Beyond "What to Retrieve": Uncertainty in Retrieval-Augmented Code Generation

Chandan Kumar Sah, Xiaoli Lian, Li Zhang

Beihang University, Beijing, China
sahchandan98@buaa.edu.cn

Abstract

Repository-level code generation relies on heterogeneous evidence whose relevance, compatibility, and completeness are inherently uncertain. Similar-code examples, repository context, and project-specific APIs may provide complementary information, but can also introduce noisy, redundant, or conflicting signals. Existing retrieval-augmented approaches primarily optimize retrieval relevance without explicitly modeling how uncertainty in retrieved evidence affects downstream generation. We introduce OpenCoder, an uncertainty-aware framework that estimates source-specific uncertainty, uses it to filter and rank heterogeneous evidence, and guides generation, verification, and repair. A factorial analysis over API knowledge, repository context, and similar-code evidence reveals no universal additive source ranking; instead, significant cross-source interactions depend on the accompanying evidence and LLM backend. On an expanded 32-task RepoExec-inline evaluation, OpenCoder improves GPT selected-output correctness over Baseline RAG from 56.25% to 78.13%. However, it matches a verification-and-repair control, and the corresponding Gemini improvement is not statistically supported, indicating backend-dependent benefits. Target-aware API refinement also substantially improves API-set retrieval. These findings support treating uncertainty as an actionable control signal for repository-level retrieval, verification, and repair.

Our code and supporting materials are available at https://github.com/Rocky5502/OpenCoder_V1.

Introduction

Large language models (LLMs) have substantially advanced automated code generation, demonstrating strong performance on function-level programming tasks (Chen et al. 2021; Roziere et al. 2023; Guo et al. 2024). However, real-world software development rarely involves isolated functions. Repository-level code generation requires models to understand cross-file dependencies, project conventions, private APIs, and execution environments that may not be captured by parametric knowledge or the local code context (Yu et al. 2024; Hai, Nguyen, and Bui 2024; Yang et al. 2024). These requirements make repository-level generation particularly challenging: relevant information is distributed across large codebases, while the context available to an LLM remains limited. Retrieval-augmented generation (RAG) addresses this challenge by supplying models with external repository knowledge (Lewis et al. 2020). Existing repository-level methods retrieve similar code, cross-file context, dependency information, or project-specific APIs. RepoCoder alternates retrieval and generation, RepoFormer selectively invokes retrieval, GraphCoder exploits structured code-context graphs, and RLCoder learns retrieval policies from generation feedback (Zhang et al. 2023; Wu et al. 2024; Liu et al. 2024; Wang et al. 2024). More recent work combines heterogeneous evidence sources and shows that repository context and API knowledge can be more useful than naively retrieved similar code (Gu et al. 2026). Collectively, these studies demonstrate the value of repository-aware retrieval, but they primarily optimize *what* information should be retrieved.

Retrieval relevance alone does not ensure reliable generation. Similar-code evidence may be semantically related yet functionally incompatible with the target task, while repository context may contain redundant or conflicting dependencies. API evidence may be incomplete, incorrectly scoped, or incompatible with the target implementation. This is particularly consequential because executable LLM-generated code can still contain substantial API misuse (Zhong and Wang 2024). Moreover, combining additional evidence can increase prompt noise and propagate retrieval errors into generation. Related RAG research has explored selective retrieval, self-reflection, corrective retrieval, and source-reliability estimation (Asai et al. 2024; Yan et al. 2024; Hwang et al. 2024), but these mechanisms have not been systematically developed for heterogeneous repository evidence and execution-based code generation. As summarized in Table 1, existing approaches use different combinations of similar code, repository context, and API knowledge, whereas OpenCoder additionally models the uncertainty associated with these evidence sources. To address this gap, we introduce OpenCoder, an uncertainty-aware framework for retrieval-augmented repository-level code generation. As illustrated in Figure 1, OpenCoder constructs a repository knowledge index, decomposes a query into implementation steps, and retrieves evidence from similar code, repository context, and project-specific APIs. It estimates source-specific uncertainty and integrates evidence according to retrieval relevance, predicted source utility, and evidence uncertainty. The resulting uncertainty trace guides evidence filtering and generation, while executable validation controls

Approach	Similar-Code	API	Repo Context	Uncertainty
A^3-CodGen (Liao et al. 2024)	×	✓	✓	×
RepoCoder (Zhang et al. 2023)	✓	×	✓	×
RepoFormer (Wu et al. 2024)	✓	×	✓	×
RepoMinCoder (Li et al. 2024)	✓	×	✓	×
RLCoder (Wang et al. 2024)	✓	×	✓	×
R^2C^2-Coder (Deng et al. 2024)	✓	×	✓	×
GraphCoder (Liu et al. 2024)	✓	×	✓	×
RepoFuse (Liang et al. 2024)	✓	✓	✓	×
AllianceCoder (Gu et al. 2026)	✓	✓	✓	×
OpenCoder (Ours)	✓	✓	✓	✓

Table 1: Retrieval evidence and uncertainty treatment in representative repository-level code-generation methods. OpenCoder jointly models similar code, repository context, API knowledge, and source-specific uncertainty.

final candidate selection and repair. OpenCoder therefore treats uncertainty as an actionable decision signal rather than only a post-hoc confidence score.

