ARKREPOBENCH: A Repository-Level Code Completion Benchmark for HarmonyOS Development

Yanlin Wang[1], Bowen Zhang[1], Yanli Wang[1], Daya Guo[1],
Terry Yue Zhuo[3], Jiachi Chen[2], Mingwei Liu[1]*, Xingong Zhang[4], Zibin Zheng[1]

[1]Sun Yat-sen University, [2]Zhejiang University,
[3]Monash University, [4]Huawei Technologies Co, Ltd

Abstract

ArkTS is the primary programming language for Huawei's HarmonyOS ecosystem, which has expanded across smartphones, tablets, and IoT devices. While large language models have demonstrated strong code generation capabilities for mainstream languages, their performance on ArkTS remains largely unexplored. To address this gap, we introduce ARKREPOBENCH, the first repository-level code completion benchmark for ArkTS, 7,519 samples collected from 20 official HarmonyOS repositories. The benchmark covers multiple difficulty levels and further categorizes completion instances into Single-File, Cross-File Independent, and Cross-File Dependent settings based on dependency analysis, distinguishing the mere presence of cross-file context from its actual necessity. Our experiments show that: (1) ArkTS completion consistently underperforms mainstream languages under our experimental settings, suggesting language-specific challenges associated with this emerging language; (2) open-source 7B models achieve performance comparable to close-source models; (3) cross-file context is a double-edged sword, with sparse retrieval (Jaccard) outperforming dense methods on ArkTS; and (4) HarmonyOS API documentation consistently improves performance, suggesting the benefits of domain-specific enhancements in low-resource settings.

1 Introduction

HarmonyOS has emerged as a rapidly growing operating system ecosystem, with deployment on over one billion devices and millions of developers. As the primary application development language for HarmonyOS, ArkTS introduces unique syntactic constructs, platform-specific APIs, and distinctive programming patterns tailored for cross-device development scenarios (Huawei Developer, 2024). Meanwhile, recent advances in LLMs have revolutionized code intelligence tasks. Both closed-source models (e.g., GPT-5, Claude-4, and Gemini-2.5) and open-source code-centric models (e.g., DeepSeek-Coder and Qwen2.5-Coder) have demonstrated strong capabilities in code understanding and generation, making LLM-powered code completion increasingly accessible. Given the rapid expansion of the HarmonyOS ecosystem, coupled with ArkTS's nature as a typical low-resource language, there is a pressing need to investigate model adaptability in this data-scarce scenario, paving the way for future technical enhancements.

Despite the maturity of LLM-based code assistance for mainstream languages, the capabilities of these models on ArkTS remain underexplored. Existing research on ArkTS has primarily focused on language translation rather than code completion (Zhou et al., 2025). Existing benchmarks (Chen, 2021; Austin et al., 2021; Lu et al., 2021) target mainstream languages exclusively, and recent repository-level benchmarks (Liu et al., 2023; Zhang et al., 2023; Liu et al., 2025) that advance cross-file code completion evaluation also do not include ArkTS. This creates a critical gap between the practical needs of HarmonyOS developers and our ability to assess LLM performance on this emerging language. Additionally, ArkAnalyzer (Chen et al., 2025), the primary static analysis tool for ArkTS, may occasionally miss certain import patterns (e.g., multiple side-effect imports in a single file), posing minor challenges for complete dependency extraction.

To bridge this gap, we introduce ArkRepoBench, the first repository-level code completion benchmark tailored for ArkTS. ArkRepoBench comprises 7,519 high-quality samples extracted from 20 official HarmonyOS repositories, reflecting realistic ArkTS development patterns. During the construction, we perform dependency-aware context extraction to associate each completion instance

* Mingwei Liu is the Corresponding author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 19409–19429

with relevant cross-file context when applicable, enabling faithful repository-level scenarios. We structure ArkRepoBench along two orthogonal dimensions to enable a comprehensive evaluation. First, regarding difficulty and granularity, we explicitly constructed three distinct tasks (Random Span, Atomic API, and Extended API Completion) to test varying spans and structural complexities. Second, for dependency analysis, we stratify instances into three categories: Standalone (SA), Cross-File Independent (CFI), and Cross-File Dependent (CFD). This taxonomy allows us to systematically investigate the model's behavior based on the necessity of external context, disentangling the mere presence of related files from their actual utility.

We conduct extensive experiments on ArkRepoBench with a diverse set of LLMs across different scales and architectures. Our empirical evaluation reveals four key findings: **(i) ArkTS code completion consistently underperforms mainstream programming languages**, with a 3–7% gap in Exact Match and a substantially larger 17–22% gap in Edit Similarity compared with Java and Python. **(ii) Open-source 7B models achieve performance comparable to closed-source models**; for example, DeepSeekCoder-7B attains similar repository-level Exact Match to GPT-4o (5.13% vs. 4.70%) and even outperforms it on context-enriched completion. **(iii) Cross-file context acts as a double-edged sword for ArkTS completion**: retrieval improves True Cross-File cases (e.g., from 11.80% to 16.36% Exact Match on 7B models) but yields marginal or negative gains for Single-File and Pseudo Cross-File cases, with sparse retrieval consistently outperforming dense methods. **(iv) Incorporating HarmonyOS API documentation consistently improves performance**, increasing overall Exact Match by more than 3% on 7B models and approaching oracle-level performance.

Our main contributions are summarized as follows:

- We introduce **ArkRepoBench**, the first repository-level code completion benchmark for ArkTS, providing a foundation for evaluating and advancing LLM assistance for the rapidly growing HarmonyOS developer community.
- We present a **benchmark construction methodology** that captures multiple difficulty levels and distinguishes between single-file and cross-file completion scenarios, enabling nuanced analysis of model capabilities.
- We conduct systematic **empirical evaluations** showing that ArkTS code completion remains challenging for current LLMs, and we derive actionable insights for retrieval augmentation and domain-specific knowledge enhancement.
- We publicly release our benchmark and evaluation framework to facilitate future research on code intelligence for OpenHarmony and ArkTs.

2 Related Work

We provide an extended review of the literature in Appendix A.

2.1 Large Language Models for Code

Large language models for code have advanced significantly, with proprietary models such as GPT-5 (OpenAI, 2025), Claude-4 (Anthropic, 2025), and Gemini-2.5-Pro (Google DeepMind, 2025) demonstrating strong coding capabilities. Open-source alternatives including DeepSeek-Coder (Guo et al., 2024) and Qwen2.5-Coder (Hui et al., 2024) have achieved competitive results through instruction tuning and fill-in-the-middle training.

2.2 Repository-Level Code Completion Benchmarks

Early benchmarks such as HumanEval (Chen, 2021), MBPP (Austin et al., 2021), and CodeXGLUE (Lu et al., 2021) evaluate code completion at the function level. Recent work introduce repository-level benchmarks including RepoBench (Liu et al., 2023), RepoEval (Zhang et al., 2023), CrossCodeEval (Ding et al., 2023), and M2RC-Eval (Liu et al., 2025) to address cross-file dependencies.

2.3 ArkTS and HarmonyOS Development

ArkTS is the primary development language for HarmonyOS, featuring declarative UI and platform-specific APIs (Huawei Developer, 2024). Despite the rapid growth of HarmonyOS, academic research has so far largely focused on language migration (Zhou et al., 2025) and ecosystem analysis (Li et al., 2025b).

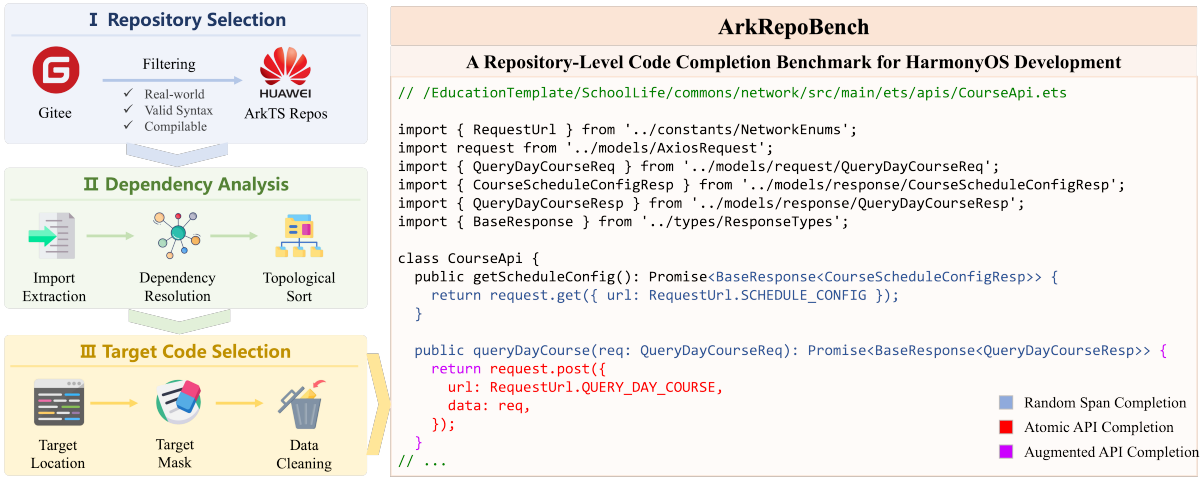

Figure 1: The three-stage construction pipeline of ArkRepoBench: (I) Repository selection from official HarmonyOS ArkTS application templates; (II) Dependency analysis via import extraction and dependency graph construction; (III) Target code selection for three benchmark types (Random Span Completion, Atomic API Completion, and Extended API Completion) with dependency-aware context.

