One Size Does Not Fit All: Revisiting Code Context Engineering for Repository-Level Code Generation

YICHEN LI, Chinese University of Hong Kong, China
QIYE LIN, Harbin Institute of Technology, China
YUN PENG, Chinese University of Hong Kong, China
ZHIHAN JIANG, Chinese University of Hong Kong, China
JINYANG LIU, Chinese University of Hong Kong, China
CHAOZHENG WANG, Chinese University of Hong Kong, China
YINTONG HUO, Singapore Management University, Singapore
CUIYUN GAO*, Harbin Institute of Technology, China

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation at the function or file level. However, they achieve limited performance on repository-level code generation due to the complicated repository context where a substantial amount of files and functions exist. To address this challenge, **code context engineering** methods are proposed to accurately extract the relevant code context required by such tasks. These methods belong to three dominant paradigms: 1) *Similarity-based In-Context Learning (ICL)*, which retrieves similar code examples in the repository as demonstrations; 2) *Static Analysis*, which captures relevant code context based on structural dependencies in the repository; and 3) *Navigation-based Paradigms*, which invokes LLMs to dynamically explore repositories for context identification.

Despite the prevalence of code context engineering approaches, their individual contributions are often coupled within complex agents in repository-level code generation, making it difficult to isolate and evaluate the effectiveness of each paradigm. This paper presents the first large-scale and systematic empirical study that compares three code context engineering paradigms independently. We evaluate ten representative code context engineering methods based on the three paradigms, with eight popular LLMs on this task. We also propose a new metric named Dependency Collection Rate (DCR) and efficiency metrics to enable the direct comparison of code context engineering methods, rather than observing their impacts only on the final code generation performance.

Our findings reveal fundamental trade-offs: static analysis provides the most reliable balance between effectiveness and cost for function-level generation, while navigation-based approaches become increasingly advantageous as task complexity grows. However, navigation requires powerful models and incurs 10-20× higher computational costs. Based on these findings, we provide actionable implications for AI coding researchers and software developers to guide the design and deployment of context-aware coding tools.

CCS Concepts: • **Software and its engineering → Software development techniques**;

Additional Key Words and Phrases: Code Generation, Large Language Models, Context Engineering

*Cuiyun Gao is the corresponding author.

Authors' Contact Information: Yichen Li, Chinese University of Hong Kong, Hong Kong, China, ycli21@cse.cuhk.edu.hk; Qiye Lin, Harbin Institute of Technology, Shenzhen, China, 25b951077@stu.hit.edu.cn; Yun Peng, Chinese University of Hong Kong, Hong Kong, China, ypeng@cse.cuhk.edu.hk; Zhihan Jiang, Chinese University of Hong Kong, Hong Kong, China, zhjiang22@cse.cuhk.edu.hk; Jinyang Liu, Chinese University of Hong Kong, Hong Kong, China, jyliu@cse.cuhk.edu.hk; Chaozheng Wang, Chinese University of Hong Kong, Hong Kong, China, adf111178@gmail.com; Yintong Huo, Singapore Management University, Singapore, Singapore, ythuo@smu.edu.sg; Cuiyun Gao, Harbin Institute of Technology, Shenzhen, China, gaocuiyun@hit.edu.cn.

ACM 2994-970X/2026/7-ARTFSE131
https://doi.org/10.1145/3808138

ACM Reference Format:
Yichen Li, Qiye Lin, Yun Peng, Zhihan Jiang, Jinyang Liu, Chaozheng Wang, Yintong Huo, and Cuiyun Gao. 2026. One Size Does Not Fit All: Revisiting Code Context Engineering for Repository-Level Code Generation. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE131 (July 2026), 24 pages. https://doi.org/10.1145/3808138

1 Introduction

Intelligent software development powered by Large Language Models (LLMs) has moved beyond concept into widespread and practical application, in which LLM-based coding tools like GitHub Copilot redefine the workflows of millions of developers [1, 16] by automatically writing code for developers. These code generation techniques significantly improve programming productivity in modern software development [31, 34, 41, 56, 61]. By aligning with the training objectives [19, 29, 35], current LLMs show remarkable performance in function-level [5] and file-level [57] code generation, where they only consider small requirement descriptions with short context before generating code solutions. However, the case is fundamentally different for repository-level code generation, a more practical and common scenario in modern software development, where the available context becomes the entire code base with many functions and files [31]. It is quite challenging to model the useful context information from the entire code base to facilitate the generation of correct code solutions at this scale.

Context engineering [38] approaches are proposed to address the challenge of modeling long and complicated context. They systematically identify, extract, and present the necessary background information required by LLMs on domain-specific tasks. In the domain of software development, we further refine the term into **code context engineering**, where the knowledge is the relevant code context in the repository [32, 34, 48, 61], such as the implementations of related APIs and invocation relationships.

With the development of repository-level code generation, the code context engineering approaches could be classified into three major paradigms. (i) **Similarity-based In-Context Learning (ICL)** was initially proposed to search similar code examples in the repository [15], inspired by the commonly used in-context learning methodologies to let LLMs learn from similar examples. (ii) **Static Analysis** is then integrated to accurately capture relevant code context based on structural dependencies in the repository [31, 32]. (iii) **Navigation-based Paradigms** have recently been proposed to directly invoke LLM to dynamically explore repository filesystems for context identification [34, 61], eliminating the need to define examples or static analysis rules manually.

Despite the widespread usage of code context engineering approaches, their individual contributions have never been systematically evaluated [24, 32, 34, 65]. Previous work focuses only on the study of a single paradigm, such as different similarity-based ICL methods [15, 52] or providing different granularity of contexts [20, 28]. There is no comprehensive comparison of multiple code context engineering paradigms in different aspects such as effectiveness, efficiency, and quality. Furthermore, the context acquired by different code context engineering approaches is usually indirectly evaluated in previous work by observing the impacts on the final code generation performance.

This paper presents the first large-scale systematic empirical study to mitigate the gap. We conduct a thorough investigation into the three code context engineering paradigms with a focus on the following research questions:

- **RQ1: Effectiveness.** How much do the three code context engineering paradigms improve the code generation effectiveness?
- **RQ2: Efficiency & Cost.** What are the differences of these paradigms in terms of time efficiency and token cost?
- **RQ3: Context Quality.** What are the differences in the context generated by each paradigm?

To answer the research questions, we implement ten representative code context engineering methods based on the three paradigms, including three adapted from prior work, along with eight predominant LLMs on 439 repository-level code generation tasks from execution-based benchmarks. Notably, we introduce and implement Nav_{Plus}, a hybrid navigation agent that integrates structural analysis capabilities directly into the tools, enabling systematic code exploration beyond basic filesystem operations. Besides, to enable direct analysis of context quality, we propose a novel metric, the **Dependency Collection Rate (DCR)**, which measures the proportion of ground-truth cross-file dependencies successfully retrieved by each method.

Based on the empirical results, we summarize the counter-intuitive findings:

- The similarity-based ICL paradigm shows minimal improvements, while the static analysis paradigm delivers substantial improvements, particularly on function-level tasks. Its relative advantage narrows on class-level generation where navigation becomes more effective. Both passive paradigms are time-efficient and consume a limited number of tokens.
- The navigation-based context acquisition paradigm shows great potential to find the accurate context but requires powerful LLM reasoning capabilities and incurs significantly higher computational costs.
- Context quality extends beyond dependency coverage alone, as navigation-based approaches with powerful models will outperform higher-DCR static analysis methods through implicit knowledge acquired during repository exploration, revealing that code context serves multiple roles beyond simply providing structural dependencies.

Based on the findings, we draw several implications for both researchers developing context-aware coding tools and software developers who use these tools. Our results reveal that static analysis offers the best balance between performance gain and computational cost, navigation approaches achieve the highest performance with powerful models, while similarity-based retrieval proves limited for repository-level tasks. We identify critical research directions including agent-specific model alignment and Language Server Protocol (LSP)-integrated tooling, while offering practical guidance for selecting appropriate paradigms based on specific development scenarios and computational constraints for users (developers).

We summarize our major contributions as follows:

- **Novelty:** To the best of our knowledge, we present the first large-scale systematic empirical study for code context engineering.
- **Framework & Benchmark:** We design and implement a comprehensive experimental framework to enable a fair comparison of 10 code context engineering methods, including three adapted from prior work, by 8,758 lines of Python code. With this framework, our study evaluates 8 predominant LLMs on 439 tasks, benchmarking the effectiveness, efficiency, and context quality.
- **Metric:** We propose a new metric, namely Dependency Collection Rate (DCR), to quantitatively measure the ability of a code context engineering approach to identify and retrieve the dependencies required.
- **Implications:** We derive several actionable implications and concrete guidance from our results for both AI Coding researchers and software developers, informing the future design and application of context-aware coding tools.

2 Background: Code Context Engineering for Repo-Level Code Generation

Repository-level code generation represents a fundamental shift from isolated function generation to contextualized programming within entire software project. Unlike function-level tasks where models operate with self-contained requirements, repository-level generation demands

understanding intricate inter-file dependencies, architectural patterns, and API usage conventions across potentially thousands of files. The generation effectiveness on repository-scale tasks is highly dependent on the quality of cross-file context. Following prior literature [43, 54, 65, 67], we use the term *repository-level code generation* to refer to tasks where the generated code depends on cross-file context (*e.g.*, imported APIs or class hierarchies). This is distinct from whole repository synthesis, focusing specifically on the accurate retrieval and utilization of project-wide code contexts.

To systematically understand the code context engineering paradigms, we conducted a literature review covering recent methods related to LLM-based code generation. While existing methods often integrate multiple agentic components (*e.g.*, prompting strategies, reflection mechanisms), our review only focuses on the code context engineering dimension. This focused lens reveals three fundamental paradigms that represent fundamentally different strategies of how LLMs interact with code repositories.

Table 1. Code context engineering approaches in LLM-based code generation methods.

