GREPSEEK: TRAINING SEARCH AGENTS FOR DIRECT CORPUS INTERACTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large Language Model (LLM) search agents have shown strong promise on knowledge-intensive tasks through iterative reasoning and retrieval. Most existing systems rely on retrievers that return ranked documents from a pre-built index. We explore a complementary paradigm in which the agent treats the corpus as the search environment and finds evidence through executable shell commands. We introduce `GrepSeek`, an optimized direct corpus interaction (DCI) agent that learns to find, filter, and compose evidence over large text corpora. To stabilize reinforcement learning (RL) over large corpora, we train in two stages: first, we initialize the policy using verified, causally grounded search trajectories generated by an answer-aware Tutor and an answer-blind Planner; then, we refine the policy using Group Relative Policy Optimization (GRPO). To make DCI practical at scale, we introduce two semantics-preserving execution optimizations: Pruned Adaptive Command Execution, which reduces shell-based search latency by up to $77\times$ on a 14GB corpus with 21 million documents using a compact auxiliary structure, and Sharded-Parallel Corpus Search, which achieves up to $7.6\times$ speedup without additional preprocessing; both preserve equivalence with sequential execution. Across eight open-domain QA benchmarks, `GrepSeek` achieves the strongest overall performance, with a statistically significant relative improvement of 5.7% over the best baseline. Our analysis shows how DCI-optimized agents conduct flexible and effective compositional search through direct corpus interaction.

1 INTRODUCTION

LLM search agents (Li et al., 2025; Jin et al., 2025) have shown promise in addressing complex information needs that require reasoning, query decomposition, and/or information synthesis from multiple sources. These agents benefit from multiple interactions with a retriever to obtain information for performing their tasks. With unstructured or semi-structured text corpora, interactions are typically in the form of keyword or natural language queries. These paradigms rely on decades of research in developing retrieval models, from lexical (Salton & Buckley, 1988; Robertson et al., 1994; Ponte & Croft, 1998) to semantic scoring based on dense (Deerwester et al., 1990; Karpukhin et al., 2020) or sparse representations (Zamani et al., 2018; Formal et al., 2021). They all operate on precomputed representations of documents to construct an index for the corpus. Relevance scores between the query and documents are also mostly computed per document.

This paper explores a fresh perspective to this problem: *information seeking on unstructured data can be done at any granularity*. In other words, instead of document-level representations, indexing, and relevance scoring, a piece of text of any size can be retrieved for each query. This enables us to perform more *"surgical"* retrieval as opposed to being restricted with pre-determined text chunks and representations. To do this, *we envision search agents that treat the corpus as an environment. Under this view, the agent can issue executable (surgical) search operations over the corpus, inspect results, refine constraints, and compose evidence across multiple steps*. This shifts retrieval from a black-box ranking procedure to an explicit sequence of controllable corpus operations. Such an interface is especially appealing for many knowledge-intensive reasoning tasks in which answering a question may require exact entity matching, lexical filtering, symbolic pattern search, or following bridge entities across documents. This perspective is closely related to recent progress in code agents, where executable search tools such as `grep` provide a simple yet effective interface for locating relevant context in code repositories (Wang et al., 2026). Inspired by this form of tool-mediated search, we ask whether a similar interaction pattern can be extended beyond code repositories to open-domain QA over large textual corpora containing millions of unstructured documents.

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.

Contemporary to our work, Sen et al. (2026) and Li et al. (2026) independently propose prompting methods to bypass precomputed retrieval indices and search a raw corpus through Unix-style shell commands. These works demonstrate that DCI can serve as an effective interface for exact matching, multi-step evidence discovery, and compositional QA. These methods are primarily built around prompting large proprietary models with strong code-generation capabilities to orchestrate search at inference time. For example, Li et al. rely on closed-weight agents such as Claude, making the resulting system computationally expensive and operationally inefficient, often requiring substantial time, sometimes even one hour or more, to complete a single query (Li et al., 2026). In contrast, we are interested in methods that are feasible in the real world, thus focusing on training compact models and efficient operation executions at large scale. In order to be consistent with these contemporary work, we also refer to this category of approaches as *Direct Corpus Interaction (DCI)*.

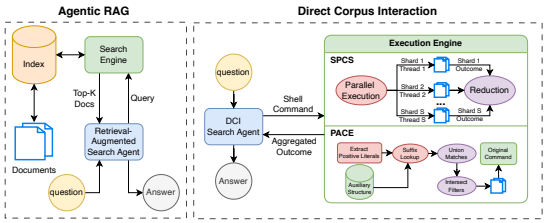

Figure 1: Comparing retrieval-based search and DCI.

To achieve effective and efficient DCI, we introduce `GrepSeek`, an optimized compact DCI agent that learns to search, filter, and compose evidence over large corpora with shell commands. This shifts DCI from inference-time prompting with proprietary LLMs (as in (Li et al., 2026)) to a learned capability of a smaller agent. Training DCI agents is challenging: naively applying RL often produces degenerate behavior, such as overly broad commands, excessive context retrieval, or unstable search behavior. To stabilize it, we first create a cold-start dataset that demonstrates successful DCI behavior. For each training question, an answer-aware Tutor model is given the ground-truth answer and constructs a backward chain of shell commands whose execution retrieves documents supporting the answer. This backward construction is useful for multi-hop questions, as it lets the Tutor model identify supporting evidence one step at a time while maintaining an explicit chain from the final answer back to the question. We then convert the backward chain into a forward, causally valid trajectory using an answer-blind Planner. The Planner generates reasoning traces and commands from the agent's observable history, simulating how the agent would solve the task at inference. The Tutor then aligns the steps with the verified backward chain. This produces trajectories that are causally grounded in the information observed so far, while allowing the Tutor model to guide the Planner towards the verified search path. Fine-tuning on the created cold-start dataset is followed by RL optimization using GRPO (Shao et al., 2024), allowing the DCI agent to improve its search behavior through direct interaction with the corpus.

For DCI to be practical, retrieval latency must remain manageable even when operating over millions of documents. However, executing shell pipelines sequentially over large multi-gigabyte collections introduces substantial I/O and processing bottlenecks, making naive execution slow for agents. We address this with two complementary semantics-preserving optimizations: Pruned Adaptive Command Execution (PACE), which narrows compatible searches to a conservative set of candidate documents before applying the original command, and a sharded-parallel execution engine that distributes compatible shell pipelines across corpus shards. Together, these mechanisms substantially reduce retrieval latency while preserving byte-exact equivalence with sequential execution.

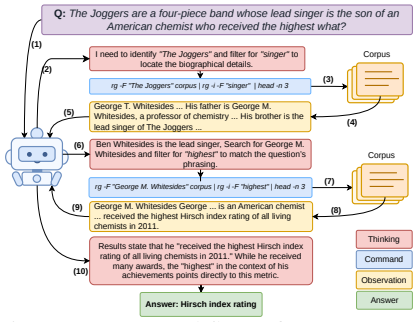

Figure 2: DCI workflow of `GrepSeek`.

To evaluate `GrepSeek`, we conduct experiments across eight knowledge-intensive QA benchmarks: Natural Questions (NQ) (Kwiatkowski et al., 2019), TriviaQA (Joshi et al., 2017), and PopQA (Mallen et al., 2023) for single-hop QA in addition to HotpotQA (Yang et al., 2018), 2WikiMultiHopQA (Ho et al., 2020), MuSiQue (Trivedi et al., 2022), Bamboogle (Press et al., 2023), and BrowseComp-Plus (Chen et al., 2025) for multi-hop QA that need iterative evidence aggregation and reasoning. Experiments show that `GrepSeek` outperforms index-based RAG methods, untrained agentic frameworks, and even search agents optimized with RL to retrieve using dense and sparse retrievers. On average, `GrepSeek` achieves the best overall performance, with a 5.7% statistically significant relative improvement over the strongest baseline, demonstrating DCI as a competitive and practical alternative to search agents that rely on index-based retrieval. Notably, it

achieves the best performance on five out of eight benchmarks—NQ, HotpotQA, 2WikiMultiHopQA, MuSiQue, and BrowseComp-Plus. The gains are pronounced on multi-hop tasks, where traditional retrieval systems frequently suffer from semantic conflation and entity ambiguity introduced by retrievers. We also show that our solutions are efficient for real-world applications. On a 14GB corpus containing 21 million documents, PACE reduces average retrieval latency from 5.39 to 0.07 seconds per query without sharding, yielding a $77\times$ speedup, at the cost of approximately 21 minutes of one-time preprocessing to construct its 2.8GB auxiliary structure. In contrast, our sharded-parallel execution requires essentially no additional preprocessing and reduces latency to 0.71 seconds with 32 shards, yielding a $7.6\times$ speedup. Thus, PACE is preferable when one-time preprocessing is affordable, while sharded execution provides substantial acceleration when additional preprocessing is undesirable. We will publicly release our codebase, the cold-start data, and model checkpoints.

2 Optimizing Direct Corpus Interaction Search Agents

Concurrent and independent of our work, Li et al. (2026) introduce DCI, where an agent bypasses precomputed retrieval indices and searches the corpus by Unix commands. This treats DCI as a prompting strategy, using strong code-generation LLMs such as Claude, making search computationally expensive and slow, taking up to an hour per query. Conversely, we study how to train smaller search agents to learn DCI capability, enabling effective corpus interaction through learned behavior.

DCI Search Agent: Figure 1(Right) provides an overview of the DCI agent and Figure 2 shows a trajectory. DCI search agent π_θ operates under ReAct (Yao et al., 2023). Given a question q and the system prompt in Figure 7 in Appendix C, the agent interacts directly with a corpus $\mathcal{C}$,[1] where each line corresponds to a document. The interaction proceeds for at most T steps, producing a trajectory $\tau = \{(t_i, a_i, o_i)\}_{i=1}^{T}$, where t_i denotes the reasoning trace, a_i the action, and o_i the observation. At step i, conditioned on q and the previous actions and observations $\tau_{<i}$, the policy π_θ generates a reasoning trace and an action: $(t_i, a_i) \sim \pi_\theta(\cdot \mid q, \tau_{<i})$. Reasoning traces are generated within `<think>` XML tags. Actions corresponding to tool invocations are emitted using the Hermes-style `<tool_call>` format, and the tool outputs are returned to the model within `<tool_response>` tags. When the agent decides to terminate, it produces the answer $\hat{y}_q$ within `<answer>` tags. The action a_i is either a corpus interaction command[2] (i.e., a shell command) or a termination that outputs the answer. Following each command, an execution engine runs the command over the corpus $\mathcal{C}$ and returns an observation o_i, which is appended to the trajectory and used for subsequent reasoning and action generation. The remainder of this section describes the training and efficient tool execution.

2.1 Training DCI Search Agent

We find that directly optimizing corpus interaction with RL is unstable: the agent generates ineffective commands and retrieves excessively large corpus segments, increasing context length and destabilizing training.[3] We adopt a two-stage procedure. First, we construct a cold-start dataset to teach effective DCI behavior. We then supervise the model on this data before RL optimization.

2.1.1 Cold-Start Data Generation

Given a dataset $D = \{(q_i, y_i)\}_{i=1}^{|D|}$ of question-answer pairs requiring information from the corpus $\mathcal{C}$, our data pipeline (Figure 3 and Algorithm 1 in Appendix C) consists of two main phases followed by a filtering stage. The process uses an answer-aware Tutor ($\mathcal{M}_T$) to construct verified evidence chains, and an answer-blind Planner ($\mathcal{M}_P$) to synthesize realistic forward reasoning trajectories.

Backward Phase: The Tutor decomposes the query q and answer y into a sequence of sub-queries (Algorithm 1, line 1; prompt in Figure 9 in Appendix C). To ensure the agent learns genuine information-seeking behavior rather than exploiting access to the answer, we construct the retrieval

[1] The corpus need not be stored as a single physical file. We expose it to the agent as one logical file for simplicity, while the execution engine maps commands to the underlying files and executes them accordingly.

[2] The action space consists of Unix tools, such as `rg`, `grep`, `find`, `sed`, `awk`, `head`, `tail`, `cat`, `ls`, `wc`, `sort`, `cut`, `uniq`, and `tr`. In practice, the agent primarily relies on `rg` and `head`.

[3] This frequently caused VRAM and host RAM OOM failures, even on systems with up to 1024 GB of RAM.