We evaluate OpenCoder using GPT and Gemini backends under a frozen, matched five-candidate protocol. Before inspecting method outputs, we audit 25 additional RepoExec tasks, retain 18 that satisfy the executable protocol, and expand RepoExec-inline from 14 to 32 matched tasks. With GPT, OpenCoder improves selected-output correctness over Baseline RAG from 56.25% to 78.13% (a 21.88-point gain; 95% CI [6.25,40.63]; nominal McNemar $p = .039$), while tying the matched *RAG + Verify/Repair* control. With Gemini, selected-output and candidate-set differences are not statistically supported, and the control is strongest at most metrics. All paired RepoExec-inline Pass@k intervals include zero. On context-limited ExecRepoBench, the control substantially outperforms OpenCoder, revealing a boundary condition under incomplete repository evidence. Target-aware API refinement nevertheless increases macro API F1 from 43.4% to 64.8% for GPT and to 59.1% for Gemini. **Our main contributions are:**

- We formulate retrieval-augmented repository-level code generation as decision-making under uncertainty over heterogeneous repository evidence.
- We introduce OpenCoder, a unified framework that estimates source-specific uncertainty and operationalizes it across evidence filtering, uncertainty-aware multi-source integration, generation, verification, and repair.
- Through a full factorial analysis of API knowledge, repository context, and similar-code evidence, we show that retrieval utility is interaction-dependent rather than governed by a fixed source ranking, empirically motivating uncertainty-aware evidence integration.
- We conduct a controlled matched evaluation across GPT and Gemini on executable repository-level tasks. On 32 RepoExec-inline tasks, OpenCoder improves GPT selected-output correctness by 21.88 percentage points over plain RAG, while target-aware refinement substantially improves API-set retrieval. Additional stress testing characterizes how these benefits depend on the LLM backend and repository evidence completeness.

OpenCoder

Problem Formulation

Given a user query q, consisting of a natural-language requirement and a target function signature, repository-level code generation aims to produce an implementation y that is consistent with the surrounding repository and passes the associated validation tests. Let $\mathcal{C}$ denote the repository knowledge space. Following prior repository-level retrieval settings (Wu et al. 2024; Gu et al. 2026), we organize $\mathcal{C}$ into three evidence sources:

$$\mathcal{C} = \mathcal{A} \cup \mathcal{X} \cup \mathcal{S}, \tag{1}$$

where $\mathcal{A}$ contains project-specific API knowledge, $\mathcal{X}$ contains contextual repository code, and $\mathcal{S}$ contains semantically similar code examples.

For each source $m \in \{\mathcal{A}, \mathcal{X}, \mathcal{S}\}$, let $\mathcal{C}_m \subseteq \mathcal{C}$ be the corresponding source-specific knowledge pool. A retriever $\mathcal{R}_m$ returns source-specific evidence

$$E_m(q) = \mathcal{R}_m(q, \mathcal{C}_m), \tag{2}$$

and the full retrieved evidence set is

$$E(q) = E_{\mathcal{A}}(q) \cup E_{\mathcal{X}}(q) \cup E_{\mathcal{S}}(q). \tag{3}$$

Conventional retrieval-augmented generation primarily ranks evidence according to relevance. In contrast, OpenCoder additionally assigns each retrieved item $e \in E_m(q)$ a source-wise uncertainty score:

$$u_m(e \mid q) \in [0, 1], \tag{4}$$

where larger values indicate lower confidence that the evidence is relevant, complete, and compatible with the target implementation. The corresponding source-level uncertainty is defined as

$$\bar{u}_m(q) = \frac{1}{|E_m(q)|} \sum_{e \in E_m(q)} u_m(e \mid q). \tag{5}$$

To score retrieved evidence, OpenCoder combines relevance, predicted source utility, and evidence uncertainty:

$$\gamma_m(e \mid s) = r_m(e, q_{s,m})\, w_m(s)\, \max\{0, 1 - \alpha u_m(e)\}, \tag{6}$$

where r_m is cosine similarity, $w_m(s)$ is the normalized source-intent weight, and $u_m(e) \in [0, 1]$ is evidence uncertainty. We set $\alpha = 0.5$, retain the top 70% per source, and merge, deduplicate, and rank the remaining evidence. For each source, let $\widetilde{E}_{s,m}$ contain the top 70% of candidates ranked by $\gamma_m(e \mid s)$. The final evidence set is

$$E^*(q) = \mathrm{TopK}_{\gamma}\left(\mathrm{Dedup}\left(\bigcup_{s,m} \widetilde{E}_{s,m}\right)\right), \tag{7}$$

where $K = 10$ is the shared fused-context budget.

The generator produces candidate code conditioned on the query, selected evidence, and source-wise uncertainty trace:

$$y \sim p_\theta\left(y \mid q, E^*(q), \mathbf{u}(q)\right), \tag{8}$$

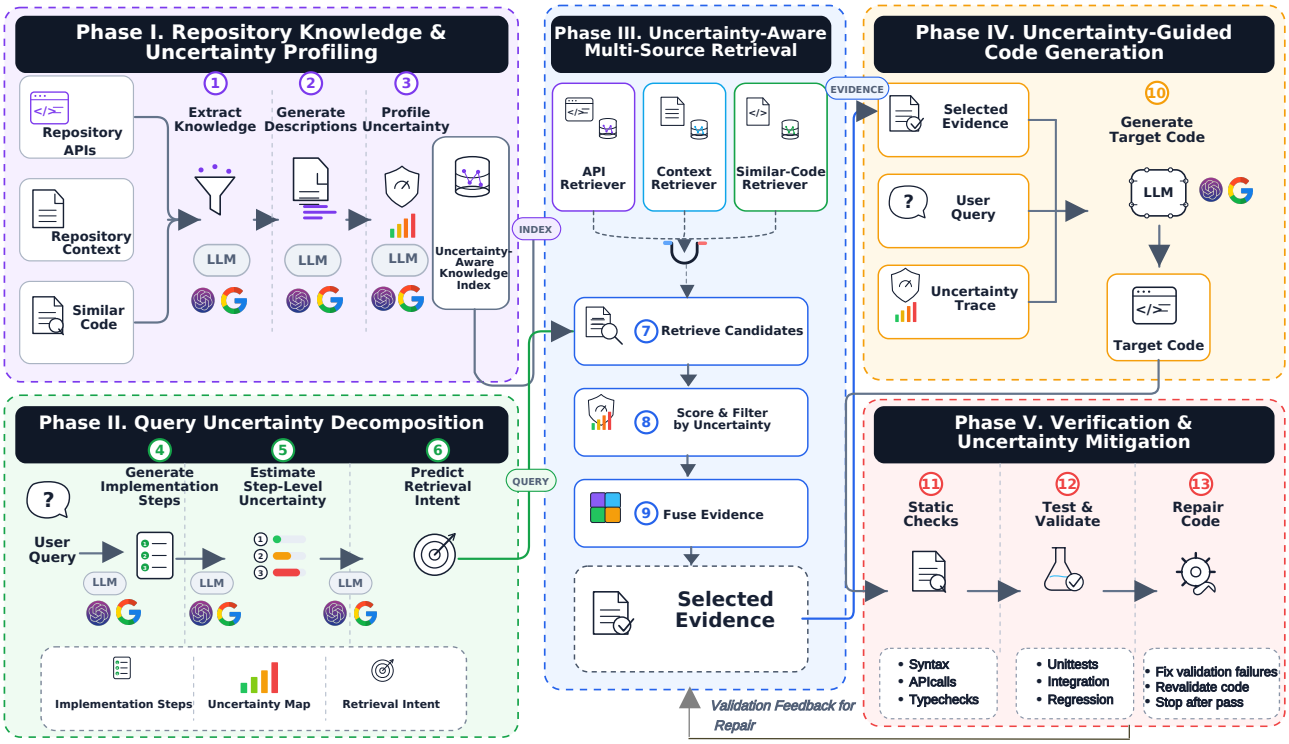


Figure 1: Overview of OpenCoder's uncertainty-aware pipeline for repository indexing, evidence retrieval and fusion, code generation, verification, and repair.

where

$$\mathbf{u}(q) = [\bar{u}_{\mathcal{A}}(q), \bar{u}_{\mathcal{X}}(q), \bar{u}_{\mathcal{S}}(q)] \tag{9}$$

is the source-wise uncertainty trace. Given five candidates $\mathcal{Y}(q) = (y_1, \ldots, y_5)$, OpenCoder validates them in generation order and returns the first passing candidate. If none passes, the earliest candidate y_{mode} from the largest normalized self-consistency group undergoes at most two repair rounds:

$$\hat{y} = \begin{cases} y_{j^*}, & j^* = \min\{j : \text{Validate}(y_j) = 1\}, \\ \text{Repair}^{(r^*)}(y_{\text{mode}}), & r^* \le 2 \text{ is the first passing repair}, \\ \text{Repair}^{(2)}(y_{\text{mode}}), & \text{otherwise.} \end{cases} \tag{10}$$

Repair is triggered only when all raw candidates fail validation.

Uncertainty-Aware Framework

As shown in Figure 1, OpenCoder is an uncertainty-aware framework for retrieval-augmented repository-level code generation. Unlike conventional RAG pipelines that mainly optimize retrieval relevance, OpenCoder explicitly estimates, propagates, and mitigates uncertainty across retrieval, generation, and repair. It comprises five phases. In Phase I, it extracts repository APIs, contextual code, and similar-code examples to construct an uncertainty-aware knowledge index. Phase II decomposes the query into implementation steps and predicts the evidence required for each step. In Phase III, OpenCoder retrieves evidence from multiple sources, scores candidates using retrieval relevance, predicted source utility, and source-specific uncertainty, and then filters, deduplicates, and ranks the retained evidence under a shared context budget. Phase IV conditions the LLM on the selected evidence and uncertainty trace, prioritizing reliable information while suppressing noisy or incompatible context. Finally, Phase V verifies generated candidates using static checks and executable tests. The earliest passing candidate is selected; when no raw candidate passes, the normalized-mode candidate undergoes at most two validation-guided repair rounds.

Experimental Setup

Benchmarks and Task Selection. We select benchmarks according to two criteria: they should support executable evaluation of functional correctness and require repository-specific context, dependencies, or APIs. Accordingly, we use CoderEval, RepoExec, and ExecRepoBench (Yu et al. 2024; Hai, Nguyen, and Bui 2024; Yang et al. 2024). CoderEval contains 230 Python and 230 Java tasks collected from real-world open-source projects and covers six levels of contextual dependency. RepoExec evaluates repository-level function generation in terms of executability, functional correctness, and dependency utilization. ExecRepoBench contains approximately 1.2K Python repository-completion samples constructed through AST-guided masking at statement, expression, and function granularities. We adapt ten suitable ExecRepoBench instances to function-level generation while preserving their available repository context and executable tests. For RepoExec, we construct a controlled executable subset through a pre-specified audit completed before ex-

amining any method outputs. Starting from 14 validated *RepoExec-inline* tasks, we assess the remaining 25 candidates and retain 18 that satisfy the frozen inclusion criteria: dependency completeness, passing reference tests, and compatibility with the evaluation harness. This yields 32 matched tasks for the final analysis. We additionally use ten ExecRepoBench tasks as a deliberately context-limited stress test to evaluate robustness under restricted repository evidence.

Evaluation Scope. The evaluation subset varies according to the artifacts required by each research question. RQ1 and RQ2 use ten execution-backed ExecRepoBench tasks with complete retrieval conditions, uncertainty traces, validation outcomes, and repair records. RQ3 evaluates the expanded 32-task RepoExec-inline set and the ten partial-context ExecRepoBench tasks. RQ4 uses 13 API-bearing RepoExec tasks and 13 API-bearing CoderEval tasks for each LLM backend. The selected ExecRepoBench tasks contain no resolvable repository-specific API calls and are therefore excluded from API-set evaluation. All task-selection decisions are completed before examining method outputs.

