SMALL AGENTS WITH SEMANTIC SEARCH: EFFICIENT MULTILINGUAL CODE LOCALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Locating relevant files from natural-language requests is a core subtask for agents operating over code repositories. We investigate whether this task can be delegated to compact, specialized models to enable on-device search while reducing the token usage, latency, and inference cost of larger agents. We show that semantic search improves file localization, with gains in accuracy, cross-language transfer, and inference efficiency. To study this setting, we introduce a training framework for file-localization agents built around `colgrep`, a local semantic search tool based on late-interaction retrieval models. Our recipe combines weighted supervised fine-tuning on teacher trajectories, assigning turn-level credit based on retrieval outcomes, with reinforcement learning on localization quality. We train three model families with fewer than two billion parameters to formulate search queries, inspect retrieved content, and identify relevant files. On localization tasks derived from SWE-bench Lite and Multi-SWE-bench Flash, `colgrep`-equipped agents substantially improve over their base models and outperform corresponding `grep`-based agents. In addition to improving localization accuracy, `colgrep` reduces mean end-to-end trajectory latency by 44.1% on CPU while using 29.1% fewer tokens, and enables better generalization to programming languages unseen during fine-tuning. These results suggest that compact, tool-specialized localization agents can provide an efficient interface between natural-language requests and large codebases. We will release model checkpoints, software, data, and training pipelines upon acceptance.

1 INTRODUCTION

Coding agents must first locate where to make a change in a repository before producing the change itself. Finding those files may require repeated directory listings, lexical searches, and file reads, making localization a substantial part of the overall repair process. CodeGrep (Chen et al., 2026), for example, reports 23 interaction rounds and 631K tokens per resolved SWE-bench Verified issue for a 30B OpenHands agent (Jimenez et al., 2024). A natural way to make this process more efficient is to use a small, specialized agent dedicated to the localization step. This agent can return candidate files, allowing a larger and more expensive model to begin from a focused context rather than searching the repository from scratch. Our paper isolates file localization and measures localization accuracy, trajectory length, and latency.

This separation is also consistent with prior work showing that repository-level repair can be decomposed into code localization and editing (Xia et al., 2024; Xie et al., 2025). Treating localization as a standalone learned decision problem, in turn, lets us compare two interfaces available to compact agents: lexical tools (Sutawika et al., 2026) and semantic retrievers (Vijay et al., 2025; Reddy et al., 2025). These interfaces place different demands on the policy. A lexical tool such as `grep` requires a string or regular expression, whereas semantic search accepts a natural-language description. The latter may reduce the need to infer repository-specific identifiers (Mahmud et al., 2024). However, lexical retrieval remains effective with capable policies and suitable access to results (Hsu et al., 2026), and task-specific training may further narrow the difference. We therefore ask: *How do search interfaces and training signals shape file-localization policies below two billion parameters?*

To study this question, we introduce a two-stage training framework for compact localization policies centered on `colgrep`, a local search tool based on late-interaction retrieval (Sourty, 2026;

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.

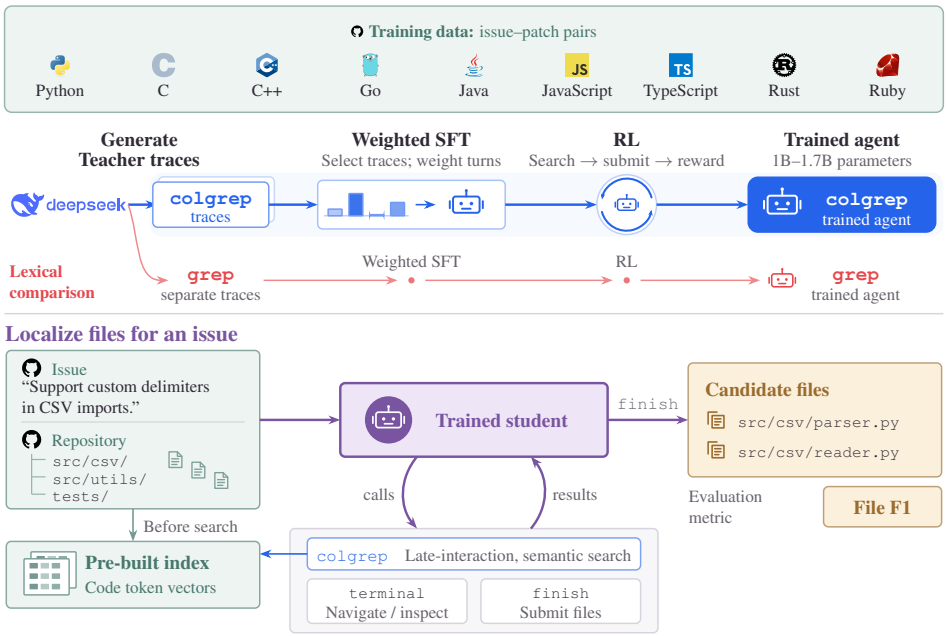

Figure 1: **Training compact agents for file localization.** Given an issue and repository, the policy searches indexed code with `colgrep`, inspects candidates in a read-only terminal, and submits file paths with `finish`. The `grep` branch represents separately generated demonstrations and lexical-only policies.

Khattab and Zaharia, 2020). A `colgrep` call combines a natural-language query with an optional regular-expression filter, allowing the policy to pair a behavioral description with an exact identifier. Supervised fine-tuning (SFT) teaches the policy to formulate queries, inspect retrieved code, and produce file submissions, with assistant turns weighted by their retrieval outcomes. Reinforcement learning (RL) then directly optimizes the quality of the final submission through repository interaction. We apply this framework to three model families with 1–1.7B parameters and evaluate file localization on SWE-bench Lite and Multi-SWE-bench Flash (Jimenez et al., 2024; Zan et al., 2026).

Our contributions are: **(i)** a systematic comparison of semantic and lexical search for compact file-localization agents, demonstrating consistent gains from semantic search across training stages on SWE-bench Lite and Multi-SWE-bench Flash; **(ii)** a cross-language generalization study showing that semantic search enables substantially stronger transfer than lexical search to seven programming languages unseen during fine-tuning; **(iii)** an efficiency analysis demonstrating that, after training, semantic search improves CPU/GPU latency and token usage alongside localization accuracy ; and **(iv)** a training recipe for small, efficient semantic-search agents, including a weighted SFT objective that improves file F1 across all three student models and both benchmarks at unchanged fine-tuning compute.

2 Related Work

Code Localization. Separating localization from editing is well established in repository-level issue resolution. Agentless decomposes the workflow into localization, repair, and validation (Xia et al., 2024), while SWE-Fixer trains separate file-retrieval and editing modules (Xie et al., 2025). Beyond this decomposition, interactive methods expose additional repository structure or support repeated retrieval. Building on graph-based localization, LocAgent uses graph-guided navigation (Chen et al., 2025), while OrcaLoca combines code-graph search with prioritized actions (Yu et al., 2025). More recently, SweRank+ complements these approaches with multilingual ranking and multi-turn localization (Reddy et al., 2025). In parallel, learning dedicated search policies has emerged as an active direction: ToolTrain combines rejection-sampled SFT with tool-integrated RL (Ma et al., 2025); CodeScout (Sutawika et al., 2026) trains code-search agents using a standard

Unix terminal; and CodeGrep (Chen et al., 2026) learns multi-turn lexical retrieval for a downstream coding agent. Together, these lines of work motivate our focus on the interaction between compact policies, semantic access, and supervision.