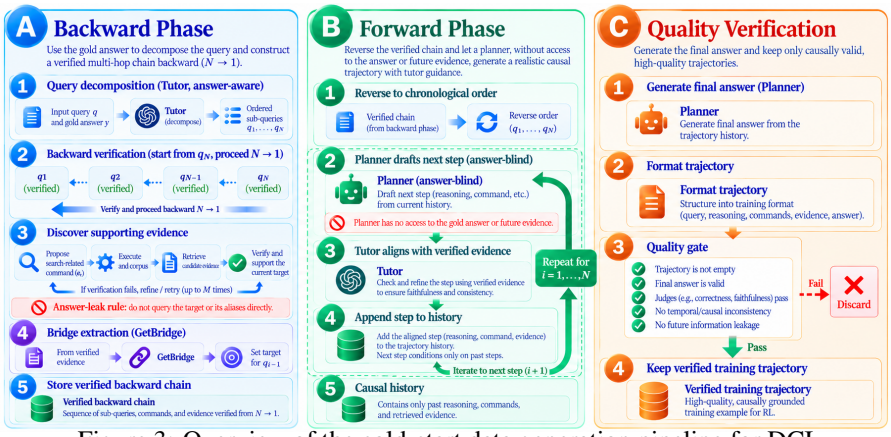

Figure 3: Overview of the cold-start data generation pipeline for DCI.

trajectory in reverse ($N \to 1$; lines 3–7). At each backward step, the Tutor proposes a shell command c_i to retrieve a document d_i that entails the current target answer a (lines 4 and 22). Crucially, we enforce an *answer-leak rule* during command generation (prompt in Figure 19 in Appendix C): the proposed command must be target-masked, forbidding the use of the target a or any of its aliases F as retrieval terms. This is necessary as the backward process has access to future information through y; without masking, it can retrieve evidence by querying the answer, resulting in unrealistic retrieval behavior that doesn't reflect causal inference. To improve retrieval robustness, the Tutor is allowed up to M attempts (lines 21–25). At each attempt, the Tutor proposes a command, executes it, and verifies whether retrieved documents support the answer using a verification step (line 23; prompt in Figure 10 in Appendix C). This increases the likelihood of obtaining valid evidence while filtering out brittle or spurious retrieval trajectories. Once at least one valid document is identified, a bridge extraction step determines the antecedent entity in d_i that answers the preceding sub-query q_{i-1} (lines 6–7; prompt in Figure 12 in Appendix C). The extracted entity then becomes the answer for the next backward hop. After completing all steps, it produces a multi-hop chain connecting the original query to the final answer through a set of shell command steps.

Forward Phase: Upon finding a verified chain of documents and commands, the sequence is reversed (Algorithm 1, line 10) to simulate the chronological information flow available to the agent in inference. Although the chain is created backward for verification purposes, a deployed agent only observes past interactions and retrieved evidence when making decisions. Reversing the trajectory ensures training trajectories faithfully match the inference behavior. At each forward step, the answer-blind Planner drafts a reasoning trace θ_d and action conditioned solely on the current causal history $\mathcal{H}$ (line 12; prompt in Figure 13 in Appendix C). Because the Planner does not have access to the verified evidence chain or future retrieval states, its proposed reasoning often lacks the precision necessary to justify the optimal shell command c_i. To bridge this gap, the Tutor model performs an alignment step that edits the Planner's reasoning trace to logically motivate c_i while remaining grounded in the observable history (line 13; prompt in Figure 16 in Appendix C). The resulting trajectory combines the realism of forward causal reasoning with the reliability of backward-verified evidence.

Filtering: To ensure the dataset provides a stable behavior for RL, it undergo a filtering (Algorithm 1, lines 16–18). First, the Planner generates a final answer $\hat{y}$ from the history $\mathcal{H}$ that achieves non-zero token-level overlap with the ground-truth y ($\mathrm{F}_1(\hat{y}, y) > 0$). This ensures trajectories contain enough information for answer generation. Second, the trajectories are evaluated by the Tutor for causal/logical errors (line 18; prompt in Figure 17 in Appendix C). The judge enforces temporal boundaries, discarding trajectories whose reasoning or commands reveal entities or facts not yet observable in the agent's current history. This is because the backward process has access to future information via the gold answer and verified evidence chain, and without verification, subtle forms of future leakage may persist even under explicit answer masking. Examples of data are in Appendix F.

2.1.2 Optimization of DCI Search Agent

SFT on Cold-Start Data: After creating the data, we perform supervised fine-tuning on them. Each example consists of the full interaction, including reasoning, tool invocations, tool responses, and

the final answer. The objective is to initialize the agent with stable retrieval and reasoning behavior before RL. In particular, SFT teaches the agent to produce concise and causally grounded search commands and avoid pathological retrieval behavior such as excessively broad corpus scans.

RL with GRPO: We optimize the SFT-trained policy with GRPO (Shao et al., 2024).[4] For each query q, π_θ samples $n = 5$ trajectories $\tau^{(i)} \sim \pi_\theta(\cdot \mid q)$, each containing reasoning, tool interactions, and a final answer. Each $\tau^{(i)}$ receives an answer reward $R_{\text{ans}}(\tau^{(i)})$ based on token-level F_1 (Rajpurkar et al., 2016) between the predicted answer $\hat{y}^{(i)}$ and the gold answer set $\mathcal{Y}$. To enforce the expected format, we define a binary indicator $\phi(\tau^{(i)}) \in \{0, 1\}$ that checks whether the trajectory satisfies the required structural constraints. The final trajectory reward is therefore $R(\tau^{(i)}) = \phi(\tau^{(i)})\, R_{\text{ans}}(\tau^{(i)})$, which assigns zero reward to structurally invalid trajectories. Details of the reward are provided in Appendix C.4. GRPO computes group-relative advantages as $A^{(i)} = (R(\tau^{(i)}) - \mu_R)/(\sigma_R + \epsilon)$, where μ_R and σ_R denote the mean and standard deviation of rewards within the group. This encourages trajectories that outperform other samples for the same query. Initializing from the SFT policy improves optimization stability by providing structured retrieval behavior before RL.

2.2 Efficient Corpus Interaction

Unlike RAG, which retrieves from a precomputed index, DCI searches the corpus directly by executing commands over millions of documents.[5] The agent uses tools such as `rg`, `grep`, `sort`, `uniq`, `head`, and `tail`. Since a trajectory may require multiple corpus scans, execution efficiency is critical. We apply semantics-preserving optimizations that reduce command latency while receiving outputs identical to execution over the original corpus. We use two complementary strategies.

Pruned Adaptive Command Execution (PACE): To reduce the cost of repeatedly scanning the full corpus, we introduce PACE, a semantics-preserving optimization inspired by inverted-index search (Manning et al., 2008; Zobel & Moffat, 2006). PACE builds a lightweight auxiliary structure with two components: posting lists that map each vocabulary term to the documents containing it, and a suffix array (Manber & Myers, 1993) that identifies vocabulary terms containing a given substring. To support substring matching in `rg`/`grep`, PACE first finds matching vocabulary terms through the suffix array, then combines their posting lists to identify candidate documents. This is important because shell search is not limited to exact token matches; for example, a query for `16` must also consider terms such as `2016` or `b16`. At inference time, PACE identifies filters that require documents to contain a specified string, such as `rg -F "London"`, which searches for the exact string `London`. Inverted filters, such as `rg -v "London"`, instead retain documents that do not contain the specified string. PACE excludes these filters from candidate construction because they do not identify a string that matching documents must contain. For each filter requiring a specified string, PACE splits that string into query terms. For each query term, two binary searches on the suffix array identify all vocabulary terms containing it as a substring. PACE then takes the *union* of their posting lists to obtain candidate documents for that query term. These candidate sets are *intersected* across query terms and filters, because a document must meet all required conditions to match the original command. The resulting candidate set contains every document that could match, but may also contain false positives. PACE reads the candidate documents from the original corpus and applies the original shell filters to determine the final results. The auxiliary structure determines only *where* the command executes; the original filters determine which documents are returned. Commands that cannot be safely pruned fall back to full-corpus execution. Details of the auxiliary structure, candidate construction, and execution are provided in Appendix C.2.

Sharded-Parallel Corpus Search (SPCS): To accelerate corpus interaction, we execute compatible shell pipelines in parallel over S line-aligned corpus shards while preserving byte-exact equivalence with sequential execution (Algorithm 3 in Appendix C.3). The engine decomposes each pipeline, parallelizing it only when the first stage is a search operator (`rg` or `grep`) and all subsequent stages are shard-independent; otherwise, it falls back to sequential execution. Shard-local outputs are then reconstructed using operation-specific reductions: concatenation for stateless pipelines, local-plus-global truncation for `head -n`, summation for counts such as `wc -l`, and deterministic

[4] Our method supports any standard RL algorithm; we use GRPO for its memory efficiency.

[5] Our experiments use a 21M-passage Wikipedia corpus (∼14GB).

k-way merging for `sort`/`uniq` pipelines. This improves throughput without changing the agent's observations. Implementation details are provided in Appendix C.3.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETUP

Datasets & Evaluation: We evaluate on eight diverse QA datasets: three single-hop—NaturalQuestions (NQ) (Kwiatkowski et al., 2019), TriviaQA (Joshi et al., 2017), and PopQA (Mallen et al., 2023)—and five multi-hop—HotpotQA (Yang et al., 2018), 2WikiMultiHopQA (2Wiki) (Ho et al., 2020), MuSiQue (Trivedi et al., 2022), Bamboogle (Press et al., 2023), and BrowseComp-Plus (Chen et al., 2025). We obtain all datasets except BrowseComp-Plus from FlashRAG (Jin et al., 2024). Following Jin et al. (2025), we use only NQ and HotpotQA for training, evaluating generalization on the remaining datasets. Dataset statistics are provided in Table 2 in Appendix A. For the first seven datasets, we use the 2018 Wikipedia dump (Karpukhin et al., 2020) containing 21M passages.[6] For BrowseComp-Plus, we use its accompanying corpus. We report token-level F_1 (Rajpurkar et al., 2016) as the primary metric for the first seven datasets and the official script with GPT-4.1 (OpenAI, 2024) judged accuracy for BrowseComp-Plus.

Training & Inference Settings: We use Qwen3.5-9B (Qwen Team, 2026) and Qwen3-4B (Team, 2025) as base models. For SFT, we construct a 10K-example dataset with a balanced mixture of NQ and HotpotQA. Qwen3.5-27B serves as both Tutor and Planner, with up to five refinement steps, and we train for one epoch. We optimize the policy with GRPO (Shao et al., 2024) for 200 steps on NQ and HotpotQA using a group size of $n = 5$. At inference, we use nucleus sampling (Holtzman et al., 2020) with temperature 0.6. The agent is allowed up to $T = 6$ turns and a 16,384-token context for FlashRAG datasets, and $T = 16$ turns and a 131,072-token context for BrowseComp-Plus. Full SFT, GRPO, and inference configurations are provided in Tables 3, 4, and 5 in Appendix C.5.

Baselines: Following Jin et al. (2025), we compare against a range of baselines, including (1) a direct LLM inference without external search, and (2) retrieval-augmented methods: RAG (Lewis et al., 2020), IRCoT (Trivedi et al., 2023), Search-O1 (Li et al., 2025), rejection sampling with a search engine and Search-R1 (Jin et al., 2025) (GRPO-optimized).[7] All baselines use the same backbone LLMs and are trained (when applicable) and evaluated under the same settings as our method for a fair comparison (Appendix C.5). We use three retrievers to retrieve top-3 documents from the corpus: BM25 (Robertson et al., 1994) as a sparse lexical baseline, E5 (Wang et al., 2022) as a dense embedding (110M parameters), and a large-scale Qwen3 embedding model (Zhang et al., 2025) (4B parameters). This allows us to systematically assess performance across both traditional and strong neural retrievers. Dense retrievers are implemented using FAISS (Douze et al., 2025) with HNSW index (Malkov & Yashunin, 2020) ($M = 32$, efConstruction $= 128$, efSearch $= 128$).

3.2 MAIN FINDINGS

Comparison of Performance with Baselines: We compare `GrepSeek` against retrieval-augmented agentic search baselines in Table 1. `GrepSeek` outperforms non-agentic approaches (Direct and RAG), untrained agents (IRCoT and Search-O1), and trained baselines (Rejection Sampling), with Search-R1 as the strongest competitor. Overall, `GrepSeek` achieves the best performance on 5 of 8 benchmarks, with statistically significant gains on 4 ($^{\uparrow}$), particularly on multi-hop tasks. These gains highlight the effectiveness of DCI for iterative evidence aggregation and entity-level precision, such as distinguishing subsidiaries from parent companies (Example 2) and avoiding name-collision errors (Example 3) in Appendix E. Performance is slightly lower on TriviaQA and Bamboogle without statistical significance, while PopQA is the only dataset with a significant drop ($^{\downarrow}$). These trade-offs reflect the lexical nature of `GrepSeek`. DCI is highly effective when queries contain

[6] Available at: `https://hf.co/datasets/PeterJinGo/wiki-18-corpus`, ∼14 GB of text.

[7] While recent agentic frameworks improve reasoning (Jin et al., 2026; Sun et al., 2025) or multi-agent orchestration (Chen et al., 2026), our goal is to isolate the effect of retrieval. We therefore compare methods that differ in how evidence is retrieved—traditional sparse and dense retrievers versus DCI. These orchestration improvements are largely orthogonal and could be incorporated into either paradigm.

Table 1: Performance across datasets (Qwen3.5 backbone). We report F_1 for the FlashRAG datasets and GPT-4.1-judged accuracy for BrowseComp-Plus. EM is reported in Table 6 in Appendix D. Superscript * denotes datasets used during training, while $^{\uparrow}$ and $^{\downarrow}$ denote statistically significant improvement and degradation, compared to the best baseline ($p < 0.05$).