3 ArkRepoBench

We construct ArkRepoBench through a three-stage pipeline (Figure 1). The benchmark comprises three task types: Random Span Completion, Atomic API Completion, and Extended API Completion. Each instance is classified into dependency categories based on cross-file requirements.

3.1 Respository Selection

We collect 20 official ArkTS application templates (API version 5.0.0–5.0.4) from Huawei's Gitee repository[1]. All repositories satisfy three criteria: (1) successful compilation and execution; (2) syntactically valid ArkTS code; (3) representative real-world development patterns. We exclude GitHub repositories due to version incompatibility and compilation failures in preliminary analysis. The selected repositories cover diverse application scenarios and UI/API usage patterns, and the benchmark is constructed at the file and completion-instance level, yielding 7,519 samples across different dependency categories and target granularities.

3.2 Dependency Analysis

We perform dependency analysis to extract cross-file relationships essential for repository-level completion. Since existing parsers (e.g., Tree-sitter) do not support ArkTS and ArkAnalyzer provides limited coverage, we implement a custom parser for ArkTS-specific constructs.

[1] https://gitee.com/appgallery_connect/agc-template-market-harmonyos-demos

Algorithm 1 Dependency Analysis for ArkTS

Require: Project root path P
Ensure: File sequence S in topological order

$M \leftarrow \text{PARSEMODULECONFIGS}(P)$
$F \leftarrow \text{FINDETSFILES}(P)$
for each file $f \in F$ **do**
  $I \leftarrow \text{EXTRACTIMPORTS}(f)$
  for each import $i \in I$ **do**
    $t \leftarrow \text{RESOLVEPATH}(i, M)$
    $G[f] \leftarrow G[f] \cup \{t\}$
  end for
end for
$S \leftarrow \text{TOPOLOGICALSORT}(G)$ **return** S

Dependency Graph Construction. Algorithm 1 outlines our approach. We first parse `oh-package.json5` configuration files to build module name-to-entry mappings. For each ETS file, we extract import statements and resolve them to target files. Relative imports (e.g., `./component`) probe `.ets` extensions or `Index.ets` entries. Module imports use pre-built mappings. This yields a dependency graph G with 4,950 nodes and 30,221 edges across all repositories.

Dependency Category Classification. We apply topological sorting to G and classify instances into three categories based on their position and dependency requirements.

- **Standalone (SA):** Files at the top of topological order with no incoming edges. These completions require only local context.

- **Cross-File Independent (CFI):** Files with dependencies exist, but the completion target uses only locally defined entities.
- **Cross-File Dependent (CFD):** Completions require entities from dependency files (`related_files`).

3.3 Target Code Selection

We construct three completion tasks with varying granularity and difficulty. All tasks attach dependency files as `related_files` to enable repository-level evaluation.

3.3.1 Random Span Completion Data

This task targets arbitrary code spans to evaluate general completion capabilities. For each file f, we tokenize the source code C_f using regex-based segmentation, producing tokens $\mathbf{t} = [t_1, \ldots, t_N]$. We sample a random span from the token sequence as follows:

$$p_{\text{start}} \sim \text{Uniform}(\lceil \alpha N \rceil, \lfloor \beta N \rfloor), \quad (1)$$

$$l'_{\max} = \min(l_{\max}, N - p_{\text{start}} - \delta), \quad (2)$$

$$l \sim \text{Uniform}(l_{\min}, l'_{\max}), \quad (3)$$

$$C_{\text{target}} = C[p_{\text{start}} : p_{\text{start}} + l], \quad (4)$$

where α, β define boundaries that exclude file edges to ensure sufficient context, and δ prevents buffer overflow.

3.3.2 Atomic API Completion Data

This task focuses on single API call completion. We identify dot-chained API expressions (e.g., `object.method()`) in each file using our custom parser. To emphasize cross-file dependencies, we prioritize API calls whose root objects are imported from other modules. If no such calls exist, we sample from all detected APIs. Each file contributes at most one instance to prevent over-representation. The masked target corresponds exactly to the API span (e.g., `method(args)`). The surrounding code serves as context.

3.3.3 Extended API Completion Data

This task extends Atomic API Completion to longer, syntactically complete fragments. After selecting an API call, we expand the masked span rightward. We incorporate subsequent tokens until reaching syntactic boundaries (e.g., balanced parentheses, semicolons) or exceeding a predefined token limit. This produces targets averaging 22.67 tokens compared to 12.53 for atomic APIs (Table 1).

Benchmark	Category	#Samples	#Tokens
Random Span Completion	SA	1,531	47.20
	CFI	336	44.48
	CFD	2,496	54.80
	Overall	*4,363*	*49.89*
Atomic API Completion	SA	284	12.38
	CFI	453	11.95
	CFD	856	13.11
	Overall	*1,593*	*12.53*
Extended API Completion	SA	284	22.88
	CFI	270	21.05
	CFD	1,039	23.48
	Overall	*1,593*	*22.67*

Table 1: Statistics of ArkRepoBench. SA: Standalone, CFI: Cross-File Independent, CFD: Cross-File Dependent. #Samples: number of instances; #Tokens: average target length (regex-based tokenization).

Data Cleaning: During construction, we filter instances involving logging APIs (e.g., `console.log`). These calls contain arbitrary user-defined strings that introduce evaluation noise.

3.4 Benchmark Statistics

Table 1 summarizes ArkRepoBench statistics. Random Span Completion has the longest targets (49.89 tokens average). Atomic API Completion has the shortest (12.53 tokens). Extended API Completion lies in between (22.67 tokens). The distribution across dependency categories reflects real-world ArkTS development patterns.

4 Experimental Setup

This section describes the experimental configurations used throughout the evaluation. We introduce models, retrieval methods, benchmarks, and metrics below.

4.1 Models

To evaluate code completion performance across different model families and parameter scales, both open-source and proprietary LLMs are included in the study.

Open-source Models: **DeepSeek-Coder-Base** (1.3B, 7B)and **Qwen2.5-Coder-Base** (0.5B, 1.5B, 7B) are evaluated, covering model sizes from 0.5B to 7B parameters.

Closed-Source Models: **GPT-4o**, **Claude 3.5-Haiku**, and **Gemini 2.5-Flash** are selected as representative proprietary models from three major LLM providers, due to their relatively favorable trade-offs between performance, cost, and inference speed.[2]

4.2 Retrieval Methods

To quantify the impact of cross-file context on code completion, several retrieval strategies are compared under a unified retrieval pipeline. For all retrieval methods, the **top-10** relevant code snippets are retrieved from related files within the same project.

Snippet Construction and Prompt Assembly. Candidate snippets are segmented deterministically from `related_files`. Each file is first split by blank lines into paragraph-level segments; segments longer than 15 physical lines are further split, and adjacent short segments are merged while keeping each final block no longer than 15 lines. The retrieval query consists of the target file path, imports, and the in-file left context before the masked span, reflecting a realistic left-to-right IDE completion setting. Retrieved blocks are ordered by retriever score and prefixed with file path and line-range metadata before being inserted into the prompt. When the combined prompt exceeds the model context budget, lower-ranked retrieved blocks are removed first, while the target-file left context is preserved. Full prompt templates are provided in Appendix B.

No Retrieval (Baseline): This setting uses only in-file context without any cross-file information, serving as the baseline for evaluating the contribution of retrieval-augmented approaches.

Oracle: This setting directly provides the ground-truth function definition as cross-file context, serving as an approximate upper bound on retrieval quality.

Sparse Retrieval: BM25 (Robertson et al., 2009) and **Jaccard Similarity** are evaluated as sparse retrievers. BM25 is a classical lexical retrieval method based on term frequency and inverse document frequency, while Jaccard Similarity measures token overlap between the query and candidate code snippets.

