CONTEXTFORGE: LEARNING WHAT TO RETRIEVE FROM DIVERSE VALIDATED PATCHES

Anonymous authors
Paper under double-blind review

ABSTRACT

Search subagents can reduce the expensive repository performed by coding agents, yet they are commonly trained and evaluated using overlap with locations modified in a single solution patch. This proxy omits useful context in unmodified callers, interfaces, tests, and invariants. It also overlooks that there can be many alternative test-valid repairs that depend on different repository evidence. To address these limitations, we introduce a novel reward function for context retrieval that scores edit locations, supporting code, and explanations using patch-conditioned information gains averaged over multiple test-validated repairs. We implement this reward in CONTEXTFORGE, a training framework that constructs diverse repair sets through offline generation and test execution. Using GRPO, we train a Qwen3.5-4B search agent and evaluate it with OpenHands and Mini-SWE-Agent on SWE-bench Pro, Verified, and Multilingual. Across all six benchmark-scaffold settings, CONTEXTFORGE achieves the lowest inference cost and best or tied Pass@1 among evaluated context retrieval agents, and reduces end-to-end cost by up to 29.1% compared to the no-subagent baseline.

1 INTRODUCTION

LLM-based coding agents have made rapid progress in software engineering and can now resolve real-world issues in large codebases (Jimenez et al., 2024; Deng et al., 2025; Zan et al., 2025; Yang et al., 2024; Wang et al., 2025a). Their success depends not only on the ability to generate code, but also on how effectively they understand an issue and identify relevant context, e.g., files, symbols, tests, and cross-file dependencies. Unlike standalone code generation, where the necessary context and requirements are provided upfront, real-world issue resolution requires the coding agent to discover relevant context distributed across a repository. Consequently, effective context retrieval has become a central bottleneck in automated software issue resolution (Zhang et al., 2023; Wu et al., 2024; Chen et al., 2025; Li et al., 2026a; Zhang et al., 2026b). An analysis of Mini-SWE-Agent on SWE-bench Verified finds that read and search operations account for 76.1% of total token usage with Claude Sonnet 4.5 and 67.5% with GLM-4.6. (Wang et al., 2026b).

Since local project exploration consumes a substantial portion of an agent's inference budget, recent approaches delegate it to a specialized search agent that returns a compact context package to a more capable main agent (Sutawika et al., 2026; Chen et al., 2025; Jiang et al., 2025; Zhang et al., 2025; 2023; Wu et al., 2024; Ouyang et al., 2025; Gangi Reddy et al., 2026). Existing intra-repository context retrieval approaches use locations extracted from a single reference patch for training and evaluation, typically based on F1 scores for file, module, and function level retrieval. This supervision assumes the locations changed in one developer's implementation are the only useful context.

This assumption is unnecessarily restrictive and limits the model's ability to actively explore more necessary context. It introduces two limitations: **(1) patch locations and useful context are related but not equivalent**. Code that is essential for understanding a repair, e.g., callers, interface definitions, tests, and analogous implementations, may not be modified by the reference patch at all. Therefore, context that is highly relevant to implementing a correct repair can receive a low patch F1 score because it does not appear in the recorded diff. **(2) one historical patch captures only one repair path** (Smith et al., 2015; Tian et al.). A software issue often admits multiple correct repairs that intervene at different abstraction layers, modify different locations, or rely on different repository evidence (Ye et al., 2021; Renzullo et al., 2024). In these cases, training against only one candidate

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

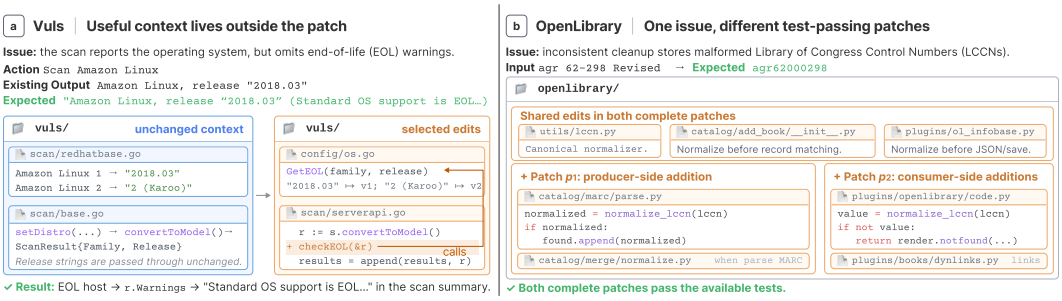


Figure 1: **Why one reference diff is an incomplete retrieval target.** (a) **Vuls:** implementing the end-of-life lookup requires understanding release strings produced and propagated by unchanged code. (b) **OpenLibrary:** the LCCN-normalization issue admits multiple test-passing repairs that satisfy the same requirement. The two audited patches share normalization, import, and persistence edits, but additionally intervene in MARC parsing versus lookup handlers.

solution can only reward the provided edits while penalizing other relevant context that supports different valid solutions. Prior work has shown that models trained on a single reference solution generalize worse than models exposed to multiple semantically equivalent solutions (Li et al., 2026b; Ni et al., 2023; Jain et al., 2021). Figure 1 illustrates both limitations. In the Vuls scanning library, the repair depends on unchanged code that defines its input contract, which is not included in labels generated from a repair patch. In OpenLibrary, the issue can be resolved by multiple possible patches with different edit locations.

Our approach. We introduce CONTEXTFORGE, a training framework with a patch-conditioned information-gain (PCIG) reward over sets of test-validated repairs (Figure 2). For each training issue, CONTEXTFORGE generates alternative patches, executes the available tests, and removes duplicate diffs. The resulting repair set provide repair references for separately scoring proposed edit locations, supporting context, and explanations.

There are two key advantages of our reward. **(1) Score evidence by how much it increases solution likelihood.** Following likelihood-gain rewards for search and memory (Wang et al., 2026a; Han et al., 2026), PCIG measures the change in a validated patch's log likelihood with and without retrieved evidence. This avoids requiring predefined context labels, which can penalize useful evidence simply because it differs from an annotated target. **(2) Score against multiple validated repairs.** A likelihood reward conditioned on only one patch can still favor the context needed for that implementation. We therefore aggregate PCIG over diverse patches that pass the same available tests, broadening the repair locations and implementations that define useful evidence.

Results. We train a Qwen3.5-4B search agent with CONTEXTFORGE using GRPO. We evaluate it across two agent scaffolds and three SWE-bench variants, utilizing GPT-5.4-high as the main coding agent. Across all six benchmark–scaffold settings, CONTEXTFORGE matches or improves the Pass@1 of the no-subagent setting while always achieving the lowest cost among all subagent baselines and reducing end-to-end inference cost by up to 29.1% compared to the no-subagent baselines (Section 4.2). CONTEXTFORGE-trained search agent costs 17.1% less than the next-lowest reported search-agent overhead (Section 4.4). Repeated evaluation on a 150-instance benchmark mixture further confirms that these gains are stable, with comparable Pass@1 to the no-subagent setting while reducing mean end-to-end inference cost by 17.2% (Section 4.3).

2 MOTIVATING EXAMPLES

Figure 1 illustrates two ways in which a reference diff can omit useful retrieval targets.

Useful context beyond edit locations. In Vuls, adding operating-system end-of-life (EOL) warnings requires interpreting release labels, e.g., `2018.03` and `2 (Karoo)`, produced by unchanged distribution-detection code. The 'ScanResult' object stores these labels without modifying them, e.g., `2018.03` for Amazon Linux 1 and `2 (Karoo)` for Amazon Linux 2. A correct repair must

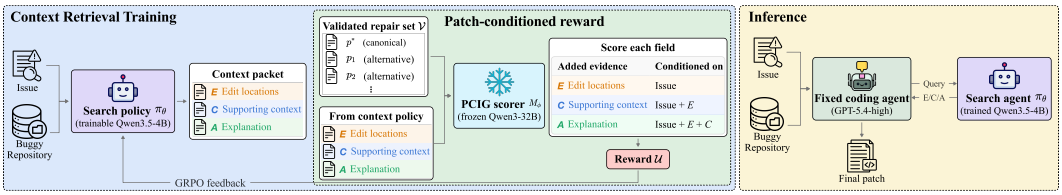


Figure 2: CONTEXTFORGE trains context retrieval against multiple validated repairs. The search policy π_θ samples search trajectories, each returning edit locations (E), supporting context (C), and an explanation (A). A separate PCIG Scorer M_ϕ measures each field's contribution to patch log-likelihood, conditioned on the evidence shown. These gains are combined into reward $\mathcal{U}$ for GRPO training. At test time, the main coding agent calls the trained policy as a context retrieval tool and uses the returned packet to construct and test a patch, requesting context again as needed.

recognize these formats and map them to the corresponding lifecycle versions for the appropriate EOL policy. Reading the detection and propagation code reveals this input contract, yet those unchanged files receive no reference-diff overlap credit.

Multiple repair paths for one issue. The OpenLibrary issue concerns inconsistent normalization of Library of Congress Control Numbers (LCCNs). The two audited patches both modify normalization, import, and persistence code. Their additional interventions differ. Patch p_1 normalizes MARC catalog records before import, while p_2 normalizes book-page and dynamic-link queries so they match stored identifiers. Both complete patches pass the available benchmark tests, although their behavior can differ on untested inputs. Thus, a single reference diff omits additional edit locations used by another test-passing repair.

3 METHOD

We first define the retrieval task (Section 3.1) and the patch-conditioned information gain (PCIG; Section 3.2). We then extend likelihood-gain supervision in two ways: separate rewards for edit locations, supporting context, and explanations (Section 3.3), and aggregation over test-equivalent repairs (Section 3.4). Finally, we describe offline candidate construction (Section 3.5) and policy training and inference (Section 3.6). See Figure 2 for the framework illustration.

