FuseSearch: Learning Adaptive Parallel Execution for Efficient Code Localization

Anonymous ACL submission

Abstract

Code localization is a primary bottleneck in automated software development. While parallel tool execution can accelerate discovery, existing agents suffer from a 34.9% redundant tool invocation rate, negating the benefits of parallelism. We introduce **FuseSearch**, which reframes parallel code localization as a **quality–efficiency co-optimization** problem. By defining **tool efficiency**—the ratio of novel information gain to total invocations—we employ a two-stage SFT and RL pipeline to train models in adaptive parallel strategies. Unlike fixed-breadth methods, FuseSearch dynamically adjusts search breadth based on task context, transitioning from exploration to refinement. On SWE-bench Verified, FuseSearch-4B matches SOTA performance (84.7% file-level and 56.4% function-level F_1 scores) while being 93.6% faster, using 67.7% fewer turns and 68.9% fewer tokens. Our findings demonstrate that efficiency-aware training inherently boosts quality by eliminating noisy, redundant signals, enabling high-performance, low-cost localization agents. Code: https://anonymous.4open.science/r/FuseSearch-2BDD.

1 Introduction

Code localization—identifying the relevant code entities needed to resolve a given issue—is a critical bottleneck in automated software development (Jimenez et al., 2024; Xia et al., 2024). Recent studies show that state-of-the-art agents devote more than 50% of their computational resources to this task, highlighting the need for more efficient strategies (Pan et al., 2025). In response, recent work has proposed specialized localization agents that operate as dedicated search components, decoupling localization from downstream repair or generation steps (Chen et al., 2025; Jiang et al., 2025). These agents typically rely on multi-turn interactions with **sequential tool execution** (e.g., code retrieval and analysis), iteratively refining queries, inspecting intermediate results, and narrowing down candidate files or functions to achieve high localization accuracy. However, this iterative paradigm introduces a fundamental trade-off: aggressive constraints on the number of allowed tool interactions (i.e., tight turn budgets) are increasingly necessary to meet real-world *computational cost* requirements, as emphasized in recent benchmarks for production-grade agent systems (Gao and Peng, 2025). Under such tight budgets, agents often fail to gather sufficient contextual evidence before exhausting their interaction quota—a phenomenon we refer to as information starvation. Consequently, further reductions in computational cost come at the cost of sharp accuracy degradation, limiting the deployability of current localization approaches in time-sensitive settings.

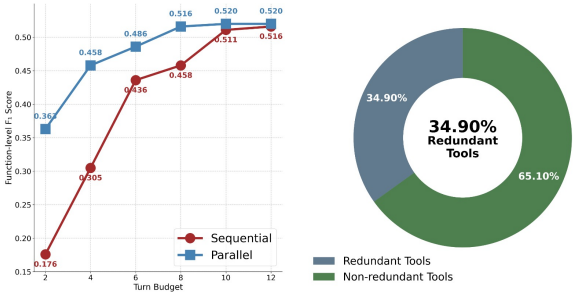

Figure 1: Parallel execution solves sequential search's information starvation under limited turns. However, 34.9% of enforced parallel tools are redundant, exhibiting redundancy.

Parallel tool execution presents a promising avenue for addressing the cost–accuracy trade-off by enabling the simultaneous invocation of multiple tools within a single interaction turn, thereby increasing the information density per turn. As illustrated in Figure 1(a), under tight turn budgets, parallel execution significantly outperforms sequential search in terms of localization accuracy. Despite this potential, most existing agents only provide technical support for parallelism—allowing concurrent tool calls—without consistently harnessing

its benefits in practice (Pan et al., 2025).

Moreover, naive parallelization schemes that enforce a fixed number of tool calls per turn can be highly inefficient. As shown in Figure 1(b), such approaches incur more than 34.9% redundant tool invocations. These unnecessary calls not only waste computational resources but also introduce noisy or irrelevant signals that can degrade localization performance. This raises a key question: how can parallelization be made both **comprehensive** and **non-redundant**—*maximizing information coverage* to avoid starvation under tight turn budgets while *eliminating redundant exploration* of previously examined code?

To address this challenge, we introduce **FuseSearch**, a code localization agent that achieves superior quality–efficiency trade-offs through *learned adaptive parallel execution*. Instead of prescribing a fixed degree of parallelism, FuseSearch *dynamically adjusts the breadth of parallel tool invocations* by explicitly optimizing **tool efficiency**—the ratio of tool calls that yield novel, relevant information to the total number of invocations. We instantiate tool efficiency as a reward that credits exploration of distinct code regions while penalizing redundancy and failed queries, enabling RL-based joint optimization of localization accuracy (measured by F_1) and search efficiency. FuseSearch adopts a minimalist design, relying only on three language-agnostic, read-only tools—`grep`, `glob`, and `read_file`—and requires no auxiliary infrastructure such as code graphs or language-specific parsers.

Building on this formulation, we employ a two-stage training pipeline that combines SFT with RL to train compact models (4B and 30B parameters) to decide *which* tools to invoke and *how many* to run in parallel at each turn, balancing exploration breadth with precision. Notably, optimizing tool efficiency not only reduces search cost but also improves localization quality: by discouraging wasteful calls, the efficiency-driven reward guides the agent toward more targeted and accurate search strategies.

Experimental results on SWE-bench Verified (Jimenez et al., 2024) show that FuseSearch (train) substantially outperforms RepoSearcher (Ma et al., 2025) under the Qwen3-4B backbone (Team, 2025). In terms of localization quality, FuseSearch delivers substantial gains, improving file-level F_1 from 38.1% to 84.7% and function-level F_1 from 21.7% to 56.4%, indicating markedly stronger precision in pinpointing both relevant files and target functions. Meanwhile, FuseSearch is significantly more efficient: it reduces overall interaction turns by 67.7%, cuts time by 93.6%, and lowers token consumption by 68.9%. These results suggest that the learned adaptive parallel execution not only boosts localization accuracy but also streamlines the search process, enabling faster and more targeted exploration with substantially less redundant tool usage. Our main contributions are:

- We propose tool efficiency to quantify information novelty in code search and integrate it into an **efficiency-aware training framework** (via SFT and RL), enabling the joint optimization of search effectiveness and computational efficiency.
- We introduce FuseSearch, a minimalist localization agent that uses only three read-only tools (`grep`, `glob`, `read_file`) yet matches or exceeds the performance of far more complex systems.
- We demonstrate that high-quality, low-latency localization significantly accelerates downstream agent workflows, cutting interaction turns by 23.1% and end-to-end task time by 28.5% without sacrificing success rates.