[2]Specific model versions: `gpt-4o-2024-11-20`, `claude-3-5-haiku-20241022`, and `gemini-2.5-flash` (with thinking disabled).

Dense Retrieval: Following recent retrieval-augmented generation frameworks (Phan et al., 2025), we utilize **UniXcoder** (Guo et al., 2022) for dense retrieval to efficiently encode code snippets into fixed-dimensional vectors.

Documentation Retrieval: For documentation-augmented settings, we extract fenced ArkTS/TypeScript example blocks from official OpenHarmony API documentation and associate each example with its API path or module heading. Import-based Doc parses import statements from the target file to map local identifiers to `@ohos`, `@kit`, and related HarmonyOS namespaces, then retrieves examples by exact API-chain match when available, falling back to prefix or module-level matches otherwise. Oracle Doc retrieves examples associated with the ground-truth target API identifier. Both settings use the same context budget and do not access the ground-truth completion text itself.

4.3 Benchmark

To enable a standardized comparison between ArkTS and other programming languages under the same evaluation protocol, **CCEval** is adopted as the benchmark suite, and the **Python**, **Java**, and **TypeScript** subsets are selected as reference languages for evaluation.

4.4 Evaluation Metrics

To capture both strict correctness and partial overlap between predictions and ground truth, two complementary metrics are reported.

Exact Match (EM): This metric measures the proportion of generated completions that exactly match the ground truth. It is a strict evaluation criterion, requiring character-level equivalence.

Edit Similarity (ES): This metric computes similarity between the generated completion and ground truth based on Levenshtein distance: $ES = 1 - \frac{\text{Levenshtein}(pred, gt)}{\max(|pred|, |gt|)}$. It is more lenient toward partially correct completions.

In addition, we conduct a compilation-based validation on a representative subset using the official HarmonyOS build toolchain (Section 5.3).

4.5 Implementation Details

To ensure reproducibility and consistent decoding behavior across all settings, a unified inference configuration is applied.

All models are configured with temperature 0 (greedy decoding) and a maximum generation length of 256 tokens. Experiments with open-source models are conducted on two NVIDIA A100 GPUs (80GB).

5 Evaluation Results

5.1 ArkTS vs. Mainstream Languages

To understand how LLM-based code completion on ArkTS compares with other mainstream programming languages, we conduct a cross-language comparison between ArkTS and several widely used languages. CCEval serves as the baseline benchmark, from which Java, Python, and TypeScript are selected for comparison.For evaluation, we employ DeepSeek-Coder-Base at two scales (1.3B and 7B parameters) as the backbone model and UniXCoder for cross-file context retrieval.

For a fair comparison, we align the evaluation settings between ArkRepoBench and CCEval. For Java, Python, and TypeScript, the **Atomic API Completion** subset is used as the task formulation, and the average completion length is comparable to that in CCEval. We control the model family, decoding configuration, retrieval setting, and target-span formulation to the extent possible. Nevertheless, this comparison should not be interpreted as isolating programming language syntax alone: static versus dynamic typing, API naming conventions, library ecosystems, and benchmark-construction pipelines may also influence model performance. We therefore use the comparison as evidence of ArkTS-specific practical difficulty under aligned completion settings, while acknowledging that some cross-language factors cannot be fully controlled.

From the experimental results presented in Table 2, **we observe that ArkTS consistently underperforms relative to mainstream programming languages.** Specifically, ArkTS trails Java and Python by margins of 3% to 7% in Exact Match (EM) scores and 18% to 22% in Edit Similarity (ES). Although the EM scores for ArkTS are comparable to those of TypeScript (14.44% compared to 15.41% on the 1.3B model), the disparity in ES remains significant at 43.92% compared to 52.45%. This divergence indicates that even when ArkTS completions achieve similar rates of exact correctness, the incorrect predictions tend to deviate more substantially from the ground truth. The consistently pronounced gap in ES across all comparisons suggests that errors in ArkTS completions are qualitatively more severe. These findings underscore the distinct challenges ArkTS presents as an emerging language with limited representation in pretraining corpora.

5.2 Performance of Different LLMs on ArkTS

To investigate the performance of different LLMs on ArkTS code completion, we evaluate a diverse set of both closed-source and open-source models on the ArkTS benchmark. The evaluated models cover multiple model families and parameter scales, including GPT-4o, Gemini 2.5 Flash, Claude 3.5-Haiku, DeepSeek-Coder-Base (1.3B, 7B), and Qwen2.5-Coder-Base (0.5B, 1.5B, 7B). For retrieval-augmented code completion, we adopt UniXcoder as the unified retriever across all experiments to ensure consistency. All models are assessed under the same evaluation protocol to enable a fair comparison of their code completion capabilities on ArkTS.

Based on the experimental results in Table 3, we draw the following conclusions. A finer-grained breakdown by dependency category is provided in Appendix D.1, revealing how model performance varies across SA, CFI, CFD.

(1) Impact of Model Scale: Within a single model family, increasing parameter size consistently correlates with enhanced performance across all benchmarks. For the Qwen2.5-Coder series, scaling from 0.5B to 7B parameters yields steady improvements; specifically, Exact Match (EM) scores rise from 3.52% to 5.32% on the Repo-Level task and from 13.64% to 18.14% on Atomic API Completion. Analogous trends are evident in the DeepSeekCoder family, suggesting that model capacity is a critical determinant of efficacy in ArkTS code completion.

(2) Closed-Source versus Open-Source Models: While proprietary models generally exhibit superior performance, with Claude 3.5 Haiku leading on both Atomic API Completion (22.38% EM) and Extended API Completion (10.30% EM). Gemini 2.5-Flash achieving the highest results on Random Span (6.53% EM) Notably, Qwen2.5-Coder-7B nearly matches GPT-4o on the Random Span task, achieving 5.32% EM compared to 5.37%, and actually surpasses it on Extended API, scoring 8.35% against 6.49%. This indicates that lightweight open-source models offer a viable alternative to large proprietary models for ArkTS code completion, particularly in scenarios prioritizing deploy-

Model	ArkTS		Java		Python		TypeScript	
	EM	ES	EM	ES	EM	ES	EM	ES
DeepSeekCoder-1.3B	14.44	43.92	17.71	61.88	17.67	65.91	15.41	52.45
DeepSeekCoder-7B	16.26	48.06	22.39	65.75	23.33	69.93	19.34	56.29

Table 2: Cross-language comparison of code completion performance.

Model	Random Span		Atomic API		Extended API	
	EM	**ES**	**EM**	**ES**	**EM**	**ES**
DeepSeekCoder-1.3B	2.87	36.24	14.44	43.03	6.40	40.18
DeepSeekCoder-7B	5.13	40.11	16.26	48.06	7.53	43.03
Qwen2.5-Coder-0.5B	2.82	36.19	13.18	40.41	6.09	36.23
Qwen2.5-Coder-1.5B	4.10	38.93	15.44	44.33	7.16	39.93
Qwen2.5-Coder-7B	4.61	41.52	17.83	49.56	8.29	43.79
GPT-4o	4.70	42.19	20.02	52.76	6.71	41.94
Gemini 2.5-Flash	**5.91**	**46.04**	**22.67**	**57.76**	7.84	45.25
Claude 3.5 Haiku	4.84	43.51	20.71	56.74	**10.29**	**50.76**

Table 3: Cross-benchmark comparison of code completion performance.

ment cost, inference latency, or data privacy.

(3) Performance Variability Across Benchmarks: No single model establishes dominance across all evaluation metrics. Gemini 2.5-Flash secures the best performance on Random Span (6.53% EM), whereas Claude 3.5 Haiku excels in Atomic API Completion (22.38% EM) and Extended API (10.30% EM). This variability indicates that different models possess distinct strengths: Gemini proves more effective in Random Span contexts, while Claude demonstrates superior capability in API-centric tasks. Consequently, model selection for ArkTS code completion should be tailored to the specific nature of the task rather than relying on a universally dominant architecture.

5.3 Compilation-Based Validation

To complement lexical metrics, we additionally perform build validation on a random subset of 100 Extended API Completion instances, focusing on GPT-4o and Qwen2.5-Coder-7B because they are closest under EM/ES in this split. For each generated completion, we substitute the prediction into the original repository and invoke the official HarmonyOS DevEco build toolchain, which covers project configuration loading, module resolution, ArkTS type checking, and compilation.