3.1 PROBLEM STATEMENT

Given an issue description q and a buggy repository snapshot $\mathcal{R}$, our goal is to train a search agent to retrieve evidence that helps a fixed downstream coding agent resolve the issue. The search policy π_θ receives $(\mathcal{R}, q)$, explores the repository, and returns a context set $y = (E, C, A)$. E and C are proposed edit locations and supporting context, both specify repository-relative file paths and 1-based inclusive line ranges. A is a short explanation connecting the retrieved code to the issue. For a collection of locations D, $\mathrm{code}(D)$ denotes the corresponding repository code rendered from the buggy snapshot $\mathcal{R}$. Training additionally provides a ground-truth repair $p^\star$ and executable tests, from which we construct a set $\mathcal{V}$ of validated repair candidates. These patches and validation records supply supervision.

3.2 PATCH-CONDITIONED INFORMATION GAIN

Let $p^{1:L}$ be a candidate repair serialized as a unified diff with L tokens. For a conditioning block b, the frozen PCIG scorer M_ϕ assigns the length-normalized, teacher-forced log likelihood

$$s_\phi(p^{1:L} \mid q, b) = \frac{1}{L} \sum_{t=1}^{L} \log M_\phi(p^t \mid q, b, p^{1:t-1}). \tag{1}$$

Only the patch tokens contribute to this score. For an additional evidence block h, we define the PCIG using the log-likelihood difference (Wang et al., 2026a; Han et al., 2026):

$$\mathrm{PCIG}_\phi(p; h \mid q, b) = s_\phi(p \mid q, b, h) - s_\phi(p \mid q, b). \tag{2}$$

The two prompts share the issue, baseline code, and scoring instruction while only the evidence block changes. A positive value means that h makes the validated repair more predictable under the scorer.

3.3 Rewarding Edit Locations, Supporting Context, and Explanations

A useful context package (E, C, A) identifies what to edit, supplies supporting code, and explains why the evidence matters. These roles require different supervision: useful evidence may not be an edit target, and retrieving relevant code does not guarantee a useful explanation. Substituting each field's added evidence and baseline into Equation (2) gives

$$\mathrm{PCIG}_E(p) = s_\phi(p \mid q, \mathrm{code}(E)) - s_\phi(p \mid q), \tag{3}$$

$$\mathrm{PCIG}_C(p) = s_\phi(p \mid q, \mathrm{code}(E), \mathrm{code}(C)) - s_\phi(p \mid q, \mathrm{code}(E)), \tag{4}$$

$$\mathrm{PCIG}_A(p) = s_\phi(p \mid q, \mathrm{code}(E \cup C), A) - s_\phi(p \mid q, \mathrm{code}(E \cup C)). \tag{5}$$

$\mathrm{PCIG}_E(p)$ rewards how much code at the proposed edit locations explains the repair, relative to the issue alone. $\mathrm{PCIG}_C(p)$ rewards what supporting code adds beyond the proposed edit code, without requiring those supporting locations to be edited. $\mathrm{PCIG}_A(p)$ rewards what the rationale adds beyond the retrieved code, with duplicate snippets removed from its baseline. All baselines are fixed across repair candidates for a given package, and the issue is included in every scoring prompt.

3.4 Extending the Reward over Test-Equivalent Repairs

One patch identifies only one implementation of a repair. Let $T(p)$ be the vector of test results obtained after applying p to $\mathcal{R}$. Two applicable patches are test-equivalent when

$$p \sim p' \quad \Longleftrightarrow \quad T(p) = T(p'). \tag{6}$$

We construct $\mathcal{V}$ from the groundtruth repair and generated alternatives that apply successfully and pass every required fail-to-pass and pass-to-pass test, removing duplicate normalized diffs. Normalization repairs hunk line counts and ensures a final newline without changing source edits. This is semantic equivalence *with respect to the available test observations*, in the spirit of equivalence modulo inputs (Le et al., 2014). The finite set $\mathcal{V}$ samples feasible repairs rather than enumerating a complete semantic equivalence class.

For each field $k \in \{E, C, A\}$, its candidate-average gain is $g_k = \frac{1}{|\mathcal{V}|} \sum_{p \in \mathcal{V}} \mathrm{PCIG}_k(p)$. Averaging gives each retained repair equal weight and prevents the reward from growing merely because a task has more candidates.

After aggregating over candidates, we divide each gain by the token count of its *added* evidence and apply a sigmoid. This favors shorter evidence when the positive gain is comparable, while also reducing the token cost of the resulting context package (Xu et al., 2024; Jiang et al., 2024). The final training reward U for the produced context package y thus combines all these scores and an assisted format reward:

$$U(y; \mathcal{V}) = \sum_{k \in \{E,C,A\}} \lambda_k \sigma\left(\alpha_k \frac{g_k}{n_k}\right) + R^{\mathrm{fmt}}(y), \tag{7}$$

where $\sigma(z) = (1 + e^{-z})^{-1}$. The token counts are taken over $\mathrm{code}(E)$, $\mathrm{code}(C)$, and the explanation text A, respectively, each with a minimum denominator of one. The weights λ_k and scales α_k control the importance and sigmoid sensitivity for each component. For positive gains, evidence-length normalization favors compact useful context and is additional to the patch-length normalization in Equation (1). The sigmoid bounds each valid component between zero and one before weighting. Missing or invalid fields are excluded and contribute zero; a valid field with zero gain contributes $\lambda_k \sigma(0) = \lambda_k / 2$. R^{fmt} rewards compliance with the structured output schema.

3.5 Generating Diverse Patches and Training Tuples

Candidate construction runs offline on the training split. Figure 3 summarizes three steps:

1. **Generate an alternative.** Given the buggy repository, issue, prompted test lists (Section A.1), one structural strategy, and previously accepted diffs, the coding agent investigates and proposes edits following that strategy.

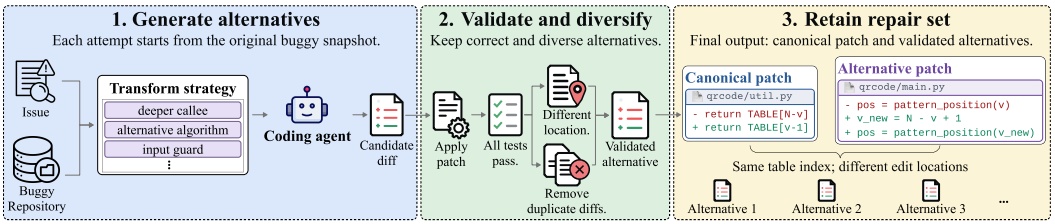


Figure 3: Offline construction of diverse, validated repair targets. Each attempt generates a patch from a fresh buggy repository. Accepted alternatives guide later attempts. Validation and diversity filtering produce a repair set with distinct edit targets. Code is abbreviated.

2. **Validate independently.** Capture the diff and run the test harness in the original task environment. Reject patches that fail any required test, including tests omitted from the prompt.
3. **Filter for diversity and repeat.** Normalize each passing diff, remove duplicates, and retain sufficiently distinct alternatives using the criteria below. Accepted alternatives guide later attempts, each of which again starts from the original buggy snapshot. Stop when the framework reaches the attempt budget.

Structural diversity. The strategy catalog asks the generator to repair a different file, intervene at a caller, intervene at a deeper callee, guard an input boundary, extract a helper, normalize an output boundary, use a different algorithm, or change a default. The attempt index selects strategies cyclically. After accepting an alternative, selection cycles through the first five strategies to encourage new intervention locations. A patch signature $S(p)$ contains its modified $(\text{file}, \text{symbol})$ pairs, recovered from diff hunk headings and newly introduced functions or classes. Module-level edits use a module marker. Every generated alternative must pass the tests and have a minimum Jaccard distance greater than 0.34 from the signatures of the canonical repair and all previously accepted alternatives. An alternative with an identical edit-location signature is rejected regardless of how much its diff text changes. This location-based criterion applies to every generation strategy as textual diversity alone cannot substitute for a different repair location.

Constructing the supervision. Each training record contains the issue and buggy repository, the buggy code, and the retained patches with their edit locations. We derive edit locations from deleted or replaced source lines, using pre-patch hunk anchors for files containing only insertions. We reconstruct the buggy code from unchanged and deleted diff lines, excluding added code. Candidate records retain validation results and generation metadata for auditing. The search policy sees only the issue and repository. Supporting context and explanations are produced by its rollouts and scored as described in Section 3.3, without reference annotations for these fields.

3.6 Policy Training and Inference

Policy training. We optimize π_θ with GRPO using the reward in Equation (7). For each sampled task, we generate a group of trajectories and compute a reward for each completed context package. GRPO compares each trajectory's reward with its rollout group's mean and applies the resulting advantage to policy-generated tokens, including search actions and the final submission. Repository-tool observations are excluded from the loss. The candidate bank and scorer remain fixed, and patches serve only as reward references. We initialize the search policy from Qwen3.5-4B (Qwen Team, 2026) and use Qwen3-32B (Team, 2025) as the PCIG scorer. See Section A.5 for more details.

Interface and inference. The agent searches with a terminal tool and finishes with one `localization_finish` call containing `edit_locations` (E), `supporting_context` (C), and `explanation` (A). The prompt requests compact verified line ranges and a concise rationale within a bounded search budget (Section A.2). At inference, the fixed coding agent invokes the trained search policy through a retrieval tool and uses the returned context to construct and test the final patch (Section 3.6). The search-agent architecture, repository tools, and output budget are held fixed against the single-reference baseline. Candidate construction and reward scoring are

training-only: the inference search agent receives neither reference patches nor the validation oracle, and additional training candidates introduce no test-time patch sampling.

4 EVALUATION

We fix GPT-5.4-high as the main coding agent and vary the searh agent that handles delegated context-retrieval tool calls. Specifically, we evaluate the **(1) effectiveness and efficiency** across three benchmarks and two scaffolds with illustrative case studies (Section 4.2), **(2) stability** across repeated runs (Section 4.3), **(3) retrieval overhead** from search agent inference (Section 4.4), **(4) generalization** to repository question answering (Section 4.5), and **(5) component contributions** through ablations of CONTEXTFORGE design (Section 4.6).