Paradigm	Methodology	Year/Avenue
Similarity-based In-Context Learning	ReACC [36] uses a dual-encoder for semantic retrieval.	2022/ACL
	RepoCoder [66] uses retrieval-generation with refinement.	2023/EMNLP
	Demo Construction [15] benchmarks similarity-based ICL methods.	2023/ASE
	LoGenText-Plus [14] retrieves similar log statement.	2023/TOSEM
	IRCoCo [26] uses RL for better utilizing retrieved context.	2024/FSE
	UniLog [59] retrieves similar log statement.	2024/ICSE
	ProCC [50] adaptively selects from multiple retrieval perspectives.	2024/TOSEM
	RLCoder [55] trains a retriever using code perplexity.	2025/ICSE
	Wang et al. [52] benchmarks similarity-based ICL methods with tuning methods.	2025/FSE
	...	
Static Analysis	TypeGen [45] extracts inter-file type dependencies.	2023/ASE
	CoCoMic [13] models cross-file context using a static analysis tool.	2024/LREC-COLING
	DRACO [8] builds a repo-specific context graph via dataflow analysis.	2024/ACL
	DroidCoder [64] enriches context from static analysis of Android app.	2024/ASE
	SCLogger [30] extracts inter-method context using static analysis of call graphs.	2024/FSE
	Blinn et al. [4] integrates LLM with an LSPs to provide precise static context.	2024/OOPSLA
	IDECoder [31] leverages native IDE-derived static context.	2024/LLM4Code
	A3-CodGen [32] identifies three types of dependencies as context.	2024/TSE
	AlphaTrans [23] uses static dependency analysis to guide code translation.	2025/FSE
	...	
Navigation-based	CodeAgent [67] developed programming tools for codebase navigation	2024/ACL
	SWE-Agent [62] Autonomously resolves issues using a basic filesystem.	2024/NeurIPS
	RepoAgent [37] navigates the repository for easy reference and comprehension.	2024/EMNLP
	AutoFL [25] prompts an LLM to use function calls to navigate a repository	2024/FSE
	LocAgent [7] navigates a pre-built structural code graph.	2025/ACL
	RepoGraph [43] manages a repository-level structure for LLMs	2025/ICLR
	CodexGraph [33] integrates LLM agents with code repository graph database	2025/NAACL
	...	

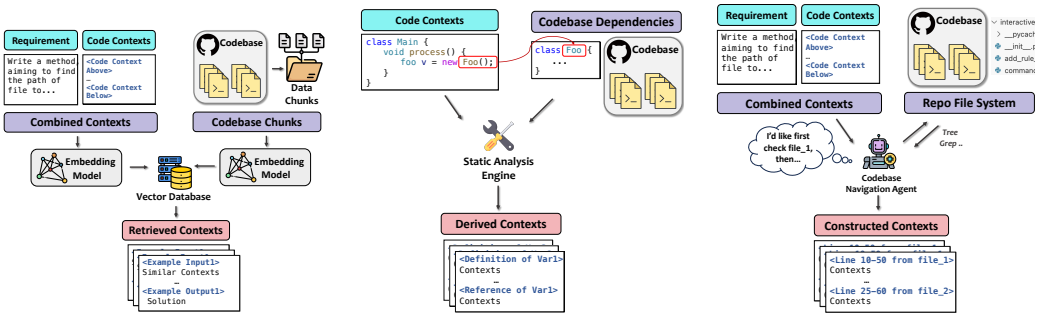
As shown in Table 1, we identify three distinct paradigms that represent fundamentally different approaches to code context engineering within these comprehensive systems. While each listed method may incorporate other innovations, our analysis extracts and categorizes the context acquisition strategies to enable systematic comparison.

Our analysis reveals the clear difference in context engineering approaches. The first two paradigms are **passive paradigms** without active repository exploration for the context. In contrast, the third **active paradigm** dynamically discovers repository context through iterative exploration guided by the model's reasoning process.

2.1 Similarity-Based In-Context Learning (ICL)

The earliest and most intuitive approach treats repository-level generation as the source of few-shot learning. As illustrated in Figure 1(a), this paradigm is founded on the principle of **learning**

from example. It treats the existing codebase as an indexed knowledge base of code snippets and uses a retriever to find solutions to problems that are similar to the current task [15, 52]. These retrieved snippets, typically identified based on lexical (*e.g.*, BM25) or semantic similarity, are then provided to the LLM as few-shot demonstrations to guide the generation process. The effectiveness of this paradigm is dependent on the definition of "similarity".



(a) Similarity-based ICL paradigm (b) Static Analysis paradigm. (c) Navigation-based paradigm.

Fig. 1. An illustration of the three core paradigms for repo-level context engineering evaluated in this study.

2.2 Static Analysis

In contrast, the Static Analysis paradigm, shown in Figure 1(b), operates on the principle of **learning from dependency**. It aims to provide the LLM with a relatively precise, programmatically-derived map of the codebase's architecture. This approach involves parsing source code to build structural representations like Abstract Syntax Trees (ASTs) and call graphs, from which it extracts explicit dependencies such as imported modules, inherited classes, and definition-use chains [30–32]. By explicitly including function definitions, the model can better understand the expected behavior and usage patterns of the called functions. While the target function body is not yet generated in the repository-level code generation task, this paradigm relies on analyzing the surrounding context, such as file-level import statements, class inheritance hierarchies and the function signature, to infer potential dependencies. By explicitly including function definitions resolved from these visible symbols, the model can better understand the expected behavior and usage patterns of the called functions. While this method can provide highly accurate context, its capability to analyze dependency is inherently static and cannot capture runtime information (*e.g.*, indirect invocation) or dynamic behaviors. This mirrors the real-world setting, where runtime invocations are unknowable when coding.

2.3 Navigation-Based Paradigm

As shown in Figure 1(c), the third paradigm represents a fundamental shift from passive context provision to active discovery, operating on the principle of **learning by exploration**. It reframes the LLM as an autonomous agent that mimics the workflow of a human developer to acquire context [34, 48, 61]. This is achieved by empowering the LLM with a toolkit of pre-defined commands, such as for file system exploration (`ls`) or content searching (`grep`).

The agent's operational cycle is an iterative feedback loop based on the modern concept of tool calling. At each step, the LLM's task is to select a tool from its toolkit and formulate the arguments for that tool. An external system then safely executes the invoked tool and returns the output to the LLM. The agent uses this new knowledge to update its understanding and decide on its next action. When the agent determines it has gathered sufficient knowledge, it concludes the exploration phase.

It then synthesizes the constructed contexts with raw code snippet and its reasoning for why they are useful, as illustrated in Figure 1(c).

This dynamic and autonomous approach fundamentally differs from the passive paradigms. Its effectiveness is highly contingent on the backbone LLM's reasoning and planning capabilities. Furthermore, the iterative, multi-turn nature of the process and the empowerment of the LLM to invoke tools introduce a distinct profile of trade-offs, which must be empirically evaluated.

3 Study Design

To systematically evaluate the trade-offs inherent in the three paradigms of code context engineering, our experimental design is structured around three fundamental dimensions: the **effectiveness** of the generated code, the practical **efficiency** of the process, and the intrinsic **quality** of the context itself.

Our evaluation is guided by the following research questions (RQs).

- **RQ1**: How much do the three code context engineering paradigms improve the code generation effectiveness?
- **RQ2**: What are the differences of these paradigms in terms of time efficiency and token cost?
- **RQ3**: What are the differences in the context generated by each paradigm?

To answer these questions, we conducted a comprehensive empirical study. For **RQ1**, we evaluate the code generation effectiveness of each paradigm by comparing multiple implementation variants, such as different retrieval algorithms for *similarity-based ICL* and different agentic strategies for *navigation*. For **RQ2**, we measure the associated trade-offs by quantifying the one-time offline preparation time (*i.e.*, indexing), the online preparation time (*i.e.*, retrieval), and the LLM workloads (*e.g.*, number of calls and tokens). Finally, for **RQ3**, we perform a qualitative manual analysis on the generated contexts to understand the root causes of the performance differences, focusing on aspects like relevance, noise.

3.1 Datasets

Table 2. Details of the studied repo-level datasets and comparison with other notable benchmarks.

Dataset	Task Type	# of Problems	# of Repositories	Language(s)	Executable
CoderEval [63]	Function Generation	460	53	Python, Java	Yes
DevEval [28]	Function Generation	1,874	115	Python	Yes
RepoClassBench [11]	Class-level Generation	97†	10	Python, Java, C#	Yes
CrossCodeEval [12]	Code Completion	10,000	~1k	Python, Java, TS, C#	No
RepoEval [66]	Code Completion	373	6	Python	Yes
CoCoMIC [13]	Code Completion	6,888	60,891	Python	No

† RepoClassBench contains 97 Python class-level tasks. We successfully restored execution environments for 94 of them.

We establish three core principles for dataset selection:

(1) **Repository-Level Context:** The benchmark must be built upon real-world code repositories to ensure that the tasks require repo-level context.
(2) **Sufficient Task Complexity:** The benchmark must focus on function-level code generation rather than simpler tasks like single-line completion, ensuring that the choice of context engineering strategy is meaningful.
(3) **Execution-Based Evaluation:** The benchmark must provide a reliable mechanism for functional correctness evaluation.