Search Interfaces and Agent Training. Agent performance depends on the tools and observations exposed by the harness, as studied by SWE-agent (Yang et al., 2024). Lexical tools provide exact matching and familiar navigation, while semantic tools enable retrieval from descriptions that need not share the target's identifiers. `colgrep` implements such semantic retrieval using late interaction in the ColBERT family (Khattab and Zaharia, 2020), with an indexing backend based on PLAID (Santhanam et al., 2022). We evaluate semantic-equipped policies against terminal-only policies, treating the retriever and its tool feedback as part of the intervention. For training, teacher distillation supplies executable examples of search and submission. Whole-trajectory filtering determines which demonstrations are retained, whereas weighted supervision adjusts their contribution to the loss. Importance-weighted SFT provides related motivation (Qin and Springenberg, 2025), but our weights are fixed task-specific credits for discovery and inspection rather than model-derived importance ratios. Subsequent RL uses GRPO (Shao et al., 2024) with dynamic removal of uninformative rollout groups, following the sampling principle used in DAPO (Yu et al., 2026). We study the resulting policies and reward trade-offs rather than proposing a new policy-optimization algorithm.

3 Task and Experimental Setup

3.1 Agent Harness and Search Interfaces

We design our own harness for the code localization task, as presented in Figure 1. The harness specifies the system prompt, tool schemas (Appendix A.6), read-only sandbox, sampling parameters, and interaction budget, thereby defining the actions available to the model and the feedback it receives. Following prior work favoring small tool sets for localization (Sutawika et al., 2026; Zhang et al., 2026), we keep the harness minimal. We use the same localization harness for both training and evaluation.

Colgrep: Semantic Search. `colgrep` (Sourty, 2026) is the semantic search engine used in our harness. It is a single Rust binary built on NextPlaid, a multi-vector search engine implementing the PLAID index, and runs entirely locally without a server or API. We use `colgrep` 1.7.0 and build all indexes once before training and evaluation, so inference-time retrieval only requires query encoding and index lookup. We build one index per repository at the instance's base commit. `colgrep` parses source files with tree-sitter and splits them into code units—functions, methods, classes, constants, and remaining blocks—across 35 languages. In all experiments, we use the same `LateOn-Code-edge` encoder (Chaffin, 2026), a 17M-parameter late-interaction model (Khattab and Zaharia, 2020). Each code unit is represented as a sequence of token vectors and scored with MaxSim: for each query token, MaxSim selects the highest similarity to any token in the candidate unit and sums these scores across the query. This preserves lexical, identifier, and semantic matches while enabling efficient CPU indexing and search.

Tools and session protocol. The agent has access to three tools. `colgrep`, as detailed above, performs semantic search over a pre-built index from a natural-language query, with optional features such as *path*, file-type *include*, or regex *pattern*. `terminal` provides a read-only shell with standard Unix utilities for navigation, lexical search, and file inspection. Access is restricted to the repository at the instance's base commit. `finish` submits file paths with optional line ranges or symbol names. The system prompt asks the model to identify locations relevant to an issue without modifying the repository; the user turn provides the issue description and repository root listing. The model may interact for up to 10 turns, with at most 5 parallel tool calls per turn. Malformed tool calls receive corrective feedback and may be retried up to 3 times. A reminder is injected near the end of the interaction budget, and if `finish` has not yet been called, a rescue prompt requests a final submission. If the model still fails to submit, the inspected files are used as a fallback submission so that every session produces a scorable output.

Retrieval interface. A search typically uses a `query` argument containing the natural-language query passed to the semantic search model and returns the top-k matching units. The optional `k` argument controls the number of results, with a default of 10 and a maximum of 25. Results are returned as `file:start-end` locations with short snippets. Three optional arguments can further restrict the candidate set before ranking: `path` limits retrieval to a subdirectory or file, `include` applies a file glob, and `pattern` applies a `grep`-style regular-expression pre-filter. This supports hybrid lexical and semantic search: the regex selects candidates exactly, while the semantic model ranks them by the intent expressed in `query`. The policy can therefore combine an identifier, symbol, or error string in `pattern` with a natural-language description of the desired behavior in `query`.

3.2 Training

We construct a corpus spanning multiple programming languages by pairing code repositories with issue descriptions and corresponding gold commits. For SFT, we sample issue–patch instances from SWE-Fixer (Xie et al., 2025) and SWE-rebench (Badertdinov et al., 2025). The resulting corpus spans 2,688 repositories and 13,452 distinct repository–commit checkouts, comprising 4,288 Python instances and 9,164 instances across seven additional languages: C, C++, Go, Java, JavaScript, Rust, and TypeScript, as shown in Table 4. For RL, we construct a separate corpus by mining bug-fix commits from public code repositories, yielding approximately 11,000 instances from 33 repositories across Python, Go, JavaScript, TypeScript, C, Ruby, Rust, and Java. For each instance, the parent commit defines the repository state used as the search environment, the issue or commit description provides the query, and the corresponding patch defines the target locations.We deduplicate by instance ID and repository/base-commit/patch hashes, decontaminate by excluding benchmark repositories and shared commit histories, and screen problem statements with long-n-gram matching. We apply our recipe to three students, ranging from 1–1.7B parameters: Qwen3-1.7B (Team, 2025), MiniCPM5-1B (MiniCPM, 2025), and LFM2.5-1.2B-Instruct (AI, 2025).

3.3 Evaluation Protocol

We evaluate file localization from code problem statements by adapting benchmarks originally designed for end-to-end issue resolution, SWE-bench Lite (Jimenez et al., 2024) and Multi-SWE-bench Flash (Zan et al., 2026). Each instance pairs a GitHub issue with the human-written pull request that resolves it at a fixed base commit. We use the eligible pre-patch files and line ranges modified by that pull request as gold localization targets and ask the agent to identify them rather than produce a fix. SWE-bench Lite contains 300 issues from 12 Python repositories. Multi-SWE-bench Flash contains 300 instances spanning 24 repositories and seven languages: C, C++, Java, Go, Rust, JavaScript, and TypeScript. Our primary metric is file-level F1, reflecting our goal of selecting relevant files for downstream inspection and editing while balancing coverage and precision. For each instance, we compute file-level precision, recall, and F1, with empty predictions scored as zero, and macro-average scores across instances. On SWE-bench Lite, we additionally report class- and function-level F1, defining gold classes and functions as the tree-sitter definitions enclosing modified lines at the base commit.

We evaluate our students using a 10-turn search budget, native function calling through vLLM (Kwon et al., 2023), a 32k-token context, and at most 2,048 completion tokens per turn. Sampling uses temperature 0.6 and top-$p = 0.95$, with `thinking` disabled where supported. We compute the average of five seeds for each evaluation.

4 Training Compact Localization Agents

4.1 Supervised Fine-Tuning from Teacher Search Trajectories

We use SFT to teach our student models to search, inspect results, and submit relevant locations through the harness. Tool descriptions alone are often insufficient: models may continue to rely on familiar shell commands such as `grep` even when semantic search is available. We therefore use Teacher demonstrations to show how to formulate `colgrep` queries, use retrieved evidence to guide subsequent actions, and produce correctly formatted submissions. We study two ways of using relevance labels during SFT: trajectory selection and turn-level weighting.

Teacher demonstration generation. We use DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026) to generate the training instances described in Section 3.2 through the localization harness. Trajectories are generated with temperature 0.3, at most 10 turns, 5 parallel tool calls per turn, and 2,048 completion tokens per turn. We slightly adapt the prompt at generation time to encourage the Teacher to use `colgrep` rather than `grep`. Furthermore, compatible `grep` calls are replaced with regex-filtered `colgrep` calls using the *pattern* argument, while incompatible shell searches are resampled. Before training, we restore the prompt and tool schema used at evaluation.

Filtering and turn-level credit assignment. We compare several trajectory-selection variants. *Unfiltered* retains all valid teacher trajectories, while *F1 filtering* requires a submitted-item F1 score of at least 0.5, at least one search action, and a `finish` submission within the interaction budget. We also revisit quality-weighted supervision (Qin and Springenberg, 2025) through *weighted SFT*, assigning credit to assistant turns based on their tool outcomes. We use the following weight assignment: discovering a new gold file receives weight 1.0, re-inspecting an already found gold file 0.6, retrieving only non-relevant files 0.3, and empty results, errors, or repeated commands 0.05. The final submission turn is weighted by its recall over the gold files. Prompt and tool-observation tokens are masked from the loss, and gold annotations are used only to construct training weights and are unavailable at evaluation. Let $\mathcal{T}_i$ be the assistant-token positions in turn i, w_i its credit, and CE_t the token-level cross-entropy loss. We minimize,

$$\mathcal{L}_{\mathrm{SFT}} = \frac{\sum_i w_i \sum_{t \in \mathcal{T}_i} \mathrm{CE}_t}{\sum_i w_i |\mathcal{T}_i|}. \tag{1}$$

Training configuration. We perform a separate full fine-tune for each student and supervision variant using TRL (von Werra et al., 2020). We use AdamW (Loshchilov and Hutter, 2017) with a learning rate of 2×10^{-5}, cosine decay, 3% warmup, zero weight decay, gradient clipping at 1.0, bf16 precision, FlashAttention-3 (Shah et al., 2024), sequence packing, and gradient checkpointing for one epoch.