Method	Retriever	Single-hop			Multi-hop					Average
		NQ*	TriviaQA	PopQA	HotpotQA*	2Wiki	MuSiQue	Bamboogle	BrowseComp+	(macro)
Direct	—	0.2733	0.5565	0.2364	0.2837	0.3353	0.1151	0.1648	0.0036	0.2461
RAG	BM25	0.3329	0.6660	0.3239	0.4434	0.3469	0.1305	0.2841	0.0916	0.3274
	E5-110M	0.5068	0.7072	0.4468	0.4206	0.3227	0.1494	0.3383	0.0398	0.3664
	Qwen3-4B	0.5002	0.7212	0.5046	0.4548	0.3498	0.1609	0.3484	0.0964	0.3920
IRCoT	BM25	0.2607	0.5518	0.2757	0.3098	0.1356	0.0725	0.1494	0.0952	0.2313
	E5-110M	0.4473	0.6238	0.3970	0.2983	0.1360	0.0921	0.2549	0.0277	0.2846
	Qwen3-4B	0.4379	0.6509	0.4548	0.3464	0.1641	0.1210	0.2770	0.0867	0.3173
Search-O1	BM25	0.4261	0.7283	0.4003	0.4893	0.3735	0.2271	0.5011	0.1867	0.4165
	E5-110M	0.4771	0.7316	0.4322	0.4706	0.3558	0.2262	0.6158	0.0614	0.4213
	Qwen3-4B	0.4622	0.7290	0.4731	0.4828	0.4009	0.2517	0.6103	0.0867	0.4371
Rejection Sampling	BM25	0.3825	0.7228	0.3827	0.5530	0.3972	0.2396	0.5819	0.2651	0.4406
	E5-110M	0.4238	0.7256	0.4298	0.5353	0.3678	0.2438	0.6441	0.1410	0.4389
	Qwen3-4B	0.4294	0.7258	0.4630	0.5442	0.4255	0.2697	0.6569	0.1759	0.4613
Search-R1	BM25	0.4592	0.7733	0.4194	0.5659	0.4028	0.2510	0.6013	0.2663	0.4674
	E5-110M	0.5069	**0.7734**	0.4747	0.5424	0.3727	0.2720	0.6763	0.1446	0.4704
	Qwen3-4B	0.5067	0.7693	**0.5101**	0.5591	0.4299	0.2878	**0.6989**	0.1940	0.4945
`GrepSeek`	—	**0.5223**$^{\uparrow}$	0.7673	0.4861$^{\downarrow}$	**0.6231**$^{\uparrow}$	**0.5178**$^{\uparrow}$	**0.3006**	0.6212	**0.3422**$^{\uparrow}$	**0.5226**$^{\uparrow}$

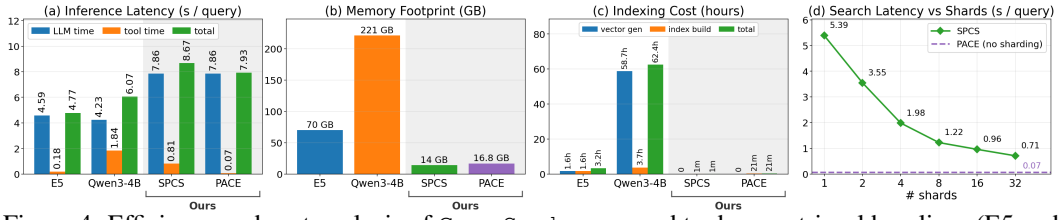

Figure 4: Efficiency and cost analysis of `GrepSeek` compared to dense retrieval baselines (E5 and Qwen3-4B). (a) Inference latency per query, broken down into LLM generation and tool execution time. (b) Memory footprint (RAM) required for the retrieval index. (c) Offline preprocessing time, measured in A100 GPU-hours for the dense retrievers and CPU wall-clock hours for SPCS and PACE. (d) Search tool latency of `GrepSeek` scaling with the number of shards.

strong textual anchors, including rare formulas (Example 1), distinctive phrasing (Example 4), and exact entity names (Example 7) in Appendix E. However, limited lexical overlap and form variation remain challenging: on PopQA, diacritics can prevent exact matches (Example 8), while overloaded keywords can cause `rg` to surface earlier but less authoritative matches because it lacks relevance ranking (Example 6). Dense retrievers are better suited to such cases by mapping lexical variants into a shared embedding space. Despite these limitations, `GrepSeek` achieves the strongest overall score (0.5226), significantly outperforming the best dense retrieval baseline ($p < 0.05$), showing that DCI provides a precise and effective alternative for open-domain QA and multi-hop reasoning.

Comparison of Latency with Best-Performing Baselines: Based on Table 1, Search-R1 with E5 and Qwen3 embeddings are the strongest dense retrieval baselines. We compare their efficiency with our two DCI execution strategies—SPCS and PACE—in Figure 4. All inference experiments use 32 CPU cores over 50 examples from each FlashRAG dataset (350 total), while dense-retriever indexing uses a single NVIDIA A100 GPU. As shown in Figure 4(a), SPCS has an end-to-end latency of 8.67 s per query, including 0.81 s of tool execution, while PACE reduces tool execution to only 0.07 s and total latency to 7.93 s. In comparison, E5 and Qwen3-4B require 4.77 s and 6.07 s, respectively; the remaining DCI overhead is primarily due to longer LLM reasoning trajectories. Figure 4(b–c) highlights the memory and preprocessing trade-offs. SPCS requires only the 14 GB raw corpus, while PACE requires 16.8 GB, including its compact auxiliary structure. In contrast, E5 and Qwen3-4B require 70 GB and 221 GB, respectively, to store their retrieval structures. SPCS requires approximately 1 minute of CPU wall-clock time for corpus setup, while constructing the PACE auxiliary structure takes approximately 21 minutes of CPU wall-clock time. Both avoid expensive embedding generation, which requires 3.2 and 62.4 A100 GPU-hours for E5 and Qwen3-4B, respectively. Finally, Figure 4(d) compares search latency across execution strategies. SPCS scales from 5.39 s with one shard to 0.71 s with 32 shards, with diminishing gains at higher shard

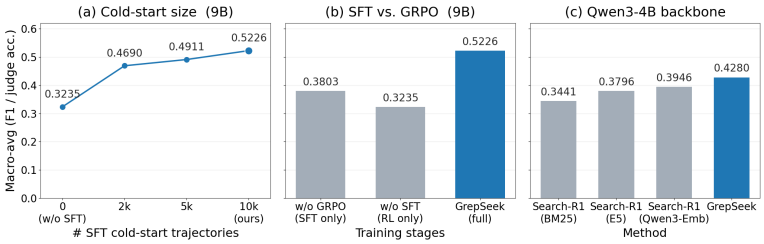

Figure 5: Ablation of `GrepSeek` using the average performance on all datasets. (a) Effect of cold-start SFT data size using the 9B backbone. (b) Contribution of the SFT and GRPO training stages for the 9B model. (c) Comparison of `GrepSeek` with Search-R1 variants using the Qwen3-4B backbone. Per-dataset results for these analyses are reported in Appendix D.

counts due to system-level bottlenecks such as memory bandwidth and merging overhead. PACE, even without sharding, reduces average search latency to 0.07 s by restricting execution to candidate documents rather than scanning the full corpus. These results show that the two optimizations are complementary: SPCS accelerates full-corpus execution through parallelism, whereas PACE avoids most of the scan altogether through candidate pruning.

Ablations of `GrepSeek`: We study the contribution of each training stage using macro-average performance in Figure 5(b), with full per-dataset F_1 and EM results reported in Tables 7 and 8 in Appendix D.[8] The full `GrepSeek` reaches a macro-average score of 0.5226, compared with 0.3803 without GRPO and 0.3235 without SFT, showing that both cold-start supervision and subsequent RL optimization are important. The larger degradation without SFT further highlights the importance of structured trajectory initialization before reinforcement learning. We further vary the cold-start set from 0 to 2k, 5k, and 10k trajectories while keeping the subsequent GRPO configuration fixed (Figure 5(a)). Macro-average performance increases from 0.3235 without SFT to 0.4690 with only 2k trajectories, then improves more gradually to 0.4911 and 0.5226 with 5k and 10k trajectories, respectively. Finally, to test whether these gains generalize across model scales, we repeat the comparison using the smaller Qwen3-4B backbone (Figure 5(c)). `GrepSeek` again achieves the highest macro-average score (0.4280), outperforming Search-R1 with BM25 (0.3441), E5 (0.3796), and Qwen3 embeddings (0.3946). Per-dataset 4B results are provided in Table 10 in Appendix D, indicating that the benefits of DCI persist across different backbone sizes.

Training Dynamics: Figure 6 shows the training dynamics over 200 GRPO steps. `GrepSeek` consistently achieves higher rewards than Search-R1 variants using E5, BM25, and Qwen3-Emb-4B (Figure 6(a)), but generates longer sequences due to extended reasoning and raw corpus context, reducing throughput (Figure 6(b)). Its retrieval behavior also evolves differently: while baselines increase their number of queries before stabilizing, `GrepSeek` gradually reduces the number of executed commands (Figure 6(c)): the agent shifts from multiple independent searches toward more expressive multi-stage shell pipelines, extracting more information per command invocation.

3.3 Analysis

Retrieval Behavior: Because DCI interacts with the corpus with commands, its retrieval behavior is directly interpretable. Across the seven FlashRAG benchmarks, `GrepSeek` consistently uses `| head` to limit output, relies on fixed-string matching (`-F`) in nearly all searches, and uses cascaded filtering in 70% of commands. It adapts search effort to task difficulty, issuing more commands on multi-hop datasets (2.8–3.8) than on single-hop datasets (2.3–2.7). Under the longer-horizon BrowseComp-Plus setting, which allows up to 16 turns, this increases to 6.9 commands per query. To separate behaviors learned during SFT from those refined by RL, we track trajectory statistics across training (Table 9 in Appendix D). Pipeline structure—pipe depth, fixed-string matching, truncation, and cascaded filtering—remains largely stable, suggesting these retrieval primitives are established during SFT. RL instead shapes higher-level behavior: the agent reduces commands per trajectory ($3.06 \rightarrow 2.56$), retrieves more context per query (`head -n` from roughly 5 to 10 lines), and allocates more tokens to reasoning ($4{,}251 \rightarrow 6{,}409$).

[8] As discussed earlier, directly optimizing the base model without SFT initialization was highly unstable. For the *w/o SFT* setting, we therefore report results from the final checkpoint before training collapse.

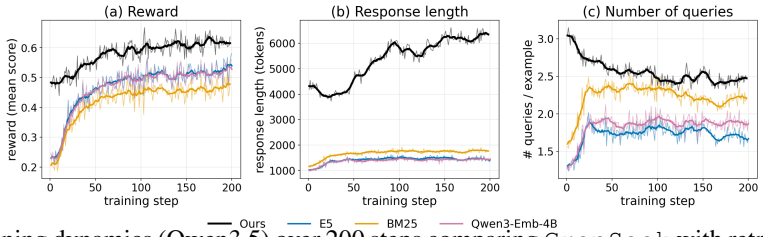

Figure 6: Training dynamics (Qwen3.5) over 200 steps comparing `GrepSeek` with retrieval baselines (E5, BM25, and Qwen3-Emb-4B). (a) Mean reward score during training. (b) Average response length measured in tokens. (c) Average number of search queries generated per example.

Case Studies: To qualitatively analyze `GrepSeek`, we compare its trajectories with the strongest baseline, Search-R1 with Qwen3-Emb-4B (Appendix E). The traces highlight the lexical precision of DCI: `GrepSeek` can isolate rare symbolic patterns (Example 1), exact entity names (Example 7), compose keywords across multiple hops (Example 4), and resolve entity collisions (Examples 2 and 3). However, lexical search has clear limitations. Since Unix tools return matches in file order rather than relevance, important evidence may be preceded by less informative passages (Example 6), while spelling variations can prevent a match (Example 8). Dense retrieval can be more robust in these cases by generalizing across lexical variations in embedding space.

4 Related Work

A more detailed discussion is provided in Appendix B.

Retrieval-Augmented Agentic Search: RAG augments LLMs with external knowledge by retrieving relevant evidence from a corpus, typically through sparse or dense index-based retrieval (Lewis et al., 2020; Zamani et al., 2022; Robertson et al., 1994; Wang et al., 2022; Zhang et al., 2025). For complex information needs, recent work extends this paradigm to multi-step search, where reasoning and retrieval are interleaved across multiple turns (Trivedi et al., 2023; Jin et al., 2025; Li et al., 2025; Maddela et al., 2025). Existing agentic-search methods primarily improve how an LLM plans, formulates queries, or interacts with a conventional retriever, with some methods additionally optimizing the retrieval component itself (Jin et al., 2025; Li et al., 2025; Zeng et al., 2026). In contrast, our work trains a compact open-weight model to interact directly with the corpus through deterministic shell-based search operations. We evaluate this capability on single-hop and multi-hop question answering, following established evaluation settings for retrieval-augmented search agents.