Based on our selection principles, we identified **CoderEval** [63] and **DevEval** [28] as the most suitable benchmarks for our study. Due to the cost of evaluating multiple context engineering methods across numerous LLMs, we sample one task per repository from DevEval. To further validate our

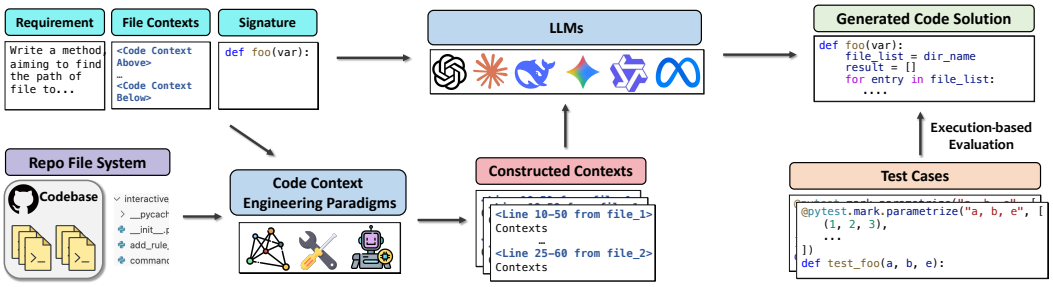


Fig. 2. Task formulation for repository-level code generation.

findings beyond function-level generation, we additionally include **RepoClassBench** [11], a class-level code generation benchmark where each task requires generating an entire class with cross-file dependencies. We use its Python split under the *detailed* description setting and evaluate with Pass@1 against the provided test cases. In total, our evaluation encompasses 439 repository-level tasks: 230 from CoderEval, 115 sampled tasks from DevEval, and 94 from RepoClassBench.

As illustrated in Figure 2, our task formulation follows the file infilling setting [28] for method-level code generation where models are provided with method signatures, requirement, and local file contexts. Each context engineering paradigm then constructs additional repository-level contexts that are combined with the base input and sent to the LLM for code generation. The generated solutions are evaluated against test cases for functional correctness.

To ensure experimental validity and prevent data leakage, we preprocessed all repositories by removing test-related directories and any files containing test cases. This preprocessing ensures that no context engineering paradigm could inadvertently access test cases or solution hints.

3.2 Studied Methods

To conduct our comparative study, we designed and implemented representative versions of each of the three paradigms. This subsection details the specific design and implementation choices for each method used in our experiments.

Similarity-based ICL. Following previous work [15, 27, 52], we implement three distinct and representative retrieval strategies, each designed to retrieve the top-5 examples with the highest similarity scores [15, 18, 52]. Specifically, these strategies are:

- ICL_{BM25} uses the classical BM25 [46] algorithm for lexical retrieval.
- $\text{ICL}_{\text{UniXcoder}}$ uses the UniXcoder [19] model for semantic retrieval.
- $\text{ICL}_{\text{CoCoSoDa}}$ employs the CoCoSoDa [47] model for contrastive learning-based retrieval.

We additionally adapt RepoCoder [65], an iterative retrieval approach. The LLM first generates draft code, then replaces the signature as query to retrieve the code snippets.

Static Analysis. As mentioned in Section 2.2, we define a dependency as any cross-file program entity (*e.g.*, function definitions, class instantiations, or global variables) necessary for the target task. Following previous works [31, 32, 52], we implemented two variants to explore the impact of dependency context granularity: $\text{SA}_{\text{Method}}$ and SA_{File}, which parse the target file's imports and symbol table to identify potential cross-file references and resolves them to their definitions.

- $\text{SA}_{\text{Method}}$ provides a dense context by extracting only the source code of dependent methods.
- SA_{File} provides a broader context by extracting the full source files of dependencies.

We additionally adapt A3-CodGen [32], which extends cross-file dependency resolution with third-party library awareness. It extracts function signatures from installed third-party packages via runtime introspection.

Table 3. Overview of the Toolkit for the Nav_{Raw}.

Category	Tool	Description
Repository Exploration	LIST_DIR	Lists the files and directories within a specified path.
	READ_DIR_TREE	Provides a structured, recursive tree view of the directory hierarchy.
	GLOB	Searches for files across the repository using wildcard patterns (*e.g.*, SRC/**/*.PY).
Content Access	READ_FILE	Reads the content of a file, with optional support for specific line ranges.
	GREP_SEARCH	Performs a content-based search within a file using a regular expression.

Navigation-based Paradigms. This approach represents the paradigm of learning by exploration [7, 34, 62], where the LLM actively discovers context. A key design choice in our implementation is that the same LLM serves dual roles: it both collects the contexts and generates the code solution. We designed and implemented two distinct agents, allowing batch invocation of independent tool calls within a single LLM request:

- Nav_{Raw} serves as our basic navigation context collection agent, equipped with a basic toolkit (detailed in Table 3) to mimic a developer operating on a raw repository, demonstrating the pure repository exploratory and context collection and summarization paradigm.
- Nav_{Plus} is an enhanced agent equipped with advanced tools. Instead of relying solely on text-based commands, its core principle is to integrate the capabilities of static analysis and semantic search directly into its action space. This is achieved by providing the agent with an augmented toolkit (detailed in Table 4) that operates on a pre-built structural [7] and semantic [65] index of the repository. Consequently, the agent can perform advanced actions, such as acquiring the function definition (*i.e.*, FIND_DEFINITION) or finding semantics-related code (*i.e.*, SEMANTIC_SEARCH), rather than reading raw files.

We additionally adapt the SWE-Agent [62] for agentic method, retaining its bash-style navigation tools (`open`, `scroll`, `search_file/dir`, `find_file`) with windowed file viewing, but removing editing commands.

Table 4. Overview of the Toolkit for the Nav_{Plus}.

Category	Tool	Description
Repository Exploration	LIST_DIR	Lists the files and directories within a specified path.
	READ_DIR_TREE	Provides a structured, recursive tree view of the directory hierarchy.
	GLOB	Searches for files across the repository using wildcard patterns.
Structural Navigation	FIND_DEFINITION	Locates an identifier's definition using the structural index.
	FIND_REFERENCES	Finds all usages of a identifier using the structural index.
	READ_CLASS_STRUCTURE	Extracts a high-level view of a class's methods and fields.
Semantic & Content Access	READ_FILE	Reads raw file content, supporting line ranges (foundational).
	READ_FUNCTION	Extracts only the source code of a specific function.
	READ_CLASS	Extracts the full source code of a specific class.
	SEMANTIC_SEARCH	Searches for semantically similar code snippets using a vector index.

3.3 Selected LLMs

We evaluate our context engineering paradigms across 8 popular and powerful LLMs. Our selection includes both open-source and commercial models. Table 5 summarizes the models, their sizes, and context window capacities.

We include four open-source models: Llama-3.3-Instruct (70B) [39], Qwen3-Instruct (235B-A22B) [60], Qwen3-Coder-Instruct (480B-A35B) [22], and GPT-OSS (120B) [42]. These models

Table 5. The LLMs we evaluate in this paper. Models highlighted in gray are closed-source models.

Model	Size	Context Size	Model	Size	Context Size
Llama-3.3-Instruct	70B	128K	Qwen3-Instruct	235B-A22B	128K
Qwen3-Coder-Instruct	480B-A35B	256K	GPT-OSS	120B	128K
DeepSeek R1	671B-A37B	128K	GPT-4o	-	128K
Claude 4 Sonnet	-	200K	Gemini 2.5 Pro	-	1M+

provide transparency in our evaluation and enable reproducibility. For commercial models, we evaluate DeepSeek R1 (671B-A37B) [10], GPT-4o [42], Claude 4 Sonnet [2], and Gemini 2.5 Pro [17], which represent the current frontier of commercial LLM capabilities. We invoke all those models through their official APIs or openly deployed platforms. We set the temperature at 0 to ensure reproducibility and follow the default model thinking budget. To verify the reliability of our single-run results, we conducted a stability analysis by repeating the generation 3 times on a random subset of 50 tasks. We observed zero variance in the generated code, confirming that our Temperature=0 setting ensures deterministic and reproducible outcomes.

Throughout our results, we use the following abbreviations: **L-3.3** (Llama-3.3-70B-Instruct), **Q-3** (Qwen3-235B-A22B-Instruct), **QC-3** (Qwen3-Coder-480B-A35B-Instruct), **G-O** (GPT-OSS-120B), **DS-R1** (DeepSeek R1), **GPT-4o**, **C-S-4** (Claude 4 Sonnet), and **G-P-2.5** (Gemini 2.5 Pro).

3.4 Evaluation Metrics

To comprehensively answer our research questions, we employ a suite of metrics.

Effectiveness Metrics (RQ1).

- **Pass@1:** The metric for functional correctness of generated code. It measures the percentage of problems for which the first generated solution passes all defined test cases.

Efficiency Metrics (RQ2). We decompose efficiency into three aspects: one-time offline preparation, per-task online latency, and LLM workload. We **deliberately exclude** LLM inference time itself, as it is a confounding variable dependent on external factors (*e.g.*, API provider server load, network latency) and out of scope for evaluating context engineering strategies.

- **Offline Preparation Time ($T_{\text{offline\_prep}}$):** The one-time, per-repository computational cost to prepare the system before requesting LLM.
- **Online Preparation Time ($T_{\text{online\_prep}}$):** The wall-clock time spent preparing context for a single task *before* LLM invocation.
- **LLM Invocations (N_{llm}):** The number of LLM calls required to complete a single task. This directly impacts both latency and cost.
- **Token Cost:** The total input and output tokens processed by the LLM across all invocations per task.

We measure time performance over 12 runs and average results after excluding the highest and lowest values [44].

Context Quality (RQ3). To quantitatively evaluate the quality of the cross-file context generated by each paradigm, we **propose a new metric** that evaluates its completeness. Our proposed DCR metric is conceptually distinct from the Dependency Invocation Rate (DIR) [20]. While DIR evaluates whether the LLM *uses* the provided context, our DCR is designed to evaluate the code context engineering process itself. We define ground-truth dependencies as the set of cross-file functions that are actually invoked during the execution of the ground-truth solution, captured through code instrumentation.

- **Dependency Collection Rate (DCR):** This metric measures the proportion of ground-truth code dependencies that are collected correctly during code context engineering. A higher DCR indicates that the context engineering strategy is more successful in identifying and including the necessary code dependencies for the task.