4.2 Reinforcement Learning for Efficient Localization

Reinforcement learning (RL) allows the student to refine its use of `colgrep` through direct interaction with the search environment. Starting from an SFT checkpoint, the student explores alternative queries and inspection strategies and is rewarded according to the quality of its final submission. This enables it to develop search behavior beyond the teacher demonstrations.

RL rewards and turn penalty. We compare multiple reward functions for code localization. *File F1* rewards a balance between retrieving the correct files and avoiding incorrect ones, while *File* F_2 places more weight on recall, encouraging the agent to recover as many gold files as possible. And *File+Class+Function F1* which follows CodeScout (Sutawika et al., 2026) and sums localization scores at the file, class, and function levels to provide finer-grained supervision. We use this combined reward for Python instances, where class and function annotations are available, and file F1 alone for other languages. All rewards also include a format penalty of -0.5 when a rollout performs no search action or fails to submit through the `finish` tool, encouraging the agent to gather evidence and return a structured prediction. We also conduct an efficiency-focused file-F1 run, where we apply a bounded penalty based on the number of assistant turns T. The first two turns are unpenalized; each additional turn reduces the task reward by 1%, up to a maximum reduction of 8%. Here, R_{task} denotes the task reward, R_{format} the format penalty, T the number of assistant turns, T_0 the unpenalized turn threshold, λ the per-turn penalty rate, and $p_{\max}$ the maximum penalty.

$$R = R_{\mathrm{task}} \left[1 - \min\left(p_{\max}, \lambda \max(0, T - T_0)\right)\right] + R_{\mathrm{format}} \tag{2}$$

Training configuration. We use Prime-RL (Intellect, 2025) to train the student with GRPO (Shao et al., 2024), with several modifications: we remove the KL penalty and standard-deviation normalization of group-relative advantages, following Dr. GRPO (Liu et al., 2025), and apply zero-advantage filtering with dynamic sampling, following DAPO (Yu et al., 2026). Each batch contains eight groups of 16 rollouts, totaling 128 rollouts, sampled at temperature 1.0 with at most 1,024 completion tokens per turn. Rollouts are generated asynchronously and discarded when they are more than four policy updates behind the current student. We update all model parameters for 500 steps with AdamW at a learning rate of 10^{-6}, using two H100 GPUs for training and six for inference.

Model	Training	Search tool	SWE-bench Lite File F1	SWE-bench Lite File R	SWE-bench Lite File P	SWE-bench Lite Class F1	SWE-bench Lite Function F1	Multi-SWE-bench Flash File F1	Multi-SWE-bench Flash File R	Multi-SWE-bench Flash File P
Qwen3-1.7B	Base	`grep`	18.75	26.13	16.66	12.21	**5.65**	12.38	14.32	14.31
		`colgrep`	**34.51**	**56.53**	**29.30**	**13.64**	2.81	**29.39**	**43.18**	**31.45**
	Weighted SFT	`grep`	50.13	60.93	45.60	**33.58**	**18.00**	32.84	36.47	38.53
		`colgrep`	**55.09**	**71.67**	**48.46**	28.62	8.00	**44.58**	**54.26**	**47.31**
	+ RL: file F1	`grep`	63.75	70.60	60.64	**45.05**	**25.05**	50.16	49.14	62.84
		`colgrep`	**67.29**	**72.67**	**64.77**	33.09	9.53	**54.00**	**52.81**	**67.07**
DeepSeek-V4-Flash-0731	-	`colgrep`	63.96	90.73	53.26	50.30	30.51	59.12	74.85	57.34
Qwen3-32B	-	`colgrep`	63.50	80.33	57.00	36.59	14.15	54.02	60.48	60.43

Table 1: Comparison of `grep` and `colgrep` across training stages for Qwen3-1.7B. Bold indicates the better search interface for each metric within a training stage.

5 Results and Analysis

5.1 Training Effects on Search Interfaces

We compare semantic search with `colgrep` and lexical search by constructing a matched `grep` variant of the agent harness. We remove the `colgrep` tool, adjust the system prompt (Appendix A.6) to describe the remaining lexical-search interface, and regenerate teacher trajectories with DeepSeek-V4-Flash-0731 under this modified harness. We then apply the same weighted-SFT and RL pipeline to these trajectories. This makes the comparison symmetric in the sense that each agent is trained on demonstrations generated with the tool available to it, while preserving the remaining training and evaluation protocol.

Table 1 shows that `colgrep` outperforms `grep` on file F1, our primary metric, on both benchmarks at every training stage. This advantage also holds across all three model families after weighted SFT (Table 5). These results suggest that semantic retrieval reduces the search burden on small policies: the agent specifies the desired behavior, while the retriever matches and ranks candidate code. With `grep`, by contrast, the policy must infer repository-specific terms and iteratively refine its queries based on returned matches. The training dynamics support this interpretation. Under the turn penalty, `colgrep`-trained models converge to only a few search calls, suggesting that these are sufficient for file selection, whereas `grep`-trained models continue to use more turns and tool calls (Figure 4). At finer granularity, however, the `grep`-based Qwen3-1.7B achieves higher class and function F1 after SFT and RL on SWE-bench Lite. This may reflect the value of exact lexical matching for distinguishing specific symbols within relevant files. Although `colgrep` supports analogous filtering through its `pattern` argument, the student rarely uses it. Training also substantially narrows the interface gap: on Multi-SWE-bench Flash, `colgrep`'s file-F1 advantage decreases from 17.01 points for the base policy to 11.74 after weighted SFT and 3.84 after RL. Qualitative examples of these different search behaviors are provided in Appendix A.7.

Figure 4 shows that, as RL progresses, the Qwen3-1.7B policy with `colgrep` converges to much shorter search trajectories. After a few hundred updates, rollouts average roughly two `colgrep` calls and three assistant turns, suggesting that only a few semantic searches are needed to identify relevant files before inspecting candidates and submitting the final answer.

5.2 What Does The Agent Add Beyond Direct Retrieval?

To isolate the contribution of agentic search beyond a single retrieval call, we compare our policy with `colgrep` and BM25 direct-retrieval baselines. Each baseline uses the full original issue text as a single query and returns an oracle subset of the top-10 retrieved files. Because this subset is selected using the gold files, it represents an upper bound for that fixed retrieval call rather than a deployable system. We evaluate the Qwen3-1.7B policy trained with file-F1 reward and a turn penalty in two settings. In the single-turn setting, the model executes only its first parallel batch of tool calls, primarily `colgrep` queries, and then submits immediately. In the multi-turn setting, it follows the evaluation protocol from Section 3.3, with a maximum of 10 turns, though the model's average interaction length is only 3.7 turns.

Configuration	Code		
	SWE F1	**Multi-SWE F1**	**Avg. retrieval calls**
BM25 oracle, direct retrieval (top-10)	53.67	34.77	1.00
`colgrep` oracle, direct retrieval (top-10)	59.67	46.79	1.00
Qwen3-1.7B RL, single-turn	61.86	51.25	2.00
Qwen3-1.7B RL, multi-turn	**67.34**	**53.37**	2.06

Table 2: File F1 (%) and average retrieval calls for direct top-10 retrieval with oracle selection and single- and multi-turn Qwen3-1.7B RL agents on SWE-bench Lite and Multi-SWE-bench Flash.