2 Preliminary

2.1 Task Formulation

Code localization aims to identify the specific code entities—such as files, functions, or code snippets—that require modification to resolve a given issue. We formulate this problem as a repository-level *information-seeking task*. Unlike static retrieval approaches, this process involves an agent that actively interacts with the repository to progressively accumulate relevant context. Through iterative tool calls, the agent retrieves and analyzes various code entities, narrowing down the search space to produce the final localization result.

Formally, an agent operates over discrete turns $t = 1, \ldots, T$. A search trajectory is defined as:

$$\tau = (q, a_1, o_1, \ldots, a_T, o_T, \mathcal{A}) \qquad (1)$$

where q is the issue description, a_t is the set of tool calls at turn t, o_t is the aggregated observation containing the retrieved code content, and $\mathcal{A}$ is the final localization result identifying the target entities for modification.

2.2 Parallel Tool Execution

Traditional sequential agents invoke one tool per turn, leading to prolonged search duration when comprehensive exploration is needed. Parallel tool execution enables simultaneous invocation of multiple tools within a single turn, increasing information density per interaction. In this paradigm, agents generate multiple tool calls in one response, each formatted as a JSON object. All tools within a turn execute concurrently—their read-only nature eliminating synchronization concerns—and their results are aggregated before the next agent response. This design reduces the total number of interaction turns required and shortens overall search time.

3 FuseSearch

We present FuseSearch, a minimalist framework for efficient code localization through learned parallel execution. This section is organized as follows: Section 3.1 introduces our minimalist tool set. Section 3.2 defines efficiency metrics and dual-objective optimization framework—the key innovation enabling joint quality-efficiency optimization. Section 3.3 details the training approach implementing these objectives. Figure 2 illustrates the overall architecture.

3.1 Minimalist Tool Set

We employ a minimalist architecture comprising three read-only tools that enable effective code localization without infrastructure dependencies:

- `grep`: Regex-based pattern search in file contents
- `glob`: File path pattern matching
- `read_file`: Reading file contents with optional line range specification

This minimalist design offers several practical advantages. First, the language-agnostic nature requires no parsers or runtime environments, enabling immediate deployment across diverse codebases. Second, the simplicity reduces learning complexity for models, allowing training resources to focus on strategic tool orchestration rather than intricate tool semantics. Third, it eliminates dependency on pre-computed structures like ASTs or dependency graphs, avoiding the overhead of building and maintaining auxiliary infrastructure.

3.2 Dual-Objective Metrics for Code Localization

While parallel tool execution increases information throughput, naive parallelization often results in substantial redundancy. As shown in Figure 1, over 34.9% of tools in enforced parallel execution provide no incremental value. To optimize parallel search strategies, we must quantify search effectiveness along two complementary dimensions: output quality and process efficiency. We define metrics for both dimensions and establish their joint optimization framework.

Localization Quality Following standard practice (Xia et al., 2024; Chen et al., 2025), we measure localization quality using precision $\mathcal{P}$, recall $\mathcal{R}$, and their harmonic mean F_1 at file-level and function-level granularities. Let $\hat{\mathcal{A}}$ denote the predicted entity set and $\mathcal{A}$ the ground truth:

$$\mathcal{P} = \frac{|\hat{\mathcal{A}} \cap \mathcal{A}|}{|\hat{\mathcal{A}}|}, \quad \mathcal{R} = \frac{|\hat{\mathcal{A}} \cap \mathcal{A}|}{|\mathcal{A}|}, \quad F_1 = \frac{2\mathcal{P}\mathcal{R}}{\mathcal{P} + \mathcal{R}} \tag{2}$$

The F_1 score balances precision and recall, rewarding models that identify relevant code comprehensively (high recall) without excessive over-prediction (high precision).

Tool Efficiency To quantify search efficiency, we measure the information gain of each tool call relative to search progress. During execution, we maintain a history of discovered code entities, including files accessed and content regions examined. For each tool call, we compute its *information gain* g_i by comparing returned entities against this cumulative history:

$$g_i = \begin{cases} \frac{|\mathcal{E}_i \setminus \mathcal{H}|}{|\mathcal{E}_i|} & \text{if } |\mathcal{E}_i| > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $\mathcal{E}_i$ denotes the set of entities returned by tool i, and $\mathcal{H}$ represents the union of all entities discovered in preceding turns. The term $\mathcal{E}_i \setminus \mathcal{H}$ thus quantifies the incremental knowledge gain provided by the tool.

Tool calls exhibit diverse effectiveness patterns. Some discover entirely new content ($g_i = 1.0$), others retrieve only previously-seen entities ($g_i = 0$), and many return mixed results with partial overlap ($0 < g_i < 1$). We define tool efficiency e as the mean information gain across all tool invocations:

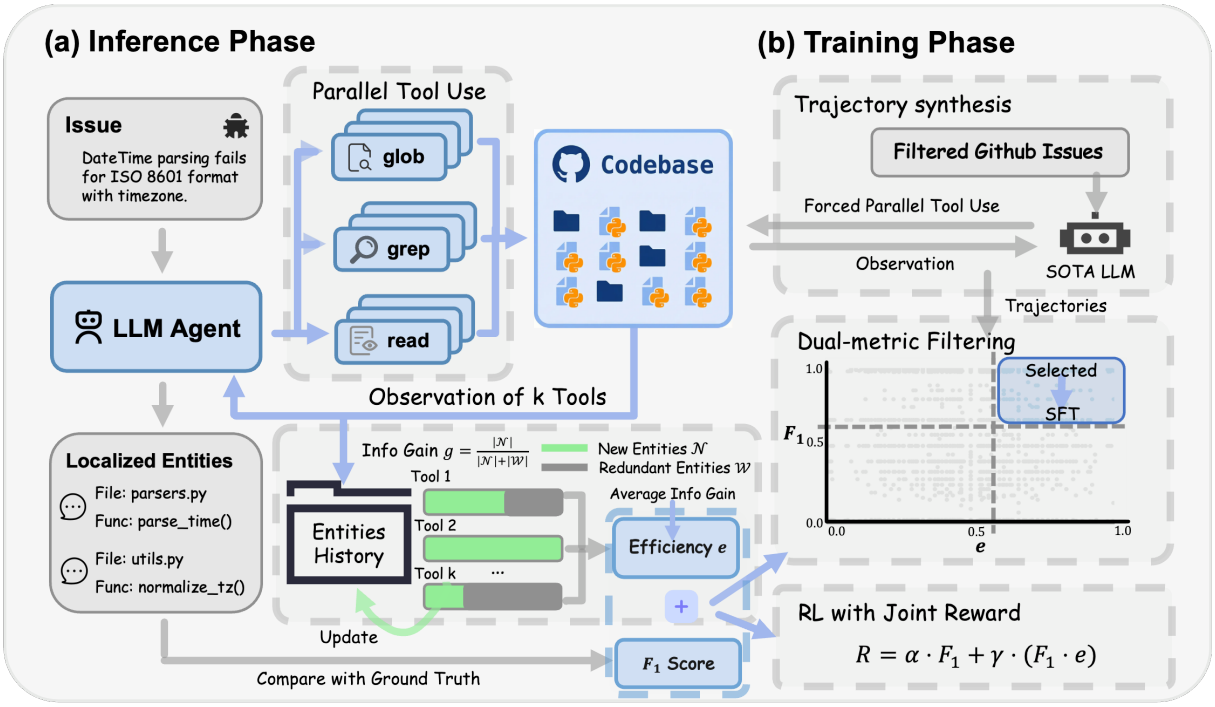

Figure 2: FuseSearch framework overview. (a) **Inference**: Agent executes three minimalist tools in parallel, with each tool's information gain tracked to compute trajectory efficiency e. (b) **Training**: Dual-metric filtering selects high-quality trajectories for SFT, followed by RL optimization with joint F_1-efficiency reward.

$$e = \frac{1}{k} \sum_{i=1}^{k} g_i \tag{4}$$

where k is the total number of tool calls. This metric credits tools proportionally to their novel contributions: higher e indicates exploration discovering distinct code regions, while lower e reveals wasteful redundancy.

Joint Optimization Objectives We formulate the agent learning objective as jointly maximizing localization quality F_1 and tool efficiency e.

This joint objective induces mutually reinforcing optimization dynamics. High F_1 requires collecting sufficient yet focused information: insufficient exploration yields incomplete coverage, while excessive unfocused exploration creates context overload that degrades answer precision. High e requires each tool to explore distinct regions of the search space, avoiding redundant queries.

These objectives are complementary rather than conflicting. An agent cannot achieve high e through low-information tools—empty results or duplicated observations yield zero information gain. However, high e alone is insufficient: brute-force strategies like sequentially reading distinct but irrelevant files achieve high efficiency (each file is novel) yet produce low F_1, as context accumulation with off-target exploration degrades answer precision. The joint objective thus enforces *focused, non-redundant exploration*: agents must efficiently locate relevant code without wasteful detours.

3.3 Joint Quality-Efficiency Training

To optimize the dual objectives defined in Section 3.2—achieving high localization quality F_1 and search efficiency e—we employ a two-stage training approach. SFT provides initial capabilities for parallel tool execution and establishes a strong baseline for both metrics. RL then refines the policy to jointly maximize F_1 and e through strategic exploration.

Training Data Construction We construct a repository-level code localization dataset from 233 high-quality GitHub repositories, ensuring no overlap with our evaluation benchmarks. To ensure data quality, we exclude samples where (1) patches introduce entirely new files or functions, (2) issue descriptions are incomplete or overly brief, or (3) no code changes occur. From ∼21K filtered samples, we extract ground truth localization targets as the modified files, functions/methods, and line ranges from each patch.

Quality-Efficiency Guided Fine-Tuning Base language models exhibit weak parallel tool usage capabilities, occasionally generating at most a few

tool calls per turn. To bootstrap reliable parallel execution while ensuring high initial quality and efficiency, we perform SFT on trajectories filtered by both F_1 and e metrics.

We use a capable teacher model (Kimi-K2-Instruct) to synthesize training trajectories. Since even advanced models exhibit inconsistent parallel behavior, we employ system-level guidance to increase parallel execution frequency: for 6K randomly sampled training queries, we generate multiple trajectories per query, each explicitly guided to use 2-8 tools per turn, yielding approximately 24K candidate trajectories. We then apply dual-metric filtering, retaining only trajectories satisfying:

$$F_1 \geq \rho_F \quad \text{and} \quad e \geq \rho_e \tag{5}$$

This filtering ensures demonstration data exhibits both accurate localization and high tool efficiency. The resulting ∼6K high-quality trajectories are used for fine-tuning Qwen3 base models (4B and 30B-A3B).

The resulting SFT models serve dual purposes: (1) they reliably generate parallel tool calls (2-8 tools per turn), addressing the base model's limited parallel execution capability, and (2) they provide high-quality initialization with reasonable F_1 and e values, enabling effective trajectory sampling during subsequent RL training.

RL with Joint Reward Building on the SFT initialization, we apply group relative policy optimization (GRPO) (Shao et al., 2024) to further optimize both localization quality and search efficiency. GRPO samples multiple outputs per query, computes advantages based on reward signals, and updates the policy to favor high-reward behaviors while maintaining proximity to a reference policy through KL regularization.

To jointly optimize localization quality (F_1) and search efficiency (e), we consider a general reward function encompassing both linear and interactive contributions:

$$R(\tau) = \alpha \cdot F_1(\tau) + \beta \cdot e(\tau) + \gamma \cdot (F_1(\tau) \cdot e(\tau)) \tag{6}$$

where $\alpha, \beta, \gamma \geq 0$ are weighting coefficients.

For code localization, we impose a strict boundary condition: a trajectory that fails to identify relevant code ($F_1 = 0$) provides zero utility, regardless of how "efficiently" it executed. This constraint necessitates setting $\beta = 0$, yielding:

$$R(\tau) = \underbrace{\alpha \cdot F_1(\tau)}_{\text{Base Guarantee}} + \underbrace{\gamma \cdot (F_1(\tau) \cdot e(\tau))}_{\text{Efficiency Bonus}} \tag{7}$$

The linear term guarantees a baseline reward for correct localization, preventing the vanishing reward problem when efficiency is low. The interaction term acts as a soft gate, amplifying the reward only when high quality is achieved with high efficiency.