Direct Interaction with Corpus: DCI bypasses conventional retrieval and allows an agent to explicitly search, filter, and compose evidence over the corpus. Earlier work studied direct textual search in code repositories and regular-expression matching over text collections (Di Grazia & Pradel, 2023; Wang et al., 2026; Agarwal et al., 2015; Navarro, 2003). More recent work has begun applying DCI-style interfaces to open-domain QA (Sen et al., 2026; Li et al., 2026; Subramanian et al., 2025). While these demonstrate that LLMs can use direct corpus operations, they focus on exposing capable proprietary models to a direct-search interface or designing the surrounding agent harness. Our work instead focuses on *learning* this capability: we train a compact open-weight model to develop multi-step direct-search behavior, including how to formulate, sequence, and refine corpus operations during reasoning. Thus, rather than treating DCI primarily as an inference-time interface for an already capable agent, we study how direct corpus interaction itself can be acquired through training.

5 Conclusion

We introduced `GrepSeek`, a DCI agent that operates directly over raw corpora with Unix commands. Cold-start SFT and GRPO yield effective, interpretable, and lexically precise retrieval programs that lead overall across our QA benchmarks, particularly on multi-hop tasks. To make DCI efficient at scale, PACE narrows search to conservative candidate sets, while sharded parallelism accelerates full-corpus scans. Both preserve sequential semantics while offering complementary preprocessing–latency trade-offs. Our analysis also highlights the limitations of lexical retrieval, particularly sensitivity to surface-form variation and the lack of semantic relevance ranking.

AI USE STATEMENT

Generative AI tools were used throughout the development of this work for several purposes. They were used to assist with writing and editing, including drafting and revising portions of the manuscript, improving clarity and presentation, and preparing and refining figures and visualizations. They were also used during software development to assist with code generation, debugging, implementation, and the development of experimental and analysis utilities. In addition, generative AI tools were used for research ideation and execution, including brainstorming methodological alternatives, discussing implementation choices, and assisting with the retrieval and discovery of relevant prior work. Generative models also form part of the research methodology itself. In particular, as described in the paper, we use language models to generate synthetic cold-start search trajectories through the Tutor–Planner pipeline. These trajectories are subsequently verified and filtered before being used for supervised fine-tuning. Generative AI was therefore used for synthetic data generation in addition to its use as a general research and writing assistant.

All AI-assisted writing was reviewed and edited by the authors. AI-suggested research ideas, technical claims, and related work were independently checked by the authors against the relevant literature and primary sources. AI-assisted code was reviewed, executed, and tested as part of the experimental pipeline, and experimental results were produced and validated using our implemented evaluation procedures rather than accepted directly from generative AI outputs. Figures and visualizations produced with AI assistance were manually inspected and revised to ensure that they accurately represented the underlying methodology and results. Synthetic trajectories generated by language models were subjected to the verification and filtering procedures described in the paper before being included in the training data.

The authors made all final decisions regarding the methodology, experiments, analysis, interpretation, and presentation of the work. We take responsibility for the final content of this paper, including all text, claims, code, data, and artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work does not involve human-subject experiments or the collection of new personally identifiable information. Our experiments use existing question-answering benchmarks and text corpora, and therefore inherit potential biases, inaccuracies, and other limitations present in these resources. The language models used to generate synthetic training trajectories may similarly reproduce biases or factual errors from their training data. To reduce the impact of erroneous synthetic supervision, our cold-start generation pipeline includes explicit execution-based verification and filtering before generated trajectories are used for training.

GrepSeek allows an agent to interact with a corpus through executable shell commands. Although our experiments restrict this interaction to a controlled corpus-search environment, more general deployment of executable agents may introduce security risks if commands are given unrestricted access to external systems, files, or sensitive data. Appropriate sandboxing, access control, command restrictions, and auditing should therefore be employed when adapting such systems to real-world environments. As with other retrieval and question-answering systems, downstream users should also consider the licensing, privacy, and usage requirements of the corpora on which the system is deployed. We have aimed to document the capabilities and limitations of the proposed approach, including cases in which lexical matching may fail because of ambiguity or surface-form variation.

REPRODUCIBILITY STATEMENT

We provide the methodological and experimental details required to reproduce our results throughout the paper and appendices. Section 2 describes the DCI agent, the cold-start data generation procedure, supervised fine-tuning, reinforcement learning with GRPO, and the optimized corpus-execution mechanisms. The corresponding algorithms, prompts, reward specification, and implementation details are provided in Appendix C. Section 3.1 describes the datasets, corpus construction, evaluation metrics, baseline configurations, retrievers, training setup, and inference settings, with complete SFT, GRPO, and inference hyperparameters reported in the appendix. Additional evaluation results, ablations, and qualitative examples are provided in the remaining appendices. An anonymized version of our implementation, covering cold-start data generation, SFT and GRPO training, inference and

evaluation, PACE, and sharded-parallel corpus search, is available at `https://anonymous.4open.science/r/grepseek-BA75`. We will publicly release the implementation, model checkpoints, cold-start data, and supporting artifacts necessary to reproduce the experiments.

REFERENCES

Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 337–350, Oakland, CA, May 2015. USENIX Association. ISBN 978-1-931971-218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/agarwal.

Yiqun Chen, Lingyong Yan, Zixuan Yang, Erhan Zhang, Jiashu Zhao, Shuaiqiang Wang, Dawei Yin, and Jiaxin Mao. Beyond monolithic architectures: A multi-agent search and knowledge optimization framework for agentic search, 2026. URL https://arxiv.org/abs/2601.04703.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41 (6):391–407, 1990. doi: https://doi.org/10.1002/(SICI)1097-4571(199009)41:6<391::AID-ASI1>3.0.CO;2-9.

Luca Di Grazia and Michael Pradel. Code search: A survey of techniques for finding code. *ACM Comput. Surv.*, 55(11), February 2023. ISSN 0360-0300. doi: 10.1145/3565971. URL https://doi.org/10.1145/3565971.

Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2025. URL https://arxiv.org/abs/2401.08281.

Peter Elias. Efficient storage and retrieval by content and address of static files. *J. ACM*, 21(2): 246–260, April 1974. ISSN 0004-5411. doi: 10.1145/321812.321820. URL https://doi.org/10.1145/321812.321820.

R.M. Fano. *On the Number of Bits Required to Implement an Associative Memory*. Computation Structures Group Memo. MIT Project MAC Computer Structures Group, 1971. URL https://books.google.com/books?id=07DeGwAACAAJ.

Thibault Formal, Benjamin Piwowarski, and Stéphane Clinchant. Splade: Sparse lexical and expansion model for first stage ranking. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '21, pp. 2288–2292, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450380379. doi: 10.1145/3404835.3463098.

Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps. In Donia Scott, Nuria Bel, and Chengqing Zong (eds.), *Proceedings of the 28th International Conference on Computational Linguistics*, pp. 6609–6625, Barcelona, Spain (Online), December 2020. International Committee on Computational Linguistics. doi: 10.18653/v1/2020.coling-main.580. URL https://aclanthology.org/2020.coling-main.580/.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training LLMs to reason and leverage search engines with reinforcement learning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=Rwhi91ideu.

Jiahe Jin, Abhijay Sai Paladugu, and Chenyan Xiong. Beneficial reasoning behaviors in agentic search and effective training methods to obtain them. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 28080–28097, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1400. URL https://aclanthology.org/2026.findings-acl.1400/.

Jiajie Jin, Yutao Zhu, Guanting Dong, Yuyao Zhang, Xinyu Yang, Chenghao Zhang, Tong Zhao, Zhao Yang, Zhicheng Dou, and Ji-Rong Wen. Flashrag: A modular toolkit for efficient retrieval-augmented generation research. *CoRR*, abs/2405.13576, 2024. URL https://arxiv.org/abs/2405.13576.

Mandar Joshi, Eunsol Choi, Daniel Weld, and Luke Zettlemoyer. TriviaQA: A large scale distantly supervised challenge dataset for reading comprehension. In Regina Barzilay and Min-Yen Kan (eds.), *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1601–1611, Vancouver, Canada, July 2017. Association for Computational Linguistics. doi: 10.18653/v1/P17-1147. URL https://aclanthology.org/P17-1147/.

Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. Dense passage retrieval for open-domain question answering. In *Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP)*, pp. 6769–6781, 2020.

Tom Kwiatkowski, Jennimaria Palomaki, Olivia Redfield, Michael Collins, Ankur Parikh, Chris Alberti, Danielle Epstein, Illia Polosukhin, Jacob Devlin, Kenton Lee, Kristina Toutanova, Llion Jones, Matthew Kelcey, Ming-Wei Chang, Andrew M. Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 7:452–466, 2019. doi: 10.1162/tacl_a_00276. URL https://aclanthology.org/Q19-1026/.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Xiaoxi Li, Guanting Dong, Jiajie Jin, Yuyao Zhang, Yujia Zhou, Yutao Zhu, Peitian Zhang, and Zhicheng Dou. Search-o1: Agentic search-enhanced large reasoning models. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 5420–5438, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.276. URL https://aclanthology.org/2025.emnlp-main.276/.

Zhuofeng Li, Haoxiang Zhang, Cong Wei, Pan Lu, Ping Nie, Yi Lu, Yuyang Bai, Shangbin Feng, Hangxiao Zhu, Ming Zhong, Yuyu Zhang, Jianwen Xie, Yejin Choi, James Zou, Jiawei Han, Wenhu Chen, Jimmy Lin, Dongfu Jiang, and Yu Zhang. Beyond semantic similarity: Rethinking retrieval for agentic search via direct corpus interaction, 2026. URL https://arxiv.org/abs/2605.05242.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Mounica Maddela, Lingjue Xie, Daniel Preotiuc-Pietro, and Mausam. STARQA: A question answering dataset for complex analytical reasoning over structured databases. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 34487–34499, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. doi: 10.18653/v1/2025.emnlp-main.1749. URL https://aclanthology.org/2025.emnlp-main.1749/.

Yu A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42 (4):824–836, April 2020. ISSN 0162-8828. doi: 10.1109/TPAMI.2018.2889473. URL https://doi.org/10.1109/TPAMI.2018.2889473.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9802–9822, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.546. URL https://aclanthology.org/2023.acl-long.546/.

Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi: 10.1137/0222058. URL https://doi.org/10.1137/0222058.

Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

Gonzalo Navarro. Regular expression searching on compressed text. *Journal of Discrete Algorithms*, 1(5):423–443, 2003. ISSN 1570-8667. doi: https://doi.org/10.1016/S1570-8667(03)00036-4. URL https://www.sciencedirect.com/science/article/pii/S1570866703000364.

OpenAI. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '98, pp. 275–281, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 1581130155. doi: 10.1145/290941.291008. URL https://doi.org/10.1145/290941.291008.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5687–5711, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378/.

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

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. SQuAD: 100,000+ questions for machine comprehension of text. In Jian Su, Kevin Duh, and Xavier Carreras (eds.), *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pp. 2383–2392, Austin, Texas, November 2016. Association for Computational Linguistics. doi: 10.18653/v1/D16-1264. URL https://aclanthology.org/D16-1264/.

Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at trec-3. In *Text Retrieval Conference*, 1994. URL https://api.semanticscholar.org/CorpusID:3946054.

Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988. ISSN 0306-4573. doi: https://doi.org/10.1016/0306-4573(88)90021-0. URL https://www.sciencedirect.com/science/article/pii/0306457388900210.

Sahil Sen, Akhil Kasturi, Elias Lumer, Anmol Gulati, and Vamse Kumar Subbiah. Is grep all you need? how agent harnesses reshape agentic search, 2026. URL https://arxiv.org/abs/2605.15184.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *ArXiv*, abs/2402.03300, 2024. URL https://api.semanticscholar.org/CorpusID:267412607.

Shreyas Subramanian, Adewale Akinfaderin, Yanyan Zhang, Ishan Singh, Mani Khanuja, Sandeep Singh, and Maira Ladeira Tanke. Keyword search is all you need: Achieving rag-level performance without vector databases using agentic tool use, 2025. URL https://arxiv.org/abs/2602.23368.

Shuang Sun, Huatong Song, Yuhao Wang, Ruiyang Ren, Jinhao Jiang, Junjie Zhang, Fei Bai, Jia Deng, Wayne Xin Zhao, Zheng Liu, Lei Fang, Zhongyuan Wang, and Ji-Rong Wen. SimpleDeepSearcher: Deep information seeking via web-powered reasoning trajectory synthesis. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 13705–13720, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.739. URL https://aclanthology.org/2025.findings-emnlp.739/.

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

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. MuSiQue: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 10:539–554, 2022. doi: 10.1162/tacl_a_00475. URL https://aclanthology.org/2022.tacl-1.31/.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10014–10037, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.557. URL https://aclanthology.org/2023.acl-long.557/.

Sebastiano Vigna. Quasi-succinct indices. In *Proceedings of the Sixth ACM International Conference on Web Search and Data Mining*, WSDM '13, pp. 83–92, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450318693. doi: 10.1145/2433396.2433409. URL https://doi.org/10.1145/2433396.2433409.