As shown in Table 2, the single-turn agent outperforms both direct-retrieval baselines, exceeding BM25 by 8.19 file-F1 points on SWE-bench Lite and 16.48 points on Multi-SWE-bench Flash, and the stronger `colgrep` baseline by 2.19 and 4.46 points, respectively. The agent formulates shorter, more targeted `colgrep` queries and averages two retrieval calls in its first turn, allowing different queries to retrieve complementary evidence. Multi-turn interaction provides further gains, reaching 67.34 and 53.37 file F1, respectively. These gains require little additional retrieval: the multi-turn agent averages 2.06 `colgrep` calls, compared with 2.00 for the single-turn variant. The additional interaction steps allow the model to verify and refine retrieved candidates, often using `sed` commands to inspect file contents before making its final prediction.

5.3 How Do Supervision and Reward Choices Affect the Policy?

Table 3 shows that turn-weighted SFT achieves the highest file F1 for all three students on both benchmarks. This supports the value of turn-level supervision for multi-turn search: unsuccessful trajectories can still contain useful retrieval steps, while successful ones may include unproductive turns. Weighting each turn by its observed retrieval outcome preserves informative behavior without assigning equal importance to every action. We therefore use the weighted SFT checkpoint to initialize all subsequent RL experiments.

		SWE-bench Lite					Multi-SWE-bench Flash		
		File			Class	Function	File		
Student	**Method/Filtering**	**F1**	R	P	F1	F1	**F1**	R	P
Qwen3-1.7B	Base	34.51	56.53	29.30	13.64	2.81	29.39	43.18	31.45
	Unfiltered	54.87	71.33	48.08	27.69	7.58	43.85	53.27	46.82
	F1	53.50	68.53	47.31	25.14	6.70	42.15	49.40	46.13
	Weighted	**55.09**	**71.67**	**48.46**	**28.62**	**8.00**	**44.58**	**54.26**	**47.31**
MiniCPM5-1B	Base	7.72	20.93	5.67	2.69	0.47	7.50	13.93	7.27
	Unfiltered	45.23	59.47	39.62	20.18	**3.26**	38.06	45.48	43.00
	F1	43.85	55.80	39.00	18.92	2.79	36.80	41.90	42.74
	Weighted	**47.10**	**60.53**	**41.77**	**20.93**	2.52	**40.77**	**47.84**	**45.84**
LFM2.5-1.2B-Instruct	Base	11.29	18.13	9.04	0.85	0.32	6.75	8.00	7.80
	Unfiltered	45.71	59.07	40.36	**21.24**	4.09	37.79	45.21	41.89
	F1	43.98	55.47	39.20	18.07	3.02	35.50	41.87	40.16
	Weighted	**46.87**	**60.73**	**41.34**	20.97	**4.20**	**39.06**	**47.08**	**42.04**
DeepSeek-V4-Flash-0731	-	63.96	90.73	53.26	50.30	30.51	59.12	74.85	57.34

Table 3: **Comparison of supervision filtering methods** (unfiltered, F1, and turn-weighted) across three model families on code localization. Bold denotes the best result for each student and metric.

RL reward design reveals distinct trade-offs (Table 6). Relative to weighted SFT, optimizing file F1 improves file F1 by 12.20 points on SWE-bench Lite and 9.42 points on Multi-SWE-bench Flash. Optimizing F_2 achieves the highest recall on both benchmarks but reduces file F1, while the combined file, class, and function reward yields the highest class and function F1 on SWE-bench Lite. Adding the turn penalty keeps file F1 within 0.63 points of the unpenalized file-F1 reward while producing much shorter trajectories.

5.4 Search Behavior and Measured Efficiency

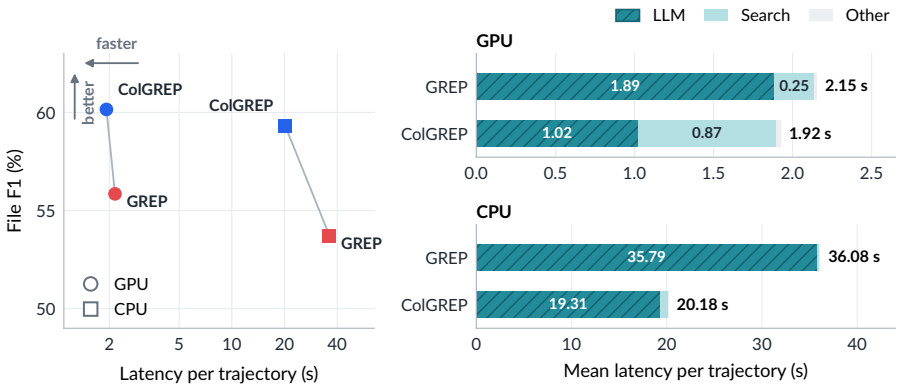

Figure 2: **File-localization performance and latency for Qwen3-1.7B after RL** with the turn penalty. The latency breakdown separates LLM inference, search (`grep`/`colgrep` calls), and other runtime overhead.

We study the search behavior and efficiency of the `grep`- and `colgrep`-based policies after RL training with file-F1 reward and our turn penalty. We compare trajectory length, end-to-end latency, and latency composition to understand how the search interface affects both localization accuracy and inference cost. For latency measurements, trajectories are executed sequentially in both GPU and CPU settings, with the student model served through vLLM on GPU or llama.cpp with BF16 GGUF weights on CPU. In both settings, `colgrep` runs entirely on CPU over precomputed indexes; see Appendix A.5 for the full measurement protocol.

End-to-end efficiency. Averaged equally over the two benchmarks, `colgrep` is substantially more efficient than `grep`, with the largest gains appearing in the CPU-only setting. On CPU, mean end-to-end latency decreases from 36.08 to 20.18 seconds per trajectory, a reduction of 44.1%. With GPU inference, latency also decreases, from 2.15 to 1.92 seconds, corresponding to a 10.6% reduction. Trajectories are also considerably shorter, with mean length decreasing from 7,857 to 5,574 tokens, a reduction of 29.1%. Together with the higher file F1 shown in Figure 2, these results indicate that `colgrep` reaches better predictions with both fewer tokens and lower end-to-end latency.

Latency composition. We further examine how end-to-end latency is distributed between LLM inference and search calls, i.e., time spent executing `grep` or `colgrep`. With GPU inference, LLM inference accounts for 87.6% of `grep` latency, compared with 53.1% for `colgrep`. `colgrep` cuts LLM inference time nearly in half, which more than compensates for the higher cost of `colgrep` calls, reducing total latency from 2.15 to 1.92 seconds. This difference becomes much more pronounced on CPU, where LLM inference accounts for 99.2% of `grep` latency, compared with 95.7% for `colgrep`. Since `colgrep` requires substantially less LLM inference, the end-to-end latency reduction grows from 10.6% with GPU inference to 44.1% on CPU. The reported `colgrep` latency also includes avoidable retrieval overhead, since each call currently launches a new process and reloads the retrieval model; keeping the model resident across calls could further reduce latency.

5.5 Generalization to Unseen Programming Languages

While our training recipe uses multilingual data for both SFT and RL, we also study SFT restricted to a single programming language. Specifically, we fine-tune each student with weighted SFT on Python-only trajectories generated by DeepSeek-V4-Flash-0731. For each setup reported in Table 4, we collect additional decontaminated Python-only teacher trajectories to match the sample count used for multilingual weighted SFT. We then evaluate on the seven non-Python languages in Multi-SWE-bench Flash and compare these policies with their multilingual weighted-SFT counterparts, isolating the effect of excluding evaluation languages from SFT while holding the training sample count fixed.