In practice, F_1 is computed as a weighted combination of file-level and function-level localization accuracy:

$$F_1 = \lambda_{\text{file}} \cdot F_1^{\text{file}} + \lambda_{\text{func}} \cdot F_1^{\text{func}} \tag{8}$$

where F_1^{file} and F_1^{func} measure precision and recall at their respective granularities. The efficiency metric e is computed as defined in Section 3.2. By explicitly coupling search efficiency with localization quality, this objective aligns the RL signal with our dual goals, encouraging the model to maximize information gain per action without compromising the validity of the final result.

4 Experiments

Datasets We evaluate on SWE-bench Verified (Jimenez et al., 2024), a curated benchmark for repository-level issue resolution. Following Suresh et al. (2025), we exclude examples where patches introduce entirely new files or functions, retaining 386 of 500 examples.

Metrics We evaluate localization quality using precision, recall, and F_1 scores at both file-level and function-level granularities. We measure search cost through wall-clock time ($T(s)$), interaction turns (#Turn), and total tokens consumed (Tok.(k)) per instance, capturing both latency and computational overhead (averaged over three runs).

Baselines We compare against three categories: (1) **Workflow-based**: Agentless (Xia et al., 2024); (2) **Agent-based**: LocAgent (Chen et al., 2025), CoSIL (Jiang et al., 2025), and RepoSearcher (Ma et al., 2025). Implementation details are provided in Appendix C.

4.1 Overall Performance

Table 1 presents our main results on SWE-bench Verified. We highlight three key findings:

Parallel vs. Sequential Execution Comparing sequential and parallel execution modes with identical toolsets, parallel invocation achieves comparable or superior localization quality while significantly reducing search time (e.g., 60% on Haiku

Model	Method	File (%)			Func (%)			#Turn	T(s)	Tok.(k)	Mode
		P	R	F_1	P	R	F_1				
Proprietary Models											
Haiku 4.5	Agentless	38.82	91.71	54.55	21.48	61.37	31.83	2.00	7.32	10.6	Seq
	CoSIL	19.62	96.63	32.62	18.37	69.41	29.05	7.22	38.7	**53.0**	Seq
	LocAgent	61.57	87.56	72.30	41.29	65.49	50.64	17.3	318	567	Seq
	RepoSearcher	19.55	97.41	32.57	20.33	67.65	31.26	19.7	114	193	Seq
	FuseSearch*	86.38	62.44	72.48	66.30	47.45	55.31	23.9	90.3	270	Seq
	FuseSearch	73.54	94.50	**82.71**	48.58	70.61	**57.56**	**6.24**	**36.2**	110	Par
Open-Source Models											
Kimi-K2	Agentless	34.53	92.23	50.25	28.22	56.27	37.59	2.00	11.8	8.33	Seq
	CoSIL	21.33	95.60	34.88	23.04	70.20	34.69	6.8	58.8	94.0	Seq
	LocAgent	55.39	95.85	70.21	33.10	73.33	45.61	14.9	261	447	Seq
	RepoSearcher	20.61	96.11	33.94	25.20	72.35	37.39	15.6	94.5	108	Seq
	FuseSearch*	77.33	79.53	78.42	51.87	49.02	50.40	15.0	71.3	216	Seq
	FuseSearch	75.11	89.31	**81.60**	51.00	54.90	**52.88**	**7.92**	**43.6**	**62.1**	Par
Qwen3-4B	Agentless	28.11	76.68	41.14	11.62	34.12	17.33	2.00	4.24	8.33	Seq
	CoSIL	21.21	94.82	34.66	18.17	65.49	28.45	10.8	50.8	63.9	Seq
	LocAgent	39.10	56.22	46.12	27.37	31.18	26.97	6.09	109	135	Seq
	RepoSearcher	31.99	47.15	38.12	16.92	30.39	21.74	14.8	85.3	99.2	Seq
	FuseSearch (base)	64.75	64.25	64.50	43.95	34.90	38.91	**4.24**	6.12	47.9	Par
	FuseSearch (train)	83.59	85.75	**84.65**	59.91	53.33	**56.43**	4.78	**5.43**	**30.9**	Par
Qwen3-30B	Agentless	24.22	87.56	38.19	15.34	55.10	24.00	2.00	8.04	32.0	Seq
	CoSIL	21.22	96.11	34.77	18.40	66.86	28.86	11.1	49.2	66.8	Seq
	LocAgent	45.16	62.95	52.59	29.96	32.75	31.32	11.4	112	136	Seq
	RepoSearcher	40.25	50.78	44.90	20.07	34.71	25.43	16.7	92.4	113	Seq
	FuseSearch (base)	70.41	79.53	74.70	53.27	46.27	45.65	7.50	14.9	80.1	Par
	FuseSearch (train)	83.12	82.90	**83.01**	66.58	52.35	**58.62**	**5.77**	**10.6**	**43.2**	Par

Table 1: Localization performance and efficiency comparison on SWE-bench Verified. For agent-based and workflow-based methods, we evaluate using Kimi-K2-Instruct (abbr. as Kimi-K2) and Claude Haiku 4.5 (abbr. as Haiku 4.5). For FuseSearch, we compare base Qwen3 models (Qwen3-4B-Instruct and Qwen3-30B-A3B-Instruct) with their trained counterparts. FuseSearch* denotes using the FuseSearch framework with sequential prompts to contrast the two execution modes.

4.5) and requiring substantially fewer interaction turns. This demonstrates that parallelization provides efficiency gains beyond mere latency reduction—simultaneous information gathering enables better-informed decisions at each search step.

Minimalist Toolset Effectiveness Even in sequential mode, our minimalist toolset achieves competitive performance compared to specialized agent-based methods with graph navigation or AST parsing. This indicates that language-agnostic primitives suffice for effective code localization, while being simpler to deploy and maintain.

Training Effects Targeted training with joint F_1 and efficiency optimization substantially improves both precision and recall. Our trained Qwen3 models (4B and 30B) achieve 83-84% file F_1 and 56-58% function F_1, matching Claude Haiku 4.5's performance while being significantly faster and more token-efficient. Additional evaluation on LocBench (Chen et al., 2025) further confirms the superior performance of our trained models (Appendix D).