Baoyi Wang, Xingliang Wang, Guochang Li, Chen Zhi, Junxiao Han, Xinkui Zhao, Nan Wang, Shuiguang Deng, and Jianwei Yin. Greprag: An empirical study and optimization of grep-like retrieval for code completion, 2026. URL https://arxiv.org/abs/2601.23254.

Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei. Text embeddings by weakly-supervised contrastive pre-training. *ArXiv*, abs/2212.03533, 2022. URL https://api.semanticscholar.org/CorpusID:254366618.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (eds.), *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, Brussels, Belgium, October-November 2018. Association for Computational Linguistics. doi: 10.18653/v1/D18-1259. URL https://aclanthology.org/D18-1259/.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Hamed Zamani, Mostafa Dehghani, W. Bruce Croft, Erik Learned-Miller, and Jaap Kamps. From neural re-ranking to neural ranking: Learning a sparse representation for inverted indexing. In

Proceedings of the 27th ACM International Conference on Information and Knowledge Management, CIKM '18, pp. 497–506, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360142. doi: 10.1145/3269206.3271800.

Hamed Zamani, Fernando Diaz, Mostafa Dehghani, Donald Metzler, and Michael Bendersky. Retrieval-enhanced machine learning. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '22, pp. 2875–2886, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450387323. doi: 10.1145/3477495.3531722. URL https://doi.org/10.1145/3477495.3531722.

Hansi Zeng, Liam Collins, Bhuvesh Kumar, Neil Shah, and Hamed Zamani. Cosearch: Joint training of reasoning and document ranking via reinforcement learning for agentic search. *arXiv preprint arXiv:2604.17555*, 2026.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, Alban Desmaison, Can Balioglu, Pritam Damania, Bernard Nguyen, Geeta Chauhan, Yuchen Hao, Ajit Mathews, and Shen Li. Pytorch fsdp: Experiences on scaling fully sharded data parallel, 2023. URL https://arxiv.org/abs/2304.11277.

Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2): 6–es, July 2006. ISSN 0360-0300. doi: 10.1145/1132956.1132959. URL https://doi.org/10.1145/1132956.1132959.

Table 2: Dataset sizes for training and evaluation. Datasets marked with an asterisk (*) indicate those utilized during the training phase for SFT and RL optimization.

	Training Size		
Dataset	**SFT**	**RL**	**Evaluation Size**
NQ*	5,000	79,168	3,610
TriviaQA	–	–	11,313
PopQA	–	–	14,267
HotpotQA*	5,000	90,447	7,405
2WikiMultiHopQA	–	–	12,576
MuSiQue	–	–	2,417
Bamboogle	–	–	125
BrowseComp-Plus	–	–	830
Total	**10,000**	**169,615**	**52,543**

A DATASETS

Following prior work (Jin et al., 2025), we evaluate our method on a comprehensive suite of eight knowledge-intensive benchmark datasets. These datasets are carefully selected to evaluate both single-step fact retrieval and complex, multi-step reasoning. To standardize the formatting and evaluation protocol, all datasets are obtained from the FlashRAG[9] and BrowseComp-Plus[10] repositories.

A.1 EVALUATION BENCHMARKS

The evaluation suite is divided into single-hop and multi-hop datasets to isolate the agent's ability to perform targeted retrieval versus iterative corpus exploration.

Single-Hop Datasets These tasks require retrieving a single, highly relevant fact or document to answer the user's query:

- **Natural Questions (NQ) (Kwiatkowski et al., 2019):** A dataset of real user queries issued to Google Search. We use the open-domain split, which requires systems to retrieve relevant Wikipedia passages to answer questions formulated by users without prior knowledge of the target.
- **TriviaQA (Joshi et al., 2017):** A collection of complex trivia questions authored by trivia enthusiasts. While the questions often contain compositional linguistic structures, the answers can typically be derived from a single retrieved document.
- **PopQA (Mallen et al., 2023):** An entity-centric QA dataset specifically designed to probe long-tail knowledge. The dataset is constructed from Wikidata triples and focuses on rare entities where parametric knowledge in LLMs typically fails, strictly necessitating accurate external retrieval.

Multi-Hop Datasets These tasks require the agent to execute multiple interdependent search queries, gathering partial information to inform subsequent retrieval steps:

- **HotpotQA (Yang et al., 2018):** A dataset requiring reasoning across at least two distinct Wikipedia articles. Questions are specifically designed to require information from multiple sources to synthesize and produce the correct final answer.
- **2WikiMultiHopQA (2Wiki) (Ho et al., 2020):** Constructed using Wikidata properties to generate compositional questions. This dataset introduces explicit logical structures to the multi-hop reasoning process, requiring the agent to follow strict reasoning chains across multiple documents.
- **MuSiQue (Trivedi et al., 2022):** A rigorous multi-hop QA dataset designed to minimize "shortcut" reasoning. The questions are composed by chaining multiple single-hop questions, heavily filtered to ensure that models cannot guess the answer through lexical overlap or single-document retrieval.

[9] Available at: `https://hf.co/datasets/RUC-NLPIR/FlashRAG_datasets`
[10] Available at: `https://huggingface.co/datasets/Tevatron/browsecomp-plus`

DCI Agent System Prompt

```
You are a research agent that searches a Wikipedia corpus to answer
    multi-hop questions by issuing shell commands.

Corpus file: corpus.jsonl

Treat the corpus as a large text file with one Wikipedia passage per
    line. The file has 21 million lines, so always cap output (use `|
    head -n 3` for narrow searches, up to `| head -n 8` when you need
    to scan more chunks of the same article).

Useful command patterns:
- Substring search:
    rg -F "distinctive phrase" corpus.jsonl | head -n 3
- AND-narrow with a second grep:
    rg -F "phrase1" corpus.jsonl | rg -i -F "phrase2" | head -n 3
- Count first, then narrow if too many results:
    rg -F "pattern" corpus.jsonl | wc -l
- Useful flags: -F (fixed string, no regex), -i (case-insensitive), -w
     (whole word).

Allowed shell tools: rg, grep, find, sed, awk, head, tail, cat, ls, wc
    , sort, cut, uniq, tr.
You MAY pipe with |. You MAY NOT use redirection (>, <), command
    chaining (; && ||), or command substitution (\$(...), `...`).

For every step, write a short paragraph of reasoning in plain prose
    (2-5 sentences) -- what you have learned from prior tool outputs,
    what's still missing, what you'll search for next. Then output
    exactly one of:

<tool_call>
{"name": "shell", "arguments": {"command": "your single-pipeline shell
     command, no newlines"}}
</tool_call>

OR (only when you have enough information to answer):

<answer>
the final answer (concise -- typically a short noun phrase, name, or
    date)
</answer>

Always reason first, then exactly one action block. Do not skip the
    reasoning.
```

Figure 7: System prompt for `GrepSeek`.

- **Bamboogle (Press et al., 2023):** A smaller but highly challenging dataset consisting of questions manually authored to defeat standard search engines. It requires deep, multi-step evidence gathering that cannot be resolved using surface-level web snippets or simple entity linking.
- **BrowseComp-Plus (Chen et al., 2025):** A controlled deep-search benchmark derived from BrowseComp, featuring challenging long-horizon, multi-hop questions over a fixed corpus. It requires iterative retrieval and reasoning across multiple pieces of evidence while enabling reproducible evaluation without live web search.

For datasets from FlashRAG, we use the 2018 Wikipedia dump (Karpukhin et al., 2020) of 21M documents as the corpus.[11] For BrowseComp-Plus, we use its accompanying corpus of 100,195

[11]Available at: `https://hf.co/datasets/PeterJinGo/wiki-18-corpus`

Corpus description (shared by all roles)

```
Corpus file: corpus.jsonl
Treat the corpus as a large text file with one Wikipedia passage per
    line. The file has 21 million lines, so always cap output (use `|
    head -n 3` for narrow searches, up to `| head -n 8` when you need
    to scan more chunks of the same article).
Useful command patterns:
- Substring search:
    rg -F "distinctive phrase" corpus.jsonl | head -n 3
- AND-narrow with a second grep:
    rg -F "phrase1" corpus.jsonl | rg -i -F "phrase2" | head -n 3
- Count first, then narrow if too many results:
    rg -F "pattern" corpus.jsonl | wc -l
- Useful flags: -F (fixed string, no regex), -i (case-insensitive), -w
     (whole word).
Allowed shell tools: rg, grep, find, sed, awk, head, tail, cat, ls, wc
    , sort, cut, uniq, tr.
You MAY pipe with |. You MAY NOT use redirection (>, <), command
    chaining (; && ||), or command substitution (\$(...), `...`).
```

Figure 8: System prompt describing the corpus and allowed shell tools.

documents, chunked into 864,837 passages using 700-word windows with a stride of 630 words (70-word overlap).

A.2 Data Splits and Training Protocol

To evaluate the generalization capabilities of our approach, we employ a strict split between in-distribution training datasets and out-of-distribution evaluation datasets. For the training phase, the agent is trained exclusively on a combined dataset consisting of the training splits of **NQ** ($79,168$ examples) and **HotpotQA** ($90,447$ examples). This provides the agent with exposure to both fundamental single-hop retrieval dynamics and complex multi-hop reasoning strategies, totaling $169,615$ training examples. During inference, we evaluate the system across all eight datasets to test both held-out in-domain performance and generalization to unseen datasets. We utilize the official test splits for datasets where they are publicly available. The final evaluation encompasses $52,543$ total queries across the eight benchmarks. Full dataset sizes and split statistics are detailed in Table 2.

A.3 Evaluation Metrics

For datasets from FlashRAG, we evaluate generated answers using Exact Match (EM) and token-level F_1 (Rajpurkar et al., 2016). For BrowseComp-Plus, we follow its evaluation protocol and report LLM-judged accuracy using GPT-4.1,[12] which determines whether the predicted final answer is correct given the reference answer.

- **Exact Match (EM):** The percentage of predictions that exactly match a gold answer after standard normalization, including lowercasing, punctuation removal, and removal of articles (e.g., "a", "an", "the").
- **F_1 Score:** Token-level F_1 measures partial overlap between the prediction and reference answer after the same normalization. When multiple references are available, we report the maximum score across references.
- **BrowseComp-Plus Accuracy:** Following the benchmark evaluation protocol, GPT-4.1 acts as an LLM judge to assess whether the predicted final answer matches the reference answer.

In the main paper, we report F_1 for the seven FlashRAG datasets and judge-based accuracy for BrowseComp-Plus; EM results for all eight datasets are provided in the appendix.

[12] https://platform.openai.com/docs/models/gpt-4.1

B Detailed Related Work

Retrieval-Augmented Agentic Search: Knowledge-intensive question answering is challenging for LLMs because many questions require facts that may be missing, outdated, or unreliable in the model's parametric memory (Mallen et al., 2023). Access to external knowledge is central to improving factuality and coverage. Retrieval-Augmented Generation (RAG) (Lewis et al., 2020) addresses this need by retrieving relevant evidence from an external corpus and conditioning generation on the retrieved context. More broadly, this can be viewed as an instance of retrieval-enhanced machine learning (Zamani et al., 2022). In this paradigm, the retrieval module determines what information is exposed to the model, typically through an index-based interface such as sparse lexical retrieval with BM25 (Robertson et al., 1994) or dense retrieval with embedding models such as E5 (Wang et al., 2022) and Qwen3-Embedding (Zhang et al., 2025). This retrieval step gives the model access to non-parametric knowledge and has become a standard approach for knowledge-intensive tasks.

However, a single retrieval step is often insufficient for complex questions. Many information needs require intermediate reasoning: the system may need to identify an entity, use that entity to form a follow-up query, retrieve additional evidence, and then compose information across documents. This motivates retrieval-augmented reasoning methods, where retrieval is not merely a preprocessing step but part of a multi-turn reasoning process. Such multi-turn behavior is important for both unstructured corpora and structured settings, where systems may need to inspect intermediate results, refine constraints, and issue follow-up operations over databases or tables, as in STARQA (Maddela et al., 2025). IRCoT (Trivedi et al., 2023) interleaves chain-of-thought reasoning with retrieval, while more recent search-agent systems extend this idea to longer-horizon tool-use trajectories, including Search-R1 (Jin et al., 2025) and Search-O1 (Li et al., 2025).

Recent agentic search methods differ in which parts of the pipeline are optimized. Some systems treat both the LLM and the retriever or search engine as black boxes, relying on prompting or inference-time orchestration to decide when and how to search, as in Search-O1 (Li et al., 2025). Other methods train the language model to issue better search queries and reason over retrieved evidence while keeping the underlying retriever fixed, as in Search-R1 (Jin et al., 2025). A third line of work jointly optimizes the reasoning policy and the retrieval or ranking component, as in CoSearch (Zeng et al., 2026). These approaches primarily differ in how they improve reasoning, planning, or retrieval over a conventional search interface. Our work studies a complementary direction: instead of training an agent to better use a fixed retriever, we train a compact open-weight model to interact directly with the corpus through deterministic shell-based search operations.