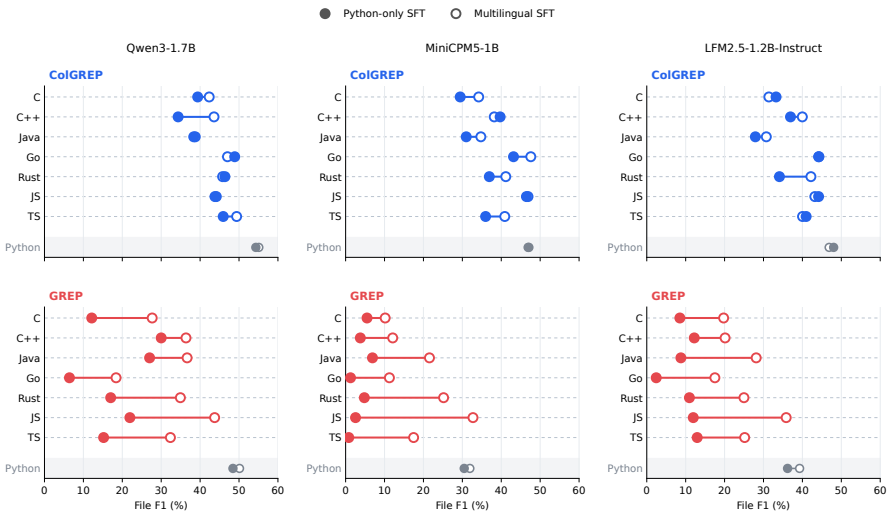

Figure 3: Performance across seven programming languages after Python-only versus multilingual SFT, using the `colgrep` and `grep` harnesses.

Figure 3 shows that restricting SFT to Python has a substantially smaller effect on `colgrep`-based policies. Averaged across the three students, benchmark file F1 drops by only 2.13 points with `colgrep`, compared with 14.90 points with `grep`. This pattern holds for all three students: the benchmark-level drop ranges from 1.47 to 3.04 points with `colgrep`, versus 14.53 to 15.27 points with `grep`. Under Python-only training, `colgrep` also outperforms `grep` across all seven languages and all three students, with an average benchmark advantage of 28.82 points. These results suggest that, in our setting, a multilingual code retriever substantially reduces the dependence on multilingual policy fine-tuning. The agent formulates natural-language queries, while the retriever matches them to code across languages. With `grep`, the agent must instead generate search patterns that account for language-specific syntax and identifiers.

6 Discussion and Limitations

While this work shows that compact agents benefit from semantic repository search, several limitations remain. First, we restrict our study to models below two billion parameters, reflecting our focus on fast, inexpensive agents that can run locally on consumer hardware. Evaluating larger models would help determine whether these benefits persist as the underlying policy becomes more capable, but is beyond the scope of this work. We also conduct all experiments using a single agent harness and therefore do not assess the sensitivity of our findings to different agent implementations, prompting strategies, or tool interfaces. Finally, our evaluation measures localization against files modified by a reference patch, which captures only one possible solution and may omit useful contextual files. Our experiments are also limited to code repositories. Extending the framework to other agent harnesses and retrieval-intensive domains, such as general document search, remains an important direction for future work.

7 Conclusion

We presented a training framework for compact code-localization agents combining weighted supervised fine-tuning with reinforcement learning around `colgrep`. Across SWE-bench Lite and Multi-SWE-bench Flash, semantic search improves localization over lexical search, while weighted supervision improves file F1 across three model families below two billion parameters. Semantic search also reduces trajectory length by 29.1% and mean latency by 10.6% on GPU and 44.1% on CPU, supporting resource-constrained deployment. Our analyses further show stronger generalization to unseen programming languages and that multi-turn interaction outperforms both single-turn search and direct retrieval. Overall, these results support compact semantic-search agents as an efficient approach to multilingual file localization.

AI USE STATEMENT

In this work, we used generative AI tools as part of our methodology, in particular to generate synthetic training data with DeepSeek, as described in Section 4. We also used AI assistants to support the preparation of the manuscript, including proofreading and polishing the writing, improving clarity and phrasing, and suggesting possible section titles; all final titles and formulations were selected and reviewed by the authors. Where AI-assisted outputs were used in the research workflow or manuscript preparation, they were reviewed by the authors before inclusion. We did not rely on generative AI to determine the scientific conclusions of the work. We take full responsibility for the final content of this work, including all text, claims, analyses, and artifacts produced with the assistance of generative AI.

ETHICS STATEMENT

This work studies code localization using public software repositories and benchmarks. It does not involve human participants, personal information, or sensitive data. We do not identify any specific ethical concerns associated with the methods or experiments presented in this paper.

REPRODUCIBILITY STATEMENT

We describe the evaluation benchmarks and metrics in Section 3.3, the construction and decontamination of the training data in Section 3.2, and the agent harness, retrieval configuration, and training procedure in Sections 3.1 and 4. To support reproducibility, we will release model checkpoints, software, data, and training pipelines upon acceptance.

REFERENCES

Liquid AI. Lfm2 technical report. *arXiv preprint arXiv:2511.23404*, 2025.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Antoine Chaffin. Lateon-code: a family of state-of-the-art late interaction code retrieval models, 2026. URL https://huggingface.co/collections/lightonai/lateon-code.

Wuya Chen, Yang Cao, Yue Lin, et al. Codegrep: An rl-trained retrieval agent for llm coding agents. *arXiv preprint arXiv:2608.05886*, 2026.

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)*, pages 8697–8727, 2025.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Tz-Huan Hsu, Jheng-Hong Yang, and Jimmy Lin. Rethinking agentic search with pi-serini: Is lexical retrieval sufficient?, 2026. URL https://arxiv.org/abs/2605.10848.

Prime Intellect. prime-rl, 2025. URL https://github.com/PrimeIntellect-ai/prime-rl.

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.

Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '20, page 39–48, New

York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380164. doi: 10.1145/3397271.3401075. URL https://doi.org/10.1145/3397271.3401075.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective. *arXiv preprint arXiv:2503.20783*, 2025.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

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.

Junayed Mahmud, Nadeeshan De Silva, Safwat Ali Khan, Seyed Hooman Mostafavi, S. M. Hasan Mansur, Oscar Chaparro, Andrian Marcus, and Kevin Moran. On using gui interaction data to improve text retrieval-based bug localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024. doi: 10.1145/3597503.3608139.

Team MiniCPM. Minicpm4: Ultra-efficient llms on end devices. *arXiv preprint arXiv:2506.07900*, 2025.

Chongli Qin and Jost Tobias Springenberg. Supervised fine tuning on curated data is reinforcement learning (and can be improved). *arXiv preprint arXiv:2507.12856*, 2025.

Revanth Gangi Reddy, Ye Liu, Wenting Zhao, JaeHyeok Doo, Tarun Suresh, Daniel Lee, Caiming Xiong, Yingbo Zhou, Semih Yavuz, and Shafiq Joty. Swerank+: Multilingual, multi-turn code ranking for software issue localization. *arXiv preprint arXiv:2512.20482*, 2025.

Keshav Santhanam, Omar Khattab, Christopher Potts, and Matei Zaharia. Plaid: an efficient engine for late interaction retrieval. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pages 1747–1756, 2022.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=tVConYid20.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Raphaël Sourty. Nextplaid, colgrep: Multi-vector search, from database to coding agents., 2026. URL https://github.com/lightonai/next-plaid.

Lintang Sutawika, Aditya Bharat Soni, Apurva Gandhi, Taha Yassine, Sanidhya Vijayvargiya, Yuchen Li, Xuhui Zhou, Yilin Zhang, Leander Melroy Maben, Graham Neubig, et al. Codescout: An effective recipe for reinforcement learning of code search agents. *arXiv preprint arXiv:2603.17829*, 2026.

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

Supriti Vijay, Aman Priyanshu, Anu Vellore, Baturay Saglam, and Amin Karbasi. Think before you retrieve: Learning test-time adaptive search with small language models. *arXiv preprint arXiv:2511.07581*, 2025. URL https://arxiv.org/abs/2511.07581.

Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. TRL: Transformers Reinforcement Learning, 2020. URL https://github.com/huggingface/trl.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 1123–1139, 2025.

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

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Zhongming Yu, Hejia Zhang, Yujie Zhao, Hanxian Huang, Matrix Yao, Ke Ding, and Jishen Zhao. Orcaloca: An llm agent framework for software issue localization. *arXiv preprint arXiv:2502.00350*, 2025.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2026.