4.1 SETUP

Benchmarks. We evaluate on three repository-level issue-resolution benchmarks: SWE-bench Pro (Deng et al., 2025), SWE-bench Verified (Jimenez et al., 2024), and SWE-bench Multilingual (Yang et al., 2025). We use 200 instances from Pro, 200 from Verified, and the 300-instance Multilingual test split spanning nine programming languages. Together, these benchmarks cover difficult long-horizon issues, curated real-world tasks, and cross-language repository navigation.

Agent scaffolds and main model. We evaluate each context configuration under both Mini-SWE-Agent (Yang et al., 2024) and OpenHands (Wang et al., 2025a). Within every benchmark–scaffold setting, we use `gpt-5.4-2026-03-05` (OpenAI, 2026a)) with `high` reasoning effort and keep all main-agent settings and the execution budget unchanged. The search policy is exposed as a retrieval tool that the main agent can invoke during repair. Each invocation returns a bounded context package.

Search-agent baselines. We compare CONTEXTFORGE against four representative configurations. *w/o subagent* measures the main agent operating without delegated retrieval. *GPT-5.4* uses the same API and `high` reasoning effort as the main agent. *CodeScout-4B* uses a learned repository-search policy (Sutawika et al., 2026), while *CoSIL* uses LLM-guided repository-graph search (Jiang et al., 2025) with a Qwen3.5-4B backend. All search-agent methods receive the same issue and repository and are evaluated under the same output budget where comparable runs are available.

Metrics. Our primary effectiveness metric is Pass@1, the fraction of instances for which the single generated patch passes the benchmark test harness. Our primary efficiency metric is *Cost*, the end-to-end inference cost per evaluated task, defined as the sum of the main-agent and search-agent inference costs. For the no-subagent baseline, *Cost* equals the main-agent cost. We additionally report main-agent tokens, turns, and tool calls, together with the search agent cost when available. See Section A.6 for details on cost calculation.

Comparison protocol. Comparisons are made only within the same benchmark and scaffold. The no-subagent baseline and every search-agent configuration use the same main model and task environment. The search-agent backend and returned evidence vary, and the main agent's subsequent actions can differ. This controlled design measures the downstream utility of retrieval rather than only its overlap with the solution patch. Candidate generation, validation, and candidate-set aggregation for CONTEXTFORGE occur offline during training and add no test-time patch sampling or ensembling.

4.2 MAIN RESULTS

We compare CONTEXTFORGE with no subagent, a GPT-5.4 context subagent, CodeScout (Sutawika et al., 2026), and CoSIL (Jiang et al., 2025). The main-agent model and benchmark split remain fixed within each block. Under both harnesses, CONTEXTFORGE obtains the lowest end-to-end cost on all three benchmarks and the highest Pass@1.

4.2.1 CASE STUDIES

We present two case studies illustrating how CONTEXTFORGE benefits agentic context retrieval.

Table 1: End-to-end results with GPT-5.4-high fixed as the main agent. Cost is the average combined main-agent and context-agent inference cost per task. Parenthesized values report relative Pass@1 and Cost changes versus *w/o subagent* under the same scaffold. The best and second-best subagent results in each column are marked in **bold** and underlined, respectively.

	SWE-bench Pro (200)		SWE-bench Verified (200)		SWE-bench Multilingual (300)	
Subagent	Pass@1 (%)	Cost/task ($)	Pass@1 (%)	Cost/task ($)	Pass@1 (%)	Cost/task ($)
Mini-SWE-Agent Scaffold						
w/o subagent	48.5	0.515	71.0	0.298	71.0	0.379
GPT-5.4	52.0 (+7.2%)	0.591 (+14.8%)	72.0 (+1.4%)	0.304 (+2.0%)	74.3 (+4.6%)	0.417 (+10.0%)
CodeScout-4B	48.5 (+0.0%)	0.552 (+7.2%)	72.5 (+2.1%)	0.272 (-8.7%)	69.7 (-1.8%)	0.375 (-1.1%)
CoSIL	51.0 (+5.2%)	0.576 (+11.8%)	**74.0** (+4.2%)	0.295 (-1.0%)	70.3 (-1.0%)	0.374 (-1.3%)
CONTEXTFORGE	**52.5** (+8.2%)	**0.490** (-4.9%)	**74.0** (+4.2%)	**0.266** (-10.7%)	**74.7** (+5.2%)	**0.361** (-4.7%)
OpenHands Scaffold						
w/o subagent	76.0	1.260	74.0	0.466	58.3	0.714
GPT-5.4	78.5 (+3.3%)	0.999 (-20.7%)	74.0 (+0.0%)	0.514 (+10.3%)	63.7 (+9.3%)	0.644 (-9.8%)
CodeScout-4B	**80.0** (+5.3%)	0.982 (-22.1%)	75.0 (+1.4%)	0.460 (-1.3%)	61.7 (+5.8%)	0.614 (-14.0%)
CoSIL	77.0 (+1.3%)	1.252 (-0.6%)	76.0 (+2.7%)	0.452 (-3.0%)	60.0 (+2.9%)	0.622 (-12.9%)
CONTEXTFORGE	**80.0** (+5.3%)	**0.893** (-29.1%)	**76.5** (+3.4%)	**0.427** (-8.4%)	**64.7** (+11.0%)	**0.593** (-16.9%)

Table 2: (a) Mean ± standard deviation over three runs with GPT-5.4-high as the downstream coding agent on 50 tasks each from SWE-bench Pro, Verified, and Multilingual. (b) Repository question answering. Score is assigned by GPT-5.4 as an LLM judge, and F1 measures answer overlap. In both panels, cost is end-to-end per task. Tokens and turns measure the main agent.

(a) Repeated evaluation on 150 instances

Subagent	Pass@1(%)	Cost/task($)	Main tokens(K)	Turns
w/o subagent	68.0 ± 1.2	0.58 ± 0.012	54.2 ± 1.25	20.1
GPT-5.4	66.3 ± 1.0	0.67 ± 0.019	54.8 ± 2.06	20.4
CodeScout-4B	67.6 ± 1.5	0.57 ± 0.021	55.5 ± 3.20	20.6
CoSIL	67.3 ± 2.0	0.62 ± 0.014	58.6 ± 2.54	20.9
CONTEXTFORGE	68.2± 2.0	**0.48 ± 0.020**	**50.4 ± 2.83**	20.2

(b) Generalization to repository question answering.

Subagent	Score	F1	Cost	Turns	Tokens(K)
w/o subagent	**84.83**	0.342	0.481	13.38	526
GPT-5.4	84.61	0.327	0.432	13.93	450
CodeScout-4B	84.55	0.348	0.405	**13.19**	455
CoSIL	84.44	0.348	0.367	13.43	**421**
CONTEXTFORGE	84.72	**0.350**	**0.346**	13.52	457

Guiding semantic understanding. For `pydata__xarray-4687`, both retrievers identify the relevant computation module. The CoSIL-conditioned main agent follows an existing `.where()` implementation and transfers its attribute-retention setting. With our context identifying attribute-handling helpers, the main agent continues into the merging policy and tests which operand supplies the retained metadata. Its initial patch still copies attributes from the Boolean condition. After observing this behavior, it revises the patch to preserve the data operand's attributes and passes the regression. See Section A.8 for more details.

Covering overlooked implementations. For `django__django-16256`, CodeScout retrieves ordinary related-manager code, while our subagent also identifies the generic related manager in a separate module. The main agent follows these additional locations and extends the async-method repair to generic relations. Both submissions correctly fix the ordinary related managers, but the CodeScout-conditioned patch leaves the generic path unfixed and fails regression tests, while the CONTEXTFORGE-conditioned patch passes all. See Section A.9 for more details.

4.3 REPEATED-RUN STABILITY

To assess the stability of the observed results, we constructed a randomly sampled 150-instance evaluation set consisting of 50 instances from SWE-bench Verified, 50 from SWE-bench Multilingual, and 50 from SWE-bench Pro. As shown in Table 2a, each configuration is evaluated over three runs.

Relative to using no subagent, CONTEXTFORGE improves mean Pass@1 by 0.3% while reducing end-to-end cost by 17.2%. Across repeated runs, the effectiveness difference remains modest. One explanation is that the strong main agent can often compensate for incomplete or incorrect context by

Table 3: Retrieval overhead and reward ablations. (a) Context-agent cost averaged across SWE-bench Pro, Verified, and Multilingual. (b) Ablation cost is end-to-end per task, including main-agent and context-agent inference; Δ is the relative Pass@1 change from Full.

(a) Context-agent cost across 3 benchmarks.

Search agent	Cost ($/task)
GPT-5.4	0.0322
CodeScout-4B	0.0075
CoSIL	0.0035
CONTEXTFORGE	**0.0029**

(b) Ablation on the training reward.

Training configuration	Pass rate	Δ	Cost ($/task)
Full CONTEXTFORGE	**68.22%**	—	**0.48**
w/o Diverse Patches	67.33%	-1.3%	0.48
w/o Supporting Context	66.44%	-2.6%	0.52
w/o Explanation	67.56%	-1.0%	0.50

retrieving missing evidence or rejecting misleading evidence through its own repository exploration. Such recovery can preserve task success, but requires additional inference.

4.4 SEARCH AGENT COST

Delegated retrieval is useful only if the provided context saves more inference than it consumes. Each search tool invocation may involve multi-step repository exploration, so retrieval overhead can erase downstream reductions in main-agent search. Search agent cost thus determines whether improved retrieval translates into lower end-to-end deployment cost. We therefore report the search-agent cost component separately, averaged across SWE-bench Pro, Verified, and Multilingual. As shown in Table 3a, CONTEXTFORGE is 17.1% below the next-lowest search agent cost.