3.5 Implementation

All experiments were conducted on a workstation, simulating a powerful developer's environment (DevBox) in real-world development scenario. The DevBox is a single Linux server running Ubuntu 20.04.4 LTS, equipped with a multi-core Intel Xeon W-series CPU (32 cores @ 3.00GHz), 64 GB of RAM and a single NVIDIA A100 (80GB) GPU. This setup is representative of a powerful, modern workstation used for both CPU-intensive tasks (*e.g.*, lexical retrieval, static parsing, and local builds) and GPU-accelerated tasks (*e.g.*, generating embeddings). Crucially, to mirror real-world development scenarios [6, 40, 53] where developers interact with powerful models, all LLM invocations in our study were made via external APIs.

4 Experiment Results

4.1 RQ1: Effectiveness

Table 6. *Pass@1* results for different context acquisition methods on the DevEval, CoderEval, and RepoClass-Bench datasets (RQ1). The *Avg-Improve* column shows the averaged relative improvement over the *Base* performance. The greener the cell, the better the performance on that dataset.

Dataset	Method	L-3.3	Q-3	QC-3	G-O	DS-R1	GPT-4o	C-S-4	G-P-2.5	Avg-Improve
DevEval (Infilling)	Base	54.78	61.74	66.09	58.26	65.22	59.13	66.09	67.83	—
	ICL_{BM25}	56.52	63.48	70.43	62.61	65.22	62.61	68.7	67.83	3.73%↑
	$ICL_{UniXcoder}$	57.39	64.35	69.57	60.0	64.35	61.74	66.96	71.3	3.34%↑
	$ICL_{CoCoSoDa}$	55.65	66.09	66.96	63.48	64.35	60.87	71.3	68.7	3.71%↑
	RepoCoder	57.39	65.22	70.43	63.48	64.35	64.35	69.57	68.7	4.87%↑
	SA_{Method}	60.0	66.96	69.57	67.83	70.43	60.0	70.43	69.57	7.28%↑
	SA_{File}	62.61	69.57	70.43	69.57	71.3	61.74	73.04	71.3	10.29%↑
	A3-CodGen	61.74	67.83	70.43	68.7	70.43	60.87	70.43	69.57	8.19%↑
	Nav_{Raw}	55.65	65.22	67.83	65.22	66.96	64.35	73.04	73.04	6.98%↑
	Nav_{Plus}	56.52	68.7	71.3	67.83	68.7	66.96	73.91	73.91	9.77%↑
	SWE-Agent	54.78	67.83	68.7	66.96	67.83	65.22	73.91	73.04	7.82%↑
CoderEval (Infilling)	Base	41.74	47.83	46.96	48.26	44.35	42.61	49.56	52.17	-
	ICL_{BM25}	40.43	48.7	49.13	51.74	49.57	46.96	52.61	56.09	5.77%↑
	$ICL_{UniXcoder}$	43.04	48.26	52.61	53.04	50.87	45.65	53.91	57.83	8.43%↑
	$ICL_{CoCoSoDa}$	42.17	48.26	52.61	53.48	50.87	46.96	51.74	56.52	7.80%↑
	RepoCoder	40.87	50.0	50.87	52.17	50.87	47.39	53.91	56.96	7.85%↑
	SA_{Method}	52.17	54.35	56.52	54.78	53.91	49.13	56.96	60.43	17.51%↑
	SA_{File}	53.91	55.22	56.09	55.22	54.35	52.17	58.26	58.7	19.19%↑
	A3-CodGen	53.48	54.78	57.39	55.22	53.91	50.0	57.83	60.87	18.95%↑
	Nav_{Raw}	35.22	52.61	53.91	50.87	50.0	57.83	63.04	60.87	14.47%↑
	Nav_{Plus}	47.39	54.35	53.91	55.22	52.6	56.09	64.78	62.61	17.17%↑
	SWE-Agent	38.26	53.48	53.91	53.04	51.3	56.96	63.91	61.74	15.61%↑
RepoClassBench (Class-level)	Base	24.47	29.79	30.85	26.60	31.91	27.66	30.85	32.98	—
	ICL_{BM25}	23.40	30.85	32.98	27.66	35.11	29.79	35.11	38.30	7.22%↑
	$ICL_{UniXcoder}$	23.40	29.79	32.98	26.60	31.91	27.66	30.85	38.30	2.34%↑
	$ICL_{CoCoSoDa}$	23.40	29.79	32.98	26.60	30.85	28.72	38.30	36.17	4.61%↑
	RepoCoder	-	-	-	-	-	-	-	-	-
	SA_{Method}	-	-	-	-	-	-	-	-	-
	SA_{File}	26.60	32.98	35.11	29.79	35.11	30.85	37.23	37.23	12.54%↑
	A3-CodGen	-	-	-	-	-	-	-	-	-
	Nav_{Raw}	19.15	35.11	31.91	28.72	37.23	34.04	43.62	42.55	14.72%↑
	Nav_{Plus}	21.28	39.36	37.23	31.91	40.43	39.36	45.74	43.62	26.16%↑
	SWE-Agent	20.21	31.91	35.11	29.79	36.17	37.23	44.68	42.55	17.17%↑

‡ RepoCoder, SA_{Method}, and A3-CodGen are not applicable to class-level tasks (see text).

Table 6 presents the Pass@1 results across all three benchmarks. Without repository-specific context, LLMs achieve an average Pass@1 of 62.27% on DevEval, 46.69% on CoderEval, and 29.39% on RepoClassBench, confirming that repository-level tasks remain challenging even for capable models. We first analyze the function-level benchmarks (DevEval and CoderEval), then examine how these patterns extend to class-level generation (RepoClassBench).

Function-level generation. Our evaluation begins with the widely adopted similarity-based ICL paradigm. While generally beneficial, its effectiveness is markedly limited, aligned with previous work [18]. In some cases, ICL even degrades performance: Llama-3.3 drops from 41.74% to 40.43% with BM25 on CoderEval. The improvements are modest across all ICL variants, including RepoCoder [65], with average improvements ranging from 3.34% to 4.87% on DevEval and 5.77% to 8.43% on CoderEval. Since the model is already provided with substantial local context under the infilling setting, the retrieved snippets appear to serve as task demonstrations (*i.e.*, few-shot prompting [15]) rather than as sources of critical missing information, a role that diminishes with more capable LLMs. Notably, even RepoCoder's iterative query refinement (4.87% on DevEval) remains well below SA_{Method} (7.28%), suggesting that the paradigm's bottleneck lies in retrieval granularity rather than query quality.

Finding 1: For the code generation task, similarity-based retrieved contexts provide minimal performance impact, even with iterative query refinement. The retrieved code snippets appear to act as task demonstrations rather than providing critical contexts. The utility of this demonstrative role diminishes with more capable LLMs.

In contrast, the Static Analysis (SA) paradigm proves remarkably effective and reliable. This approach provides a programmatically-derived dependency map of the codebase's architecture. SA_{File}, which gathers file-level dependent contexts from structural dependencies, led to significant gains across all models. For instance, SA_{File} increased the *Pass@1* score of *Qwen-3* on DevEval from 61.74% to 69.57% (12.68% relative improvement), and *Claude Sonnet 4* from 66.09% to 73.04% (10.52% relative improvement). On average, the SA paradigm improved performance by 7.28% to 10.29% on DevEval and 17.51% to 19.19% on CoderEval, with SA_{File} consistently achieving the highest gains. A3-CodGen [32], which extends SA_{Method} with third-party library signatures, falls between SA_{Method} and SA_{File} (8.19% on DevEval), indicating that file-level context richness is a more dominant factor than coverage completeness.

Finding 2: Providing dependency contexts derived from static analysis, which captures the structural properties of the codebase, proves to be a highly effective paradigm for function-level code generation, though its relative advantage narrows on class-level tasks where the target code dominates the file context.

Shifting from passive context provision to active context discovery, the navigation-based paradigm achieves the highest individual scores in our evaluation: Nav_{Plus} reaches 73.91% on both Claude Sonnet 4 and Gemini 2.5 Pro for DevEval, matched by SWE-Agent [62] on Claude Sonnet 4. On average, Nav_{Plus} achieves improvements of 9.77% on DevEval and 17.17% on CoderEval. However, the paradigm's effectiveness is highly dependent on the backbone LLM's reasoning capabilities. Weaker models struggle significantly: Llama-3.3 drops from 41.74% to 35.22% with Nav_{Raw} on CoderEval. This pattern persists across all three navigation variants with distinct tool interfaces, confirming that the model sensitivity is an intrinsic paradigm characteristic.

Class-level generation. Three methods are not applicable to class-level tasks: SA_{Method} and A3-CodGen rely on sibling code in the target file to resolve fine-grained dependencies, which is unavailable when the entire class body is removed; RepoCoder's iterative refinement depends

on function-level draft generation that does not transfer to class-level. The remaining methods reveal an important shift in the paradigm hierarchy. The ICL paradigm remains modestly effective, consistent with function-level findings. However, the relative advantage of Static Analysis narrows considerably: on function-level benchmarks, SA_{File}'s average improvement is 2.8× that of BM25; on class-level tasks, this gap narrows to 1.7×. We attribute this narrowing to the target-to-file ratio: SA_{File} traces cross-file dependencies by analyzing sibling definitions in the target file. When the target is a single function within a large file, abundant sibling code provides rich dependency signals. When the target is an entire class that dominates its file, removing the class body leaves few sibling definitions for SA to analyze, causing it to partially degenerate toward the baseline.

This structural limitation creates an opportunity for navigation. Because agents discover context through active exploration rather than file-level parsing, they are unaffected by the target-to-file ratio. The lower baseline on class-level tasks (29.39% vs. 62.39% on DevEval) also provides more headroom for context-driven improvement. On class-level tasks, Nav_{Raw} surpasses SA_{File}, reversing the function-level ordering, and Nav_{Plus} achieves the largest improvement across all paradigms. SWE-Agent falls between the two navigation variants, maintaining the consistent within-paradigm ordering. The model-dependency pattern also persists: Llama-3.3 drops below baseline for all navigation variants on class-level tasks as well.