Compared Methods. Our primary controlled baseline, *Baseline RAG*, retrieves API knowledge, repository context, and similar-code evidence using the same retrieval and candidate budgets as OpenCoder, but directly concatenates the retrieved items with the generation prompt. It does not use uncertainty-aware filtering, uncertainty-aware evidence integration, verification-guided selection, or repair. *RAG + Verify/Repair* receives the identical retrieved evidence as Baseline RAG and applies the same executable validation and maximum two-round repair budget as OpenCoder, but omits uncertainty-aware filtering, uncertainty-aware evidence integration, and target-aware refinement. This matched control isolates the contribution of uncertainty-aware evidence processing from downstream verification and repair. For RQ4, *OpenCoder w/o API Refinement* removes target-aware API refinement while retaining the remaining pipeline.

LLM Backends and Generation Configuration. We evaluate `gpt-4o-mini` and `gemini-2.5-flash`. RQ1–RQ2 use temperature 0.2 for controlled diagnostic analysis, whereas RQ3–RQ4 use temperature 0.7 with five candidates per task and $k \in \{1, 3, 5\}$. RQ1 evaluates all 2^3 combinations of API knowledge, repository context, and similar-code evidence across ten tasks and two backends, yielding 160 task–condition runs and 480 executed generations. RQ2 compares six paired component configurations. RQ3 evaluates Baseline RAG, RAG + Verify/Repair, and OpenCoder on 32 RepoExec-inline tasks and ten context-limited ExecRepoBench tasks using identical manifests, tests, and a maximum of two repair rounds. The frozen protocol is applied without retuning prompts, thresholds, or retrieval budgets.

Retrieval Configuration. We use UniXcoder (Guo et al. 2022) to embed code, natural-language queries, and API descriptions in a shared dense space. Each source-specific retriever returns up to eight candidates from API knowledge, repository context, or similar-code evidence, and the fused context retains at most ten items under a common prompt budget. OpenCoder ranks candidates using retrieval relevance, predicted source utility, and source-specific uncertainty, filtering low-confidence evidence before generation. Target-aware refinement further removes self-retrieved, redundant, and intent-incompatible APIs.

Uncertainty Estimation and Mitigation. OpenCoder estimates source-specific retrieval uncertainty for API, repository-context, and similar-code evidence. Generation uncertainty combines token entropy, self-consistency, and semantic variance with fixed weights 0.4, 0.4, and 0.2 across all benchmarks and backends. These signals guide evidence filtering and generation. Final candidate selection is validation-driven: the earliest passing candidate is returned, and when no raw candidate passes, the normalized-mode candidate undergoes at most two repair rounds.

Evaluation Metrics. Functional correctness is measured using Pass@k (Chen et al. 2021). For each task, we generate n candidate implementations, of which c pass all executable tests, and compute

$$\text{Pass@}k = 1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}. \tag{11}$$

We average the task-level estimates across benchmark instances and report $k \in \{1, 3, 5\}$ whenever $n \geq k$. RepoExec-inline and the partial-context ExecRepoBench setting use executable tests in RQ3; CoderEval is excluded from this validated functional-correctness comparison. For RQ1, we report task-paired marginal effects of adding each retrieval source on aggregate uncertainty and Pass@1, together with pairwise cross-source interaction effects. For RQ2, we distinguish *effective Pass@1*, which evaluates the final verification-selected or repaired output, from *raw-sample Pass@1*, which measures the proportion of sampled candidates that pass before selection and repair. We additionally report Expected Calibration Error (ECE) (Guo et al. 2017), failure-detection AUROC, and the proportion of post-selection failures recovered through repair. For RQ3, we report candidate-set Pass@k and correctness of the single selected output under a matched five-candidate budget. For RQ4, predicted and ground-truth API sets are compared using macro-averaged precision, recall, F1, and exact API-set match. API-count reliability is categorized as over-, exact-, or under-retrieval. We further evaluate API-specific uncertainty using AUROC, AUPRC, and ECE on a held-out task-level split, treating incorrectly retrieved API items as the positive class.

Statistical Analysis. All evaluations use task-level paired comparisons. For RQ1, we estimate marginal and pairwise interaction effects under the 2^3 factorial design, with Holm correction within each backend and outcome family. For RQ2, changes in binary correctness use two-sided exact McNemar tests with 95% paired-bootstrap confidence intervals. For RQ3, selected-output comparisons use paired-bootstrap intervals and two-sided exact McNemar tests; candidate-set Pass@k comparisons use paired-bootstrap intervals and exact paired tests. The reported RQ3 McNemar values are nominal and unadjusted for multiple comparisons, so we emphasize intervals and treat $p = .039$ as backend-specific support

LLM	Evidence source	ΔU (p_{H})	ΔPass@1 (p_{H})
GPT	API knowledge	$+0.002$ (1.000)	$+8.3$ (.623)
GPT	Repository context	$+0.003$ (1.000)	0.0 (.984)
GPT	Similar code	-0.018 (1.000)	-8.3 (.623)
Gemini	API knowledge	$+0.003$ (1.000)	-5.0 (1.000)
Gemini	Repository context	$+0.009$ (1.000)	-1.7 (1.000)
Gemini	Similar code	-0.010 (1.000)	-1.7 (1.000)

Table 2: Task-paired marginal effects of adding each retrieval source. Parentheses contain Holm-adjusted p-values. Pass@1 effects are reported in percentage points.

rather than universal significance. Additional scope-specific results are provided in the Technical Supplement. For RQ4, AUROC and ECE are computed exclusively on a held-out task split.

Results

RQ1: Influence of Retrieved Evidence

We evaluate all 2^3 combinations of API knowledge, repository context, and similar-code evidence on ten execution-backed ExecRepoBench tasks for each LLM backend. This factorial evaluation comprises 160 task–condition runs and 480 test-executed candidate generations. Table 2 reports the exact task-paired marginal effects, while Figure 2 visualizes both the marginal and pairwise interaction effects with 95% bootstrap confidence intervals. After Holm correction, no individual source exhibits a statistically significant marginal effect on either aggregate uncertainty or Pass@1.