As shown in Table 4, most generated completions pass compilation, and all EM-correct completions also compile successfully. However, build success alone is not sufficiently discriminative: many compilable predictions still use plausible but incorrect APIs or fail to match the intended behavior. These results support using EM and ES as strict and reproducible indicators while highlighting the need for richer semantics-aware evaluation in future ArkTS benchmarks, such as unresolved-symbol rate, API-signature consistency, and task-specific runtime tests.

Model	EM	ES	Build Pass
GPT-4o	6.00	39.48	94%
Qwen2.5-Coder-7B	10.00	45.13	93%

Table 4: Compilation-based validation on 100 Extended API Completion instances. EM and Build Pass Rate are reported as percentages.

5.4 Impact of Cross-File Context on ArkTS Completion

To investigate how cross-file context affects code completion performance, we conduct experiments on the **Atomic API Completion** benchmark, which represents the most typical cross-file completion scenario. We evaluate DeepSeekCoder-1.3B and DeepSeekCoder-7B with different retrieval strategies: no context (Baseline), sparse retrieval methods (BM25, Jaccard), dense retrieval (UniXcoder), and Oracle retrieval that provides ground-truth relevant code snippets. To gain deeper insights into when cross-file context is truly beneficial, we further analyze the results across the three dependency categories: SA, CFI, CFD.

Based on the experimental results in Table 4, we draw the following conclusions. We also conduct a supplementary analysis in Appendix D.2 that further partitions samples by whether they require

Model	Retriever	SA		CFI		CFD		Overall	
		EM	ES	EM	ES	EM	ES	EM	ES
DeepSeekCoder-1.3B	No Retrieval	19.37	**47.78**	**14.13**	**44.93**	10.28	40.93	12.99	43.29
	BM25	19.37	47.62	13.69	43.73	12.15	41.20	13.87	43.06
	Jaccard	**19.72**	47.65	14.13	43.84	14.95	43.28	15.57	44.22
	UniXcoder	19.37	47.73	13.91	44.36	13.08	42.42	14.44	43.92
	Oracle	19.37	47.67	13.91	44.27	**19.86**	**50.08**	**18.08**	**48.00**
DeepSeekCoder-7B	No Retrieval	20.77	51.14	15.45	**47.82**	11.80	44.94	14.44	46.87
	BM25	20.77	51.22	**16.34**	47.05	13.79	46.03	15.76	47.25
	Jaccard	20.77	51.21	15.89	47.28	16.36	47.93	17.01	48.33
	UniXcoder	20.77	**51.27**	15.89	47.60	14.95	47.24	16.26	48.06
	Oracle	20.77	51.21	15.89	47.60	**23.95**	**56.37**	**21.09**	**52.96**

Table 5: Impact of cross-file context on API completion across different dependency categories.

Model	Context	SA		CFI		CFD		Overall	
		EM	ES	EM	ES	EM	ES	EM	ES
DeepSeek-1.3B	No Retrieval	19.37	47.78	14.13	44.93	10.28	40.93	12.99	43.29
	UniXcoder	19.37	47.73	13.91	44.36	13.08	42.42	14.44	43.92
	Import-based Doc	**21.48**	**54.63**	**15.01**	48.91	11.10	43.06	14.06	47.13
	UniXcoder + Doc	21.13	54.54	14.79	48.46	**13.32**	**43.63**	**15.03**	47.17
	Oracle Doc	**21.48**	54.49	14.79	**49.00**	11.33	43.21	14.09	**47.24**
DeepSeek-7B	No Retrieval	20.77	51.14	15.45	47.82	11.80	44.94	14.44	46.87
	UniXcoder	20.77	51.27	15.89	47.60	14.95	47.24	16.26	48.06
	Import-based Doc	22.89	55.97	19.65	54.32	14.25	48.72	17.25	51.55
	UniXcoder + Doc	**23.24**	**56.11**	**20.97**	**55.65**	**16.36**	**50.00**	**18.96**	**52.64**
	Oracle Doc	22.89	56.07	19.65	54.52	13.90	48.60	17.11	51.55

Table 6: Effectiveness of HarmonyOS API documentation on code completion. Import-based Doc retrieves API documentation based on `import` statements; Oracle Doc provides ground-truth API documentation.

official HarmonyOS API knowledge, revealing the underlying factors that determine retrieval effectiveness.

(1) Cross-File Context is a Double-Edged Sword for ArkTS: The impact differs across dependency categories. For SA samples, retrieval shows minimal variance (EM stays flat around 20%) as no external dependencies are involved. For CFI samples, the effect is mixed; retrieval hampers the 1.3B model but improves the 7B model, implying that irrelevant API snippets can act as noise for smaller models that lack the capacity to filter out distracting context. Only for CFD samples does retrieval consistently improve performance across all models. This indicates that effective ArkTS completion requires accurately identifying when HarmonyOS API context is truly necessary, motivating adaptive retrieval strategies that condition on the dependency profile of each completion point.

(2) Sparse Retrieval Outperforms Dense Retrieval on ArkTS: In contrast to standard assumptions, the dense retriever UniXcoder does not outperform sparse methods on ArkTS. On CFD samples, Jaccard achieves 14.95% and 16.36% EM on DeepSeekCoder-1.3B and 7B respectively, surpassing UniXcoder's 13.08% and 14.95%. A likely reason is that UniXcoder was primarily trained on mainstream languages and therefore may not adequately capture ArkTS-specific constructs, while the strict naming conventions of HarmonyOS APIs make lexical overlap an unusually reliable signal for relevance. This points to the necessity for retrieval models specifically adapted to emerging languages like ArkTS.

(3) Retrieval Quality is one of the Key Bottlenecks for ArkTS Completion: Oracle retrieval substantially outperforms all automatic methods, with the gap widening for larger models. On CFD samples, the best automatic retriever (Jaccard) achieves 14.95% and 16.36% EM, while Oracle reaches 19.86% and 23.95%, a gap of approximately 5–8 percentage points. Larger models benefit more from Oracle context (12.15 compared to 9.58 percentage points improvement over baseline), indicating they can better utilize HarmonyOS API documentation when correctly retrieved. This demonstrates that for ArkTS, improving retrieval accuracy, particularly for HarmonyOS-

specific APIs and type definitions, is more critical than scaling model size alone, and future ArkTS assistants are likely to gain more from retriever improvements than from further increases in model capacity.

5.5 Effectiveness of Domain-Specific Enhancements for ArkTS

Building on RQ3's finding that retrieval quality is a key bottleneck, we explore enhancing code completion with domain-specific knowledge from official HarmonyOS documentation. We extract API example code from the official documentation and retrieve relevant examples based on `import` statements in the current file, simulating a realistic IDE scenario. We evaluate five configurations: (i) Baseline with no context, (ii) UniXcoder retrieval, (iii) Import-based Doc with API documentation, (iv) UniXcoder + Doc combining both, and (v) Oracle Doc providing ground-truth API documentation as an upper bound.

The results in Table 6 demonstrate that integrating HarmonyOS API documentation significantly enhances ArkTS code completion. Specifically, the Import-based Doc strategy consistently improves performance over the baseline, while its combination with code retrieval yields the optimal outcome. This synergy indicates that documentation supplies essential semantic knowledge regarding HarmonyOS specifications, whereas code retrieval captures project-specific usage patterns. Notably, the Import-based approach marginally outperforms the Oracle Doc setting; we attribute this to the broader API context helping the model comprehend the wider HarmonyOS ecosystem, thereby compensating for its limited pretraining exposure more effectively than narrowly focused ground-truth documentation. However, despite these gains, the analysis in Appendix D.3 reveals that a performance disparity persists for official API-dependent samples, suggesting that while external knowledge injection is beneficial, it cannot entirely substitute for the fundamental lack of ArkTS representation in pretraining corpora. Importantly, Oracle Doc is an oracle over the target API identifier rather than an oracle over the final code answer. This explains why Import-based Doc can slightly outperform Oracle Doc in some cases without implying leakage from the ground-truth completion.

5.6 Side-Effect Import Extraction Coverage

Side-effect imports (e.g., `import './module'`) are commonly used in ArkTS for module registration and initialization. We evaluate the extraction coverage of side-effect imports across all 20 repositories. As shown in Table 7, ArkAnalyzer captures approximately 77% of side-effect imports due to its internal storage mechanism that overwrites entries when multiple side-effect imports exist in the same file. Our augmented pipeline addresses this limitation through source-level parsing, achieving 100% extraction coverage.

Method	Coverage
ArkAnalyzer	77%
Ours	100%

Table 7: Side-effect import extraction coverage.

6 Conclusion

In this paper, we present ArkRepoBench, the first repository-level code completion benchmark specifically designed for ArkTS, 7,519 carefully curated samples distributed across three difficulty levels, featuring a fine-grained categorization of cross-file dependencies.