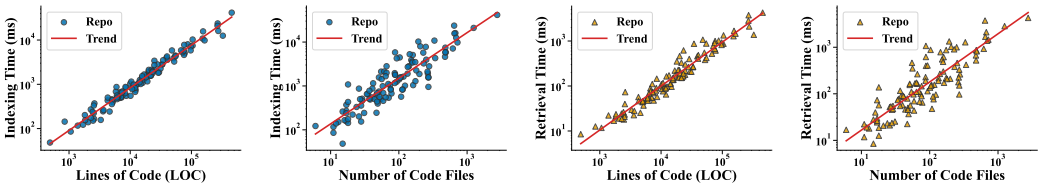
Finding 3: Navigation-based paradigm demonstrates the potential to outperform all passive strategies with powerful commercial models, particularly when equipped with advanced tools (*i.e.*, Nav_{Plus}). This advantage amplifies on class-level tasks, where passive dependency resolution becomes less effective. However, this paradigm's effectiveness is highly dependent on both the backbone LLM's reasoning capabilities and the tool interface design, while weaker models may experience performance degradation across both task granularities.

4.2 RQ2: Efficiency

Table 7. Time efficiency on DevEval. Offline costs are one-time per repository. Online costs are per-task. Navigation costs (*marked with* *) are averaged across all evaluated language models, as they are model-dependent.

Method Paradigm		Offline Phase	Online Phase (Per-Task)	
		$T_{\text{offline\_prep}}$ (ms)	$T_{\text{online\_prep}}$ (ms)	N_{llm} (# Invocations)
Base	—	N/A	≈ 0	1
Similarity-based ICL	ICL_{BM25}	133	11	1
	$ICL_{UniXcoder}$	7811	56	1
	$ICL_{CoCoSoDa}$	5175	47	1
	RepoCoder	487	127	2
Static Analysis	SA_{Method}	3011	383	1
	SA_{File}	2756	269	1
	A3-CodGen	4687	421	1
Navigation	Nav_{Raw}	N/A	1426*	9.85*
	Nav_{Plus}	9380	2075*	4.62*
	SWE-Agent	N/A	1386*	11.47*

Beyond effectiveness, the practical viability of any context engineering paradigm hinges on its efficiency. In this RQ, we provide a holistic analysis of the efficiency trade-offs. We dissect efficiency



(a) Indexing Time vs. LOC. (b) Indexing Time vs. File Count. (c) Retrieval Time vs. LOC. (d) Retrieval Time vs. File Count.

Fig. 3. Performance evaluation of static analysis indexing and retrieval times across codebases of varying sizes. Each point represents a repository.

into three key components: the one-time **Offline Preparation Time** for repository preparation, the per-task **Online Preparation Time** (*i.e.*, retrieval time) perceived by the user, and the **Token Cost** driven by token consumption. Time efficiency (indexing and retrieval latency) depends on the method and repository, not on the generation target, so we report it on function-level benchmarks only. Token cost depends on both, so we analyze it across both function-level and class-level generation.

4.2.1 *Time Efficiency.*

Offline Preparation Time. The one-time computational investment varies significantly across the different paradigms. The lexical similarity-based BM25 is exceptionally lightweight, requiring only 133 ms to index a repository; RepoCoder additionally constructs a sliding-window snippet corpus for its iterative retrieval (487 ms). In contrast, methods that rely on embedding models incur a more substantial, multi-minute cost: UnixCoder is the most expensive at 7,811 ms, while SA_{File} represents a moderate middle ground at 2,756 ms. A3-CodGen adds third-party signature extraction on top of SA, increasing offline cost to 4,687 ms. As shown in Figure 3, static analysis costs scale predictably with repository size, demonstrating nearly linear growth patterns ($O(N)$) with both lines of code and file count. Note that this offline time pattern represents the cold start scenario for repo-level coding task. In practical development, the code index maintenance is incremental, where only modified files are re-parsed, rendering the amortized cost for daily code changes negligible.

The navigation paradigm presents a dichotomy: Nav_{Raw} and SWE-Agent require zero offline preparation, while Nav_{Plus} reintroduces offline costs to build structural and semantic indexes.

Online Time and LLM Invocations. The efficiency landscape completely reverts during the online phase. Passive paradigms exhibit remarkably low online preparation time under 425 ms. The lexical ICL_{BM25} achieves exceptional speed at just 11 ms, while static analysis methods range from 269 ms (SA_{File}) to 421 ms (A3-CodGen). RepoCoder requires 127 ms but uniquely needs two LLM invocations (draft generation + final generation); all other passive methods require exactly one LLM invocation per task.

In contrast, navigation-based methods incur dramatically higher online costs. Nav_{Raw} requires 1,426 ms and averages 9.85 LLM calls per task, while Nav_{Plus} reduces invocations to 4.62 at the expense of higher preparation time (2,075 ms). SWE-Agent averages 11.47 calls per task, the highest among all methods, reflecting the overhead of its windowed file viewing that requires multiple scroll operations per file. Since each LLM call introduces substantial inference delays, the multi-invocation nature of navigation creates multiplicative latency effects that fundamentally distinguish it from the single-call efficiency of passive paradigms.

Finding 4: A fundamental efficiency trade-off separates the passive and active paradigms. Passive methods leverage offline pre-computation to resolve tasks within one or two LLM calls, minimizing time latency. In contrast, the active paradigm's iterative nature inherently requires 4.62–11.47 times LLM invocations on average, leading to substantially higher user-perceived latency.

4.2.2 Token Cost Efficiency.

Table 8 reveals that API token costs vary by more than two orders of magnitude across paradigms, creating significant implications for practical deployment. The basic approach consumes only 4,977 tokens, establishing the minimal cost floor for function-level code generation.

Table 8. Model cost efficiency analysis (Logarithmic Scale Heatmap). For RAG and Static Analysis, the token count is model-agnostic. For Navigation, the token count is model-dependent. Costs are estimated based on public pricing models and their own tokenizers.

Paradigm		Avg. Total Tokens	Averaged Cost per Task ($)							
			L-3.3	Q-3	QC-3	G-O	DS-R1	GPT-4o	C-S-4	G-P-2.5
Base	–	4,977	.0009	.0011	.0012	.0007	.0028	.0113	.0395	.0332
Similarity-based ICL	ICL_{BM25}	28,412	.0059	.0065	.0069	.0074	.0104	.0647	.1129	.0802
	$ICL_{UniXcoder}$	29,790	.0064	.0072	.0076	.0081	.0114	.0711	.1239	.0835
	$ICL_{CoCoSoDa}$	28,581	.0058	.0071	.0069	.0074	.0102	.0647	.1160	.0793
	RepoCoder	33,247	.0067	.0077	.0080	.0082	.0131	.0762	.1519	.1131
Static Analysis	SA_{Method}	8,195	.0016	.0020	.0020	.0026	.0053	.0253	.0539	.0455
	A3-CodGen	9,695	.0019	.0024	.0024	.0035	.0065	.0318	.0606	.0512
	SA_{File}	43,497	.0084	.0107	.0095	.0112	.0216	.0958	.2506	.1787
Navigation	Nav_{Raw}	92,147	.0161	.0385	.0281	.0314	.1272	.3184	.7136	.4158
	SWE-Agent	82,463	.0168	.0361	.0254	.0296	.1095	.2743	.6483	.3651
	Nav_{Plus}	56,785	.0107	.0274	.0196	.0240	.0539	.1972	.5295	.2619

[a] A deeper red color indicates a higher cost.

Function-level generation. Passive paradigms exhibit clear cost-granularity trade-offs. Static analysis presents the internal contrast: SA_{Method} emerges as the most token-efficient context engineering approach at 8,195 tokens (65% increase over baseline), providing dense structural information with minimal overhead. A3-CodGen [32] adds third-party library signatures for a modest increase to 9,695 tokens. Conversely, SA_{File} becomes the most expensive passive method at 43,497 tokens (774% increase), reflecting the cost of including complete dependent file contents.

In contrast, Similarity-based ICL methods operate on a different principle, providing similar examples rather than abstract structures. This approach is necessarily more verbose than SA_{Method}, increasing the total prompt size to ≈29–33k tokens, where RepoCoder [65] incurs the highest cost due to its two sequential LLM calls. This higher token count is a direct result of **our file infilling setting**, which adheres to the common practice of retrieving similar granularity examples [15, 18, 52] to provide contexts.

Navigation methods impose the highest token burden. Nav_{Raw} consumes 92,147 tokens (1,751% increase over baseline), driven by extensive codebase exploration and multiple file reads during iterative discovery. Nav_{Plus} reduces consumption to 56,785 tokens (1,041% increase) through fewer invocations and specialized tools that provide structured information without verbose file operations. SWE-Agent [62] falls between the two at 82,463 tokens, confirming that the high token consumption is an inherent paradigm characteristic regardless of tool interface design. However, even the most optimized version remains at least 1.3x more expensive than all passive methods. For premium models like *Claude Sonnet 4*, navigation costs reach $0.53-$0.71 per task with Nav_{Raw} and $0.26-$0.53 with Nav_{Plus}, compared to $0.04-$0.25 for passive methods. This cost differential creates a significant barrier for large-scale deployment.

Actual cost multipliers exceed token count ratios because navigation generates substantially more output tokens through iterative reasoning [2, 9]: Nav_{Raw} uses 2.1x more tokens than SA_{File}

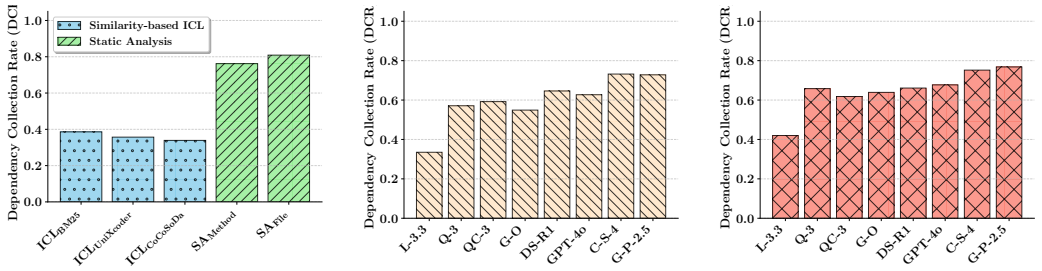
but costs 2.8x more with premium models. Navigation also risks exceeding model context windows (128k–200k tokens), posing practical constraints for large repositories.