Question answering is widely used to evaluate retrieval-augmented and agentic search systems because it directly tests whether a system can retrieve sufficient evidence and synthesize the correct answer. Multi-hop QA benchmarks are especially relevant because they require multi-turn search behavior: a model must often retrieve one piece of evidence, use it to identify a new information need, and then combine evidence across steps. Following prior search-agent work such as Search-R1 (Jin et al., 2025), we evaluate on both single-hop and multi-hop QA benchmarks.

Direct Interaction with Corpus: Direct Corpus Interaction (DCI) provides a different way to connect language models with external information. Instead of relying on a retriever to rank passages, the agent issues explicit operations over the raw corpus and controls how evidence is matched, filtered, and composed. Prior work has studied direct textual search in code and repository settings (Di Grazia & Pradel, 2023; Wang et al., 2026), as well as recent DCI-style agents for open-domain retrieval over large-scale corpora (Sen et al., 2026; Li et al., 2026; Subramanian et al., 2025). This direction is also related to systems work on efficient string and regular-expression search over large or compressed text collections, including Succinct and Swift (Agarwal et al., 2015; Navarro, 2003). We view these systems primarily as efficiency-oriented predecessors rather than agentic-search baselines: they improve the execution substrate for direct search, while our focus is on whether an LLM can learn when and how to use direct corpus operations as part of multi-step reasoning.

C GREPSEEK'S IMPLEMENTATION DETAILS

This section provides implementation details and settings used for `GrepSeek`.

C.1 PROMPTS

The main system prompt for the DCI agent is shown in Figure 7. The prompts described in this section collectively define the instructional framework used to generate synthetic cold-start training trajectories. All agent roles share a common system prompt that specifies the corpus structure, interaction format, and permissible shell tools (Figure 8).

In **Phase A**, the pipeline employs an Answer-Aware Tutor to decompose multi-hop questions into ordered single-hop sub-queries (prompt in Figure 9). Guided by system instructions that enforce the anti-leak constraint (prompt in Figure 19), the Tutor iteratively constructs a backward retrieval trajectory through an initial command proposal stage (prompt in Figure 20) followed by targeted refinement steps (prompt in Figure 11). Each retrieved document is validated using a per-hop entailment judge (Figure 10), after which a dedicated bridge extraction prompt identifies the intermediate entity required to connect the current retrieval step to the preceding sub-query (prompt in Figure 12).

After constructing a verified retrieval path, **Phase B** transitions to an Answer-Blind Planner operating under a separate forward-acting system prompt (prompt in Figure 13). At each step, the Planner generates an initial history-conditioned reasoning trace and retrieval action proposal (prompt in Figure 14). A Tutor-edit stage then refines the Planner's reasoning to align it with the verified retrieval command while remaining strictly grounded in the observable interaction history (prompt in Figure 16). This process produces trajectories that preserve realistic forward causal reasoning while avoiding future-state leakage introduced by backward evidence construction.

Finally, in **Phase C**, the Planner generates a final answer conditioned exclusively on the accumulated interaction trajectory (prompt in Figure 15). The completed trajectory is then evaluated by a Trajectory Coherence Judge (prompt in Figure 17), which enforces strict temporal consistency constraints and rejects trajectories that implicitly reveal target entities, retrieval terms, or unobserved facts before they become available within the agent's causal history.

Decomposition

```
You are decomposing a multi-hop question into the ordered chain of
    single-hop sub-questions an agent would need to solve in order to
    reach the answer.

Question: {question}
Final answer: {answer}

Rules:
- Output 1 to 3 ordered sub-questions; each one's answer is what an
    agent would need before it can solve the next one.
- Order MUST match the order an agent would solve the question (sub_q[
    i+1] depends on sub_q[i]'s answer).
- The LAST sub-question's answer is the final answer above.
- Do NOT include expected_answers -- only the sub-questions themselves.
     The intermediate answers will be derived later.
- Do NOT make a sub-question whose answer is already given in the
    question (e.g., do not ask "Who is X?" when the question already
    names X).

Output ONLY a JSON array, no prose:
[{{"sub_question": "..."}}, {{"sub_question": "..."}}, ...]
```

Figure 9: Tutor prompt for decomposing multi-hop questions into single-hop steps.

Per-hop judge (does the doc confirm the answer?)

```
You are judging whether a tool output confirms the expected answer to
    a sub-question.
Sub-question: {sub_question}
Expected answer (primary): {expected_answer}
Acceptable alternative forms (any of these counts as confirming): {
    acceptable_forms}
Tool output:{tool_output}
Read the tool output. Does it contain a passage that clearly answers
    the sub-question with the primary expected answer OR any of the
    acceptable alternative forms? Be strict -- a tangential mention or
     a different entity with a coincidentally similar name does NOT
    count. But if the same underlying entity is referenced under any
    of the alternative forms (e.g., a stage name, a real name, or an
    alias listed above), that DOES count as confirming.
Output exactly one JSON object:
{{"verdict": "YES" or "NO", "reasoning": "<one short sentence>"}}
```

Figure 10: Prompt for judging if the retrieved document entails the target answer.

Backward command – refine attempt

```
Your previous command did not retrieve a passage that confirms the
    expected answer.
Sub-question: {sub_question}
Expected answer (do NOT search for this string): {expected_answer}
Alternative forms (also forbidden as search terms):{forbidden_forms}
Previous attempts:{prior_attempts}
Most recent attempt: Command: {last_command} Output:{last_output}
Judge said it does NOT confirm the answer because:{judge_reasoning}
Propose a different command. Common fixes:
- Your phrase may not appear verbatim -- try a shorter or differently-
    worded substring from the sub-question.
- The query may be too broad -- add a second piped rg with another
    distinctive phrase to AND-narrow.
- The query may be too narrow -- drop a filter or shorten the phrase
    to broaden.
- Try synonym variants of words from the sub-question (e.g., "
    headquartered in" vs "head office", "based in", "located in").
Same hard rules as before (do NOT use the expected answer as a search
    term; keep output <= 5 lines via head; one pipeline; whitelisted
    tools only).
Output exactly:<reasoning>why the previous command failed and what is
    different about your new approach</reasoning><command>
your new single-pipeline shell command</command>
```

Figure 11: Tutor prompt for refining a failed retrieval command.

C.2 Pruned Adaptive Command Execution (PACE)

PACE accelerates DCI by reducing the corpus scanned for each command while preserving correctness. It conservatively identifies a superset of candidate lines that may match, then executes the original command only on those candidates. Thus, pruning may introduce false positives but never removes true matches. Commands that cannot be safely or efficiently pruned fall back to execution over the full corpus. We next describe the data structures enabling this pruning.

Bridge-entity extraction

```
You are reading a passage that was retrieved to help answer a multi-
    hop question. Your job is to identify the entity the passage uses
    as the bridge -- the answer to the previous sub-question.
Original multi-hop question: {question}
Earlier sub-question (whose answer we are identifying): {sub_q_prev}
Later sub-question (already solved): {sub_q_next}
Answer to the later sub-question: {expected_next}
Passage retrieved for the later sub-question:{doc_next}
Read the passage. The passage answers the later sub-question;
    somewhere in it, the entity named in the answer to the EARLIER sub-
    question appears (since the later sub-question depends on it).
    Extract that entity exactly.
Also collect:
- "aliases" -- alternative names of the SAME entity that the passage
    explicitly mentions (e.g., stage names, real names, abbreviations,
     parenthetical "also known as"/"born ..." forms). Only include
    forms that literally appear in the passage. Do NOT invent aliases.
- "alternates" -- at most 2 OTHER plausible candidate entities from
    the passage that could also be the bridge if your primary pick is
    wrong. Use this only when the passage genuinely supports multiple
    readings.
Output a JSON object, nothing else: {{"bridge_entity": "<short noun
    phrase>", "aliases": ["<alt-name-1>", ...], "alternates": ["<other-
    candidate-1>", ...], "evidence": "<the exact short snippet from
    the passage that names the primary entity>"}}
If you cannot extract a clear bridge entity (the passage truly does
    not contain enough to derive the previous answer), output:
{{"bridge_entity": null, "aliases": [], "alternates": [], "evidence":
    null}}
```

Figure 12: Prompt for extracting the bridging entity required for backward chaining.

Planner – system (answer-blind agent)

```
You are a research agent that searches a Wikipedia corpus to answer
    multi-hop questions by issuing shell commands.
{corpus_description}
For every step, write a short paragraph of reasoning in plain prose
    (2-5 sentences) -- what you have learned from prior tool outputs,
    what's still missing, what you'll search for next. Then output
    exactly one of:
<tool_call>
{{"name": "shell", "arguments": {{"command": "your single-pipeline
    shell command, no newlines"}}}}
</tool_call>
OR (only when you have enough information to answer):
<answer>
the final answer (concise -- typically a short noun phrase, name, or
    date)
</answer>
Always reason first, then exactly one action block. Do not skip the
    reasoning.
```

Figure 13: System prompt for the Planner agent used during forward assembly.

Planner – step user prompt

```
Question: {question}

Trace so far:
{history}

Produce the next step.
```

Figure 14: Standard user prompt for the Planner agent during trajectory generation.

Final answer user prompt

```
Question: {question}

Trace so far:
{history}

You now have enough information. Produce a brief reasoning paragraph
    synthesizing the answer from the trace, then output exactly:

<answer>
the final answer (concise -- just a name, date, or short noun phrase)
</answer>
```

Figure 15: User prompt for the final answer formulation step.

C.2.1 Offline Auxiliary Structure Construction

Pruned Adaptive Command Execution (PACE) constructs a compact auxiliary structure once for each corpus. This structure is used only to identify candidate document identifiers; the original corpus remains unchanged and is consulted during command execution.

Term Catalog and Posting Lists: The corpus is processed document-by-document. Each document is case-folded and decomposed into maximal Unicode alphanumeric runs, and terms shorter than two characters are omitted. For each distinct term v in the corpus, we record a posting list $P(v)$ containing the identifiers of documents in which the term occurs. This is inspired by inverted indices in retrieval systems such as BM25. Since documents are processed in corpus order and each term is recorded at most once per document, the posting lists are naturally sorted and duplicate-free.

Substring-Searchable Vocabulary: Exact vocabulary lookup is insufficient for shell-based search because `rg` and `grep` perform substring matching. For example, a search for `16` must also consider vocabulary terms such as `2016` and `b16`.[13] To support such lookups efficiently, we first serialize the vocabulary into a single byte sequence. Specifically, given vocabulary terms $v_1, \ldots, v_m$, we construct: $S = v_1 \,\|\, \backslash 0 \,\|\, v_2 \,\|\, \backslash 0 \,\|\, \cdots \,\|\, v_m \,\|\, \backslash 0$, where $\backslash 0$ is a separator that does not occur inside a vocabulary term. We also retain a mapping from each position in S to the vocabulary term containing that position. We then construct a suffix array (Manber & Myers, 1993) over S. Each suffix-array entry therefore corresponds to a starting position within one of the vocabulary terms. Because suffixes are sorted lexicographically, all suffixes beginning with the same byte sequence are contiguous. At query time, for a query term q, two binary searches locate the interval of suffixes having q as a prefix. Since a suffix can start at any character position within a vocabulary term, this interval identifies exactly the vocabulary terms that contain q as a substring. The corresponding term identifiers can then be mapped to their posting lists. We further compact the suffix array by discarding entries that can never match a valid query. PACE considers only query terms of length at least two. Consequently, a suffix beginning at a separator cannot match any query, and a suffix beginning at the final character of a vocabulary term contains fewer than two term characters before reaching the separator and

[13] This is necessary because `rg` and `grep` also support substring matching.