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, 2026. URL `https://arxiv.org/abs/2512.20957`.

A Appendix

A.1 Supervised Fine-Tuning Data

Table 4 provides additional details on the supervised fine-tuning data used in our experiments. In particular, we report the language composition of the teacher trajectories retained under the different filtering setups

	`grep`	`colgrep`		
Language	Weighted	Unfiltered	Weighted	File-F1 ≥ 0.5
Python	4,374	4,288	4,288	3,157
Go	2,056	2,083	2,083	1,402
Rust	1,840	1,841	1,841	1,291
JavaScript	1,824	1,822	1,822	1,224
TypeScript	1,697	1,684	1,684	1,056
Java	1,453	1,433	1,433	811
C	192	186	186	113
C++	114	115	115	70
Total	13,550	13,452	13,452	9,124

Table 4: Language distribution of teacher trajectories generated with DeepSeek-V4-Flash-0731 after cleaning and selection for each SFT mode.

A.2 Search Tool Comparison

Table 5 compares `grep` and `colgrep` after weighted supervised fine-tuning across three small language models. Across both SWE-bench Lite and Multi-SWE-bench Flash, `colgrep` consistently improves file-level localization performance. The gains are particularly pronounced for the smaller LFM2.5-1.2B-Instruct and MiniCPM5-1B models.

Benchmark	**Model**	**Search tool**	**File**			**Class**	**Function**	**Turns**
			F1	R	P	F1	F1	
SWE-bench Lite								
	Qwen3-1.7B + SFT	`grep`	50.13	60.93	45.60	**33.58**	**18.00**	**7.89**
		`colgrep`	**55.09**	**71.67**	**48.46**	28.62	8.00	8.53
	LFM2.5-1.2B-Instruct + SFT	`grep`	39.28	47.33	36.16	**25.14**	**10.75**	**7.79**
		`colgrep`	**46.87**	**60.73**	**41.34**	20.97	4.20	7.83
	MiniCPM5-1B + SFT	`grep`	31.93	41.60	28.18	20.20	**9.25**	7.78
		`colgrep`	**47.10**	**60.53**	**41.77**	**20.93**	2.52	**7.30**
Multi-SWE-bench Flash								
	Qwen3-1.7B + SFT	`grep`	32.84	36.47	38.53	–	–	**7.94**
		`colgrep`	**44.58**	**54.26**	**47.31**	–	–	8.56
	LFM2.5-1.2B-Instruct + SFT	`grep`	24.59	26.87	30.11	–	–	**7.88**
		`colgrep`	**39.06**	**47.08**	**42.04**	–	–	7.91
	MiniCPM5-1B + SFT	`grep`	18.84	20.40	23.44	–	–	8.12
		`colgrep`	**40.77**	**47.84**	**45.84**	–	–	**7.12**

Table 5: Comparison of `grep` and `colgrep` after weighted SFT across three small language models. Bold indicates the better result for each model and metric.

A.3 Reinforcement Learning Reward Ablations

We next study how the choice of reward affects localization performance. Table 6 compares file-level F1, an F_2 reward that places greater weight on recall, a hierarchical reward combining file-, class-, and function-level F1, and a file-F1 reward augmented with a turn penalty.

	SWE-bench Lite					Multi-SWE-bench Flash		
	File			Class	Function	File		
Training / reward	**F1**	R	P	F1	F1	**F1**	R	P
Weighted SFT	55.09	71.67	48.46	28.62	8.00	44.58	54.26	47.31
+ RL: file F1	67.29	72.67	64.77	33.09	9.53	**54.00**	52.81	**67.07**
+ RL: file F_2	55.47	**75.80**	47.02	28.70	8.85	48.34	**58.89**	49.90
+ RL: file + class + function F1	65.22	72.20	61.99	**35.31**	**13.80**	53.10	52.97	64.64
+ RL: file F1 + turn penalty	**67.34**	71.60	**65.29**	32.48	10.28	53.37	51.83	66.51

Table 6: Effect of RL reward design on localization performance for Qwen3-1.7B. Results are reported on SWE-bench Lite and Multi-SWE-bench Flash.

A.4 Reinforcement Learning Dynamics

Figure 4 tracks the evolution of trajectory length, assistant turns, tool calls, and search calls over 500 RL optimization steps for Qwen3-1.7B when training with a turn penalty. We report the dynamics for both the `grep` and `colgrep` harnesses.

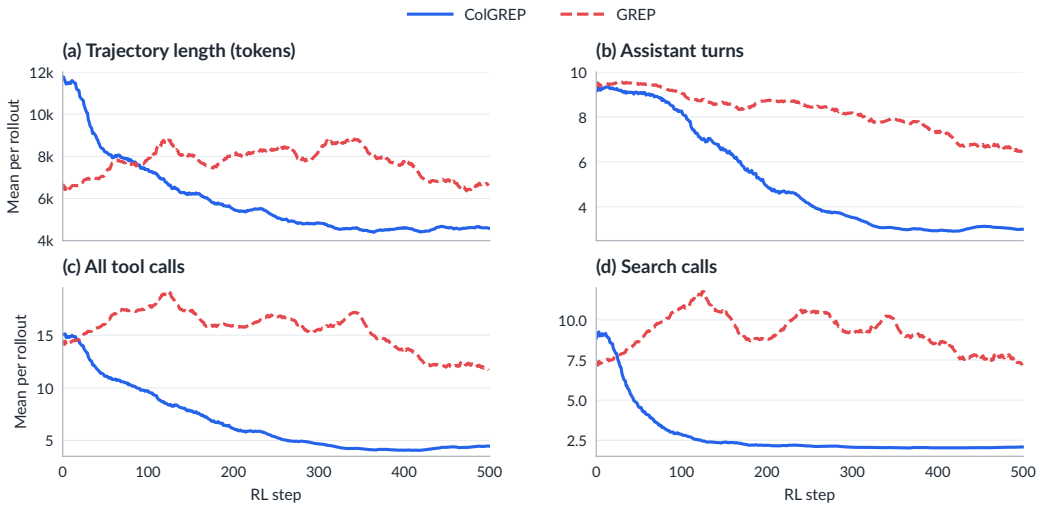

Figure 4: **Evolution** of mean trajectory length, assistant turns, tool calls, and search calls per rollout for Qwen3-1.7B over 500 RL steps with a turn penalty, using the `grep` and `colgrep` harnesses.

A.5 Latency Measurement Protocol

Latency was measured on hosts with two Intel Xeon Platinum 8462Y+ processors, using 32 CPU threads for model inference. GPU inference used one NVIDIA H100 80,GB with vLLM 0.23.0 and BF16 weights; CPU inference used llama.cpp with BF16 GGUF weights. ColGREP ran on CPU with four threads in both settings. Each configuration processed 300 instances per benchmark sequentially; reported values average the two benchmark means equally. `grep` and `colgrep` used the same host within each benchmark and backend. Model weights, repositories, and precomputed indexes were preloaded into memory, with inference caching enabled. Timing covered the complete agent session, including search subprocess startup and retriever loading, but excluded staging, server startup, warm-up, and indexing.

A.6 Evaluation Prompts

ColGREP system prompt

```
You are a code-search agent.  Given a software problem statement,
find the exact code locations | files and line ranges | a
developer must read and edit to solve it.  This is a RETRIEVAL
task:  locate the relevant context; do NOT write a fix.

Follow this method:
• Start broad:  use semantic search to survey the codebase and
identify candidate files related to the problem.
• Narrow down:  refine searches with path scopes, file-type
filters, or regex pre-filters to isolate the most relevant code.
• Read and verify:  open promising files and read the exact lines
to confirm they are the right locations.
• Iterate:  if results are too broad or off-target, reformulate
your query from a different angle and search again.

You MUST search the repository before answering.  Never answer
from prior knowledge.

### Finishing
When you have identified the relevant locations, you MUST use the
`finish` tool to submit your answer.  Do NOT simply reply with
text | call the tool.
The `finish` tool expects a `locations` list; each entry is a
string:
- `path/to/file`
- `path/to/file:START-END` (line range)
- `path/to/file:ClassName.method_name` (symbol)
Call `finish` once when done.  Do not keep searching after you
know the answer.
```

Table 7: Complete system prompt for the ColGREP code-localization configuration.