Class-level generation. On RepoClassBench, the paradigm cost hierarchy is preserved but absolute costs increase substantially. Class-level tasks require generating entire class bodies with multiple methods, increasing both the input context and the output tokens. Navigation methods are most affected: with *Claude 4 Sonnet*, the average cost per task reaches $1.47 for navigation agents, approximately 1.8× the function-level cost. The amplified cost gap further reinforces the practical trade-off: while navigation agents achieve the highest effectiveness, their deployment cost on complex generation tasks makes paradigm selection a critical economic decision.

Finding 5: Token consumption varies by over two orders of magnitude across paradigms, with navigation methods consuming 10-20x more tokens than passive ones. This cost hierarchy persists on class-level generation, where navigation costs reach $1.47 per task with premium models. While enhanced toolkits can mitigate costs, the navigation paradigm's multi-invocation nature creates substantial monetary expenses and context window risks that challenge large-scale deployment.

4.3 RQ3: Context Quality

To answer RQ3, we evaluate the quality of the context generated by each paradigm using our proposed Dependency Collection Rate (DCR) metric, which measures the proportion of cross-file, ground-truth dependencies that are successfully captured by the context engineering strategy. We report DCR on the function-level benchmarks, where ground-truth dependencies are captured by instrumenting individual function test cases at runtime. For class-level tasks, the ground-truth dependency set becomes the union of all methods runtime invocations, producing a substantially larger denominator that makes DCR values incomparable with function-level results. We therefore leave class-level DCR analysis to future work.



(a) Passive paradigm (b) Nav_{Raw} paradigm. (c) Nav_{Plus} paradigm.

Fig. 4. DCR on DevEval and CoderEval.

Our analysis, presented in Figure 4, reveals significant differences in the ability of each paradigm to collect relevant dependencies. Similarity-based ICL methods consistently achieve the lowest DCR scores, ranging from 33.9% to 38.6% across different retrieval strategies. This result confirms our hypothesis: *these methods optimize for syntactic or semantic similarity to retrieve demonstration examples, not dependency completeness.* While retrieved examples may occasionally contain required function definitions through lexical overlap, this occurs incidentally rather than systematically. The retrieved code snippets typically lack the specific cross-file dependencies needed to resolve the target task's requirements.

In contrast, Static Analysis methods demonstrate substantially higher dependency collection capability. The comprehensive SA_{File} approach achieves the highest DCR among passive methods at

80.9%, while the more targeted SA_{Method} reaches 76.2%. However, static analysis faces fundamental limitations that prevent further improvement. Beyond its inability to resolve runtime behaviors, static analysis struggles when dependencies are co-located with the target generation context within the same file or module, where local context may lack sufficient semantic signals for dependency identification. This limitation is particularly pronounced in dynamically-typed languages such as Python, where duck typing and the absence of mandatory type annotations mean that some dependencies cannot be traced through import statements or explicit type signatures. In statically-typed languages like Java, strict type declarations and interface contracts would provide stronger semantic signals for dependency resolution, potentially narrowing the gap between SA and navigation-based approaches.

Navigation-based paradigms exhibit strong model-dependent variation. Nav_{Raw} ranges from 33.5% DCR with Llama-3.3 to 73.2% with Claude 4 Sonnet, as weaker models struggle to explore repositories systematically with basic text-based tools. When equipped with powerful tools (*e.g.*, FIND_DEFINITION), Nav_{Plus} improves across all models: even Llama-3.3 reaches 42.0% DCR, while the strongest models reach 76.8%. This confirms that well-designed tools can compensate for model limitations in dependency discovery.

To further validate DCR as a context quality metric, we analyze the correlation between DCR and Pass@1 across all method-model configurations. The Spearman rank correlation coefficient is $\rho = 0.78$ ($p < 0.001$), confirming a statistically significant positive relationship. However, the moderate correlation strength indicates that definition coverage alone does not fully explain performance differences, motivating the deeper analysis of what navigation agents acquire beyond dependency definitions. By design, the three adapted baselines employ the same core retrieval or exploration mechanisms as their paradigm counterparts, and their DCR scores follow the same patterns.

Finding 6: Static Analysis and Nav_{Plus} demonstrate superior structural dependency collection, with DCR scores reaching 80.9% and 76.8% respectively. Similarity-based ICL proves ineffective for dependency retrieval, achieving only 33.9-38.6% DCR. Context quality extends beyond dependency coverage alone, as navigation-based approaches with powerful models can outperform higher-DCR static analysis methods through implicit knowledge acquired during repository exploration.

We conduct a context composition analysis to investigate this gap. For each method, we classify the code fragments in its constructed context into three mutually exclusive categories based on ground-truth dependencies (identified via test-case instrumentation, consistent with DCR): *definition context*, containing the syntactic definition of at least one dependency; *reference context*, containing call-site invocations of a dependency but no dependency definition; and *other*, containing neither. When a fragment contains both, we assign it to definition, as this dimension is already captured by DCR.

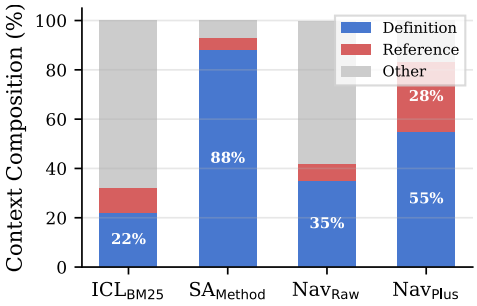


Fig. 5. Context composition by paradigm.

As shown in Figure 5, the four methods exhibit strikingly different compositions. SA_{Method} consists of 88% definition context with only 5% reference, confirming that import-based resolution locates *definitions* but does not search for where dependencies are *called*. ICL_{BM25} is dominated by 68% other content, reflecting similarity-based retrieval's inability to target structural dependencies. Among navigation agents, Nav_{Plus} acquires 28% reference context while Nav_{Raw} achieves only 7%,

with 58% of its context consumed by exploration overhead from coarse-grained tools. This contrast isolates FIND_REFERENCES as the mechanism driving reference acquisition.

Finding 7: Context quality comprises two complementary dimensions: *definition coverage* (measured by DCR) and *reference coverage* (call-site usage patterns). Nav_{Plus} acquires 28% reference context through FIND_REFERENCES, while $\text{SA}_{\text{Method}}$ contexts consist of 88% definitions and only 5% references. On the tasks where Nav_{Plus} outperforms SA_{File} despite lower DCR, the decisive factor is context *composition* rather than *volume*. Future context engineering methods should explicitly target both dimensions.

5 Discussion

5.1 Threats To Validity

A threat to construct validity arises from our specific implementations of the context engineering paradigms. The performance we measure could be an artifact of our implementation choices rather than the inherent properties of the paradigms themselves. To mitigate this, we did not rely on a single implementation per paradigm. Instead, we implemented ten representative variants based on foundational principles from the literature, including three adapted from established tools [32, 62, 65]. By including both our own implementations and adapted prior work within each paradigm, we bracket the design space: consistent trade-offs across variants with different engineering origins confirm that our findings reflect inherent paradigm characteristics rather than implementation artifacts.

5.2 Implications

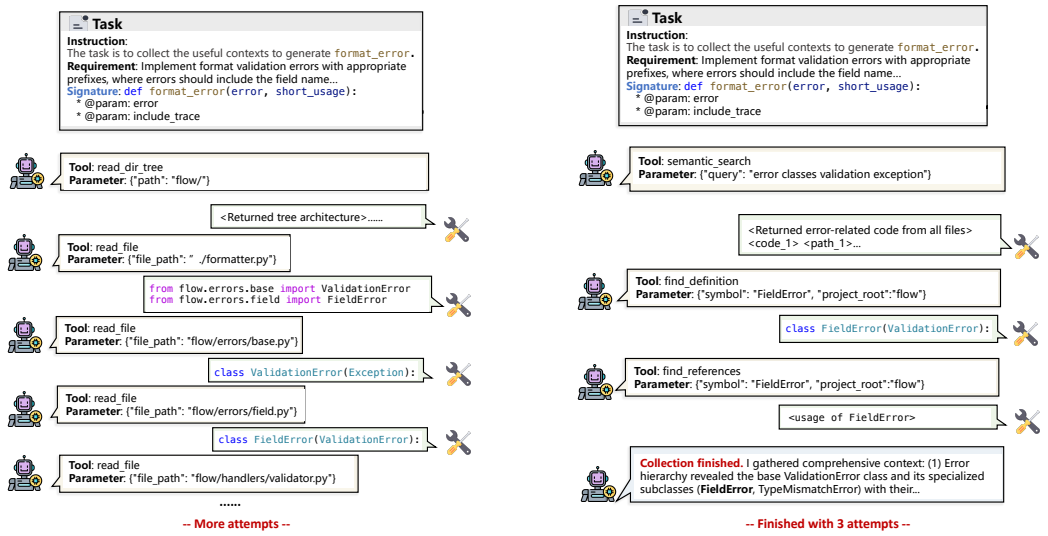
Based on the findings we conclude in Section 4, we provide some implications for researchers who build AI Coding methods and practitioners who use LLMs in software development.

5.2.1 *For AI Coding Researchers.*

Conduct agent-specific training or alignment. The performance gaps across models highlight that many LLMs weren't specially aligned for systematic exploration. Models are normally optimized for single-shot generation with reward signals that prioritize immediate correctness over iterative discovery. Navigation paradigms require fundamentally different alignment objectives: rewarding thorough exploration, penalizing premature termination, and valuing information-gathering steps even when they don't immediately produce code. This alignment mismatch explains why some models perform well in direct generation, but fail in repository navigation.