Stage	File F_1	Func F_1	e	#Turn	#Tool	T(s)	Tok.(k)
Qwen3-4B							
Base	64.50	38.91	59.50	4.24	1.63	6.12	47.9
RL	70.11	40.18	54.01	**3.16**	3.44	7.10	31.7
SFT	78.86	47.94	68.46	4.96	3.59	9.17	54.8
SFT+RL	**84.65**	**56.43**	**69.00**	4.78	2.15	**5.43**	**30.9**
Qwen3-30B-A3B							
Base	74.70	45.65	54.92	7.50	1.65	14.9	80.1
RL	79.17	47.67	49.21	**4.23**	1.24	6.63	51.0
SFT	81.13	51.17	59.80	5.49	3.40	11.7	65.2
SFT+RL	**83.01**	**58.62**	**64.53**	5.77	3.44	**10.6**	**43.2**

Table 2: Progressive training effects. SFT establishes parallel tool usage from high-quality trajectories, while RL refines search strategies through F_1 optimization.

4.2 Training Analysis

Table 2 reveals key insights about our training framework.

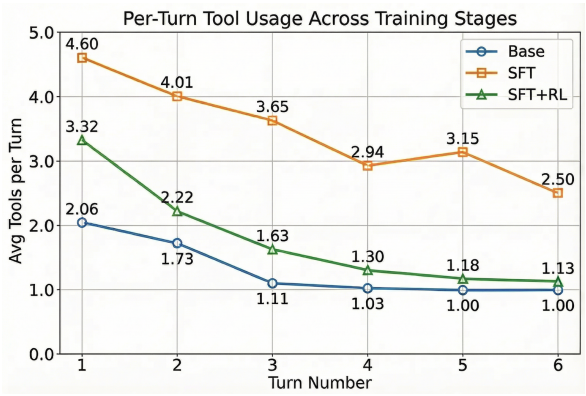

Figure 3: Evolution of average tools per turn across training stages. RL learns adaptive parallelism: high initial exploration transitioning to focused refinement.

Complementary Training Stages The two-stage training exhibits clear complementarity. SFT establishes parallel tool usage capability and substantially improves F_1, but introduces redundancy that degrades efficiency and increases search cost. RL then resolves this trade-off through joint optimization: it further improves F_1 while simultaneously recovering efficiency and reducing time cost. This validates our design—neither stage alone achieves optimal quality-efficiency balance, but their combination enables effective parallelization without sacrificing precision.

Evolving Parallel Strategies Figure 3 reveals how parallel search strategies evolve across training. Base models use minimal parallelism; SFT shifts to uniformly aggressive parallelism, explaining both recall gains and efficiency drops. RL produces qualitatively different behavior: adaptive parallelism that begins with broad exploration and rapidly transitions to focused refinement. This breadth-first-to-depth-first pattern emerges from joint optimization, demonstrating that models learn not just to parallelize, but when and how much.

4.3 Ablation Studies

We validate key design choices through systematic ablations on Qwen3-4B.

Parallel vs. Sequential Execution To isolate execution mode impact, we train sequential variants (1 tool/turn) with identical data and configurations. Table 3 shows parallel execution consistently outperforms sequential. Sequential requires nearly twice as many turns yet achieves lower F_1 and consumes more tokens. This confirms parallel execution offers fundamental advantages beyond latency—simultaneous information gathering enables better decisions at each step.

Mode	Stage	File F_1	Func F_1	#Turn	T(s)	Tok.(k)
Seq	SFT	74.02	47.15	9.82	10.42	95.9
	SFT+RL	78.82	50.21	7.52	8.03	59.4
Par	SFT	78.86	47.94	4.96	9.17	54.8
	SFT+RL	84.65	56.45	5.60	5.43	30.9

Table 3: Parallel vs sequential execution with identical training configurations. Parallel consistently outperforms sequential across both training stages.

Filtering	File F_1	Func F_1	e	T(s)	Tok.(k)
No filtering	75.44	43.52	55.77	9.84	60.7
Filter F_1	78.55	45.43	56.72	10.53	73.2
Filter e	76.74	42.63	60.14	12.94	61.8
Joint filtering	78.86	47.94	62.03	9.17	54.8

Table 4: SFT performance under different filtering strategies. Joint filtering achieves optimal F_1 and efficiency simultaneously.

SFT Data Filtering We evaluate four filtering strategies, each yielding 6K trajectories for SFT: (A) no filtering, (B) F_1-only filtering, (C) efficiency-only filtering, (D) joint filtering (ours). Table 4 shows joint filtering produces SFT models with superior F_1 and efficiency simultaneously. Single-metric filtering improves its target metric but degrades the other, while unfiltered data yields suboptimal initialization on both. This validates that dual-metric filtering provides high-quality starting points for subsequent RL optimization.

Reward Design We compare three RL reward formulations: (A) F_1 only, (B) additive $F_1 + e$, (C) multiplicative $F_1 + F_1 \cdot e$ (ours). Table 5 reveals distinct trade-offs. F_1-only optimization improves over SFT with reduced token usage but moderate time reduction. Additive $F_1 + e$ achieves highest efficiency, but reduced search quality necessitates more tool invocations, increasing both time and token costs. Our multiplicative reward delivers the best F_1, highest efficiency, and lowest time and token costs simultaneously. This validates the multiplicative reward structure for practical deployment scenarios where both quality and efficiency matter.

4.4 Downstream Task Applications

To evaluate FuseSearch's practical impact, we test its effectiveness in assisting Kimi-K2-Instruct on SWE-bench Verified issue resolution. We compare three configurations: (A) **No Localization**, where the main agent performs full-stack exploration; (B) **Pre-Search**, where FuseSearch-4B conducts ini-

Reward Type	File F_1	Func F_1	e	T(s)	Tok.(k)
SFT	78.86	47.94	62.03	9.17	54.8
F_1 only	81.84	54.90	59.66	7.28	39.4
$F_1 + e$	79.22	51.98	66.62	9.40	45.7
$F_1 + F_1 \cdot e$ (ours)	84.65	56.45	69.00	5.43	30.9

Table 5: Reward design ablation. The composite reward balances quality and efficiency optimally.

Method	Pass Rate (%)	#Turn	T(s)	Tok.(k)
No Localization	68.4	41.1	312	1053
Pre-Search	68.1	31.6	223	562
Sub-Agent	68.7	31.9	290	713

Table 6: Impact of FuseSearch-4B on Kimi-K2's issue resolution performance. Localization reduces token consumption and time while maintaining quality.

tial localization before the main agent begins; and (C) **Sub-Agent**, where the main agent dynamically invokes FuseSearch-4B during task execution.