Submission reminder

```
You have 2 turns left.  You MUST call the `finish` tool now with
your best `locations` list | an unsubmitted session scores zero.
Do not run another search.
```

Table 8: User-role reminder with two turns remaining. The count and singular/plural form vary with the remaining budget.

grep system prompt

```
You are a code-search agent.  Given a software problem statement,
find the exact code locations | files and line ranges | a
developer must read and edit to solve it.  This is a RETRIEVAL
task:  locate the relevant context; do NOT write a fix.

Follow this method:
• Start from the task's concrete tokens:  grep for identifiers,
error strings, function or class names, and other literal text the
problem mentions (e.g.  `grep -rn "TokenError" .`).
• Narrow down:  refine patterns with path scopes or file filters
to isolate the most relevant code.
• Read and verify:  open promising files and read the exact lines
to confirm they are the right locations.
• Iterate:  if results are too broad or off-target, reformulate
your pattern from a different angle and search again.

You MUST search the repository before answering.  Never answer
from prior knowledge.

### Finishing
When you have identified the relevant locations, you MUST use the
`finish` tool to submit your answer.  Do NOT simply reply with
text | call the tool.
The `finish` tool expects a `locations` list; each entry is a
string:
- `path/to/file`
- `path/to/file:START-END` (line range)
- `path/to/file:ClassName.method_name` (symbol)
Call `finish` once when done.  Do not keep searching after you
know the answer.
```

Table 9: Complete system prompt for the grep code-localization configuration.

ColGREP tool descriptions

TOOL: colgrep
Semantic code search over the repository (finds code by MEANING, not just text). Give a natural-language query describing the code this task needs.
query (string; optional): Natural-language description, e.g. "where session token expiry is validated".
k (integer; optional): Number of results to return (default 10, max 25).
path (string; optional): Optional subdirectory or file to scope the search.
include (string; optional): Optional file glob, e.g. "*.py".
pattern (string; optional): Optional grep-style regex pre-filter (colgrep -e), combined with `query`: the regex narrows the candidate documents, the query ranks them semantically. Use for exact names, numbers, or rare terms, e.g. "Art Deco|South Beach". Case-insensitive; if it matches nothing, retry without it.
command (string; optional): Alternative: a full colgrep CLI string.

TOOL: terminal
Run one read-only shell command (ls/cd/cat/head/tail/pwd/grep/find/...). Use colgrep for semantic search.
command (string; required): One shell command.

TOOL: finish
Submit your final answer: the code locations a developer must read and edit. Call this once, when you are done searching. Each location is a string: 'path/to/file', 'path/to/file:START-END' (line range), or 'path/to/file:ClassName.method_name' (symbol).
locations (array of strings; required): The relevant locations, most important first.

Table 10: Complete tool descriptions and parameters exposed to the ColGREP agent. All `colgrep` arguments are optional in the schema; `command` provides the alternative CLI interface.

grep tool descriptions

TOOL: terminal
Run one read-only shell command (grep/find/ls/cd/cat/head/tail/pwd/...). Search the repository with grep, e.g. `grep -rn "TokenError" .`.
command (string; required): One shell command.

TOOL: finish
Submit your final answer: the code locations a developer must read and edit. Call this once, when you are done searching. Each location is a string: 'path/to/file', 'path/to/file:START-END' (line range), or 'path/to/file:ClassName.method_name' (symbol).
locations (array of strings; required): The relevant locations, most important first.

Table 11: Complete tool descriptions and parameters exposed to the grep agent. Lexical searches are passed to `terminal`.

A.7 EXAMPLES OF COLGREP VS. GREP TRAJECTORIES

User (problem statement): pydata__xarray-4248

Feature request: show units in dataset overview

Here's a hypothetical dataset:
It would be really nice if the units of the coordinates and of the data variables were shown in the `Dataset` repr, for example as:

Original display

```
<xarray.Dataset>
Dimensions:  (time: 3, x: 988, y: 822)
Coordinates:
  * x         (x) float64 ...
  * y         (y) float64 ...
  * time      (time) datetime64[ns] ...
Data variables:
    rainfall  (time, y, x) float32 ...
    max_temp  (time, y, x) float32 ...
```

Requested display

```
<xarray.Dataset>
Dimensions:  (time: 3, x: 988, y: 822)
Coordinates:
  * x, in metres      (x) float64 ...
  * y, in metres      (y) float64 ...
  * time              (time) datetime64[ns] ...
Data variables:
    rainfall, in mm    (time, y, x) float32 ...
    max_temp, in deg C (time, y, x) float32 ...
```

Find the code locations relevant to this problem.

colgrep harness — 4 assistant turns

Agent (Turn 1)

```
{"query": "show units in dataset overview repr
 coordinates data variables"}
{"query": "dataset repr units dimensions
coordinates variables"}
```

Tool (Turn 1; excerpts)

```
[...]
[6] xarray/core/formatting.py:256-678  (
function dataset_repr)
[...]
```

Agent (Turn 2)

Let me look at the `dataset_repr` function in formatting.py, which is the core of what needs to be edited.

```
sed -n '256,678p' xarray/core/formatting.py
sed -n '187,200p' xarray/core/dataset.py
```

Tool (Turn 2; excerpts)

```
[...]
def summarize_variable(
    name: Hashable, var, col_width: int,
marker: str = " ", max_width: int = None
):
    """Summarize a variable in one line, e.g.,
 for the Dataset.__repr__."""
[...]
    first_col = pretty_print(f"  {marker} {
name} ", col_width)
    if var.dims:
        dims_str = "({}) ".format(", ".join(
map(str, var.dims)))
    else:
        dims_str = ""
    front_str = f"{first_col}{dims_str}{var.
dtype} "
[...]
```

Agent (Turn 3)

Let me see the rest of the `dataset_repr` function, particularly the part that builds the dataset summary.

```
[...]
sed -n '678,720p' xarray/core/formatting.py
```

Tool (Turn 3; excerpts)

```
return "\n".join(summary)
```

Agent (Turn 4)

I have all the relevant locations.

```
[...]
```

finish

```
{"locations": ["xarray/core/formatting.py
:256-678"]}
```

File F1: 1.00 Exact match to the gold file set.

grep harness — 7 assistant turns

Agent (Turn 1)

```
grep -rn "unit" xarray/ --include=*.py
grep -rn "repr" xarray/ --include=*.py
```

Tool (Turn 1; excerpts)

```
xarray/backends/api.py:386:        If True,
decode variables and coordinates with time
units in
[...]
xarray/backends/api.py:382:        represented
 using ``np.datetime64[ns]`` objects.  If False
, always
[...]
```

[Turns 2–4 omitted]

Agent (Turn 5)

```
grep -rn "units" xarray/core/
grep -rln "__repr__" xarray/core/
```

Tool (Turn 5; excerpts)

```
[...]
xarray/core/dataset.py:4800:
                'attrs': {'units':'s'}}},
[...]
xarray/core/dataset.py
xarray/core/formatting.py
[...]
```

Agent (Turn 6)

I have a good understanding. The key locations are in `xarray/core/dataset.py` where the Dataset's __repr__ is defined, and where units might be extracted from variables.

```
[...]
sed -n 4750,4820p xarray/core/dataset.py
grep -rln "units" xarray/core/dataset.py
```

Tool (Turn 6; excerpts)

```
[...]
def to_dict(self, data=True):
    """
    Convert this dataset to a dictionary
following xarray naming
    conventions.
[...]
xarray/core/dataset.py
```

Agent (Turn 7)

I have all the information needed.

```
[...]
```

finish

```
{"locations": ["xarray/core/dataset.py
:4750-4820"]}
```

File F1: 0.00 Submitted file differs from gold.

