the target assembly, while the exemplar demonstrations remain unchanged. The remainder of the generation process is identical to that of ICL4D-R.

D Additional Robustness Results

Figure 5 reports the complementary robustness analysis stratified by lines of code. The same pattern holds: ICL4D-R remains above the baseline in every bin on both datasets.

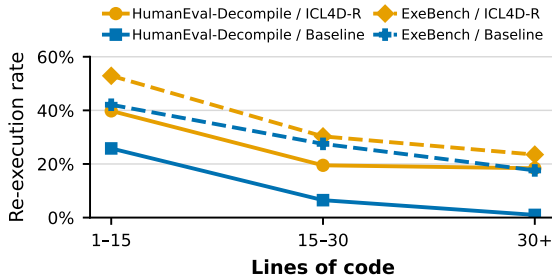


Figure 5: Re-execution success rate by lines of code. Solid lines denote HumanEval-Decompile and dashed lines denote ExeBench; ICL4D-R consistently outperforms the baseline.

E Assumptions and Scope

ICL4Decomp assumes function-level assembly as input. This follows the dominant evaluation protocol in neural decompilation, where a disassembler or benchmark pipeline first extracts candidate functions and linearized assembly before source recovery is evaluated. Under this protocol, function boundary recovery, code–data separation, and robust disassembly are upstream concerns rather than components of the source-generation model.

This separation is consistent with prior binary-analysis work that studies function recovery or decompiler backends as distinct stages, such as ByteWeight (Bao et al., 2014), rev.ng (Di Federico et al., 2017), and D-LIFT (?). ICL4Decomp is therefore best viewed as a semantic recovery layer that can be composed with stronger upstream function-boundary detectors or disassemblers.

The retrieval component uses assembly embeddings rather than source symbol names. As a result, exemplar selection does not depend on debug information or preserved identifiers, which is important for stripped binaries. However, the current evaluation does not cover adversarial obfuscation, indirect calls, inline assembly, complex exception handling, or repository-level recovery. Extending retrieval to callers, callees, global variables, and library/API signatures is a natural path toward these broader settings.

LLM Judge Prompt

LLM-as-a-Judge evaluation

Evaluate the quality of the recovered code against ground truth.

Score (0–100): 70% semantic correctness, 20% compilability, 10% readability.

Ground Truth: ``c <GT\_CODE> ``

Candidate: ``c <PRED\_CODE> ``

Return JSON only: { "score": <0-100>, "verdict": "<perfect|good|partial|poor|wrong>", "reason": "<short reason>" }

F LLM Judge Prompt for GitHub2025

GitHub2025 functions do not include executable test cases for every sampled function, so the supplementary real-world evaluation uses the following structured judge prompt. The score is reported only as a secondary metric and does not replace executable success rate on HumanEval-Decompile and ExeBench.