4.5 GENERALIZATION

Real-world repository agents must retrieve useful evidence for tasks beyond issue resolution. A search agent trained only from repair outcomes may overfit to edit-oriented cues and fail when the requests are not relevant to issue resolution. We therefore evaluate whether CONTEXTFORGE's repair-derived supervision transfers to repository question answering. Its diverse repair targets may discourage reliance on a single implementation pattern, while its explicit rewards for supporting context and explanations may promote evidence about program behavior that is useful beyond patch generation. To test this transfer, we evaluate on SWE-QA, where the downstream agent answers repository-level questions using the retrieved evidence. We use GPT-5.4 as the LLM-as-a-judge evaluator and assess generalization along two dimensions: (1) answer quality, measured by judge score and F1, and (2) efficiency, measured by end-to-end cost, turns, and token usage.

As shown in Table 2b, compared to the no-subagent baseline, CONTEXTFORGE improves F1 by 2.3% while reducing end-to-end cost by 28.1% and main token usage by 13.1%. Among all subagent methods, it ranks first on both quality metrics and reduces end-to-end cost by 5.7% relative to the next-lowest-cost configuration.

4.6 ABLATION

We ablate three design components in CONTEXTFORGE: (1) supervision over diverse, test-validated repairs, (2) explicit supporting context, and (3) explanations that connect the retrieved evidence to the issue.

Specifically, we consider the following variants to isolate the contribution of each component: (1) *w/o Diverse Patches* uses only the groundtruth repair as the training reference while retaining the structured context components; (2) *w/o Supporting Context* retains diverse-repair supervision and the remaining output components but omits explicit supporting context; (3) *w/o Explanation* retains diverse-repair supervision and supporting context but omits the explanation component; and (4) *Full* combines diverse-repair supervision with all structured context components.

As shown in Table 3b, each component contributes to the full system. Supporting context yields the largest gain, improving Pass@1 by 2.7% while reducing end-to-end cost by 7.7%. Diverse-patch supervision improves Pass@1 by 1.3% relative to its ablation. Explanations provide a smaller but consistent benefit, improving Pass@1 by 1.0%.

5 Related Work

Repository exploration and context supervision. Context retrieval supports repository-level issue resolution by identifying code relevant to a repair. Existing approaches organize search through hierarchical localization and repository graphs (Xia et al., 2025; Ouyang et al., 2025; Chen et al., 2025; Jiang et al., 2025), or train retrievers and search agents on locations extracted from solution patches (Sutawika et al., 2026; Gangi Reddy et al., 2026; Zhang et al., 2025; Ma et al., 2025). However, a patch's edit locations alone can omit unchanged code needed to understand the repair. Recent context benchmarks evaluate broader evidence, including dependency annotations, successful exploration trajectories, and contexts conditioned on alternative test-passing repairs (Li et al., 2026a; Zhang et al., 2026b). ContextForge connects this broader notion of context to candidate-specific training of a repository-search policy.

Utility-driven retrieval and information gain. Previous work learns retrieval from its contribution to downstream generation, using target likelihood (Shi et al., 2024; Wang et al., 2025b) or quality improvements from additional context (Wu et al., 2024). IGPO and InfoMem extend this principle to search and memory agents by rewarding increases in answer log-likelihood (Wang et al., 2026a; Han et al., 2026). InfoReasoner rewards gains in the probability of a correct semantic answer class (Hu et al., 2026), while SePer extends utility estimation to multiple reference answers (Dai et al., 2025). ContextForge adapts these ideas to repository repair, using patch-conditioned information gain to reward edit locations, supporting code, and explanations by their contribution to patch likelihood.

Multiple references and repair validity. Multiple-reference learning includes both complete-target sampling in text generation and expanding program targets through execution filtering and structural deduplication (Zheng et al., 2018; Ni et al., 2023). Software repair makes the inadequacy of a single reference particularly concrete: distinct implementations can satisfy the same requirement while receiving low patch-similarity scores or being rejected by reference-based assessment (Wei et al., 2025; Ye et al., 2021). ContextForge uses validated alternatives as reward references without requiring the retriever to return the union of their edit locations.

6 Discussion and Conclusion

Test availability. ContextForge constructs repair supervision using executable tests. Its notion of equivalence is therefore limited to the behaviors these tests exercise. Incomplete coverage can admit overfitting patches (Smith et al., 2015; Renzullo et al., 2024), and human-written references can themselves be imperfect behavioral oracles (Ye et al., 2021). Such errors can propagate into the retrieval reward. We omit 50% of test names from the prompt but leave test files accessible (Section A.1); stronger tests and semantic analysis remain needed.

Benchmark familiarity and limited headroom. Latest LLMs may be familiar with SWE-bench repositories, code conventions, and repair patterns through training (Zhang et al., 2026a; Deng et al., 2025). This raises concerns about benchmark-specific overfitting. Prior knowledge or the main agent's own exploration may leave retrieval little room to improve task success, making its benefits more visible in inference cost. Upgrading main agent models on these benchmarks may therefore reduce sensitivity to search agents' quality and limit conclusions about unfamiliar repositories. Evaluating the search agent with latest models calls for fresh datasets of new issues from newly created or access-controlled repositories, and measures to limit exposure to reference solutions.

Conclusion. We introduced ContextForge, a framework for training search agents from sets of validated repairs. Its reward separately values edit locations, supporting code, and explanations, reducing dependence on predefined context labels and a single reference implementation. Across three SWE-bench benchmarks and two scaffolds with GPT-5.4-high fixed as the main agent, the ContextForge-trained 4B search agent matches or improves no-subagent Pass@1 while reducing end-to-end inference cost by up to 29.1%.

Ethics Statement

This work studies repository search and code repair using existing software-engineering benchmarks and model-generated patches. More effective and less expensive coding agents can support software maintenance, but can also introduce incorrect or insecure changes or facilitate harmful software development. Passing the available tests does not guarantee correctness or security, so generated patches require appropriate review and validation before deployment. Use and redistribution of repository code and derived artifacts must respect applicable licenses and protect sensitive information. We discuss limitations of test-based validation and possible benchmark familiarity in Section 6.

Reproducibility Statement

The reward definitions, aggregation over repair candidates, candidate-filtering criteria, and policy-training procedure are described in Section 3. Sections A.1 to A.3 provide the candidate-construction algorithm and the search-agent and patch-generation prompts. Section A.5 details the training and ablation interfaces. The benchmark splits, agent scaffolds, baseline configurations, and evaluation metrics are described in Section 4.1. We provide repeated-run results and search agent cost evaluation in Sections 4.3 and 4.4. Candidate generation is restricted to the training split, and reference patches are not supplied to the search agent at inference. We will release the ContextForge source code at `https://anonymous.4open.science/r/contextforge-7BB7`, including scripts for constructing validated repair sets, training the PCIG search agent, and evaluating it with the OpenHands and Mini-SWE-Agent scaffolds.

AI Use Statement

We used AI to assist with manuscript polishing, related work collection, LaTeX formatting, initial figure drafts, and experiment execution. The research itself also uses language models for patch generation, reward scoring, context retrieval, downstream code editing, and evaluation of repository question answering (Sections A, 3 and 4). The authors take full responsibility for the paper's claims, references, code, and reported results, including material prepared with LLM assistance.

References

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8697–8727, 2025. doi: 10.18653/v1/2025.acl-long.426. URL `https://aclanthology.org/2025.acl-long.426/`.

Lu Dai, Yijie Xu, Jinhui Ye, Hao Liu, and Hui Xiong. SePer: Measure retrieval utility through the lens of semantic perplexity reduction. In *International Conference on Learning Representations*, 2025. URL `https://arxiv.org/abs/2503.01478v5`.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Fireworks AI. Prompt caching, 2026a. URL `https://docs.fireworks.ai/guides/prompt-caching`. Accessed September 24, 2026.

Fireworks AI. Serverless pricing, 2026b. URL `https://docs.fireworks.ai/serverless/pricing`. Accessed September 24, 2026.

Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. SweRank: Software issue localization with code ranking. In *International Conference on Learning Representations*, 2026. URL `https://arxiv.org/abs/2505.07849v2`.

Tiancheng Han, Yong Li, Wuzhou Yu, Qiaosheng Zhang, and Wenqi Shao. InfoMem: Training long-context memory agents with answer-conditioned information gain. *arXiv preprint arXiv:2606.03329*, 2026. URL https://arxiv.org/abs/2606.03329v1.

Senkang Hu, Yong Dai, Yuzhi Zhao, Yihang Tao, Yu Guo, Zhengru Fang, Sam Tak Wu Kwong, and Yuguang Fang. Optimizing agentic reasoning with retrieval via synthetic semantic information gain reward. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2602.00845v3.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 5954–5971, 2021. doi: 10.18653/v1/2021.emnlp-main.482. URL https://aclanthology.org/2021.emnlp-main.482/.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1658–1677, 2024.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. CoSIL: Software issue localization via LLM-driven code repository graph searching. *arXiv preprint arXiv:2503.22424*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices*, 49(6):216–226, 2014.

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. ContextBench: A benchmark for context retrieval in coding agents. *arXiv preprint arXiv:2602.05892*, 2026a. URL https://arxiv.org/abs/2602.05892v3.

Weichen Li, Jiamin Song, Bogdan Alexandru Stoica, Arav Dhoot, Gabriel Ryan, Shengyu Fu, and Kexin Pei. Semrep: Generative code representation learning with code transformations. *arXiv preprint arXiv:2603.13640*, 2026b.

Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. Tool-integrated reinforcement learning for repo deep search. *arXiv preprint arXiv:2508.03012*, 2025. URL https://arxiv.org/abs/2508.03012v2.

Ansong Ni, Jeevana Priya Inala, Chenglong Wang, Oleksandr Polozov, Christopher Meek, Dragomir Radev, and Jianfeng Gao. Learning math reasoning from self-sampled correct and partially-correct solutions. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2205.14318v2.