Figure 5: **Example 1 — Selected ColGREP vs. Grep trajectory excerpts.** Qwen3-1.7B trained with weighted SFT + RL (file F1 with turn penalty; step 500). ColGREP reaches the gold file, `xarray/core/formatting.py`; grep follows a units match into the `to_dict` documentation in `dataset.py`. Excerpts retain the original turn numbers.

User (problem statement): cli__cli-2279

add message on `pr checks` returns no CI with non-empty PR
[...]
Update messaging when `gh pr check` run on repo w/o CI
When running this `gh pr checks [branch] -R [owner/repo]` command and there is no pull request but no CI, I get this: `no open pull requests found for branch "JJJ"`
However, when running the same command on a different branch and there is a pull request but no CI, nothing is returned.
In both cases, I think it would be more accurate to say something like "no CI checks are set up for this repository".

Find the code locations relevant to this problem.

colgrep harness — 4 assistant turns

Agent (Turn 1)

```
{"query": "pr checks returns no CI with non-
empty PR message"}
{"query": "gh pr checks command no CI checks
setup"}
```

Tool (Turn 1; excerpts)

```
[...]
[2] pkg/cmd/pr/checks/checks.go:1-227  (
function NewCmdChecks)
[...]
[1] pkg/cmd/pr/checks/checks.go:1-227  (
function NewCmdChecks)
[...]
```

Agent (Turn 2)

Let me read the relevant part of `pkg/cmd/pr/checks/checks.go` to understand the full flow and the exact code that needs to be updated.

```
sed -n '1,60p' pkg/cmd/pr/checks/checks.go
```

Tool (Turn 2; excerpt)

```
[...]
cmd := &cobra.Command{
  Use:   "checks [<number> | <url> | <branch
>]",
  Short: "Show CI status for a single pull
request",
[...]
return checksRun(opts)
[...]
```

Agent (Turn 3)

[...]
Let me verify the exact line range of `checksRun`.

```
sed -n '60,110p' pkg/cmd/pr/checks/checks.go
```

Tool (Turn 3; excerpts)

```
[...]
if len(pr.Commits.Nodes) == 0 {
  return nil
}

rollup := pr.Commits.Nodes[0].Commit.
StatusCheckRollup.Contexts.Nodes
if len(rollup) == 0 {
  return nil
}
[...]
```

Agent (Turn 4)

[...]
Let me finalize.

finish

```
{"locations": ["pkg/cmd/pr/checks/checks.go
:60-110"]}
```

File F1: 1.00 Exact match to the gold file set.

grep harness — 6 assistant turns

Agent (Turn 1)

```
grep -rn "pr checks" . --include=*.go
grep -rn "no open pull requests" .
```

Tool (Turn 1; excerpts)

```
(grep: ran, no output)
./pkg/cmd/pr/status/status.go:130:    shared.
PrintMessage(out, "  You have no open pull
requests")
[...]
```

[Turns 2–4 omitted]

Agent (Turn 5)

Now I understand the problem. The key location is the message logic in `status.go`.
[...]

```
sed -n 112,140p pkg/cmd/pr/status/status.go
grep -rn "no CI checks" .
```

Tool (Turn 5; excerpts)

```
[...]
shared.PrintHeader(out, "Created by you")
if prPayload.ViewerCreated.TotalCount > 0 {
  printPrs(out, prPayload.ViewerCreated.
TotalCount, prPayload.ViewerCreated.
PullRequests...)
} else {
  shared.PrintMessage(out, "  You have no open
 pull requests")
}
[...]
(grep: ran, no output)
```

Agent (Turn 6)

[...]
The relevant locations are the message logic in `else if currentPRHeadRef == ""` and the `shared.PrintMessage` calls. Let me finalize the locations.

finish

```
{"locations": ["pkg/cmd/pr/status/status.go
:112-124", "pkg/cmd/pr/status/status.go
:125-140"]}
```

File F1: 0.00 Submitted file differs from gold.

Figure 6: **Example 2 — Selected ColGREP vs. Grep trajectory excerpts.** Qwen3-1.7B, weighted SFT + RL (file F1 with turn penalty, step 500). ColGREP reaches the gold file, `pkg/cmd/pr/checks/checks.go`, and reads the empty-check branches; grep follows a similar message in `pr/status/status.go`. Excerpts retain the original turn numbers.

User (problem statement): vuejs__core-8538

fix(compiler-sfc): take into account imports that are used within dynamic arguments
[...]
(compiler-sfc): `<script setup>` - Import usage analyzer doesn't resolve dynamic arguments in non-inline mode
[...]
Steps to reproduce
1. Import an object/enum from another file.
2. Try to use their keys as a source of dynamic attribute name or slot name in `<script setup>` component. Note that this enum/object is **not** supposed to be used anywhere else in the component except the mentioned place.
3. Observe an error since our object/enum is not exposed to the template (you can also check the generated code and take a look at the final `__returned__` variable).

What is expected?
This object/enum should be properly exposed to the template in the final `__returned__` variable since it's already being used in the template.

What is actually happening?
An error like `Cannot read properties of undefined (reading 'someProperty')` because the variable we're trying to get access to is not exposed.
[...]
Find the code locations relevant to this problem.

colgrep harness — 3 assistant turns

Agent (Turn 1)

```
{"query": "compiler-sfc import usage analyzer
resolve dynamic arguments non-inline mode"}
{"query": "script setup import usage dynamic
argument resolution"}
```

Tool (Turn 1; excerpts)

```
[1] packages/compiler-sfc/src/script/
importUsageCheck.ts:1-115  (function
isImportUsed)
[...]
export function isImportUsed(local: string,
sfc: SFCDescriptor): boolean {
[...]
```

Agent (Turn 2)

Let me read the key files:
[...]

```
sed -n '1,80p' packages/compiler-sfc/src/
compileScript.ts
sed -n '1,120p' packages/compiler-sfc/src/
script/importUsageCheck.ts
```

Tool (Turn 2; excerpts)

```
[...]
for (let i = 0; i < node.props.length; i++) {
  const prop = node.props[i]
  if (prop.type === NodeTypes.DIRECTIVE) {
[...]
if (prop.exp) {
  code += `,${processExp(
    (prop.exp as SimpleExpressionNode).content
,
    prop.name
  )}`
}
[...]
```

Agent (Turn 3)

I have identified the key locations.
[...]

finish

```
{"locations": ["packages/compiler-sfc/src/
script/importUsageCheck.ts:1-115"]}
```

File F1: 1.00 Exact match to the gold file set.

grep harness — 8 assistant turns

Agent (Turn 1)

```
grep -rn "dynamic arguments" .
grep -rn "import usage analyzer" .
```

Tool (Turn 1; excerpts)

```
(grep: ran, no output)
(grep: ran, no output)
```

[Turns 2–3 omitted]

Agent (Turn 4)

The bug is about dynamic arguments in `<script setup>` where imports are used in non-inline mode.
[...]

```
grep -rn "dynamic arguments" packages/compiler
-sfc/src/
grep -rn "__returned__" packages/compiler-sfc/
src/
```

Tool (Turn 4; excerpts)

```
(grep: ran, no output)
packages/compiler-sfc/src/compileScript.ts
:926:      `\nconst __returned__ = ${returned}\
n` +
[...]
```

[Turns 5–7 omitted]

User (harness reminder after Turn 7)

You have 2 turns left. You MUST call the `finish` tool now with your best `locations` list — an unsubmitted session scores zero. Do not run another search.

Agent (Turn 8)

finish

```
{"locations": ["packages/compiler-sfc/src/
compileScript.ts:376-400", "packages/compiler-
sfc/src/compileScript.ts:926-929"]}
```

File F1: 0.00 Submitted file differs from gold.

Figure 7: **Example 3 — Selected ColGREP vs. Grep trajectory excerpts.** Qwen3-1.7B, weighted SFT + RL (file F1 with turn penalty, step 500). ColGREP reaches the gold file, `packages/compiler-sfc/src/script/importUsageCheck.ts`, and reads the directive traversal; grep submits `compileScript.ts` after repeated searches and a budget reminder. Excerpts retain the original turn numbers.