Our empirical evaluation reveals that ArkTS consistently underperforms compared to mainstream languages, confirming the unique challenges inherent to this emerging language. While closed-source models achieve the best performance, we find that open-source 7B models offer a competitive and cost-effective alternative. Furthermore, cross-file context proves to be a double-edged sword: it is essential for true cross-file completions but can be detrimental when it introduces noise. Interestingly, sparse retrieval outperforms dense methods on ArkTS, largely due to the strict naming conventions of HarmonyOS APIs which favor lexical matching. Finally, incorporating official API documentation significantly improves performance, with our import-based strategy achieving oracle-level results while remaining practical for IDE integration.

Future work includes developing ArkTS-specific retrieval models and adaptive context selection mechanisms to support low-resource programming languages.

Limitations

First, ArkTS is a rapidly evolving language within the burgeoning HarmonyOS ecosystem. The current iteration of ArkRepoBench is constructed using repositories compatible with API versions 5.0.0 through 5.0.4. Consequently, the benchmark does not encompass the most recent syntactic features, API enhancements, or deprecations introduced in higher SDK versions. As the language specification matures, the benchmark will require periodic updates to remain representative of state-of-the-art development patterns.

Furthermore, our evaluation still relies primarily on string-based metrics, supplemented only by compilation-based validation on a representative subset. EM and ES are strict and reproducible, but they may miss semantically equivalent completions. Conversely, compilation success is also incomplete because official template repositories contain mostly scaffold tests and buildable code can still violate developer intent or use an incorrect API. Constructing large-scale runtime tests for ArkTS requires heavier HarmonyOS SDK, device, and application-state dependencies, so richer execution-based evaluation remains important future work.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant No. 92582202, No. 62302534, No. 92582117)

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Anthropic. 2025. Introducing Claude 4. https://www.anthropic.com/news/claude-4. Claude Opus 4 and Claude Sonnet 4 released May 22, 2025.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. 2024. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, and 1 others. 2023. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691.

Linzheng Chai, Shukai Liu, Jian Yang, Yuwei Yin, Ke Jin, Jiaheng Liu, Tao Sun, Ge Zhang, Changyu Ren, Hongcheng Guo, and 1 others. 2024. Mceval: Massively multilingual code evaluation. *arXiv preprint arXiv:2406.07436*.

Haonan Chen, Daihang Chen, Yizhuo Yang, Lingyun Xu, Liang Gao, Mingyi Zhou, Chunming Hu, and Li Li. 2025. Arkanalyzer: The static analysis framework for openharmony. *arXiv preprint arXiv:2501.05798*.

Mark Chen. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Ken Deng, Jiaheng Liu, He Zhu, Congnan Liu, Jingxin Li, Jiakai Wang, Peng Zhao, Chenchen Zhang, Yanan Wu, Xueqiao Yin, and 1 others. 2024. R2c2-coder: Enhancing and benchmarking real-world repository-level code completion abilities of code large language models. *arXiv preprint arXiv:2406.01359*.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and 1 others. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Advances in Neural Information Processing Systems*, 36:46701–46723.

Yangruibo Ding, Zijian Wang, Wasi Ahmad, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2024. Cocomic: Code completion by jointly modeling in-file and cross-file context. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 3433–3445.

Aryaz Eghbali and Michael Pradel. 2024. Dehallucinator: Iterative grounding for llm-based code completion. *arXiv preprint arXiv:2401.01701*.

Google DeepMind. 2025. Gemini 2.5: Our most intelligent AI model with thinking. https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/. Gemini 2.5 Pro released March 25, 2025.

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*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and 1 others. 2025. Deepseek-r1: Incentivizing reasoning capability in

llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, and 1 others. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Huawei Developer. 2024. ArkTS development language. https://developer.huawei.com/consumer/en/arkts/.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805.

Jia Li, Xuyuan Guo, Lei Li, Kechi Zhang, Ge Li, Zhengwei Tao, Fang Liu, Chongyang Tao, Yuqi Zhu, and Zhi Jin. 2025a. Longcodeu: Benchmarking long-context language models on long code understanding. *arXiv preprint arXiv:2503.04359*.

Li Li, Xiang Gao, Hailong Sun, Chunming Hu, Carolyn Sun, Haoyu Wang, Haipeng Cai, Ting Su, Xiapu Luo, Tegawendé Bissyande, and 1 others. 2025b. Software engineering for openharmony: A research roadmap. *ACM Computing Surveys*, 58(2):1–36.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Ming Liang, Xiaoheng Xie, Gehao Zhang, Xunjin Zheng, Peng Di, Hongwei Chen, Chengpeng Wang, Gang Fan, and 1 others. 2024. Repofuse: Repository-level code completion with fused dual context. *arXiv preprint arXiv:2402.14323*.

Jiaheng Liu, Ken Deng, Congnan Liu, Jian Yang, Shukai Liu, He Zhu, Peng Zhao, Linzheng Chai, Yanan Wu, JinKe JinKe, and 1 others. 2025. M2rc-eval: Massively multilingual repository-level code completion evaluation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15661–15684.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024a. Repoqa: Evaluating long context code understanding. *arXiv preprint arXiv:2406.06025*.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *arXiv preprint arXiv:2306.03091*.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. 2024b. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

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*.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, and 1 others. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis.

OpenAI. 2025. GPT-5 system card. https://openai.com/index/introducing-gpt-5/. Released August 7, 2025.

Qiwei Peng, Yekun Chai, and Xuhong Li. 2024. Humaneval-xl: A multilingual code generation benchmark for cross-lingual natural language generalization. *arXiv preprint arXiv:2402.16694*.

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

Stephen Robertson, Hugo Zaragoza, and 1 others. 2009. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

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

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*.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

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. *arXiv preprint arXiv:2303.12570*.

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*.

Bo Zhou, Jiaqi Shi, Ying Wang, Li Li, Tsz On Li, Hai Yu, and Zhiliang Zhu. 2025. Porting software libraries to openharmony: Transitioning from typescript or javascript to arkts. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1445–1466.

A Expanded Related Work

This appendix provides an expanded discussion of related work, offering additional context and technical details that complement Section 2 in the main paper.

A.1 Large Language Models for Code

The development of large language models for code has progressed rapidly in recent years. The open-source community has made significant contributions to code LLMs. CodeGen (Nijkamp et al., 2022) introduced a family of models with multi-turn program synthesis capabilities. StarCoder (Li et al., 2023) and StarCoder2 (Lozhkov et al., 2024) pushed the boundaries by training on The Stack, a large-scale dataset of permissively licensed code. Code Llama (Roziere et al., 2023) Extended the Llama foundation models with specialized code training, offering models ranging from 7B to 70B parameters with support for long contexts up to 100K tokens.

Recent advances have focused on reasoning capabilities. DeepSeek-R1 (Guo et al., 2025) introduced reinforcement learning techniques to incentivize multi-step reasoning, showing significant improvements on complex coding tasks. The Qwen3 series (Yang et al., 2025) further pushed the boundaries with models trained on 36 trillion tokens, demonstrating exceptional performance on long-context tasks and complex reasoning. These long-context capabilities are particularly relevant for repository-level code completion, where models must reason across multiple files and understand complex dependency structures. GPT-4 (Achiam et al., 2023) demonstrated early potential for long-context code understanding with its 32K token context window, paving the way for subsequent models with even larger context capacities.

A.2 Repository-Level Code Intelligence

Repository-level code intelligence has emerged as a critical research direction, addressing the gap between isolated function understanding and holistic codebase comprehension. Early retrieval-augmented approaches laid the foundation: ReACC (Lu et al., 2022) proposed combining lexical copying with semantic retrieval from external code databases, achieving state-of-the-art performance on CodeXGLUE benchmarks. RepoFusion (Shrivastava et al., 2023) extended this by training models to incorporate multiple relevant repository contexts through a Fusion-in-Decoder approach. COCOMIC (Ding et al., 2024) established the importance of combining in-file and cross-file context through joint modeling approaches.

Subsequent research developed more sophisticated context retrieval and fusion techniques. RepoFuse (Liang et al., 2024) proposed methods for fusing dual context from different sources with relevance-guided selection. RepoHyper (Phan et al., 2025) introduced hypergraph-based context retrieval for modeling complex dependency relationships. RepoFormer (Wu et al., 2024) developed selective retrieval mechanisms that dynamically determine when and what cross-file context to incorporate. GraphCoder (Liu et al., 2024b) leveraged code context graphs consisting of control-flow and data-dependence relationships for more accurate retrieval. R2C2-Coder (Deng et al., 2024) proposed context perturbation strategies to simulate real-world repository completion scenarios and introduced the R2C2-Bench benchmark. Planning-based approaches have also shown promise: CodePlan (Bairi et al., 2024) introduced systematic planning for complex repository tasks, while CodeAgent (Zhang et al., 2024) enhanced code generation with tool-integrated agent systems. Recent advances include RLCoder (Wang et al., 2024), which learns retrieval policies through reinforcement learning, and De-hallucinator (Eghbali and Pradel, 2024), which addresses hallucination through iterative grounding.