The strongest effects instead arise from interactions among evidence sources. For GPT, repository context and similar-code evidence exhibit a positive Pass@1 interaction of 33.3 percentage points ($p_{\mathrm{Holm}} = .032$). For Gemini, combining API and similar-code evidence reduces aggregate uncertainty by 0.059 ($p_{\mathrm{Holm}} = .032$). These results indicate that the contribution of a retrieval source depends on both the accompanying evidence and the LLM backend, rather than following a universal additive ranking.

Finding 1. Retrieval utility is interaction-dependent: the contribution of API knowledge, repository context, and similar-code evidence varies with their combination and the LLM backend. This finding empirically motivates uncertainty-aware multi-source integration over fixed, source-wise weighting.

RQ2: Uncertainty Quantification and Mitigation

We isolate the effects of query decomposition, uncertainty filtering, guided generation, verified selection, and repair using six paired configurations for each task and LLM backend. Figure 3 distinguishes *effective Pass@1*, which evaluates the final selected or repaired output, from correctness measured over the three raw candidate samples. On the ten-task diagnostic subset, the complete OpenCoder pipeline increases effective Pass@1 from 10.0% to 80.0% for GPT (95% CI [40.0, 100.0], exact McNemar $p = .016$) and from 0.0% to 60.0% for Gemini (95% CI [30.0, 90.0], $p = .031$). The component analysis attributes most of the GPT improvement to verification-based candidate selection. For Gemini, the repair stage recovers 33.3% of generations that remain incorrect after selection. Uncertainty is useful as a control signal, but its reliability is strongly backend-dependent. ECE changes from 0.162 to 0.222 for GPT and from 0.191 to 0.153 for Gemini. Failure-detection AUROC is 0.167 for GPT, indicating an inverse association under the evaluated score orientation, but 0.905 for Gemini. We therefore use uncertainty operationally to control evidence selection, validation, and repair rather than assuming backend-independent probabilistic calibration.

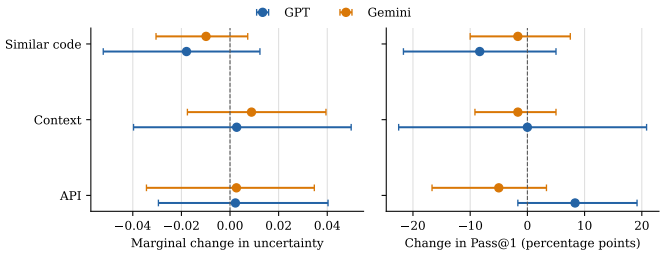
Finding 2: Uncertainty becomes effective through actionable control. By combining uncertainty-aware evidence filtering with executable verification and repair, OpenCoder substantially improves selected-output correctness on the diagnostic subset. Differences across LLM backends underscore the importance of backend-adaptive uncertainty interpretation.

RQ3: End-to-End Effectiveness

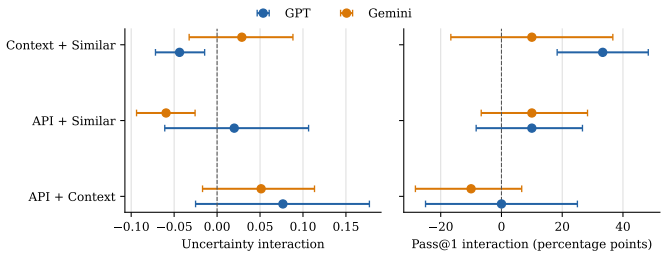
Prior to generation, we audit the remaining 25 RepoExec tasks under a frozen executable protocol and retain 18, yielding an expanded set of 32 matched RepoExec-inline tasks. We evaluate all three methods on this set and on a ten-task partial-context ExecRepoBench stress test. Table 3 reports the expanded results together with the ten-task partial-context ExecRepoBench stress test. All methods use identical task manifests, retrieval evidence, temperature, five-candidate budgets, executable tests, and at most two repair rounds.

With GPT, OpenCoder improves RepoExec-inline selected-output correctness over Baseline RAG from 56.25% to 78.13% ($\Delta = +21.88$, 95% CI [6.25,40.63], W/L/T $= 8/1/23$, nominal McNemar $p = .039$), but ties RAG+Verify/Repair at 78.13% ($\Delta = 0$, CI [-9.38,9.38]). Its Pass@1/3/5 values are 61.88/72.50/78.13, compared with 61.88/74.38/78.13 for the matched control. Thus, the expanded experiment provides backend-specific evidence for better GPT output selection than plain RAG, but not for an advantage beyond executable verification and repair. With Gemini, OpenCoder obtains selected-output correctness of 78.13%, compared with 68.75% for Baseline RAG and 84.38% for RAG+Verify/Repair. The paired differences are not statistically supported (Baseline: $\Delta = +9.38$, CI [-6.25,25.00], $p = .453$; control: $\Delta = -6.25$, CI [-15.63,0], $p = .500$). The control is also strongest across Pass@1/3/5. Across both backends, all paired RepoExec-inline Pass@k confidence intervals include zero. The partial-context ExecRepoBench results retain the previously observed boundary condition: RAG+Verify/Repair outperforms OpenCoder at every candidate-set metric and in selected-output correctness. Incomplete repository evidence can therefore make uncertainty-aware filtering suppress useful alternatives rather than improve the final decision.

Finding 3: Uncertainty-aware control strengthens selected-output reliability when repository evidence is sufficient. On the expanded 32-task RepoExec-inline set, OpenCoder improves GPT selected-output correctness by 21.88 percentage points over Baseline RAG and matches the verification-and-repair control. Results across backends and the context-limited stress test further show that these benefits depend on both LLM behavior and repository-evidence completeness.



(a) Task-paired marginal effects of adding each evidence source.



(b) Task-paired pairwise interactions between evidence sources.