Table 6 shows that both localization modes maintain comparable pass rates while substantially reducing the main agent's token consumption and total inference time. Pre-search mode offers the most efficient configuration, demonstrating that fast, accurate localization models can significantly accelerate downstream task completion without sacrificing solution quality.

5 Related Work

5.1 Code Localization Methods

Recent LLM-based approaches divide into two paradigms. **Workflow methods** like Agentless (Xia et al., 2024) employ fixed hierarchical strategies that progressively narrow search scope from files to functions, but lack adaptability to varying task complexity. **Agent-based methods** enable flexible multi-step exploration: graph-guided approaches like LocAgent (Chen et al., 2025) and CoSIL (Jiang et al., 2025) represent code as static or dynamic graphs for navigation, while general agents like OpenHands (Wang et al., 2025) and SWE-agent (Yang et al., 2024) use bash-like interfaces for repository traversal.

However, existing agents execute tool calls sequentially, leading to prolonged search duration when comprehensive exploration is needed. Graph-based methods further require language-specific preprocessing infrastructure that limits generalization across programming languages. We address these limitations through a minimalist parallel execution framework that accelerates search via concurrent tool invocation while eliminating infrastructure dependencies.

5.2 Parallel Tool Use and Agent Training

Parallel tool execution has emerged as a strategy to accelerate multi-step search processes. Commercial systems like SWE-grep (Pan et al., 2025) train specialized retrieval models with basic weighted F_1 rewards to issue fixed parallel calls, achieving speedups but requiring expensive inference infrastructure (Cerebras at 2800+ tokens/s). In web search, recent work trains models to distinguish parallelizable from sequential queries: HybridDeepSearcher (Ko et al., 2025) fine-tunes models on synthetic hybrid-hop QA data to reduce search turns, while ParallelSearch (Zhao et al., 2025) and RAG-R1 (Tan et al., 2025) apply RL with query decomposition for parallel execution.

However, existing work optimizes primarily for accuracy, leading to wasteful tool invocations that contribute nothing to search progress, or incomplete utilization of discovered information. While some approaches penalize trajectory length (Zelikman et al., 2024), this does not directly measure tool usage quality or distinguish between effective and redundant exploration.

We address this gap by introducing tool efficiency—the ratio of tools that discover new code entities to total tools invoked—as a metric that directly penalizes redundant exploration while encouraging focused information gathering. Our compact models achieve strong performance without specialized hardware by co-optimizing localization quality and search efficiency through efficiency-based RL rewards.

6 Conclusion

We introduced **FuseSearch**, a code localization agent that achieves superior accuracy-efficiency trade-offs through learned adaptive parallel execution. By optimizing **tool efficiency**—rewarding information novelty while penalizing redundancy—via SFT and RL, FuseSearch-4B achieves 84.7% file-level F_1 while completing searches 93.6% faster with 67.7% fewer turns. As preprocessing for downstream repair tasks, it reduces interaction turns by 23.1% and completion time by 28.5%. Our results demonstrate that efficiency-aware training enables accurate yet computationally practical agents—a crucial step toward production-grade automated software development.

7 Limitations

Our work is subject to limitations in current evaluation frameworks. First, ground truth derived from golden patches represents only one valid solution, potentially missing alternative correct localizations. Second, available benchmarks predominantly cover Python repositories (SWE-bench Verified); while our toolset is language-agnostic, assessing effectiveness on statically-typed languages like Java or C++ requires more diverse benchmarks and training data. Third, existing benchmarks focus exclusively on issue-driven localization, whereas code search is fundamental to broader scenarios such as repository question answering, code comprehension, and documentation generation—contexts our approach has not been evaluated on. These constraints highlight the need for comprehensive localization benchmarks spanning diverse tasks and languages.

References

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

Pengfei Gao and Chao Peng. 2025. More with less: An empirical study of turn-control strategies for efficient coding agents. *arXiv preprint arXiv:2510.16786*.

Zhonghao Jiang, Xiaoxue Ren, Meng Yan, Wei Jiang, Yong Li, and Zhongxin Liu. 2025. Issue localization via llm-driven iterative code graph searching. *Preprint*, arXiv:2503.22424.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

Dayoon Ko, Jihyuk Kim, Haeju Park, Sohyeon Kim, Dahyun Lee, Yongrae Jo, Gunhee Kim, Moontae Lee, and Kyungjae Lee. 2025. Hybrid deep searcher: Integrating parallel and sequential search reasoning. *Preprint*, arXiv:2508.19113.

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

Weichen Li, Xiaotong Huang, Jianwu Zheng, Zheng Wang, Chaokun Wang, Li Pan, and Jianhua Li. 2025. rllm: Relational table learning with llms. *Preprint*, arXiv:2407.20157.

Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. 2025. Tool-integrated reinforcement learning for repo deep search. *Preprint*, arXiv:2508.03012.

Ben Pan, Carlo Baronio, Albert Tam, Pietro Marsella, Mokshit Jain, Daniel Chiu, Swyx, and Silas Alberti. 2025. Introducing swe-grep and swe-grep-mini: Rl for multi-turn, fast context retrieval.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *Preprint*, arXiv:2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*.

Tarun Suresh, Revanth Gangi Reddy, Yifei Xu, Zach Nussbaum, Andriy Mulyar, Brandon Duderstadt, and Heng Ji. 2025. Cornstack: High-quality contrastive data for better code retrieval and reranking. *Preprint*, arXiv:2412.01007.

Zhiwen Tan, Jiaming Huang, Qintong Wu, Hongxuan Zhang, Chenyi Zhuang, and Jinjie Gu. 2025. Rag-r1: Incentivizing the search and reasoning capabilities of llms through multi-query parallelism. *Preprint*, arXiv:2507.02962.

Qwen Team. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025. Openhands: An open platform for ai software developers as generalist agents. *Preprint*, arXiv:2407.16741.

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

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D. Goodman. 2024. Quiet-star: Language models can teach themselves to think before speaking. *Preprint*, arXiv:2403.09629.

Shu Zhao, Tan Yu, Anbang Xu, Japinder Singh, Aaditya Shukla, and Rama Akkiraju. 2025. Parallelsearch: Train your llms to decompose query and search sub-queries in parallel with reinforcement learning. *Preprint*, arXiv:2508.09303.