Design task-specific code context engineering paradigm. The modest improvements from similarity-based methods (3-8%) reveal that generic retrieval misses the trend. When models already have substantial local context, they need functionally relevant code, not syntactically similar examples. Future work should focus on retrieval strategies that understand task intent and code dependencies rather than surface patterns. This means developing context selection methods that consider what the model actually needs to complete the specific task, not what looks most similar.

Design powerful tools. Our evaluation reveals that tool quality fundamentally determines navigation paradigm effectiveness. The dramatic improvement from Nav_{Raw} to Nav_{Plus} demonstrates how well-designed tools can compensate for model limitations. This suggests integrating Language Server Protocol (LSP) capabilities, which provide native code analysis infrastructure already deployed in modern IDEs. Unlike our experimental tools, LSP servers offer tested `go-to-definition`, `find-references`, and symbol resolution capabilities that can transform costly repository exploration into efficient structural queries. Future navigation systems should leverage these existing developer tools rather than reinventing code analysis from scratch.



(a) A simplified case study for Nav_{Raw} (b) A simplified case study for Nav_{Plus}

Fig. 6. A simplified case study

5.2.2 *For Users (Developers).*

Match paradigms to interaction modes. Our empirical analysis reveals a clear performance-latency trade-off that maps to distinct developer workflows. Static analysis is the optimal default for synchronous assistance (*e.g.*, IDE-level code completion); it delivers competitive accuracy (within around 4% of complex agents) at sub-second latency by efficiently capturing structural dependencies. In contrast, navigation agents demonstrate superior effectiveness for asynchronous, code context exploration-heavy tasks requiring seconds to minutes. This differentiation extends to the nature of code context engineering: single-shot static analysis is sufficient for syntactic correctness, whereas multi-step navigation is required to uncover the implicit knowledge (*e.g.*, usage patterns) needed for complex repository-level understanding.

Weigh cost against model capability. While navigation methods reach higher performance ceilings, they incur 10-20x higher API costs (up to $1.47 per task). In real development environments with hundreds of daily interactions, this adds up quickly. Crucially, our results indicate that this investment yields returns *only* when paired with reasoning-strong models (*e.g.*, Claude 4 Sonnet). Weaker models often struggle with systematic exploration, leading to performance degradation compared to passive code context engineering methods. Therefore, teams should prioritize passive paradigms for cost-effective reliability, reserving navigation for genuinely complex problems where both the model's capability and the exploration depth justify the expense.

5.3 Case Study

To illustrate the navigation paradigm's exploratory process, we examine a representative case where agents investigate implementing the `format_error` function based on task requirement.

As shown in Figure 6(a), Nav_{Raw} begins with READ_DIR_TREE to understand the repository structure, discovering the `errors` directory alongside other modules. The agent then uses READ_FILE to examine the target function with the imports: `ValidationError`, `FieldError`. However, without understanding how these classes are actually used, the agent cannot determine their significance. It reads `base.py` to discover the `ValidationError` base class with basic error handling code. Next, it examines `field.py` to find the `FieldError` subclass with `field_name`, but still lacks context

about formatting. Finally, it continues reading files to discover usage patterns file by file, eventually finding code that shows how `field_name` is used in actual error formatting after many steps.

In contrast, as shown in Figure 6(b), a Nav_{Plus} achieves the same understanding in just 3 focused steps. It first uses SEMANTIC_SEARCH with the query *"error classes validation exception"* to immediately discover the error related code, including `ValidationError` base class and `FieldError` subclass. Next, it employs FIND_DEFINITION for `FieldError` to obtain precise structural information, identifying the `field_name` attribute that enables field-specific formatting. Most crucially, it then uses FIND_REFERENCES for `FieldError` to see actual usage patterns across the codebase. This reveals code like `f"Field 'error.field_name': error.message"`, immediately showing the agent exactly how field-specific formatting should work and confirming the key information needed to start summarizing code contexts.

To contrast with passive paradigms, we examine how SA_{File} handles the same task. By parsing the target file's imports, SA_{File} correctly resolves `ValidationError` and `FieldError` to their source files and includes their complete definitions, achieving high DCR. However, SA_{File} cannot discover how these classes are *used* elsewhere in the codebase, as it only follows import chains rather than searching for call sites. The formatting pattern `f"Field '{error.field_name}': {error.message}"`, which reveals the expected output structure, resides in a file with no direct import relationship to the target. This illustrates the fundamental limitation identified in Finding 7: SA_{File} excels at definition coverage but structurally cannot acquire the reference context that Nav_{Plus} obtains through FIND_REFERENCES.

As our findings reveal, it would be nearly impossible for passive paradigms to discover the complex interdependencies and usage patterns required across large repositories. The case study reinforces our finding that tool quality fundamentally determines navigation paradigm effectiveness. This points out the future research direction of integrating more powerful tools and models specifically trained for systematic code exploration to maximize the potential of navigation paradigms and ultimately move toward fully automated software development.

6 Related Works

Despite extensive research in repository-level code generation, existing evaluations of context engineering approaches suffer from fundamental limitations about studying the individual contributions. Most studies evaluate complete, multi-component systems rather than isolating the effectiveness of specific context acquisition strategies. This section reviews the three dominant paradigms and highlights the evaluation gaps that motivate our systematic study.

6.1 Similarity-Based In-Context Learning (ICL)

The most prevalent paradigm treats repositories as indexed knowledge bases, retrieving syntactically or semantically similar code snippets as few-shot examples. Foundational works like ReACC [36] and RepoCoder [66] established dual-encoder approaches, while subsequent research refined retrieval mechanisms through reinforcement learning [54] or evolved query strategies [49]. Studies have explored what constitutes effective examples [15], with some finding in-file context more beneficial than cross-file similarities [18]. However, *these evaluations typically evaluate end-to-end system performance rather than isolating retrieval quality.* Existing metrics focus on downstream code generation success, providing no direct measure of whether retrieved examples contain the necessary contextual information. Furthermore, most studies [15, 18] evaluate only specific retrieval variants (*e.g.*, BM25 vs. semantic), lacking comprehensive comparisons across different paradigms.

6.2 Static Analysis

A growing paradigm leverages program analysis to provide precise structural context derived from codebase architecture. Early work like CoCoMic [13] used static tools to augment ICL, while recent frameworks like DRACO [8] and SCLogger [30] build comprehensive dependency graphs. Advanced approaches like IDECoder [31] integrate IDE capabilities, and systems like RepoGraph [43] create repository-wide structural representations. *Yet evaluation of static analysis approaches faces a critical limitation: the absence of ground-truth dependency baselines.* Existing studies [31, 52] measure downstream performance improvements but cannot quantify how effectively these methods capture the actual dependencies required for successful code generation. This evaluation gap makes it impossible to understand whether performance differences stem from dependency coverage, context presentation, or other factors.

6.3 Navigation-Based Paradigms

This paradigm reframes LLMs as autonomous agents that actively discover context through repository exploration. Systems like SWE-Agent [62] and AutoDev [51] provide filesystem toolkits, while advanced approaches like LocAgent [7] navigate pre-built structural graphs. Research has explored cognitive architectures [3], self-correction mechanisms [58], and multi-agent collaboration [21]. However, navigation paradigm evaluation suffers from the most severe confounding factors. These systems involve complex interactions between LLM reasoning, tool design, and exploration strategies, making it nearly impossible to isolate the contribution of the navigation approach itself. Moreover, the multi-step, model-dependent nature of these agents introduces variability that existing evaluations fail to systematically control for across different LLM capabilities.

6.4 Evaluation Gaps

Our literature review reveals three critical gaps: (i) *Paradigm isolation*: no study independently compares all three paradigms under controlled conditions; (ii) *Context quality measurement*: existing metrics only evaluate downstream performance, lacking direct measures of efficiency and context completeness or relevance; and (iii) *Systematic LLM comparison*: most evaluations miss how paradigm effectiveness varies across different model capabilities. These gaps prevent the field from understanding fundamental trade-offs between approaches and limit principled selection of context engineering strategies for different scenarios.

7 Conclusion

In this paper, we present the first large-scale systematic empirical study comparing three foundational paradigms of code context engineering for repository-level code generation. To address the lack of comprehensive evaluation, we implement 10 representative methods, including three adapted from prior work, and evaluate 8 state-of-the-art LLMs on 439 repository-level tasks from execution-based benchmarks. We also introduce a new context quality metric *DCR* that quantitatively measures the ability to identify and retrieve code dependencies. We provide actionable implications based on our findings for AI coding researchers and software developers.

Acknowledgment

This research is supported by National Natural Science Foundation of China under project (No. 62472126) and CCF-Huawei Populus Grove Fund.

Data Availability

We have made our corrected dataset, all 10 designed and implemented code context engineering methods and benchmarking framework publicly available: https://github.com/YichenLi00/CCE.

References

[1] Amazon. 2023. CodeWhisperer. https://aws.amazon.com/cn/codewhisperer/

[2] Anthropic. 2024. Claude 3.5 Sonnet. https://www.anthropic.com/news/claude-3-5-sonnet https://www.anthropic.com/news/claude-3-5-sonnet.

[3] Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B Ashok, Shashank Shet, et al. 2023. CodePlan: Repository-level Coding using LLMs and Planning. *arXiv preprint arXiv:2309.12499* (2023).

[4] Andrew Blinn, Xiang Li, June Hyung Kim, and Cyrus Omar. 2024. Statically contextualizing large language models with typed holes. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024), 468–498.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[6] Xiang Chen, Chaoyang Gao, Chunyang Chen, Guangbei Zhang, and Yong Liu. 2025. An empirical study on challenges for llm application developers. *ACM Transactions on Software Engineering and Methodology* (2025).

[7] Zhaoling Chen, Xiangru Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. Locagent: Graph-guided llm agents for code localization. *arXiv preprint arXiv:2503.09089* (2025).

[8] Wei Cheng, Yuhan Wu, and Wei Hu. 2024. Dataflow-guided retrieval augmentation for repository-level code completion. *arXiv preprint arXiv:2405.19782* (2024).