Figure 2: Factorial effects of API knowledge, repository context, and similar-code evidence on aggregate uncertainty and Pass@1. Points show task-paired effect estimates, and error bars denote 95% bootstrap confidence intervals. Positive values indicate increased uncertainty (ΔU) or improved functional correctness (Pass@1); significance is Holm-adjusted.

LLM	Method	RepoExec-inline ($N = 32$)				ExecRepoBench† ($N = 10$)			
		Pass@1	Pass@3	Pass@5	Sel.	Pass@1	Pass@3	Pass@5	Sel.
GPT	Baseline RAG	58.75	68.44	71.88	56.25	52.00	82.00	**100.00**	40.00
GPT	RAG + Verify/Repair	**61.88**	**74.38**	**78.13**	**78.13**	**62.00**	**95.00**	**100.00**	**100.00**
GPT	OpenCoder	**61.88**	72.50	**78.13**	**78.13**	28.00	39.00	40.00	40.00
Gemini	Baseline RAG	66.25	70.31	71.88	68.75	90.00	99.00	**100.00**	80.00
Gemini	RAG + Verify/Repair	**70.00**	**78.75**	**84.38**	**84.38**	**94.00**	**100.00**	**100.00**	**100.00**
Gemini	OpenCoder	65.00	73.75	78.13	78.13	48.00	68.00	80.00	80.00

Table 3: End-to-end RQ3 results under the frozen five-candidate protocol. ExecRepoBench† is evaluated under partial repository context. Sel. denotes selected-output correctness; bold marks the best result per LLM and benchmark, including ties.

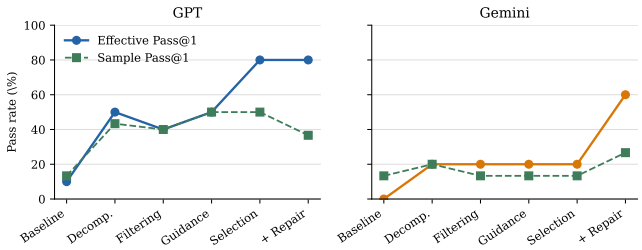


Figure 3: Effective and raw-sample Pass@1 as uncertainty-aware components are introduced. Effective Pass@1 evaluates the final selected or repaired output, whereas raw-sample Pass@1 measures correctness before verification-based selection and repair.

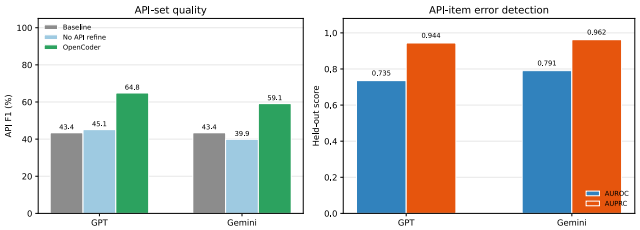


Figure 4: API-set retrieval quality (left) and held-out false-positive API detection (right).

RQ4: API Retrieval Reliability

We evaluate API retrieval on 13 API-bearing RepoExec and 13 API-bearing CoderEval tasks for each LLM backend. The selected ExecRepoBench tasks contain no resolvable repository-specific API calls and are therefore excluded from the API-F1 analysis. As shown in Figure 4, target-aware refinement increases macro-averaged API F1 from 43.4% to 64.8% for GPT and from 43.4% to 59.1% for Gemini. Exact API-set match increases from 0.0% to 57.7% and 50.0%, respectively. Removing API refinement reduces F1 to 45.1% for GPT and 39.9% for Gemini, identifying refinement as the primary contributor to the improvement. API-count reliability nevertheless varies substantially across benchmarks. OpenCoder retrieves the exact number of required APIs for 100.0% of GPT and 92.9% of Gemini RepoExec tasks, but for only 10.5% of CoderEval tasks. It over-retrieves APIs for the remaining 89.5% of CoderEval tasks. On a held-out task split, API-specific uncertainty detects incorrectly retrieved API items with AUROC values of 0.735 and 0.791 and ECE values of 0.030 and 0.041 for GPT and Gemini, respectively. This signal can therefore support filtering of false-positive evidence, but cannot identify required APIs that are absent from the retrieved candidate set. Exact API recovery is also insufficient for functional correctness: although all 13 GPT RepoExec tasks recover the exact API set, only ten generated implementations pass their executable tests.

Table 4 shows that target-aware refinement is critical for API grounding. It raises macro F1 from 45.1 to 64.8 for GPT and from 39.9 to 59.1 for Gemini, while exact-set recovery increases from 3.8% to 57.7% and from 0.0% to 50.0%,

(a) API-set retrieval quality (%)

LLM	Method	Macro F1	Exact
GPT	Baseline RAG	43.4	0.0
	OpenCoder w/o API refinement	45.1	3.8
	OpenCoder	**64.8**	**57.7**
Gemini	Baseline RAG	43.4	0.0
	OpenCoder w/o API refinement	39.9	0.0
	OpenCoder	**59.1**	**50.0**

(b) False-positive API detection

LLM	AUROC	AUPRC	ECE
GPT	0.735	0.944	0.030
Gemini	0.791	0.962	0.041

Table 4: API retrieval quality and false-positive detection. Exact denotes exact API-set recovery over 26 API-bearing tasks per backend: 13 RepoExec and 13 CoderEval tasks.

respectively. API-specific uncertainty further detects false-positive evidence with AUROC values of 0.735 and 0.791 and low ECE, supporting uncertainty-guided filtering.

Finding 4: Target-aware refinement strengthens API grounding. OpenCoder improves API-set quality through target-aware refinement, while API-specific uncertainty provides an actionable signal for identifying false-positive evidence. Together, these mechanisms enhance repository grounding, with recovery of omitted APIs depending on adequate candidate coverage.