OpenAI. GPT-5.4 model, 2026a. URL https://developers.openai.com/api/docs/models/gpt-5.4. Accessed September 24, 2026.

OpenAI. API pricing, 2026b. URL https://developers.openai.com/api/docs/pricing. Accessed September 24, 2026.

OpenAI. Counting tokens, 2026c. URL https://developers.openai.com/api/docs/guides/token-counting. Accessed September 24, 2026.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. RepoGraph: Enhancing AI software engineering with repository-level code graph. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2410.14684v2.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Joseph Renzullo, Pemma Reiter, Westley Weimer, and Stephanie Forrest. Automated program repair: Emerging trends pose and expose problems for benchmarks. *arXiv preprint arXiv:2405.05455*, 2024. URL https://arxiv.org/abs/2405.05455v1.

Weijia Shi, Sewon Min, Michihiro Yasunaga, Minjoon Seo, Richard James, Mike Lewis, Luke Zettlemoyer, and Wen-tau Yih. REPLUG: Retrieval-augmented black-box language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 8371–8384, 2024. doi: 10.18653/v1/2024.naacl-long.463. URL https://aclanthology.org/2024.naacl-long.463/.

Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 532–543, 2015. doi: 10.1145/2786805.2786825. URL https://doi.org/10.1145/2786805.2786825.

Lintang Sutawika, Aditya Bharat Soni, Bharath Sriraam R. R, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, and Graham Neubig. CodeScout: An Effective Recipe for Reinforcement Learning of Code Search Agents, March 2026. URL http://arxiv.org/abs/2603.17829. arXiv:2603.17829 [cs].

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Haoye Tian, Li Li, Hongyu Zhang, John Grundy, Jacques Klein, Tegawendé F Bissyandé, and Bach Le. Patch overfitting in program repair: A survey.

Guoqing Wang, Sunhao Dai, Guangze Ye, Zeyu Gan, Wei Yao, Yong Deng, Xiaofeng Wu, and Zhenzhe Ying. Information gain-based policy optimization: A simple and effective approach for multi-turn search agents. In *International Conference on Learning Representations*, 2026a. URL https://arxiv.org/abs/2510.14967v2.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025a.

Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. RLCoder: Reinforcement learning for repository-level code completion. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering*, 2025b. URL https://arxiv.org/abs/2407.19487v1.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Pruner: Self-adaptive context pruning for coding agents. *arXiv preprint arXiv:2601.16746*, 2026b. URL https://arxiv.org/abs/2601.16746v4.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, 2025. URL https://arxiv.org/abs/2502.18449v2.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In *International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2403.10059v2.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE):801–824, 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Fangyuan Xu, Weijia Shi, and Eunsol Choi. Recomp: Improving retrieval-augmented lms with context compression and selective augmentation. In *International Conference on Learning Representations*, volume 2024, pp. 43478–43502, 2024.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.15793.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

He Ye, Matias Martinez, and Martin Monperrus. Automated patch assessment for program repair at scale. *Empirical Software Engineering*, 26(2):20, 2021. doi: 10.1007/s10664-020-09920-w.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

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

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2026a.

Shaoqiu Zhang, Yuhang Wang, Jialiang Liang, Yuling Shi, Wenhao Zeng, Maoquan Wang, Shilin He, Ningyuan Xu, Siyu Ye, Kai Cai, and Xiaodong Gu. SWE-Explore: Benchmarking how coding agents explore repositories. *arXiv preprint arXiv:2606.07297*, 2026b. URL https://arxiv.org/abs/2606.07297v1.

Zhaoxi Zhang, Yitong Duan, Yanzhi Zhang, Yiming Xu, Zhixiang Wang, Kun Liang, Weikang Li, Jiahui Liang, Deguo Xia, Jizhou Huang, Jiyan He, and Yunfang Wu. One tool is enough: Reinforcement learning for repository-level LLM agents. *arXiv preprint arXiv:2512.20957*, 2025. URL https://arxiv.org/abs/2512.20957v6.

Renjie Zheng, Mingbo Ma, and Liang Huang. Multi-reference training with pseudo-references for neural translation and text generation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 3188–3197, 2018. URL https://aclanthology.org/D18-1357/.

A Additional Experimental Details

A.1 Candidate-Set Construction Protocol

This protocol implements the candidate construction in Section 3.5 and produces the validated repair set $\mathcal{V}$ used in Section 3.4. We use the task-level notation $(\mathcal{R}, q, p^\star)$ from Section 3.1; $\mathcal{T}$ denotes the full set of required fail-to-pass and pass-to-pass tests, whose result vector is $T(p)$ in Equation (6). Candidate generation is restricted to the training split. For each issue, an LLM samples alternative patches, after which we discard patches that do not apply or fail the available test oracle. We normalize surviving diffs, remove duplicates, and require each alternative's edit-location signature to have Jaccard distance greater than 0.34 from the canonical repair and every previously accepted alternative, as defined in Section 3.5. Textual changes at an identical edit-location signature do not qualify as a distinct repair target. Tasks with only one surviving patch remain in the training set.

Test exposure and validation. We omit the names of 50% of the provided tests from the explicit requirements in the repair agent's prompt. The remaining tests are listed as fail-to-pass or pass-to-pass requirements. This omission applies only to the prompt: test files remain accessible in the repository, so the agent can discover and run the unlisted tests during exploration or broader test execution. Candidate acceptance evaluates the full test set $\mathcal{T}$, including the unlisted tests, in a separate

validation step. Thus these tests are withheld from the explicit requirements, not hidden from the repair environment; their results do not constitute an evaluation on tests guaranteed to be unseen by the generator.

Candidate-bank and training-tuple construction. **Input:** Training tasks $(\mathcal{R}, q, \mathcal{T}, p^\star)$; repair generator; attempt budget M; alternative cap $K_{\text{alt}} = 3$; temperature and seed schedules.

1. For each task, orient $p^\star$ as a repair and verify that it applies to $\mathcal{R}$ and passes every test in $\mathcal{T}$. Initialize the accepted-alternative list $\mathcal{G} = \emptyset$.
2. Repeat steps 3–5 for attempts $a = 0, \ldots, M-1$, stopping if $|\mathcal{G}| = K_{\text{alt}}$. Select the structural operator using a and the accepted location signatures.
3. Start a fresh $\mathcal{R}$; give the generator q, the prompted subsets of the fail-to-pass and pass-to-pass test lists, the operator, and the accepted diffs/signatures. Run its investigate–edit–test loop.
4. Capture the repair as a unified diff with three context lines. Reject test-file changes, empty diffs, failed application, or failure of any test in the full set $\mathcal{T}$ during independent validation.
5. Require minimum edit-location Jaccard distance > 0.34 from the canonical repair and all accepted alternatives, including for the first alternative. Reject identical edit-location signatures regardless of textual differences. Append each accepted patch to $\mathcal{G}$ and record the diff, operator, attempt index, sampling settings, and test results.
6. Form $\mathcal{V} = \text{Unique}(\{p^\star\} \cup \mathcal{G})$, where Unique removes duplicate normalized diffs. Store $(\mathcal{R}, q, \mathcal{V})$ together with the extracted edit locations, buggy code, and audit metadata described in Section 3.5, including singleton sets.

Output: Fixed task-indexed repair sets $\mathcal{V}$ for the candidate-average PCIG reward in Equation (7).

The generator uses a fixed repair model, a bounded attempt schedule, and parallel task processing. Resumed construction preserves accepted candidates and attempt indices; the retained candidate records identify the sampling settings used by each attempt. Accepted alternatives inform subsequent prompts. Every attempt starts from the original buggy repository rather than applying earlier patches cumulatively.

Connection to policy training. For a context package $y = (E, C, A)$ sampled by π_θ, each $p \in \mathcal{V}$ is a patch-likelihood target for the frozen scorer in Equations (1) and (2). The component gains defined in Section 3.3 are averaged over this fixed set and combined by Equation (7) to supply the GRPO reward (Section 3.6). The policy receives only $(\mathcal{R}, q)$; the patch bank and validation records are training-only supervision, and no reference annotations for C or A are required.