Evaluating code intelligence across diverse programming languages has gained increasing attention. MultiPL-E (Cassano et al., 2023) pioneered multilingual code evaluation by translating HumanEval to 18 programming languages, revealing significant performance disparities across languages—models performed substantially worse on low-resource languages compared to Python and JavaScript. XCodeEval (Khan et al., 2024) introduced the largest executable multilingual multitask benchmark with 25M document-level examples covering 11 programming languages and 7 tasks including generation, translation, and retrieval. HumanEval-XL (Peng et al., 2024) established connections between 23 natural languages and 12 programming languages with 22,080 prompts, enabling assessment of cross-lingual generalization. McEval (Chai et al., 2024) extended evaluation to 40 languages with native benchmarks rather than translations, providing more authentic assessment

including languages like Perl, Haskell, and Racket. For long-context evaluation, LongCodeU (Li et al., 2025a) provides granular assessment across multiple dimensions of code understanding, while RepoQA (Liu et al., 2024a) evaluates long-context code understanding through function search tasks based on natural language descriptions.

A.3 HarmonyOS Ecosystem and ArkTS Research

The HarmonyOS ecosystem represents an emerging platform with unique software engineering challenges. As Huawei's distributed operating system designed for the Internet of Everything (IoT) era, HarmonyOS has gained significant market adoption, particularly in China, with over 100 million devices running the system. The platform introduces ArkTS as its primary application development language, which extends TypeScript with additional constraints and platform-specific APIs for enhanced performance and security.

The ArkTS language presents unique challenges for code intelligence systems.Unlike mainstream languages with extensive training corpora, ArkTS has limited publicly available code, creating a low-resource scenario that challenges conventional LLM training approaches. Furthermore, ArkTS's restrictions on dynamic typing and certain JavaScript features require models to understand not just syntax but also the semantic constraints imposed by the language design. These characteristics make ArkTS an ideal testbed for evaluating LLMs' ability to generalize to new programming languages and adapt to platform-specific development patterns. Our benchmark, ArkRepoBench, addresses this gap by providing the first systematic evaluation of repository-level code completion for the ArkTS language.

B Prompt and Retrieval Construction

This appendix provides the implementation details used to assemble retrieval-augmented prompts.

Code Snippet Chunking. For each completion instance, candidates are drawn only from its `related_files`. We split each file by blank lines into paragraph-level segments. If a segment exceeds 15 physical lines, it is further divided into consecutive subsegments. Adjacent short segments are merged greedily while keeping the resulting block within 15 lines, producing blocks with an average length of approximately 12 lines. Each block stores its source file path and line range. Retrieval methods rank these blocks with the same query and return the top 10 unless the context budget requires truncation.

Code Retrieval Query. The query is formed from information available before generation: the target file path, import statements, and the left context before the masked span. We omit the right context from the retrieval query and prompt to match left-to-right code completion in IDEs. For sparse retrieval, both the query and candidate snippets are tokenized with the same regex-based tokenizer used for benchmark construction. For dense retrieval, the raw query and candidate snippets are encoded by UniXcoder.

Base-Model Prompt. For base completion models, retrieved code and documentation are prepended directly before the target-file prefix:

```
{API_CONTEXT}
{CROSS_FILE_CONTEXT}
// file: {TARGET_FILE_PATH}
{LEFT_CONTEXT}
```

Each retrieved block in `CROSS_FILE_CONTEXT` is formatted as:

```
// file path: {SOURCE_FILE_PATH}
// lines: {START_LINE}-{END_LINE}
{CODE_BLOCK}
```

Instruction-Model Prompt. For instruction-tuned or chat models, we use the same retrieved content but wrap it in an instruction template:

```
[System]
Only output the completed code without explanations.

[User]
Relevant Code Blocks:
Block 1:
// file path: {SOURCE_FILE_PATH}
// lines: {START_LINE}-{END_LINE}
{CODE_BLOCK}

...

Complete the following ArkTS code:
// file: {TARGET_FILE_PATH}
{LEFT_CONTEXT}
```

If documentation retrieval is enabled, documentation examples are inserted before code blocks under a separate `Relevant API Documentation` header. When truncation is necessary, lower-ranked documentation or code blocks are removed before the target-file left context is shortened.

Documentation Example Extraction. We collect documentation examples from official OpenHarmony API documentation by extracting fenced

code blocks under API or module headings, yielding 24,951 candidate examples. Non-code fragments and empty examples are removed. Each retained example is indexed by its heading path, module namespace, and API-chain tokens when available. Import-based retrieval first uses the target file's imports to identify candidate HarmonyOS namespaces and then matches examples in three stages: exact API-chain match, prefix match, and module-level fallback. Oracle Doc uses the ground-truth target API identifier to select associated examples, but it does not expose the target completion text. We do not explicitly de-duplicate documentation examples against repository code, because the goal is to model realistic external documentation lookup rather than a closed-book setting.

C Data Schema

This section describes the data schema used in our benchmark. Each sample in the dataset is represented as a JSON object with the fields described in Table 8.

D Supplementary Experimental Results

D.1 RQ2: Fine-Grained Performance by Dependency Category

To provide deeper insights into model behavior across different completion scenarios, we present a fine-grained breakdown of performance by dependency category (SA, CFI, CFD) for each benchmark. Tables 9, 10, and 11 report the detailed results.

Atomic API Completion As shown in Table 9, all models achieve their highest EM scores on SA samples, where no cross-file information is required. Notably, Claude 3.5 Haiku and Gemini 2.5-Flash both achieve 25.00% EM on SA, substantially outperforming open-source models. However, the performance gap between SA and CFD varies across model families: closed-source models maintain relatively stable performance (e.g., Claude drops only 5.14% from SA to CFD), while smaller open-source models exhibit larger degradation (e.g., Qwen2.5-Coder-0.5B drops 6.16%). This suggests that closed-source models possess stronger capabilities in leveraging cross-file context for ArkTS completion.

Extended API Completion Table 10 reveals a distinctive pattern: Claude 3.5 Haiku consistently outperforms all other models across all dependency categories, achieving 10.21%, 11.48%, and 10.01% EM on SA, CFI, and CFD respectively. Interestingly, several models show higher performance on CFI than SA (e.g., Gemini 2.5-Flash: 10.00% vs. 8.10%), suggesting that for extended completion spans, additional context can occasionally provide useful signals even when not strictly necessary. The relatively uniform performance of Claude across categories indicates robust handling of longer completion targets regardless of cross-file dependency requirements.

Random Span Completion Table 11 shows the most pronounced performance degradation from SA to CFD. Gemini 2.5-Flash achieves the best SA performance (10.49% EM) but drops to 3.84% on CFD—a 6.65% absolute decrease. This pattern is consistent across all models, with the SA-to-CFD gap ranging from 1.09% (DeepSeekCoder-1.3B) to 6.65% (Gemini 2.5-Flash). The larger gaps observed for stronger models suggest that while these models excel at single-file completion, they struggle proportionally more when cross-file dependencies become essential. CFI samples show intermediate performance, indicating that the mere presence of cross-file context (without necessity) introduces some noise but less severely than true cross-file requirements.

Cross-Benchmark Observations. Comparing across benchmarks, we observe that the relative difficulty ordering (SA $>$ CFI $>$ CFD in terms of EM) holds consistently for Atomic API Completion and Random Span Completion, but is less pronounced for Extended API. This may be attributed to the longer completion spans in Extended API samples, where additional context provides more opportunities for partial credit even when not perfectly aligned with ground truth. The Edit Similarity (ES) metric shows smaller gaps across categories compared to EM, suggesting that models can approximate correct completions even when exact matches fail.

D.2 RQ3 Supplementary: Analysis by API Knowledge Requirement

While Section 5.4 examines cross-file context through dependency categories (SA, CFI, CFD), this supplementary analysis stratifies samples based on their reliance on official HarmonyOS APIs. This distinction elucidates *why* retrieval proves effective in specific scenarios and isolates the knowledge deficits that necessitate the investigation in RQ4.