[9] Deepmind. 2024. Gemini 1.5 Pro. https://deepmind.google/technologies/gemini/pro/ https://deepmind.google/technologies/gemini/pro/.

[10] DeepSeek. 2024. DeepSeek API. https://platform.deepseek.com/ https://platform.deepseek.com/.

[11] Ajinkya Deshpande, Anmol Agarwal, Shashank Shet, Arun Iyer, Aditya Kanade, Ramakrishna Bairi, and Suresh Parthasarathy. 2024. Class-Level Code Generation from Natural Language Using Iterative, Tool-Enhanced Reasoning over Repository. *CoRR* abs/2405.01573 (2024). arXiv:2405.01573 https://arxiv.org/abs/2405.01573

[12] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. 2023. CROSSCODEEVAL: A Diverse and Multilingual Benchmark for Cross-File Code Completion. *arXiv preprint arXiv:2310.11248* (2023).

[13] Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2022. Cocomic: Code completion by jointly modeling in-file and cross-file context. *arXiv preprint arXiv:2212.10007* (2022).

[14] Zishuo Ding, Yiming Tang, Xiaoyu Cheng, Heng Li, and Weiyi Shang. 2023. LoGenText-Plus: Improving Neural Machine Translation-based Logging Texts Generation with Syntactic Templates. *ACM Transactions on Software Engineering and Methodology* (2023).

[15] Shuzheng Gao, Xin-Cheng Wen, Cuiyun Gao, Wenxuan Wang, and Michael R Lyu. 2023. Constructing Effective In-Context Demonstration for Code Intelligence Tasks: An Empirical Study. *arXiv preprint arXiv:2304.07575* (2023).

[16] GitHub. 2025. GitHub Copilot: Your AI pair programmer. https://github.com/features/copilot

[17] Google. 2024. API reference provided by Google. https://ai.google.dev/gemini-api/docs/models/gemini https://ai.google.dev/gemini-api/docs/models/gemini.

[18] Wenchao Gu, Juntao Chen, Yanlin Wang, Tianyue Jiang, Xingzhe Li, Mingwei Liu, Xilin Liu, Yuchi Ma, and Zibin Zheng. 2025. What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond. *arXiv preprint arXiv:2503.20589* (2025).

[19] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[20] Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. 2024. On the Impacts of Contexts on Repository-Level Code Generation. *arXiv preprint arXiv:2406.11927* (2024).

[21] Dong Huang, Jie M Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010* (2023).

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

[23] Ali Reza Ibrahimzada, Kaiyao Ke, Mrigank Pawagi, Muhammad Salman Abid, Rangeet Pan, Saurabh Sinha, and Reyhaneh Jabbarvand. 2025. AlphaTrans: A Neuro-Symbolic Compositional Approach for Repository-Level Code Translation and Validation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 2454–2476.

[24] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? *CoRR* abs/2310.06770 (2023). arXiv:2310.06770 doi:10.48550/ARXIV.2310.06770

[25] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of LLM-based explainable fault localization. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1424–1446.

[26] Bolun Li, Zhihong Sun, Tao Huang, Hongyu Zhang, Yao Wan, Ge Li, Zhi Jin, and Chen Lyu. 2024. Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 182–203.

[27] Dongze Li, Songqiang Chen, Jialun Cao, and Shing-Chi Cheung. 2025. What Builds Effective In-Context Examples for Code Generation? *arXiv preprint arXiv:2508.06414* (2025).

[28] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. 2024. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. *arXiv preprint arXiv:2405.19856* (2024).

[29] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *CoRR* abs/2305.06161 (2023). arXiv:2305.06161 doi:10.48550/ARXIV.2305.06161

[30] Yichen Li, Yintong Huo, Renyi Zhong, Zhihan Jiang, Jinyang Liu, Junjie Huang, Jiazhen Gu, Pinjia He, and Michael R Lyu. 2024. Go static: Contextualized logging statement generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 609–630.

[31] Yichen Li, Yun Peng, Yintong Huo, and Michael R. Lyu. 2024. Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context. In *Proceedings of the 1st International Workshop on Large Language Models for Code* (Lisbon, Portugal) *(LLM4Code '24)*. Association for Computing Machinery, New York, NY, USA, 70–74. doi:10.1145/3643795.3648392

[32] Dianshu Liao, Shidong Pan, Xiaoyu Sun, Xiaoxue Ren, Qing Huang, Zhenchang Xing, Huan Jin, and Qinying Li. 2024. A 3-CodGen: A Repository-Level Code Generation Framework for Code Reuse with Local-Aware, Global-Aware, and Third-Party-Library-Aware. *IEEE Transactions on Software Engineering* (2024).

[33] Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. 2024. Codexgraph: Bridging large language models and code repositories via code graph databases. *arXiv preprint arXiv:2408.03910* (2024).

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

[35] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. StarCoder 2 and The Stack v2: The Next Generation. *CoRR* abs/2402.19173 (2024). arXiv:2402.19173 doi:10.48550/ARXIV.2402.19173

[36] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[37] Qinyu Luo, Yining Ye, Shihao Liang, Zhong Zhang, Yujia Qin, Yaxi Lu, Yesai Wu, Xin Cong, Yankai Lin, Yingli Zhang, et al. 2024. RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 436–464.

[38] Lingrui Mei, Jiayu Yao, Yuyao Ge, Yiwei Wang, Baolong Bi, Yujun Cai, Jiazhi Liu, Mingyu Li, Zhong-Zhi Li, Duzhen Zhang, et al. 2025. A survey of context engineering for large language models. *arXiv preprint arXiv:2507.13334* (2025).

[39] Meta. 2024. Llama3. https://ai.meta.com/blog/meta-llama-3/ https://ai.meta.com/blog/meta-llama-3/.

[40] Amr Mohamed, Maram Assi, and Mariam Guizani. 2025. The Impact of LLM-Assistants on Software Developer Productivity: A Systematic Literature Review. *arXiv preprint arXiv:2507.03156* (2025).

[41] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_

[42] OpenAI. 2024. GPT-4o. https://openai.com/index/hello-gpt-4o/ https://openai.com/index/hello-gpt-4o/.

[43] Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. 2024. Repograph: Enhancing ai software engineering with repository-level code graph. *arXiv preprint arXiv:2410.14684* (2024).

[44] Yun Peng, Jun Wan, Yichen Li, and Xiaoxue Ren. 2025. Coffe: A code efficiency benchmark for code generation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 242–265.

[45] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2023. Generative Type Inference for Python. *arXiv preprint arXiv:2307.09163* (2023).

[46] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[47] Ensheng Shi, Yanlin Wang, Wenchao Gu, Lun Du, Hongyu Zhang, Shi Han, Dongmei Zhang, and Hongbin Sun. 2023. Cocosoda: Effective contrastive learning for code search. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2198–2210.

[48] Ramneet Singh, Sathvik Joel, Abhav Mehrotra, Nalin Wadhwa, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. 2025. Code Researcher: Deep Research Agent for Large Systems Code and Commit History. *arXiv preprint arXiv:2506.11060* (2025).

[49] Hongjin Su, Shuyang Jiang, Yuhang Lai, Haoyuan Wu, Boao Shi, Che Liu, Qian Liu, and Tao Yu. 2024. Evor: Evolving retrieval for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2024*. 2538–2554.

[50] Hanzhuo Tan, Qi Luo, Ling Jiang, Zizheng Zhan, Jing Li, Haotian Zhang, and Yuqun Zhang. 2024. Prompt-based code completion via multi-retrieval augmented generation. *ACM Transactions on Software Engineering and Methodology* (2024).

[51] Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2024. AutoDev: Automated AI-driven development. *arXiv preprint arXiv:2403.08299* (2024).

[52] Chaozheng Wang, Zezhou Yang, Shuzheng Gao, Cuiyun Gao, Ting Peng, Hailiang Huang, Yuetang Deng, and Michael Lyu. 2025. RAG or Fine-tuning? A Comparative Study on LCMs-based Code Completion in Industry. *arXiv preprint arXiv:2505.15179* (2025).

[53] Wei Wang, Huilong Ning, Shuo Qian, Gaowei Zhang, and Yi Wang. 2024. Characterizing Developers' Behaviors in LLM-Supported Software Development. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 1168–1177.

[54] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2024. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487* (2024).

[55] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025. RLCoder: Reinforcement Learning for Repository-Level Code Completion. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 1140–1152.

[56] Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source Code Is All You Need. *CoRR* abs/2312.02120 (2023). arXiv:2312.02120 doi:10.48550/ARXIV.2312.02120

[57] Papers with Code. 2024. The Leaderboard of the Code Contests benchmark. https://paperswithcode.com/sota/code-generation-on-codecontests https://paperswithcode.com/sota/code-generation-on-codecontests.

[58] Chunqiu Steven Xia and Lingming Zhang. 2023. Conversational automated program repair. *arXiv preprint arXiv:2301.13246* (2023).

[59] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, et al. 2024. Unilog: Automatic logging via llm and in-context learning. In *Proceedings of the 46th ieee/acm international conference on software engineering*. 1–12.

[60] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. 2025. Qwen3 Technical Report. *arXiv preprint arXiv:2505.09388* (2025).

[61] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *CoRR* abs/2405.15793 (2024). arXiv:2405.15793 doi:10.48550/ARXIV.2405.15793

[62] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.

[63] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. 2024. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.

[64] Xinran Yu, Chun Li, Minxue Pan, and Xuandong Li. 2024. DroidCoder: Enhanced Android Code Completion with Context-Enriched Retrieval-Augmented Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 681–693.

[65] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation. In *Proceedings of*

the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023. Association for Computational Linguistics, 2471–2484. doi:10.18653/V1/2023.EMNLP-MAIN.151

[66] Fengji Zhang, Bei Chen, Yue Zhang, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. 2023. Repocoder: Repository-level code completion through iterative retrieval and generation. *arXiv preprint arXiv:2303.12570* (2023).

[67] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339* (2024).

Received 2025-09-12; accepted 2026-03-24