A.2 Context-Agent Training Prompt

The Full and w/o Diverse Patches settings use the same triple-scope prompt pair for training and evaluation; the other ablations remove the corresponding field requirements, as detailed below. The system prompt in Listing 1 requires the context agent to return explanatory repository evidence, precise edit targets, and a retrieval-grounded rationale in one `localization_finish` call. The user template in Listing 2 inserts the repository language, working directory, and issue statement.

```
You are a specialized code-localization agent. Identify the code that
explains an issue's behavior, the precise locations that need modification,
and a short rationale connecting the two. You have up to 4 turns to search
the repository with bash and must finish by calling `localization_finish`.

## Required triple-scope output

Your final tool call must contain all three non-empty fields:

1. `supporting_context`: functions, classes, or module-level regions that
   explain the mechanism, control flow, callers, sibling implementations, or
   configuration relevant to solving the issue. These locations are useful to
   a solver and are not required to be edited.
2. `edit_locations`: exact files and compact 1-based inclusive line ranges expected
   to be modified.
   Include imports, changed expressions, and insertion points, but do not
   include merely related code.
3. `explanation`: a concise (3-5 sentence) rationale explaining why the
   `supporting_context` and `edit_locations` above were chosen and how they
   connect to the issue. This must be grounded strictly in the code you
```

```
   retrieved -- do not invent files, symbols, or behavior that are not present
   in `supporting_context` or `edit_locations`, and do not simply restate the
   issue text back without adding retrieval-grounded reasoning.

For example, when recursive traversal ignores an existing ignore policy,
`supporting_context` can include the recursive caller/traversal and the
existing ignore helper; `edit_locations` contains only the traversal and helper
lines that must change; `explanation` states that the traversal never
consults the ignore helper before recursing, so the fix must call it before
the recursive step.

## Search and submission rules

- Use bash to search and inspect code. Use `cat -n`, `grep -n`, or `rg -n`
  before submitting line numbers.
- Return repository-relative paths without `./`.
- Every entry in `supporting_context` and `edit_locations` requires `file` and a
  non-empty `line_ranges` list.
- Use `line_ranges: [[start, end], ...]`; both endpoints are 1-based and inclusive.
- Verify every boundary with a numbered source read before submitting.
- Behavioral ranges should be the smallest coherent functions or branches needed
  to explain the behavior. Edit-target ranges should tightly bound modifications
  or insertion points.
- Use separate ranges for disjoint regions. A single line is `[[42, 42]]`.
- Do not enumerate adjacent singleton ranges, return whole-file spans, overlap
  duplicate ranges, or add speculative unrelated context.
- Keep behavioral context bounded and non-duplicative. Do not put all edit
  targets into behavioral context merely because they are relevant.
- Keep `explanation` to 3-5 sentences. Do not solve the issue with a patch in
  prose -- state *why* the locations are relevant, not a step-by-step fix.
- The final action must be the `localization_finish` tool call, with
  `supporting_context`, `edit_locations`, and `explanation` all populated together.

Example:

'''json
{
  "supporting_context": [
    {"file": "src/traversal.py", "line_ranges": [[40, 88]]},
    {"file": "src/policy.py", "line_ranges": [[12, 31], [70, 76]]}
  ],
  "edit_locations": [
    {"file": "src/traversal.py", "line_ranges": [[61, 68]]},
    {"file": "src/policy.py", "line_ranges": [[72, 72]]}
  ],
  "explanation": "The recursive traversal in `walk()` (src/traversal.py:40-88) calls itself on every
     child without checking `policy.is_ignored()` (src/policy.py:12-31), so files matched by the ignore
      policy are still visited. The fix must add an `is_ignored()` check before the recursive call at
     line 61-68 and ensure the policy lookup at line 72 uses the resolved path rather than the raw
     entry name."
}
'''
```

Listing 1: System prompt used to train the search agent.

```
I have access to a <language> code repository in the directory <working_dir>.
Consider the following issue description:

<issue_description>
<problem_statement>
</issue_description>

Search for:

1. `supporting_context`: the bounded code scope that explains the behavior
   causing this issue and would help another agent reason about a fix.
2. `edit_locations`: the exact files and inclusive line ranges that should be modified.
3. `explanation`: a concise (3-5 sentence) rationale explaining why the
   `supporting_context` and `edit_locations` above were chosen and how they
   connect to the issue.

Submit all three in one `localization_finish` call. Use paths relative to
<working_dir>. Every location entry must use
`{"file": "...", "line_ranges": [[start, end], ...]}`. Verify every range boundary
with a numbered file read. Keep ranges tight and coherent; use separate ranges for
disjoint regions and `[[line, line]]` for a single-line target. Ground the
explanation strictly in the code you retrieved -- do not describe files,
symbols, or behavior that are not present in `supporting_context` or
`edit_locations`.
```

Listing 2: Per-instance user prompt used to train the search agent. Angle-bracketed fields are populated from the training example.

A.3 PATCH-VARIANT GENERATION PROMPT

The new generator runs a repair agent in the task container. Its system prompt is reproduced in Listing 3, and its dynamically assembled user message is shown in Listing 4. Only the tests selected as explicit prompt requirements in Section A.1 populate its fail-to-pass and pass-to-pass lists; the independent validator uses the full test set. The prior-diff and used-location blocks are omitted on the first attempt. On later attempts, the message includes previously accepted diffs and their (file, symbol) signatures. It also injects one of eight structural strategies: fix in a different file, fix at the caller, fix in a deeper callee, guard at the input boundary, extract a helper, normalize at the output boundary, use an alternative algorithm, or change an appropriate default or configuration value. Every strategy is subject to the edit-location diversity requirement in Section 3.5.

```
You are an autonomous Python repair agent operating inside a fresh
Docker container that has been seeded with a buggy version of a Python
project at `/testbed`. You will be given a problem description and a
list of tests that currently fail (and a list that must continue to
pass). Your job is to **produce a patch that makes ALL the listed
tests pass**, then signal completion.

## Tools available

* `docker_terminal` -- run shell commands inside the container. Use
  this for `pytest`, `git`, `python`, `ls`, `cat`, `head`, `grep`,
  `find`, etc. Each call is a fresh shell; chain with `&&` if needed.
* `file_editor` -- view, create, and edit files in the container. Use
  this for surgical edits. **Always prefer `str_replace_editor` over
  generating raw diffs.**

## Process

1. **Orient.** Skim the relevant package: `ls /testbed`, then `cat`
   the modules named in the problem statement.
2. **Reproduce.** Run the failing tests verbatim. Read the failure
   carefully -- do NOT guess at the fix.
3. **Diagnose.** Inspect the code paths exercised by the failing
   tests. Look at the function under test and its callers.
4. **Edit.** Make the minimal change that addresses the root cause.
   Be careful: the buggy state may include subtle off-by-one or
   inverted-key bugs.
5. **Verify.** Re-run the full FAIL_TO_PASS and PASS_TO_PASS test
   lists. Both must pass. If a previously-passing test breaks, your
   edit is wrong -- revert and re-think.
6. **Signal completion.** When (and only when) **every** test in both
   lists passes, send a final assistant message that contains the
   literal sentinel `<<<DONE>>>` on its own line. The orchestrator
   captures `git diff` immediately after that signal.

## Diversity mandate

You may be asked to produce **alternative correct fixes** for the same
problem. If the user message lists prior attempts, your new patch
**must differ structurally** from them -- different file scope,
different control flow, different helper functions, different
variable names. A patch that's identical up to whitespace will be
rejected. *Same observable behavior, different surface form.*

## Constraints

* Do NOT add, delete, or rename test files. Tests are ground truth.
* Do NOT modify `pyproject.toml`, `setup.py`, or `requirements*.txt`.
* Do NOT install new packages; the container has everything it needs.
* If a test imports something that doesn't exist, the bug is missing
  code -- implement it. Do not stub out the test.
* Do NOT use `pdb`, breakpoints, or interactive REPLs.
* Single-line agents that immediately emit `<<<DONE>>>` without
  running tests are FAILURES. You must verify.

## Failure modes to avoid

* "I think this fixes it" -- without running the tests. Always verify.
* Editing the test file. Tests are immutable.
* Patches > 200 lines for a typical bug. If your fix sprawls, re-read
  the problem statement; the bug is almost certainly local.
* Silent exception swallowing (`try: ... except: pass`). Never paper
  over the bug.

## Output discipline

Use the tools, don't narrate. Keep assistant messages short. The only
required output is the final `<<<DONE>>>` sentinel after verification.
```

Listing 3: System prompt used by the structural patch-variant generator.

```
# Instance: <instance_id>
# Attempt: <attempt_index>
```

```
## Problem statement
<problem_statement>

## Tests that must START passing (FAIL_TO_PASS)
- <fail_to_pass test>

## Tests that must KEEP passing (PASS_TO_PASS)
- <pass_to_pass test>

## Strategy for THIS attempt -- <operator_title>
<operator_instruction>

## Prior attempts on THIS task -- your fix must differ structurally

### Prior attempt <i> diff
```diff
<prior_diff>
```

Edit locations ALREADY used by accepted variants -- avoid these

A fix with the same set of (file, function/class) edit locations as an
accepted variant will be REJECTED, regardless of textual differences.
Choose different edit locations; the validator requires edit-location
Jaccard distance greater than 0.34 from the canonical repair and every
accepted variant:

- `<file>` :: `<symbol>`

What to do now
Investigate, edit, run the full FAIL_TO_PASS + PASS_TO_PASS lists above,
and only when **every** one passes emit `<<<DONE>>>` on its own line.
The orchestrator captures `git diff` immediately after.
```

Listing 4: User-message template for one patch-generation attempt. Repeated test entries, prior diffs, and used locations are expanded as needed; optional sections are absent when empty.

A.4 Detailed Example: Equivalent Repairs at Different Locations

We illustrate the distinction between superficial diff variation and structural repair diversity using the SWE-Smith task `lincolnloop__python-qrcode.456b01d4.func_basic__a42xwk2d`. The injected bug causes QR-code generation to raise an `IndexError` for every version. Let N be the length of `PATTERN_POSITION_TABLE` and v the one-indexed QR-code version. The buggy helper indexes the table at $N - v$, whereas the required entry is at $v - 1$.

```
diff --git a/qrcode/util.py b/qrcode/util.py
index 02fe11d..e737298 100644
--- a/qrcode/util.py
+++ b/qrcode/util.py
@@ -136,7 +136,7 @@ def BCH_digit(data):


 def pattern_position(version):
-    return PATTERN_POSITION_TABLE[len(
-     PATTERN_POSITION_TABLE) - version]
+    return PATTERN_POSITION_TABLE[version -
-     1]


 def mask_func(pattern):
```

Listing 5: Groundtruth patch: repair the shared table lookup.

```
diff --git a/qrcode/main.py b/qrcode/main.py
index 46116b5..79a1cda 100644
--- a/qrcode/main.py
+++ b/qrcode/main.py
@@ -406,7 +406,8 @@ class QRCode(Generic[
-     GenericImage]):
                 self.modules[6][c] = c % 2 == 0

     def setup_position_adjust_pattern(self):
-        pos = util.pattern_position(self.
-     version)
+        table_size = len(util.
-     PATTERN_POSITION_TABLE)
+        pos = util.pattern_position(
-     table_size - self.version + 1)

         for i in range(len(pos)):
             row = pos[i]
```

Listing 6: Our generated patch: compensate at the caller.