Field	Type	Description
file_path	String	The file path of the source code file.
left_context	String	The code context preceding the masked position.
right_context	String	The code context following the masked position.
target_code	String	The ground truth code to be predicted.
related_files	List	List of related files for cross-file context.
language	String	The programming language (always "arkts").
task_id	String	A unique identifier for each task.

Table 8: Data schema of the ArkTS benchmark.

Model	SA		CFI		CFD		Total	
	EM	ES	EM	ES	EM	ES	EM	ES
DeepSeekCoder-1.3B	19.37	47.73	13.91	44.36	13.08	42.42	14.44	43.03
DeepSeekCoder-7B	20.77	51.27	15.89	47.60	14.95	47.24	16.26	48.06
Qwen2.5-Coder-0.5B	17.61	45.63	13.69	42.11	11.45	37.77	13.18	40.41
Qwen2.5-Coder-1.5B	17.61	46.36	15.89	45.64	14.49	42.96	15.44	44.33
Qwen2.5-Coder-7B	21.48	52.59	18.32	49.97	16.36	48.34	17.83	49.56
GPT-4o	21.13	52.24	18.76	51.87	20.33	53.41	20.02	52.76
Gemini 2.5-Flash	**25.00**	**59.24**	**23.89**	**58.72**	**21.26**	**56.76**	**22.67**	**57.76**
Claude 3.5 Haiku	**25.00**	56.89	19.65	57.02	19.86	56.55	20.71	56.74

Table 9: Performance breakdown by dependency category on Atomic API Completion.

We distinguish between samples that necessitate official API knowledge and those that do not by analyzing import statements rather than relying on superficial keyword matching. Specifically, we extract the base identifier from the target API call (e.g., `router` from `router.pushUrl`) and resolve it against the file's imports. A sample is categorized as requiring official knowledge if the source module originates from the `@ohos` or `@kit` namespaces, or includes `arkui/arkts`. This import-resolution strategy is crucial because: (1) official API usage often lacks explicit in-line markers (like `@ohos`) within the target code; and (2) file-level imports frequently mix official and third-party modules, making file-wide labels imprecise.

Experimental Results. Table 12 details the performance breakdown. We observe three primary patterns:

(1) Significant performance disparity driven by API knowledge requirements. Within the SA category, models struggle significantly with completions requiring official API knowledge, achieving an Exact Match (EM) score of only 8.90%. In contrast, samples free from such requirements see scores ranging from 30.43% to 33.33%, representing a threefold performance difference. This trend remains consistent across CFI and CFD categories, suggesting that the underrepresentation of HarmonyOS-specific data in pretraining corpora creates a fundamental barrier for completions dependent on this domain knowledge.

(2) Retrieval effectiveness in CFD settings is amplified by API knowledge needs. For CFD samples requiring official API knowledge, Jaccard retrieval raises the EM score from a baseline of 7.56% to 16.84% on DeepSeekCoder-1.3B, while Oracle retrieval further boosts performance to 23.37%. Conversely, samples not requiring this knowledge see more moderate gains, improving from 11.68% to 18.05%. These results indicate that cross-file retrieval is most impactful when it compensates for the model's internal knowledge deficits by surfacing relevant HarmonyOS API definitions from the repository.

(3) Retrieval fails to mitigate deficits in SA and CFI samples requiring official API knowledge. Regardless of the retrieval strategy—sparse or dense—performance remains stagnant for these categories. EM scores plateau at 8.90% for SA and hover between 10% and 11% for CFI. This finding demonstrates that when a completion requires HarmonyOS API knowledge but lacks cross-file dependencies, the primary bottleneck is the model's intrinsic lack of domain knowledge, which repository-level context cannot resolve.

Model	SA		CFI		CFD		Total	
	EM	ES	EM	ES	EM	ES	EM	ES
DeepSeekCoder-1.3B	8.80	44.93	8.18	39.39	5.29	39.12	6.40	40.18
DeepSeekCoder-7B	**10.21**	47.55	8.52	41.01	6.54	42.32	7.53	43.03
Qwen2.5-Coder-0.5B	7.75	39.46	8.92	37.71	4.91	35.00	6.09	36.23
Qwen2.5-Coder-1.5B	7.04	41.68	8.89	39.71	6.74	39.54	7.16	39.93
Qwen2.5-Coder-7B	9.86	49.02	8.61	42.33	7.86	43.25	8.29	43.79
GPT-4o	6.69	41.14	7.78	41.46	6.45	42.30	6.71	41.94
Gemini 2.5-Flash	8.10	46.49	10.00	47.55	7.22	44.32	7.84	45.25
Claude 3.5 Haiku	**10.21**	**50.02**	**11.48**	**50.75**	**10.01**	**50.97**	**10.29**	**50.76**

Table 10: Performance breakdown by dependency category on Extended API Completion.

Model	SA		CFI		CFD		Total	
	EM	ES	EM	ES	EM	ES	EM	ES
DeepSeekCoder-1.3B	3.53	38.05	2.98	37.70	2.44	34.96	2.87	36.24
DeepSeekCoder-7B	8.72	43.37	3.87	39.66	3.12	38.27	5.13	40.11
Qwen2.5-Coder-0.5B	3.92	37.91	3.57	37.95	2.08	34.94	2.82	36.19
Qwen2.5-Coder-1.5B	5.82	40.60	4.46	40.91	3.00	37.68	4.10	38.93
Qwen2.5-Coder-7B	5.88	42.57	4.17	43.80	3.89	40.58	4.61	41.52
GPT-4o	6.46	43.93	**4.49**	40.06	3.80	39.70	4.70	42.19
Gemini 2.5-Flash	**10.49**	**50.18**	4.33	**45.13**	3.84	**44.11**	**5.91**	**46.04**
Claude 3.5 Haiku	6.69	44.90	4.33	44.84	**3.93**	42.60	4.84	43.51

Table 11: Performance breakdown by dependency category on Random Span Completion.

Implications. These observations underscore a critical limitation: standard retrieval-augmented generation is effective only when the relevant context exists within the repository. It cannot address fundamental gaps in domain-specific knowledge. Consequently, this motivates RQ4, where we explore integrating official API documentation as an external knowledge source to support completions where internal repository retrieval proves insufficient.

D.3 RQ4 Supplementary: Analysis by API Knowledge Requirement

Section 5.5 establishes that integrating HarmonyOS API documentation enhances overall completion performance. In this supplementary analysis, we scrutinize whether this improvement targets the specific knowledge deficits highlighted in Appendix D.2—specifically, the suboptimal performance on completions necessitating official HarmonyOS API knowledge.

Experimental Results. Table 13 details the performance metrics stratified by API knowledge requirements, contrasting the baseline against retrieval-only (UniXcoder), documentation-only (Import-based Doc), and combined strategies (UniXcoder + Doc). We observe several distinct trends:

(1) Documentation integration yields the most significant gains for samples dependent on official API knowledge. For instance, in the SA category using DeepSeekCoder-7B, the Import-based Doc approach raises the Exact Match (EM) score for official samples from 8.90% to 13.01%, whereas performance on non-official samples remains static at 33.33%. This trend amplifies within the CFI category, where official samples experience a substantial increase from 10.95% to 20.00%, contrasting sharply with the stable performance of other samples. These results confirm that external documentation effectively supplies the domain-specific definitions absent from the model's pre-training data.

(2) Documentation and retrieval mechanisms offer complementary advantages. The synergistic combination of UniXcoder and documentation yields the highest accuracy for samples requiring official API knowledge, achieving 22.38% EM for CFI and 19.59% EM for CFD on DeepSeekCoder-7B. This suggests that while API documentation fills the semantic gaps regarding HarmonyOS specifications, code retrieval supplies essential project-specific context, with both elements proving indispensable for optimal code completion.