A Parallel Execution Framework

A.1 Inference Algorithm

Algorithm 1 describes the inference process of FuseSearch. At each turn, the model generates an action a that may contain multiple tool calls formatted as JSON objects. The Parse function extracts these tool calls $\{c_1, \ldots, c_n\}$, which are then executed concurrently by the environment. The aggregated observations $\{o_1, \ldots, o_n\}$ are appended to the trajectory before the next model invocation. This cycle continues until the model produces a final answer without tool calls.

Algorithm 1 Inference Process of FuseSearch

Require: Query q, model $\mathcal{M}$, environment $\mathcal{E}$
Ensure: Localized files $\mathcal{F}$

Initialize trajectory $\tau \leftarrow \emptyset$
while True **do**
  $a \leftarrow \mathcal{M}(q, \tau)$ // Generate action
  $\tau \leftarrow \tau \oplus a$ // Append action to trajectory
  if no <tool_call> in a **then**
    return $answer \leftarrow$ Extract(a)
  end if
  $\{c_1, c_2, \ldots, c_n\} \leftarrow$ Parse(a)
  $\{o_1, o_2, \ldots, o_n\} \leftarrow \mathcal{E}$.step($\{c_1, \ldots, c_n\}$)
  $\tau \leftarrow \tau \oplus \{o_1, o_2, \ldots, o_n\}$
end while

A.2 Tool Specifications

Table 7 summarizes the core parameters of our three tools. Each tool is implemented as a function with a JSON-formatted parameter schema, enabling models to invoke them through structured tool calls.

Tool	Required	Optional
`read_file`	`path`	`start_line`, `end_line`
`grep`	`pattern`	`path`, `glob`, `output_mode`
`glob`	`pattern`	`path`

Table 7: Core parameters for the three minimalist tools. All paths are absolute.

read_file Reads file contents with optional line range specification. The `path` parameter specifies the absolute file path. When `start_line` and `end_line` are provided, only the specified range is returned; otherwise, the entire file is read (up to a default limit of 1000 lines).

grep Performs regex-based content search built on ripgrep. The `pattern` parameter accepts full regex syntax. The optional `output_mode` controls result format: `files_with_matches` (default) returns only file paths, `content` returns matching lines with context, and `count` returns match counts per file. The `glob` parameter filters files by pattern (e.g., `*.py`), while `path` restricts search to a specific directory.

glob Matches files by name patterns. The `pattern` parameter accepts standard glob syntax (e.g., `**/*.js` for recursive search, `test_*.py` for prefix matching). Results are limited to 100 file paths to prevent overwhelming context windows.

B Training Configuration

B.1 Data Split

From the collected repository-level localization dataset of approximately 21K issue-patch pairs, we allocate 6K samples for SFT and the remaining 15K samples for RL. The SFT subset undergoes trajectory synthesis and dual-metric filtering as described in Section 3.3, yielding approximately 6K high-quality demonstration trajectories.

B.2 SFT

We fine-tune Qwen3-4B-Instruct and Qwen3-30B-A3B-Instruct on the filtered trajectories for 1 epoch with a batch size of 32. Both models are trained on 8×NVIDIA H20 GPUs(96GB). We use AdamW optimizer with a learning rate of 2e-5 and linear warmup for the first 10% of training steps. The maximum sequence length is set to 32768 tokens to accommodate long repository contexts and multi-tool trajectories.

B.3 RL

Infrastructure We employ vLLM (Kwon et al., 2023) as the inference engine to accelerate trajectory sampling during policy rollouts. The training framework is built on RLLM (Li et al., 2025), which leverages veRL (Sheng et al., 2024) for distributed RL.

Sampling Configuration To ensure diversity in trajectory exploration, we set the sampling temperature to 0.7 during rollouts. For each training

instance, we sample 8 trajectories (rollout=8) to compute advantage estimates for GRPO updates.

Training Hyperparameters Table 8 summarizes the key hyperparameters for GRPO training. We use a per-GPU batch size of 32, yielding a global batch size of 256 across 32 NVIDIA H20 GPUs(96GB) with rollout factor 8. The prompt length is capped at 49152 tokens to accommodate extensive repository context, while response length is limited to 32768 tokens for tool call sequences and reasoning. We train for 1 epoch over the 15K RL training samples.

Hyperparameter	Value
Training batch size	32
Rollout per instance	8
Global batch size	256
Sampling temperature	0.7
Max prompt length	49152
Max response length	32768
Training epochs	1
Training samples	15K
Learning rate	1e-6
KL coefficient (β)	0.01

Table 8: Hyperparameters for GRPO-based RL.

Reward Coefficients For the joint reward function $R(\tau) = \alpha \cdot F_1(\tau) + \gamma \cdot (F_1(\tau) \cdot e(\tau))$, we set $\alpha = 0.8$ and $\gamma = 0.2$. The F_1 score is computed as $F_1 = 0.7 \cdot F_1^{\text{file}} + 0.3 \cdot F_1^{\text{func}}$, placing higher weight on file-level accuracy while still incentivizing function-level precision.

C Baseline Implementation Details

We compare FuseSearch against four representative code localization frameworks, each employing distinct strategies and infrastructure requirements.

Agentless (Xia et al., 2024) adopts a hierarchical pipeline approach without maintaining agent state across turns. It performs localization in three sequential stages: (1) file-level filtering using keyword matching and LLM ranking, (2) class/function identification within selected files, and (3) fine-grained line-level localization. This workflow-based design eliminates the need for complex reasoning chains but requires careful tuning of each pipeline stage.

CoSIL (Jiang et al., 2025) implements an iterative agent that dynamically constructs module call graphs during exploration. Starting from entry points identified by keyword search, it progressively expands the graph by analyzing function invocations and dependencies. The agent employs context pruning to manage token limits, discarding less relevant code paths based on semantic similarity to the issue description. This dynamic graph construction enables adaptive exploration without requiring pre-built static analysis infrastructure.

LocAgent (Chen et al., 2025) pre-processes repositories into directed heterogeneous graphs encoding file, class, and function nodes along with their structural relationships (imports, inheritance, invocations). The agent navigates this graph through node-visiting actions, leveraging graph connectivity to perform multi-hop reasoning. This approach requires upfront graph construction but enables efficient traversal of complex dependency chains.

RepoSearcher (Ma et al., 2025) provides a lightweight tool suite (GetRepoStructure, SearchClass, SearchFunction, SearchClassMethod) for direct code retrieval without graph preprocessing. The agent iteratively invokes tools to gather relevant context, terminating via an explicit Exit action. This minimalist design reduces infrastructure overhead while maintaining competitive localization performance.