The groundtruth repair in Listing 5 directly replaces the erroneous expression in `pattern_position`, changing the lookup from $N - v$ to $v - 1$. This is the most local repair and changes the semantics of the shared utility for every caller. Our generated repair in Listing 6 instead leaves that helper unchanged. Before calling it, `setup_position_adjust_pattern` maps v to $v' = N - v + 1$. The unchanged helper then indexes $N - v' = N - (N - v + 1) = v - 1$, so both patches select the same table entry for every valid version.

Table 4: Groundtruth and generated repairs for the same QR-code task.

Property	Groundtruth patch	Our structural variant
Edit location	`qrcode/util.py`, `pattern_position`	`qrcode/main.py`, `QRCode.setup_position_adjust_pattern`
Intervention	Correct the table index inside the shared lookup helper.	Transform the version at the caller while retaining the existing helper.
Index presented to the table	$v - 1$	$N - (N - v + 1) = v - 1$
Generation strategy	Historical groundtruth repair	Caller-side structural operator
Validation	Benchmark reference repair	Full test oracle passed; strict audit passed

The two patches are therefore equivalent on the required behavior but imply different localization targets. The canonical target is a utility function in `qrcode/util.py`; the generated target is its caller in `qrcode/main.py`. Their file-level location signatures are disjoint, giving location Jaccard distance 1.000. Their normalized character-level Levenshtein distance is 0.601, and their token Jaccard distance is 0.804. The generated patch was accepted by the *caller-side* structural operator, passed the full fail-to-pass and pass-to-pass test oracle, and passed the strict audit. This example shows why multiple valid patches provide richer supervision: a search agent trained only on the groundtruth patch is rewarded for finding the lookup helper, while candidate-set training can also reward finding the caller where the same correction can be expressed.

A.5 Training Details

Table 5 specifies the requested fields, active PCIG rewards, repair references, and main-agent inputs for each ablation. Full ContextForge requests `edit_locations` (E), `supporting_context` (C), and `explanation` (A), and uses all three PCIG components averaged over diverse validated repairs. Each location contains a repository-relative `file` and inclusive, 1-based `line_ranges`. The explanation is a retrieval-grounded rationale of three to five sentences.

The w/o Supporting Context setting requests only E and A and retains PCIG_E and PCIG_A over the diverse repair set. It removes both the supporting-context field and its reward; the explanation is scored relative to $\text{code}(E)$, with $C = \emptyset$. The w/o Diverse Patches setting retains all three fields and PCIG components but uses only the groundtruth repair as its reference. The w/o Explanation setting requests only E and C and retains PCIG_E and PCIG_C over the diverse repair set, removing both the explanation field and its reward. All settings retain the format reward, which checks only the fields required by that setting.

The main-agent tool response renders the retained location fields as scope labels, file paths, line ranges, and source snippets when available, and appends the rationale when A is included. Thus Full and w/o Diverse Patches supply $E + C + A$, w/o Supporting Context supplies $E + A$, and w/o Explanation supplies $E + C$. Removed fields are omitted from the main-agent input as well as the training objective; failed calls and output limits can prevent complete delivery of the retained fields.

A.6 Evaluation Cost Accounting

We sum inference costs over all main-agent and context-agent calls before averaging over evaluated tasks. For GPT-5.4, we estimate inference costs using OpenAI's public API rates and reported tokens, which include reasoning tokens (OpenAI, 2026c). The standard-tier short-context rates are \$2.50/1M uncached input, \$0.25/1M cached input, and \$15.00/1M output tokens, with \$5.00, \$0.50, and \$22.50/1M tokens for prompts exceeding 272K tokens (OpenAI, 2026b;a). We serve 4B models for all evaluated agents on Fireworks and estimate their inference costs using its publicly documented 4B–16B serverless rate of \$0.20/1M uncached input and output tokens with documented 50% discount for cached inputs (Fireworks AI, 2026b;a).

Table 5: Prompt requirements, active PCIG rewards and repair references, and main-agent inputs for the settings in Table 3b. The evaluated checkpoints use $(\lambda_E, \lambda_C, \lambda_A) = (1, 1, 0.25)$, with the omitted component's weight set to zero in each component ablation; all active components use sigmoid scale $\alpha_k = 50$. All settings also retain the format reward. Main-agent fields denote observed input types across each evaluation, subject to failed calls and output truncation.

Training configuration	Prompt fields	PCIG rewards and repair references	Main-agent fields
Full CONTEXTFORGE	$E + C + A$	PCIG_E, PCIG_C, PCIG_A, diverse reference patches	$E + C + A$
w/o Supporting Context	$E + A$	PCIG_E, PCIG_A, diverse reference patches	$E + A$
w/o Diverse Patches	$E + C + A$	PCIG_E, PCIG_C, PCIG_A, groundtruth repair only	$E + C + A$
w/o Explanation	$E + C$	PCIG_E, PCIG_C, diverse reference patches	$E + C$

E: edit targets; C: supporting context; A: explanation.

A.7 TRAINING EXAMPLE WITH PCIG REWARDS

We illustrate the three PCIG components with an isodate task involving malformed ISO 8601 duration strings. The formatting code swaps year/month and hour/minute labels and mishandles format selection and fractional seconds. Table 6 reports all eight training rollouts scored under Equation (7). The detailed trace is rollout 7, the lower of the two middle rollouts ranked by PCIG utility. This example uses the same reward weights and sigmoid scales as the evaluated Full CONTEXTFORGE (Table 5).

For each field $k \in \{E, C, A\}$, we first average its PCIG over the four validated repair references to obtain $g_k = \frac{1}{4}\sum_{p\in\mathcal{V}} \text{PCIG}_k(p)$, using the added evidence and baselines in Section 3.3. The implementation already normalizes each gain by n_k, the token count of that field's added evidence with a minimum denominator of one. We denote these stored, normalized gains by $\widetilde{g}_k = g_k/n_k$, distinguishing them from the unnormalized candidate-average gains g_k in Section 3.4. All three fields are valid in these rollouts, so $\mathcal{K}(y) = \{E, C, A\}$. With sigmoid scales $\alpha_E = \alpha_C = \alpha_A = 50$ and weights $(\lambda_E, \lambda_C, \lambda_A) = (1, 1, 0.25)$, each field's score is $S_k = \sigma(50\widetilde{g}_k) = \sigma(50 g_k/n_k)$, and Equation (7) becomes

$$U = S_E + S_C + 0.25 S_A + R^{\text{fmt}}. \tag{8}$$

The stored gains therefore enter the sigmoid directly; the division by n_k is not applied a second time. All eight rollouts submit the required fields and receive format reward $R^{\text{fmt}} = 0.1$. For rollout 7, the explanation contributes $0.25 \times 0.528613 \simeq 0.132153$, and the total PCIG utility is 2.109974, computed before rounding the components.

Table 6: PCIG scores for all eight isodate training rollouts, using the same reward weights and sigmoid scales as the evaluated Full CONTEXTFORGE checkpoint. Each field applies $\sigma(50\widetilde{g}_k)$, where the stored gain $\widetilde{g}_k = g_k/n_k$ already includes evidence-length normalization as in Equation (7). Explanation PCIG has weight 0.25; the other PCIG components have unit weights. Totals use unrounded scores. The detailed example is rollout 7, selected by lower-median PCIG utility.

Rollout	Edit PCIG	Supporting PCIG	Explanation PCIG	Format	Total
0	0.999930	0.917940	0.681519	0.100000	2.188250
1	0.999930	0.888815	0.599750	0.100000	2.138682
2	0.892244	0.869374	0.534525	0.100000	1.995249
3	0.999930	0.917940	0.614221	0.100000	2.171425
4	0.540850	0.648500	0.488185	0.100000	1.411396
5	0.999930	0.942205	0.606926	0.100000	2.193866
6	0.999930	0.879729	0.454635	0.100000	2.093318
7	0.999930	0.877891	0.528613	0.100000	2.109974

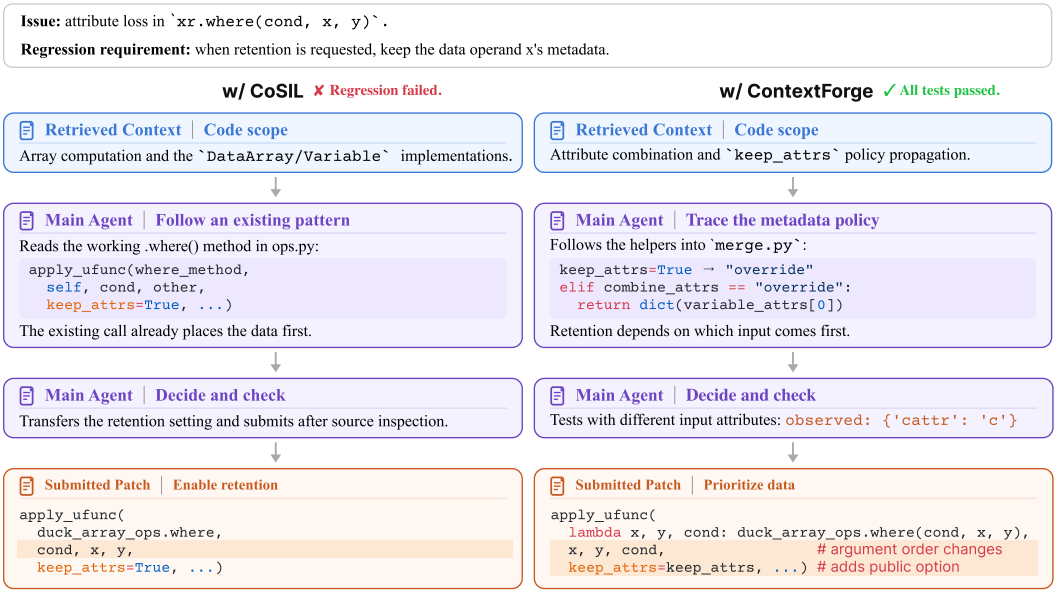


Figure 4: For the same issue, CoSIL retrieves relevant computation code, misleading the main agent to reuse a retention setting and fails the regression. Our subagent identifies attribute-handling helpers, guiding the main agent to test metadata behavior and correct operand order.