(3) A notable performance disparity persists despite documentation enhancement. Even under the optimal configuration (UniXcoder + Doc),

Model	Retriever	SA		CFI		CFD	
		Official (n=146)	**Other** (n=138)	**Official** (n=210)	**Other** (n=243)	**Official** (n=291)	**Other** (n=565)
DeepSeekCoder-1.3B	No Retrieval	8.90/38.49	30.43/57.61	10.48/39.04	17.28/50.03	7.56/34.74	11.68/44.11
	BM25	8.90/38.42	30.43/57.35	10.48/37.64	16.46/49.00	10.65/37.63	12.92/43.04
	Jaccard	8.90/38.46	31.16/57.38	**11.90**/38.10	16.05/48.79	16.84/41.81	13.98/44.03
	UniXcoder	8.90/38.63	30.43/57.35	10.48/38.19	16.87/49.70	14.78/40.58	12.21/43.37
	Oracle	8.90/38.53	30.43/57.35	10.48/38.16	16.87/49.55	**23.37**/50.70	18.05/49.75
DeepSeekCoder-7B	No Retrieval	8.90/44.71	**33.33**/57.95	10.95/**41.34**	19.34/53.41	7.90/35.65	13.81/49.73
	BM25	8.90/44.78	**33.33**/58.03	11.43/39.85	**20.58**/53.26	11.34/39.04	15.04/49.64
	Jaccard	8.90/44.75	**33.33**/58.03	11.43/39.96	19.75/53.61	17.18/43.86	15.93/50.03
	UniXcoder	**8.90/44.83**	**33.33/58.07**	11.43/39.55	19.75/**54.56**	15.46/43.02	14.69/49.42
	Oracle	8.90/44.75	**33.33**/58.03	11.43/39.55	19.75/**54.56**	23.02/**51.38**	**24.42/58.94**

Table 12: Performance breakdown by API knowledge requirement on Atomic API Completion. *Official*: samples requiring official HarmonyOS API knowledge; *Other*: samples not requiring such knowledge. Results are reported as EM/ES (%).

samples requiring official knowledge trail behind their counterparts: SA samples achieve 13.70% compared to 33.33% for non-official samples, with similar divergences observed in CFI and CFD categories. Although documentation narrows the extreme performance divide seen in the baseline—where official samples initially lagged by a factor of three to four—it does not successfully eliminate the gap.

(4) In the CFD context, combining documentation with retrieval demonstrates diminishing returns. For samples requiring official API knowledge, both UniXcoder and Import-based Doc independently improve performance to an identical 15.46% from a baseline of 7.90%. However, their combined application results in a score of 19.59%. This outcome indicates that while both strategies are beneficial, their effects are partially overlapping rather than strictly additive.

Implications. These findings delineate the boundaries of retrieval-augmented generation for low-resource programming languages. Although injecting HarmonyOS API documentation mitigates specific knowledge deficits, the enduring performance disparity implies that inference-time context cannot entirely substitute for the lack of native ArkTS exposure during the pretraining phase. Consequently, this underscores the necessity for future research focused on integrating ArkTS corpora directly into pretraining or continued pretraining stages to achieve fundamental improvements beyond what retrieval mechanisms can offer.

E Case Study

To better understand the failure modes of existing language models on ArkTS code completion, we present a qualitative analysis of representative error cases. Our analysis reveals two primary categories of errors: *cross-language contamination* and *API misunderstanding*.

E.1 Cross-Language Contamination

A significant category of errors stems from models incorrectly applying knowledge from other programming languages—particularly JavaScript and React—to ArkTS contexts. This phenomenon suggests that the models' training data, which is dominated by mainstream languages, leads to inappropriate knowledge transfer when handling ArkTS-specific syntax.

As illustrated in Figure 2, DeepSeek-Coder-7B generates React-style JSX syntax within an ArkTS builder function. The model produces code following React's component return pattern. However, ArkTS employs a declarative UI paradigm where builder functions should directly invoke UI components without explicit return statements. The ground truth demonstrates this pattern with a simple component invocation.

Figure 3 presents another manifestation of this issue. Given an incomplete import statement in an ArkTS file, GPT-4o incorrectly suggests importing React hooks such as `useState` and `useEffect`. However, ArkTS uses a fundamentally different state management mechanism. The correct completion should import domain-specific constants from the project's local modules, as shown in the figure. This error indicates that the model relies heavily on syntactic patterns from its dominant training distribution rather than understanding the target language's ecosystem.

Model	Context	SA		CFI		CFD	
		Official (n=146)	**Other** (n=138)	**Official** (n=210)	**Other** (n=243)	**Official** (n=291)	**Other** (n=565)
DeepSeekCoder-1.3B	No Retrieval	8.90/38.49	30.43/57.61	10.48/39.04	17.28/50.03	7.56/34.74	11.68/44.11
	UniXcoder	8.90/38.63	30.43/57.35	10.48/38.19	16.87/49.70	14.78/40.58	12.21/43.37
	Import-based Doc	12.33/51.78	31.16/57.65	12.38/47.76	17.28/49.90	9.97/40.97	11.68/44.13
	UniXcoder + Doc	13.01/51.85	29.71/57.39	12.38/47.29	16.87/49.48	15.12/44.42	12.39/43.23
	Oracle Doc	13.01/51.77	30.43/57.36	12.38/47.90	16.87/49.95	10.65/41.55	11.68/44.07
DeepSeekCoder-7B	No Retrieval	8.90/44.71	**33.33**/57.95	10.95/41.34	19.34/53.41	7.90/35.65	13.81/49.73
	UniXcoder	8.90/44.83	**33.33/58.07**	11.43/39.55	**19.75**/54.56	15.46/43.02	**14.69**/49.42
	Import-based Doc	13.01/54.18	**33.33**/57.86	20.00/55.48	19.34/53.31	15.46/46.75	13.63/49.74
	UniXcoder + Doc	**13.70/54.70**	**33.33**/57.60	**22.38/56.88**	**19.75/54.58**	**19.59/50.73**	**14.69**/49.63
	Oracle Doc	13.01/54.38	**33.33**/57.86	20.00/55.96	19.34/53.28	14.43/46.29	13.63/**49.78**

Table 13: Effectiveness of HarmonyOS API documentation by API knowledge requirement on Atomic API Completion. *Official*: samples requiring official HarmonyOS API knowledge; *Other*: samples not requiring such knowledge. Results are reported as EM/ES (%).

E.2 API Misunderstanding

The second category of errors involves incorrect API usage, where models demonstrate insufficient understanding of ArkTS and HarmonyOS-specific APIs. These errors often manifest as plausible-looking but semantically incorrect completions.

Figure 4 shows DeepSeek-Coder-7B attempting to complete a cloud function invocation. The model generates a generic method call pattern that does not exist in the HarmonyOS API. The correct completion requires invoking a specific query method with appropriate parameters, demonstrating that accurate code generation demands precise knowledge of the target platform's API specifications.

In Figure 5, DeepSeek-Coder-1.3B confuses two distinct API categories within HarmonyOS. When the context indicates a click handler that should trigger page navigation, the model incorrectly suggests a dialog dismissal API. The ground truth, however, correctly employs the navigation stack API to push a new page onto the routing stack. This confusion between UI control APIs and navigation APIs highlights the model's shallow understanding of HarmonyOS's architectural patterns.

Key Left Context

```
/FoodAndDrinkTemplate/TeaDrinkOrders/features/order_list/src/main/ets/pages/OrderListPage.ets
import {Constants, NavHeaderBar } from 'common';
import {OrderListComp} from '../components/OrderListComp';
@Builder
export function OrderListPageBuilder{
<MASK>
```

Ground truth

```
OrderListPage() }
```

Wrong answer

```
return(<Page><NavHeaderBar>title={Constants.OrderListPageTitle}/><OrderListComp/></Page>);}
```

Figure 2: Cross-language contamination example from DeepSeek-Coder-7B. The model generates React-style JSX syntax inside an ArkTS `@Builder` function, instead of the correct declarative UI syntax.

Key Left Context

```
/EducationTemplate/ArtTraining/Application/feature/mine/src/main/ets/types/Index.ets
import {CourseInfo, CourseStatusMap, OrderStatusMap, OrderTabsMap} from 'commonlib';
import <MASK>
```

Ground truth

```
{OrderActionMap, OrderButtonStyleMap} from '../constants/Enums';
```

Wrong answer

```
{useState,useEffect} from 'react';
```

Figure 3: Cross-language contamination example from GPT-4o. When completing an import statement, the model incorrectly suggests React hooks instead of the project-specific constants.

Key Left Context

```
/LifestyleAndServiceTemplate/AdministrativeAffairs/Application/features/information/src/main/ets/pages/InformationPage.ets
async aboutToAppear() {
    commonToast.show({
        type: 'loading',
    })
    let resp = await <MASK>
```

Ground truth

```
CloudFunction.getInformaticaList(0, '')
```

Wrong answer

```
CloudFunction.call(CloudFunctionInformationResult)
```

Figure 4: API misunderstanding example from DeepSeek-Coder-7B. The model suggests a generic cloud function call pattern instead of the correct HarmonyOS API method.

Key Left Context

```
/PhotographyTemplate/ImageProcessing/scenes/personal/src/main/ets/pages/Setting.ets
...
.onClick(() => {
<MASK>
```

Ground truth

```
this.pathStack.pushPath({ name: 'saveSetting' })
```

Wrong answer

```
this.dialogController.close()
```

Figure 5: API misunderstanding example from DeepSeek-Coder-1.3B. The model confuses dialog control APIs with navigation APIs.