Tutor edit (rewrite think to reach target command)

```
You are editing a research agent's draft reasoning so that it leads
    naturally to a known-correct next command. Your goal is to produce
     a final reasoning paragraph that: 1. Is grounded ONLY in what the
     agent has actually observed so far (the trace below). Do NOT add
    facts the agent could not know at this point. 2. Concludes with a
    clear motivation for the target command. 3. Reads as natural
    forward reasoning by an agent -- not as a justification written
    after the fact. Hedge where genuine uncertainty exists. 4. Stays
    close to the agent's draft when draft is already on track; rewrite
     freely when the draft proposes something the wrong direction.
Original question (the agent knows this): {question}
Trace observed so far by the agent: {history}
Agent's draft reasoning: {draft_think}
Agent's draft command (may be discarded): {draft_command}
Known-correct next command (what trace will use): {target_command}
What this command will return when executed (for your context only --
    the agent only sees this AFTER it runs the command, NOT while
    reasoning): {target_doc_preview}
FORBIDDEN -- your edited reasoning must NOT contain: (a) expected
    answer of current step, final answer, or any answer to be
    discovered in a future step. agent should not know these yet. (b)
    Any factual claim agent could not derive from (i) user's question,
     (ii) prior tool_responses in trace, or (iii) common knowledge.
ALLOWED -- the agent's reasoning may freely include: - Words from the
    question and synonyms/paraphrases of them ("hotel" -> "hospitality
    ","head office"->"headquarters"/"based in"). - Tentative
    hypotheses about WHERE answer might be found, phrased as guesses --
     e.g., "article likely uses phrase 'headquartered in' rather than '
    head office'", or "this is probably described in a Wikipedia
    article about family's business". - Common-knowledge inference --
    e.g., "Delhi is the capital of India" once Delhi has appeared, or
    "World Cup 2002 was the FIFA tournament" given a sports question.
Concrete pass/fail examples: Question: "The Oberoi family is part of a
     hotel company that has a head office in what city?" Trace so far:
     (no commands run yet). Target command: rg -F 'Oberoi family'
    corpus.jsonl | rg -F 'hotel' | head -n 3
LEAK (names bridge answer "Oberoi Group" before it has been seen): "I
    need to locate article on Oberoi Group to find its headquarters."
OK (describes the search in question terms with hypotheses): "I need
    to find which hotel company the Oberoi family is part of. The
    corpus likely contains passages mentioning the 'Oberoi family'
    alongside hotel-business context. I'll grep for 'Oberoi family'
    and narrow with 'hotel' to surface the relevant article."
Question: same as above. Trace so far: tool_response showed "...the
    Oberoi family is the majority shareholder in EIH Ltd, parent of
    The Oberoi Group..." Target command: rg -F 'Oberoi Group' corpus.
    jsonl | rg -F 'head office' | head -n 3
OK ("Oberoi Group" is now derivable from previous tool_response): "The
     previous result identifies family's hotel company as The Oberoi
    Group. To find its head office, I'll search for that company
    alongside 'head office', or a synonym like 'headquartered'."
When planner's draft is already valid under these rules, keep it close
     to draft. When draft proposes a substantively different direction,
     rewrite the reasoning so it naturally leads to the target command
     -- using hypotheses, synonyms, and search-strategy reasoning,
    never by naming entities the agent shouldn't know yet.
Output:<edited_reasoning>final reasoning</edited_reasoning>
```

Figure 16: Tutor prompt for steering agent reasoning toward verified actions.

Trajectory coherence judge

```
You are an expert reviewer of multi-hop QA trajectories produced by a
    research agent. The agent answers a question by issuing shell
    commands against a corpus and reasoning about the results across
    multiple turns. Your job is NOT to solve the question. Your job is
     to verify that the trajectory is INTERNALLY COHERENT.
INFORMATION FRONTIER
At any TURN k, the agent could legitimately know only:
  (a) the text of the original QUESTION,
  (b) anything appeared in the OUTPUT of any earlier turn (1 ... k-1),
  (c) generic common knowledge that any educated reader would have
    without consulting Wikipedia. Examples of (c):
    - "Delhi in India", "Brazil is country", "WW II ended in 1945".
    - "Headquarters"/"head office"/"based in" mean roughly the same.
    - Synonyms and paraphrases of words from the question.
      Counter-examples (NOT (c)):
    - "Cafu was Brazil's 2002 World Cup captain" - too specific.
    - "The Oberoi Group is based in Delhi" - too specific.
    - The bridge answer of any sub-question.
A turn explicitly tagged "[self-correction: failed attempt by design]"
     is allowed to issue a wrong command and produce confident
    reasoning that leads to that wrong command. A turn tagged "[
    recovery from TURN k]" is allowed to reference the failed turn's
    OUTPUT as observed evidence.

CHECKS: The trajectory FAILS if ANY of the following holds for ANY
    turn. Report the FIRST turn where it fails and which check failed.
CHECK 1 - REASONING leaks future facts. TURN k's REASONING names a
    specific entity, number, date, or fact not derivable from the
    information frontier (excluding self-correction turns). FAIL
    example: TURN 1 REASONING (no prior turns) says "I'll look up the
    Oberoi Group's headquarters" when the question only mentions "
    Oberoi family" - the company name has not been observed.
CHECK 2 - COMMAND leaks future facts. TURN k's COMMAND uses a search
    term not derivable from the information frontier. FAIL example:
    question asks for a track length, no prior OUTPUT has named a
    number, and the COMMAND contains rg -F "6.213" - that number is
    the answer the agent should still be searching for.
CHECK 3 - ACTION does not match REASONING. TURN k's REASONING
    explicitly states "the answer is X", "no further search is needed",
     "the question is answered", "I have the answer", and the action
    on this turn is a COMMAND rather than a FINAL ANSWER. FAIL example:
     REASONING ends with "the answer is clearly Amy Jo Johnson, no
    further tool execution is needed" and the action is a shell
    command.
CHECK 4 - FINAL ANSWER not supported. The final turn's FINAL ANSWER is
     not stated in, or clearly inferrable from, any earlier OUTPUT.
    FAIL example: FINAL ANSWER is "Roseau, Minnesota" but no earlier
    OUTPUT contains "Roseau" or supports that location.
If none of CHECK 1-4 fails on any turn, the trajectory PASSES.
When in doubt, prefer FAIL over PASS - we want clean training data,
    not high yield.
OUTPUT: Return EXACTLY one JSON object, no surrounding prose, no
    markdown fences:{{"verdict": "PASS" or "FAIL", "failing_check": 1
    or 2 or 3 or 4 or null, "first_failing_turn": <int or null>, "
    reasoning": "<one short sentence, max ~30 words>"}}
QUESTION: {question}
TRAJECTORY: {trajectory_text}
```

Figure 17: Prompt for the final quality gate checking for information leakage.

Algorithm 1 Cold-start Trajectory Generation for `GrepSeek`

Input: query q, gold answer y; corpus $\mathcal{C}$; max refinement iterations M; tutor $\mathcal{M}_T$, planner $\mathcal{M}_P$
Output: Verified training trajectory $\mathcal{T}_{\text{train}}$, or FAIL

// Phase A: Goal-Aware Decomposition and Backward Verification
$(q_1, \ldots, q_N) \leftarrow \mathcal{M}_T.\text{DECOMPOSE}(q, y)$ ▷ Generate ordered sub-queries; FAIL if unparsable
$a \leftarrow y$; $F \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$; $\mathcal{S} \leftarrow \langle\,\rangle$ ▷ a: target entity; F: aliases; $\mathcal{D}$: evidence context
for $i = N$ **down to** 1 **do**
  $(\textit{success}, c_i, d_i) \leftarrow \text{DISCOVER}(q_i, a, F, \mathcal{D})$; **if** $\neg\,\textit{success}$ **then return** FAIL
  $\mathcal{S} \leftarrow \mathcal{S} \oplus \langle i, q_i, a, c_i, d_i\rangle$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{d_i\}$
  if $i > 1$ **then**
    $(a, F) \leftarrow \text{GETBRIDGE}(q, q_{i-1}, q_i, a, d_i)$; **if** $a = \emptyset$ **then return** FAIL
  end if
end for
// Phase B: Forward Assembly (Answer-Blind Planner, Tutor-Guided)
$\mathcal{S} \leftarrow \text{REVERSE}(\mathcal{S})$; $\mathcal{H} \leftarrow \langle\,\rangle$ ▷ Restore forward order; init state history
for each $\langle i, q_i, a, c_i, d_i\rangle \in \mathcal{S}$ **do**
  $(\theta_d, c_d) \leftarrow \mathcal{M}_P.\text{DRAFT}(q, \mathcal{H})$ ▷ Answer-blind prediction of reasoning and action
  $\theta \leftarrow \mathcal{M}_T.\text{ALIGN}(q, \mathcal{H}, \theta_d, c_d, c_i, d_i)$; **if** $\theta = \emptyset$ **then return** FAIL
  $\mathcal{H} \leftarrow \mathcal{H} \oplus \langle \theta, c_i, d_i\rangle$ ▷ Ensure reasoning is strictly conditioned on causal history
end for
// Phase C: Answer Formulation and Quality Assurance
$\hat{y} \leftarrow \mathcal{M}_P.\text{ANSWER}(q, \mathcal{H})$
$\mathcal{T}_{\text{train}} \leftarrow \text{FORMAT}(q, \mathcal{H}, \hat{y})$
if $\hat{y} = \emptyset \vee \text{F}_1(\hat{y}, y) = 0 \vee \text{JUDGE}(q, \mathcal{T}_{\text{train}}) \neq \text{Pass}$ **then return** FAIL
return $\mathcal{T}_{\text{train}}$

function $\text{DISCOVER}(q', a, F, \mathcal{D})$ ▷ Identify a target command that retrieves evidence for a
  for $t = 1, \ldots, M$ **do**
    $c \leftarrow \mathcal{M}_T.\text{PROPOSE}(q', a, \{a\} \cup F, \mathcal{D})$ ▷ Strictly exclude target & aliases
    $d \leftarrow \text{EXEC}(c, \mathcal{C})$; **if** $\mathcal{M}_T.\text{CHECK}(q', a, d, F) = \text{True}$ **then return** (True, c, d)
  end for
  return $(\text{False}, \emptyset, \emptyset)$
end function

therefore cannot match either. These suffixes are removed after construction while preserving the relative lexicographic order of all retained entries. This reduces the suffix-array size by approximately 22% without changing the result of any valid substring lookup.

Compact Auxiliary Representation. PACE uses compact representations for each component needed to translate a substring match into candidate documents. First, a suffix-array lookup returns positions in the concatenated vocabulary rather than vocabulary-term identifiers. Storing an explicit term identifier for every suffix would require substantial additional memory. Instead, PACE stores a bitvector marking the null separators between vocabulary terms. For a suffix starting at byte position p, the containing term is identified by a rank query over this bitvector, which counts the number of separators preceding p. This reduces the suffix-to-term mapping from 932 MB to 44 MB on wiki18.

The resulting term identifiers are then mapped to their document posting lists. These lists are highly skewed: on wiki18, 88% of the 26.7M vocabulary terms occur in exactly one document. PACE therefore represents singleton terms directly with a single 32-bit document identifier, avoiding the metadata required for a general posting list. Posting lists containing multiple document identifiers are stored using Elias–Fano encoding (Fano, 1971; Elias, 1974; Vigna, 2013). For a sorted list of n document identifiers drawn from a corpus of u documents, Elias–Fano requires approximately $2 + \log_2(u/n)$ bits per posting while supporting efficient sequential decoding. Together, singleton specialization and Elias–Fano compression reduce posting storage from 6.11 GB to 1.68 GB on wiki18. Once candidate identifiers are recovered, PACE locates the documents in the original JSONL corpus. Because each document occupies one line, PACE records its starting byte offset. Rather than storing every offset as a 64-bit integer, documents are partitioned into blocks of 1,024: each block stores one 64-bit base offset, while individual documents store a 32-bit delta relative to that base.

This representation allows candidate documents to be accessed directly without scanning the corpus. Across all components, the resulting auxiliary structure occupies 2.79 GB for the 14.39 GB wiki18 corpus and 0.44 GB for the 3.73 GB BrowseComp-Plus corpus. All components are memory-mapped read-only, allowing concurrent execution workers to share the same physical memory pages.

C.2.2 Online Cost-Aware Candidate Construction:

During inference, PACE analyzes the emitted shell pipeline to determine whether candidate pruning can be applied safely (Algorithm 2, line 1). The first stage must be a positive `rg` or `grep` filter over the corpus. Subsequent stages are inspected while they remain document-wise filtering operations. Positive literal filters contribute constraints to candidate construction, whereas inverted filters are excluded because they do not imply that a matching document must contain a particular literal. Pipelines containing unsupported or globally stateful operations are routed directly to the original sequential execution path (line 2).

For each contributing literal, PACE applies the same case folding and tokenization used during offline construction (line 3). For example, the literal `Ring-Road` produces the query terms `ring` and `road`. For each query term q, two binary searches over the suffix array identify the vocabulary terms containing q as a substring (line 6) (Manber & Myers, 1993). The corresponding suffix positions are mapped to vocabulary-term identifiers using the separator-rank structure (line 9). Their posting lists define the candidate set $C(q) = \bigcup_{v \in V(q)} P(v)$, where $V(q)$ denotes the set of vocabulary terms containing q. The union is necessary because a substring match may occur within any such vocabulary term. When multiple positive query terms are available, their candidate sets can be intersected (line 22): $C = \bigcap_{q \in Q'} C(q)$, where $Q' \subseteq Q$ contains the terms selected by PACE. This construction is conservative: every true matching document must occur in the posting-derived candidate set of every selected positive term. Consequently, omitting a term from Q' can only enlarge C and cannot remove a true match.

PACE does not necessarily materialize every available posting list. Once the suffix-array range for q has been resolved, stored posting-list cardinalities provide the estimate $\mathrm{est}(q) = \sum_{v \in V(q)} |P(v)|$, which upper-bounds $|C(q)|$ without decoding the postings themselves (line 10). Query terms are considered in increasing order of $\mathrm{est}(q)$ so that selective constraints can reduce the candidate set before less selective ones are processed (line 14). The posting union of the most selective term initializes the candidate set (line 17). After each intersection, PACE compares the expected cost of processing another term with the cost of directly retrieving and verifying the current candidates. This process stops when the current candidate set is sufficiently small, the next term is too frequent to provide useful pruning, or further posting-list processing is estimated to be more expensive than candidate verification (line 20). If a query term has an empty suffix-array range, no vocabulary term contains it and the command can immediately be determined to have no matching document (line 7). At the opposite extreme, highly frequent substrings may match very large portions of the vocabulary and are skipped when expanding them would provide little benefit (line 8). These decisions affect only execution cost: skipping a pruning constraint enlarges the candidate superset but does not affect correctness.