Rollout 7 selects `src/isodate/isostrf.py:105-128`. Its supporting context includes the formatting helper and the `duration_isoformat` wrapper that calls it. Its edit range captures the central unit-label and fractional-second logic and the explanation highlights the swapped labels.

The four terminal calls appear in Listing 7; Listing 8 preserves the returned locations and explanation.

```
Terminal 1:
grep -rn "duration_isoformat" <repo>/ --include="*.py"
Terminal 2:
cat -n <repo>/src/isodate/isoduration.py
Terminal 3:
grep -rn "strftime" <repo>/src/isodate/isostrf.py 2>/dev/null || grep -rn "strftime\|
    D_DEFAULT" <repo>/ --include="*.py" | head -100
Terminal 4:
cat -n <repo>/src/isodate/isostrf.py
```

Listing 7: Terminal calls from rollout 7, in order. `<repo>` replaces the temporary repository root; tool outputs are omitted.

```
supporting_context:
  src/isodate/isostrf.py:93-134
  src/isodate/isoduration.py:130-148

edit_locations:
  src/isodate/isostrf.py:105-128

explanation:
  The bug is in the 'repl' function (line 99-132) of '_strfduration' which processes
  '%p' and '%P' format codes. Lines 106-109 incorrectly swap year/month labels (years
  are labelled as M, months as Y), and lines 111-128 have incorrect unit assignments
  (hours/minutes are computed in reverse order). The fix must correct the year/month
  label assignments and fix the sequential calculation of hours/minutes/seconds/days
  to produce proper ISO 8601 duration output.
```

Listing 8: Final locations and verbatim explanation from rollout 7. File ranges are abbreviated, schema metadata and null fields are omitted, and the explanation is rewrapped.

A.8 MOTIVATING EXAMPLE: CONTEXT THAT GUIDES PATCH CONSTRUCTION

Task and comparison. The issue `pydata__xarray-4687` concerns attribute loss in `xr.where(cond, x, y)`. Selecting the correct output values does not determine whose metadata should be retained: when retention is requested, the regression requires attributes from the data operand `x`. We inspect one paired historical Verified100 evaluation under Mini-SWE-Agent, using GPT-5.4 with high reasoning effort as the main model. Both conditions use identical main-agent prompts and limits, search the same Python repository, return relevant context, and submit patches that apply.

From returned context to main-agent behavior. CoSIL returns context covering the computation module and array/variable implementations. Our returned context identifies attribute combination, retention-policy normalization, and propagation through the computation helpers. Figure 4 visualizes the two observed trajectories; Table 7 details what the main agent subsequently investigates and how it constructs the patch.

Table 7: Retrieved context, semantic checks, and patch outcomes for the xarray case.

Stage	CoSIL condition	Our historical checkpoint
Retrieval	Computation and array/variable implementations.	Attribute combination and retention-policy propagation.
Investigation	Reads attribute helpers and the existing method implementation; treats its retention setting as a pattern for the top-level function.	Reads attribute helpers, follows the merging policy into `merge_attrs`, and investigates which input supplies metadata.
Initial edit	Adds `keep_attrs=True` to the existing helper call.	Exposes and forwards `keep_attrs`; this first edit still retains the condition's attributes.
Behavioral check	Submits after source inspection without executing a successful operand-metadata probe.	Tests distinct attributes on the condition and both data inputs; observes the wrong metadata source and revises the patch.
Final patch	Leaves the public retention option unsupported and the original condition-first helper call in place.	Reorders helper inputs to give the data operand precedence, using a wrapper to preserve the value computation.
Graded outcome	Regression fails; all 1,717 preservation tests pass.	Regression and all 1,717 preservation tests pass.

The assumption behind the existing pattern. The working method form passes `(self, cond, other)` to its helper, placing the data first. The top-level function instead passes `(cond, x, y)`, placing the Boolean condition first. Attribute retention follows the chain `True` → `override` → `dict(variable_attrs[0])`. Transferring the retention setting therefore requires checking operand order. In our condition, the main agent opens `xarray/core/merge.py`, verifies that policy, and later probes the behavior of its initial edit. The probe returns `{'cattr': 'c'}`, the condition's metadata. After revising the helper argument order to `(x, y, cond)` and wrapping the value operation to retain `(cond, x, y)` semantics, it obtains `{'xattr': 'x'}`, the data operand's metadata. This sequence demonstrates the additional semantic check behind the successful patch.

Interpretation and evidence scope. The example motivates returning supporting context that can encourage a main agent to follow a behavioral dependency, test its assumptions, and refine a patch. The explicit diagnosis and the additional read of `merge.py` are the main agent's work; they were not supplied as a subagent explanation. CoSIL's main agent also obtains the attribute-handling code during its own search. Thus the traces show different uses of available evidence. Both patch-generation environments initially lack dependencies; our main agent installs compatible versions and runs behavioral probes, while the CoSIL-conditioned agent proceeds by source inspection. Their

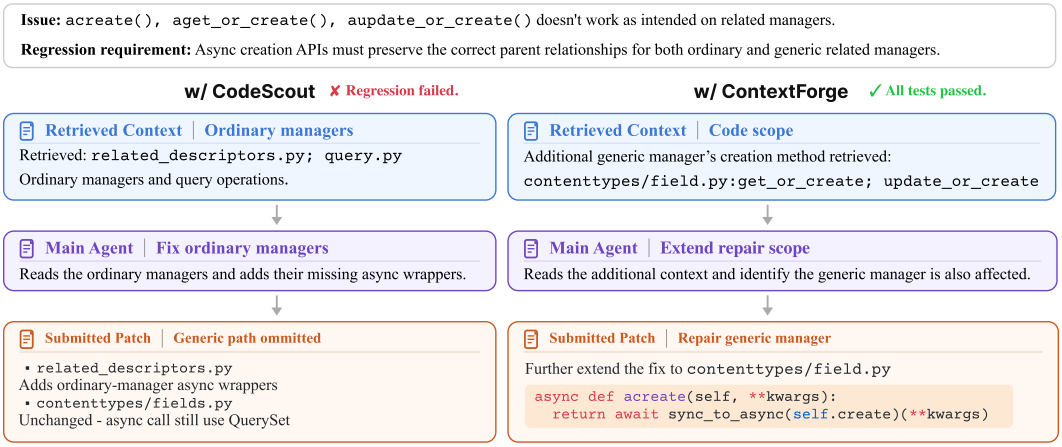


Figure 5: CodeScout returns only ordinary related-manager code, resulting in its main agent leaving the generic manager unfixed. Our subagent locates the additional generic manager, leading the main agent to inspect and repair all affected code.

main-agent API-call counts are 26 and 9, respectively, so this is a correctness example with different verification effort.

The CoSIL regression first fails on the missing public `keep_attrs` keyword, before checking metadata. The wrong-metadata observation above comes from our intermediate patch, which already supports that keyword. Both full test logs contain the same additional Pint/Dask failure outside the graded preservation set.

A.9 CodeScout Case Study: Django Async Related Managers

Task. The SWE-bench Verified issue `django__django-16256` concerns `acreate()`, `aget_or_create()`, and `aupdate_or_create()` on related managers. These inherited async methods invoke `QuerySet` operations, bypassing the managers' own synchronous methods that populate relationship fields. The issue provides example fixes for ordinary related managers in `django/db/models/fields/related_descriptors.py`. The generic related manager in `django/contrib/contenttypes/fields.py` also requires a repair.

Table 8: Retrieved context, repair scope, and graded outcomes for the Django case.

Stage	CodeScout condition	Our historical checkpoint
Retrieval	Ordinary related-manager and `QuerySet` modules.	Also identifies generic-manager creation methods in `contenttypes/fields.py`.
Investigation	Inspects the ordinary related managers.	Follows the additional locations and extends the repair to generic relations.
Shared edits	Adds async wrappers to ordinary related managers.	Makes identical edits to ordinary related managers.
Generic edits	None; inherited `QuerySet` methods remain.	Adds async wrappers and makes content-type lookup lazy.
Verification	Tests ordinary-manager paths.	Reproduces a generic initialization error, fixes it, and verifies all three async creation APIs.
Regression	6/9 pass; all three generic async tests fail.	9/9 pass, including all three generic async tests.
Preservation	53/53 pass.	53/53 pass.

Returned context and repair scope. CodeScout returns locations in the ordinary related-manager module and the `QuerySet` implementation. Our subagent also identifies the generic manager's `get_or_create` and `update_or_create` methods. The main agent follows these additional locations, reads the surrounding code, and explicitly considers extending the repair: "GenericRelation has its own related manager implementation, and since it's a related manager too, it may need the same async wrappers." The retrieved package supplies code locations; this diagnosis and the subsequent edits are the main agent's work. Figure 5 shows the resulting expansion of repair scope; Table 8 compares the two conditions.

Observed patch construction. Both submissions make identical changes to ordinary related managers, adding async wrappers around their synchronous creation methods. With our context, the main agent also adds these wrappers to the generic manager, preserving its injection of content-type and object-ID fields. Its own reproducer then reveals that synchronous content-type lookup during manager initialization raises `SynchronousOnlyOperation` in an async context. It makes this lookup lazy and verifies all three creation APIs. CodeScout's main agent fixes the ordinary managers but does not read or edit the generic-manager implementation, although its broader search output includes existing generic-relations tests.

Graded outcomes. The CodeScout-conditioned patch passes the six ordinary-manager regression tests but fails the generic-manager tests for all three async creation APIs. Each fails with an `IntegrityError` because `content_type_id` is missing, indicating that creation bypasses the generic manager's relationship-field handling. The patch produced with our context passes all nine regression tests. Both patches pass all 53 graded preservation tests, yielding unresolved and resolved outcomes, respectively. The generic-manager failures in CodeScout's evaluation are distinct from the initialization error found during our main agent's own verification.