C.1 Model Deployment

For proprietary models (Claude-3.5-Sonnet, Kimi-K2-Instruct), we invoke their official APIs with temperature set to 0 for deterministic outputs. For open-source models (Qwen3-4B-Instruct, Qwen3-30B-A3B-Instruct), we deploy local inference servers using vLLM (Kwon et al., 2023) on 8×NVIDIA H20 GPUs. We set tensor parallelism to 8 for distributed inference and enable continuous batching to maximize throughput. All models use their default system prompts and tool-calling formats as specified in their official documentation.

D Additional Experiments

D.1 Evaluation on LocBench

To further validate the generalization of FuseSearch across different benchmarks, we conduct additional evaluation on LocBench (Chen et al., 2025), a localization-focused benchmark designed to assess code localization capabilities. Following the same filtering criteria as SWE-bench Verified (excluding

patches that introduce entirely new files or functions), we retain 456 out of 560 examples for evaluation.

Table 9 presents the performance comparison between FuseSearch-4B (trained model) and Kimi-K2-Instruct on LocBench. Both models use the FuseSearch framework with parallel tool execution. The results demonstrate that our trained 4B model achieves superior localization quality while being significantly more efficient: it improves file-level F_1 by 5.4 points and function-level F_1 by 5.8 points compared to Kimi-K2, while reducing search time by 77% (6.24s vs 27.8s) and token consumption by 35% (37.5k vs 57.9k). The efficiency metric e is also higher (74.06 vs 68.97), indicating that the trained model generates fewer redundant tool calls. These results confirm that our efficiency-aware training approach generalizes well beyond SWE-bench Verified, consistently producing models that achieve better quality-efficiency trade-offs across diverse localization tasks.

Model	File F_1	Func F_1	e	#Turn	T(s)	Tok.(k)
Kimi-K2	70.33	45.65	68.97	6.60	27.8	57.9
FuseSearch-4B	**75.74**	**51.47**	**74.06**	**4.61**	**6.24**	**37.5**

Table 9: Performance comparison on LocBench (456 examples). Both models use the FuseSearch framework with parallel execution.

E Prompt Design

E.1 System Prompt and Output Format

FuseSearch employs a structured prompt design that guides the model to produce localization results in two distinct sections: Locations to Modify and Related Context. Figure 4 illustrates the complete system prompt used during inference.

`Locations to Modify` contains the core localization results—the specific files and functions that require modification to resolve the issue. The model outputs a ranked list of code entities, where higher-ranked items are deemed more likely to be the root cause. All precision, recall, and F_1 scores reported in our experiments are computed based on this section, treating it as the model's primary prediction.

`Related Context` allows the model to include additional code entities that are semantically related to the issue but do not necessarily require direct modification. For example, when localizing a bug in a data validation function, the model might include related utility functions or constants in Related Context even though the fix only requires modifying the validation logic itself. While not used for localization metric calculation, this supplementary information proves valuable for downstream issue resolution tasks: by providing repair agents with a broader context of relevant code, it enables faster and more accurate patch generation without requiring additional exploration turns.

This two-part output design reflects a key insight: effective localization should distinguish between "must-fix" locations (high precision for metrics) and "helpful context" (high utility for downstream tasks). By separating these concerns, FuseSearch simultaneously optimizes for localization accuracy and downstream task efficiency.

```
You are CodeFuse Repo DeepResearch Agent - a read-only codebase analyzer that navigates repositories to localize related code snippets.
**Core Mission**:
**Stage 1 - RECALL** (Up to 4 initial turns):
- Cast a wide net with parallel tool calls
- Prioritize breadth over precision - better to collect more than miss relevant code
**Stage 2 - RERANK** (Final turn):
- Analyze all collected snippets for relevance to user query
- Select top results ordered by relevance for final output

**Capabilities**:
- Find files via glob patterns(Glob Tool)
- Search code with regex (Grep Tool)
- Read file contents(Read_file tool)
**Output Format**: Divided into TWO subsections wrapped in XML tags:

a) **Locations to Modify** :
- Code that needs to be changed to fix the issue
- Focus on functions, methods, classes that require modification
- Wrapped in <locations_to_modify></locations_to_modify> tags
- Format: <absolute_filepath>:<ClassName>.<method_name> or <absolute_filepath>:<function_name>
- Must use absolute file paths
- Each entry on a separate line
- Ordered by relevance (most relevant first)
b) **Related Context** :
- Code necessary to understand the problem but may not need modification
- Wrapped in <related_context></related_context> tags
- Same format as above
- Include: caller/callee functions, base classes, related utilities, relevant tests
- Helps provide self-contained understanding of the issue

**Example Output**:
<locations_to_modify>
/home/user/project/src/auth/login.py:Login.__init__
/home/user/project/src/auth/session.py:Session.start
</locations_to_modify>

<related_context>
/home/user/project/src/auth/base.py:BaseAuth.validate
/home/user/project/src/auth/utils.py:check_credentials
/home/user/project/tests/test_auth.py:test_auth
</related_context>

**Important**:
- Always use absolute file path with function/method name
- For class methods: `filepath:ClassName.method_name`
- For standalone functions: `filepath:function_name`
- Results must be ordered by relevance to the user's query

**Operating Rules**:
1. **READ-ONLY**: Never create, edit, delete, or modify any files
2. **Return absolute paths** in final responses
3. **Two-Stage Approach**: Use Stage 1 (RECALL) to gather broadly, then Stage 2 (RERANK) to refine

You MUST use multiple parallel function calls, return a JSON array of function calls within <tool_call></tool_call> XML tags:
<tool_call>
{{"name": <function-name-1>, "arguments": <args-json-object-1>}}
</tool_call>
<tool_call>
{{"name": <function-name-2>, "arguments": <args-json-object-2>}}
</tool_call>

**Parallel Tool Calling Benefits**:
- Faster exploration by executing multiple searches simultaneously
- Reduce total interaction rounds
- Ideal for: searching multiple patterns, reading multiple files, or combining search + read operations
- Essential for Stage 1 (RECALL) to cast a wide net efficiently
```

Figure 4: System prompt for FuseSearch. The prompt instructs the model to output localization results in two sections: Locations to Modify (required) and Related Context (optional).