Related Work

Retrieval-Augmented Code Generation. Retrieval-augmented code generation uses related code, documentation, and API knowledge to supplement a model's internal knowledge (Hayati et al. 2018; Parvez et al. 2021; Lu et al. 2022; Zhou et al. 2022; Zan et al. 2025). Repository-level methods extend this paradigm to cross-file dependencies, project conventions, and private APIs. RepoCoder alternates retrieval and generation, whereas RepoFormer and SRACG selectively invoke or filter retrieval to reduce harmful augmentation (Zhang et al. 2023; Wu et al. 2024; Wang et al. 2026). Other systems improve context construction through multi-source fusion, structural retrieval, and learned selection (Liang et al. 2024; Liao et al. 2024; Liu et al. 2024; Deng et al. 2024; Wang et al. 2024). Most closely related, Gu et al. compare API knowledge, repository context, and similar-code evidence and demonstrate that their utility varies across source types (Gu et al. 2026). Unlike approaches centered primarily on retrieval effectiveness, OpenCoder explicitly models evidence reliability and uses it throughout generation and validation.

Uncertainty and Execution-Guided Reliability. LLM uncertainty has been studied through semantic consistency, uncertainty decomposition, and confidence calibration (Farquhar et al. 2024; Hou et al. 2024; Shen et al. 2024). In code generation, Incoherence estimates error from behavioral disagreement without an execution oracle (Valentin et al. 2026). CodeT and LEVER use generated tests or execution outcomes for candidate selection, while self-debugging iteratively repairs incorrect programs (Chen et al. 2023; Ni et al. 2023; Chen et al. 2024). These approaches primarily assess or correct uncertainty after generation; OpenCoder additionally models uncertainty in the repository evidence used to construct the generation context.

Adaptive and Reliability-Aware Retrieval. Self-RAG and CRAG determine when evidence should be retrieved, critiqued, or corrected (Asai et al. 2024; Yan et al. 2024). Probing-RAG predicts retrieval necessity, while Reliability-Aware RAG estimates heterogeneous source reliability to prioritize trustworthy evidence (Baek et al. 2025; Hwang et al. 2024). These methods establish retrieval necessity and source reliability as useful control signals, but primarily target knowledge-intensive text generation. OpenCoder specializes these ideas for repository-level code generation by estimating uncertainty over similar code, repository context, and API evidence, integrating the sources through uncertainty-aware ranking, and coupling them with executable verification and repair.

Limitations and Ethical Considerations

Our validated evaluation covers 32 RepoExec-inline tasks and a ten-task partial-context ExecRepoBench stress test. The GPT selected-output gain over Baseline RAG is the only comparison with an unadjusted confidence interval excluding zero (nominal $p = .039$); the method matches the verification-and-repair control, while the corresponding Gemini and candidate-set Pass@k differences are not statistically supported. Generalization may therefore depend on the backend, benchmark scope, and repository-evidence completeness. The framework also incurs additional inference cost, and its uncertainty estimates are not universally calibrated. Required APIs absent from the initial candidate pool cannot be recovered through filtering alone. Generated code may still contain functional, security, or licensing defects and should undergo developer review and project-specific testing before deployment.

Conclusion

We introduced OpenCoder, an uncertainty-aware framework that jointly controls heterogeneous repository evidence, generation, verification, and repair. Our factorial analysis shows that retrieval utility emerges from cross-source interactions, motivating uncertainty-aware multi-source integration rather than fixed source weighting. On 32 pre-audited RepoExec-inline tasks, OpenCoder improves GPT selected-output correctness by 21.88 percentage points over Baseline RAG and matches RAG+Verify/Repair under the same validation-and-repair budget. Target-aware API refinement further improves API-set grounding, while results with Gemini and under partial repository context show that the benefits depend on backend behavior and evidence completeness. Overall, uncertainty is most useful not as an isolated confidence estimate, but as an operational control signal coupled with evidence fusion, executable validation, and repair.

References

Asai, A.; Wu, Z.; Wang, Y.; Sil, A.; and Hajishirzi, H. 2024. Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection. In *The Twelfth International Conference on Learning Representations*.

Baek, I.; Chang, H.; Kim, B.; Lee, J.; and Lee, H. 2025. Probing-rag: Self-probing to guide language models in selective document retrieval. 3287–3304.

Chen, B.; Zhang, F.; Nguyen, A.; Zan, D.; Lin, Z.; Lou, J.-G.; and Chen, W. 2023. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*.

Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H. P. d. O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*.

Chen, X.; Lin, M.; Schärli, N.; and Zhou, D. 2024. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*.

Deng, K.; Liu, J.; Zhu, H.; Liu, C.; Li, J.; Wang, J.; Zhao, P.; Zhang, C.; Wu, Y.; Yin, X.; Zhang, Y.; Su, W.; Xiang, B.; Ge, T.; and Zheng, B. 2024. R2C2-Coder: Enhancing and Benchmarking Real-World Repository-Level Code Completion Abilities of Code Large Language Models. *arXiv preprint arXiv:2406.01359*.

Farquhar, S.; Kossen, J.; Kuhn, L.; and Gal, Y. 2024. Detecting Hallucinations in Large Language Models Using Semantic Entropy. *Nature*, 630: 625–630.

Gu, W.; Chen, J.; Wang, Y.; Jiang, T.; Li, X.; Liu, M.; Liu, X.; Ma, Y.; and Zheng, Z. 2026. What to Retrieve for Effective Retrieval-Augmented Code Generation? An Empirical Study and Beyond. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE)*. ACM.

Guo, C.; Pleiss, G.; Sun, Y.; and Weinberger, K. Q. 2017. On Calibration of Modern Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning*, 1321–1330.

Guo, D.; Lu, S.; Duan, N.; Wang, Y.; Zhou, M.; and Yin, J. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850*.

Guo, D.; Zhu, Q.; Yang, D.; Xie, Z.; Dong, K.; Zhang, W.; Chen, G.; Bi, X.; Wu, Y.; Li, Y.; et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*.

Hai, N. L.; Nguyen, D. M.; and Bui, N. D. Q. 2024. On the Impacts of Contexts on Repository-Level Code Generation. *arXiv preprint arXiv:2406.11927*.

Hayati, S. A.; Olivier, R.; Avvaru, P.; Yin, P.; Tomasic, A.; and Neubig, G. 2018. Retrieval-based neural code generation. 925–930.

Hou, B.; Liu, Y.; Qian, K.; Andreas, J.; Chang, S.; and Zhang, Y. 2024. Decomposing Uncertainty for Large Language Models through Input Clarification Ensembling. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, 19023–19042.

Hwang, J.; Park, J.; Park, H.; Park, S.; and Ok, J. 2024. Retrieval-Augmented Generation with Estimation of Source Reliability. *arXiv preprint arXiv:2410.22954*.

Lewis, P.; Perez, E.; Piktus, A.; Petroni, F.; Karpukhin, V.; Goyal, N.; Küttler, H.; Lewis, M.; Yih, W.-t.; Rocktäschel, T.; Riedel, S.; and Kiela, D. 2020. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *arXiv preprint arXiv:2005.11401*.

Li, Y.; Shi, E.; Zheng, D.; Duan, K.; Chen, J.; and Wang, Y. 2024. RepoMinCoder: Improving Repository-Level Code Generation Based on Information Loss Screening. In *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, 229–238.

Liang, M.; Xie, X.; Zhang, G.; Zheng, X.; Di, P.; Jiang, W.; Chen, H.; Wang, C.; and Fan, G. 2024. RepoFuse: Repository-Level Code Completion with Fused Dual Context. *arXiv preprint arXiv:2402.14323*.

Liao, D.; Pan, S.; Sun, X.; Ren, X.; Huang, Q.; Xing, Z.; Jin, H.; and Li, Q. 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*.

Liu, W.; Yu, A.; Zan, D.; Shen, B.; Zhang, W.; Zhao, H.; Jin, Z.; and Wang, Q. 2024. GraphCoder: Enhancing Repository-Level Code Completion via Coarse-to-Fine Retrieval Based on Code Context Graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*.

Lu, S.; Duan, N.; Han, H.; Guo, D.; Hwang, S.-w.; and Svyatkovskiy, A. 2022. ReACC: A Retrieval-Augmented Code Completion Framework. *arXiv preprint arXiv:2203.07722*.

Ni, A.; Iyer, S.; Radev, D.; Stoyanov, V.; Yih, W.-T.; Wang, S.; and Lin, X. V. 2023. LEVER: Learning to Verify Language-to-Code Generation with Execution. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, 26106–26128.

Parvez, M. R.; Ahmad, W. U.; Chakraborty, S.; Ray, B.; and Chang, K.-W. 2021. REDCODER: Retrieval Augmented Code Generation and Summarization. *arXiv preprint arXiv:2108.11601*.

Roziere, B.; Gehring, J.; Gloeckle, F.; Sootla, S.; Gat, I.; Tan, X. E.; Adi, Y.; Liu, J.; Sauvestre, R.; Remez, T.; et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950*.

Shen, M.; Das, S.; Greenewald, K.; Sattigeri, P.; Wornell, G. W.; and Ghosh, S. 2024. Thermometer: Towards Universal Calibration for Large Language Models. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, 44687–44711.

Valentin, T. J.-M.; Madadi, A.; Sapia, G.; and Böhme, M. 2026. Incoherence as Oracle-less Measure of Error in LLM-Based Code Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, 33305–33313.

Wang, M.; Ma, S.; Gong, S.; Wang, J.; Chen, R.; Cao, L.; and Cai, Y. 2026. SRACG: A Code Generation Framework

with Selective Retrieval Augmentation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, 33584–33592.

Wang, Y.; Wang, Y.; Guo, D.; Chen, J.; Zhang, R.; Ma, Y.; and Zheng, Z. 2024. RLCoder: Reinforcement Learning for Repository-Level Code Completion. *arXiv preprint arXiv:2407.19487*.

Wu, D.; Ahmad, W. U.; Zhang, D.; Ramanathan, M. K.; and Ma, X. 2024. Repoformer: Selective Retrieval for Repository-Level Code Completion. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, 53270–53290.

Yan, S.-Q.; Gu, J.-C.; Zhu, Y.; and Ling, Z.-H. 2024. Corrective Retrieval Augmented Generation. *arXiv preprint arXiv:2401.15884*.

Yang, J.; Zhang, J.; Yang, J.; Jin, K.; Zhang, L.; Peng, Q.; Deng, K.; Miao, Y.; Liu, T.; Cui, Z.; Hui, B.; and Lin, J. 2024. ExecRepoBench: Multi-level Executable Code Completion Evaluation. *arXiv preprint arXiv:2412.11990*.

Yu, H.; Shen, B.; Ran, D.; Zhang, J.; Zhang, Q.; Ma, Y.; Liang, G.; Li, Y.; Wang, Q.; and Xie, T. 2024. CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*.

Zan, D.; Chen, B.; Gong, Y.; Cao, J.; Zhang, F.; Wu, B.; Guan, B.; Yin, Y.; and Wang, Y. 2025. Private-library-oriented code generation with large language models. *Knowledge-Based Systems*, 326: 113934.

Zhang, F.; Chen, B.; Zhang, Y.; Keung, J.; Liu, J.; Zan, D.; Mao, Y.; Lou, J.-G.; and Chen, W. 2023. RepoCoder: Repository-Level Code Completion through Iterative Retrieval and Generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2471–2484.

Zhong, L.; and Wang, Z. 2024. Can LLM Replace Stack Overflow? A Study on Robustness and Reliability of Large Language Model Code Generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, 21841–21849.

Zhou, S.; Alon, U.; Xu, F. F.; Wang, Z.; Jiang, Z.; and Neubig, G. 2022. DocPrompting: Generating Code by Retrieving the Docs. *arXiv preprint arXiv:2207.05987*.