Candidate sets are represented according to their density. Sparse sets are maintained as sorted arrays of document identifiers, enabling efficient ordered intersections, whereas dense sets are represented as per-document masks so that unions and intersections can be performed using scatter and bitwise operations. Finally, PACE abandons candidate pruning when even the most selective available term is too frequent or when the resulting candidate set exceeds the configured document or byte budget (line 25). In these cases, the command is executed through the original full-corpus path.

C.2.3 Semantics-Preserving Execution

Once a candidate set has been constructed, PACE retrieves the corresponding documents directly from the original corpus (Algorithm 2, line 26). Each document identifier is translated to its byte position using the stored document-offset structure. Because each JSONL document occupies exactly one line, consecutive candidate identifiers correspond to adjacent regions of the corpus. PACE coalesces such candidates into contiguous byte ranges and reads these ranges from a memory-mapped view of the original file in increasing corpus order. Maintaining corpus order is necessary for downstream operations whose result depends on input order, such as `head -n N`. Candidate documents are

Algorithm 2 Pruned Adaptive Command Execution (PACE)

Input: Command c; corpus $\mathcal{C}$ with u documents; vocabulary $\mathcal{V}$; suffix array SA; separator-rank structure Rank; singleton postings Single; compressed postings EF; posting counts Count; document offsets Off; hyperparameters B_{SA} (maximum suffix-array range), B_{post} (maximum posting estimate), B_{target} (target candidate count), B_{doc} (maximum candidate documents), and B_{byte} (maximum candidate bytes)

Output: Output identical to sequential execution of c on $\mathcal{C}$

1: $(\tau, q) \leftarrow \textsc{Classify}(c)$ ▷ Check if can be executed fast and collect literal in command
2: **if** $\tau = \textsc{Sequential}$ **then return** $\textsc{Exec}(c, \mathcal{C})$ ▷ Fallback for unsupported commands
3: $Q \leftarrow \textsc{Tokenize}(q)$ ▷ Case-fold and extract query terms
4: $\mathcal{R} \leftarrow \emptyset$
5: **for** $q \in Q$ **do**
6: $\quad [l_q, r_q) \leftarrow \textsc{SuffixRange}(\mathcal{V}, \mathrm{SA}, q)$ ▷ Find vocabulary terms containing q
7: $\quad$ **if** $l_q = r_q$ **then return** $\textsc{ExecEmpty}(c)$ ▷ No document can match
8: $\quad$ **if** $r_q - l_q > B_{\mathrm{SA}}$ **then continue** ▷ Skip overly common substring
9: $\quad V(q) \leftarrow \textsc{Unique}(\textsc{Rank}(\mathrm{SA}[l_q : r_q]))$ ▷ Map suffixes to vocabulary terms
10: $\quad e_q \leftarrow \sum_{v \in V(q)} |P(v)|$ ▷ Estimate posting cost without decoding
11: $\quad \mathcal{R} \leftarrow \mathcal{R} \cup \{(V(q), e_q)\}$
12: **end for**
13: **if** $\mathcal{R} = \emptyset$ **then return** $\textsc{Exec}(c, \mathcal{C})$
14: $\mathcal{R} \leftarrow \textsc{SortAscending}(\mathcal{R}, e)$ ▷ Most selective term first
15: $(V, e) \leftarrow \textsc{First}(\mathcal{R})$
16: **if** $e > B_{\mathrm{post}}$ **then return** $\textsc{Exec}(c, \mathcal{C})$ ▷ Even the cheapest term is too common
17: $D \leftarrow \bigcup_{v \in V} P(v)$ ▷ Initial candidate documents
18: **for** $(V, e) \in \mathcal{R}[2:]$ **do**
19: $\quad$ **if** $|D| \le B_{\mathrm{target}} \ \vee \ e > B_{\mathrm{post}} \ \vee \ \textsc{GatherCost}(e) > \textsc{VerifyCost}(D)$ **then**
20: $\quad\quad$ **break** ▷ Further pruning is not worthwhile
21: $\quad$ **end if**
22: $\quad D \leftarrow D \cap \left(\bigcup_{v \in V} P(v)\right)$ ▷ Intersect candidates across query terms
23: $\quad$ **if** $D = \emptyset$ **then return** $\textsc{ExecEmpty}(c)$
24: **end for**
25: **if** $|D| > B_{\mathrm{doc}} \vee \textsc{Bytes}(D, \mathrm{Off}) > B_{\mathrm{byte}}$ **then return** $\textsc{Exec}(c, \mathcal{C})$ ▷ Candidate set is too large
26: $X \leftarrow \textsc{Materialize}(D, \mathrm{Off}, \mathcal{C})$ ▷ Preserve original corpus order
27: **return** $\textsc{Exec}(c, X)$ ▷ Apply exact command semantics to candidates

therefore streamed in their original order and in bounded chunks rather than assembled into a single intermediate buffer. This also preserves the streaming behavior of standard shell pipelines: downstream stages can terminate after consuming sufficient input without requiring the remaining candidate documents to be read or processed.

The original command semantics are then applied to the materialized candidate documents (line 27). Candidate pruning therefore changes only which corpus documents must be examined. The original filtering predicates are still applied before a document can contribute to the command output, while corpus ordering, output formatting, truncation, counting, and exit-status semantics are preserved.

C.3 Sharded-Parallel Corpus Search

This appendix provides a comprehensive technical overview of the system-level optimizations employed by the DCI agent's command-execution engine. The engine is designed under a strict correctness-first principle: any pipeline whose parallel execution cannot be guaranteed to be byte-identical to sequential execution is safely executed via a single-file fallback mechanism.

C.3.1 Corpus Sharding and Parallel Fan-Out

The primary bottleneck in evaluating shell pipelines over large-scale corpora (e.g., a 14 GB JSONL file) is the inherently sequential execution model of standard Unix search tools. To address this, the engine performs a one-time, idempotent, line-aligned sharding of the corpus into S disjoint partitions.

Algorithm 3 Sharded-Parallel Corpus Search

Input: command containing pipe (|) c; corpus $\mathcal{C}$ split into S contiguous shards C_i s.t. $\biguplus_i C_i = \mathcal{C}$
Output: Byte-exact output identical to sequential execution of c on $\mathcal{C}$

$(s_1, \ldots, s_m) \leftarrow$ DECOMPOSE(c) ▷ Split pipeline into m distinct stages based on pipe operator(|)
$(\tau, N) \leftarrow$ CLASSIFY$(s_1, \ldots, s_m)$ ▷ Classify the type of reduction
if $\tau =$ SEQUENTIAL **then return** EXEC$(c, \mathcal{C})$ ▷ Run on full corpus if it can only be sequentially
parallel for $i = 1, \ldots, S$ **do** $R_i \leftarrow$ EXEC(c, C_i) ▷ Perform the operation on each shard in parallel
if $\tau =$ CONCAT **then return** $\biguplus_i R_i$ ▷ Concat result of operations
else if $\tau =$ HEAD **then return** $\mathrm{TOP}(\biguplus_i R_i,\ N)$ ▷ Concat result of operations and select top N
else if $\tau =$ COUNT **then return** $\sum_i \mathrm{Int}(R_i)$ ▷ Add results of operations
else if $\tau =$ SORTHEAD **then return** $\mathrm{TOP}(\mathrm{Merge}(R_1, \ldots, R_S),\ N)$ ▷ Merge results based on value and return top N

function CLASSIFY$(s_1, \ldots, s_m)$ ▷ m is the final stage of the pipeline
  if $s_1 \notin \{\texttt{rg}, \texttt{grep}\} \vee \exists s_j$ s.t. UNSAFE(s_j) **then**
    return (SEQUENTIAL, $\emptyset$) ▷ Reject non-search or cross-line context
  end if
  $\mathcal{S} \leftarrow \{s_i \in (s_1, \ldots, s_m) \mid$ STATELESS$(s_i)\}$ ▷ e.g., `cut`, `tr`, line-wise `sed`
  if $s_m =$ `head -n` $N \wedge \forall j < m,\ s_j \in \mathcal{S}$ **then**
    return (HEAD, N) ▷ Pattern: Stateless maps → early termination
  else if $s_m =$ `wc -l` $\wedge \forall j < m,\ s_j \in \mathcal{S}$ **then**
    return (COUNT, $\emptyset$) ▷ Pattern: Stateless maps → global line count
  else if $s_{m-1} \in \{$`sort`, `sort | uniq`$\} \wedge s_m =$ `head -n` N **then**
    return (SORTHEAD, N) ▷ Pattern: Stateless maps → Top-K filter
  else if $\forall j \leq m,\ s_j \in \mathcal{S}$ **then**
    return (CONCAT, $\emptyset$) ▷ Pattern: Entire pipeline is purely stateless
  else
    return (SEQUENTIAL, $\emptyset$) ▷ Revert unsupported complex pipelines
  end if
end function

- **Line-Aligned Sharding:** The corpus is partitioned along line boundaries (e.g., via `split -d -n l/S`), ensuring that each JSON record remains intact within a single shard. By construction, concatenating all shards in order reconstructs the original corpus without byte-level modification.
- **Thread-Level Fan-Out:** At inference time, shell pipelines are executed concurrently across the S shards using a thread pool. Since each shard is processed via independent `subprocess.run` calls (which in turn invoke tools such as `rg`), threading avoids Python-level process management overhead while allowing all shard executions to proceed in parallel. This reduces end-to-end latency for full-corpus scans approximately proportional to the number of shards, up to the I/O and memory bandwidth limits of the system that is hosting the optimized engine.

C.3.2 PIPELINE CLASSIFICATION AND MERGE STRATEGIES

To guarantee exact behavioral equivalence with sequential corpus execution, the engine employs a conservative pipeline parser that dynamically classifies each shell pipeline and routes it to one of five deterministic execution strategies. If a pipeline begins with an unsupported primitive or contains stateful operations that violate shard independence—such as line-indexing flags (e.g., `-n`), count-based modes (e.g., `-c`), contextual windowing (e.g., `-A`, `-B`, `-C`), or in-place transformations (e.g., `sed -i`)—it is immediately executed via the single-file fallback path. For supported pipelines, the engine aggregates the per-shard partial outputs $\{R_1, \ldots, R_N\}$ using the following semantics:

`CONCAT`: Applied to fully stateless pipelines (e.g., `rg`, `grep`, `cut`, `tr`, `sed`). The engine executes the pipeline on each shard independently and concatenates the outputs R_i in shard order.

Table 3: Hyperparameters used for synthetic cold-start trajectory generation (Algorithm 1) and Supervised Fine-Tuning (SFT) of the DCI agent inside the `verl` FSDP training framework.

Phase	Hyperparameter	Value
Cold-Start Data Generation	Tutor ($\mathcal{M}_T$) & Planner ($\mathcal{M}_P$) Backbone	Qwen3.5-27B
	Max Refinement Iterations (M)	5
	SFT Dataset Size	10,000
	Default Top-p	1.0
	Tutor (Backward Phase) Temperature	$0.4 + (0.1 \times \text{iteration})$
	Planner (Forward Phase) Temperature	0.7
	Judge Phase Temperature	0.6
Supervised Fine-Tuning (SFT)	Policy Model (π_θ)	Qwen3.5-9B
	Epochs	1
	Optimizer	AdamW
	Optimizer Betas (β_1, β_2)	(0.9, 0.999)
	Optimizer Epsilon (ϵ)	1×10^{-8}
	Peak Learning Rate	5×10^{-6}
	Learning Rate Scheduler	Constant with Warmup
	Linear Warmup Ratio	0.05
	Weight Decay	0.01
	Gradient Clipping Norm	1.0
	Max Sequence Length	16,384
	Global Batch Size	32
	Precision	`bfloat16`
	Hardware Parallelism	Ulysses (size = 4)

`HEAD`: Applied to pipelines terminating in `head -n K`. The engine applies the truncation locally per shard to bound memory usage to $K \times N$ lines. The bounded outputs are concatenated in shard order, and a final global `head -n K` is applied.

`COUNT`: Applied to pipelines terminating in counting operations (e.g., `wc -l`). The engine extracts the scalar count from each shard and computes the global sum.

`SORTHEAD`: Applied to top-K retrieval pipelines containing `sort`, optionally `uniq`, and terminating in `head -n K`. The engine applies `sort | head -n K` to each shard. The resulting sorted streams are merged using a deterministic k-way merge (`sort -m`), followed by an optional global `uniq` and a final global `head -n K`.

`SEQUENTIAL`: Applied to any unrecognized, unparseable, or globally stateful pipeline. The pipeline is executed sequentially against the unified single-file corpus to guarantee correctness.

C.3.3 I/O and System-Level Optimizations

Because the engine predominantly evaluates fixed-string filtering operations, retrieval latency is largely determined by memory bandwidth and I/O access patterns. To maximize throughput, the system implements a tiered I/O optimization stack.

RAM-Resident Corpus Placement: When system memory permits, the corpus and all shards are staged in a RAM-backed filesystem (e.g., `/dev/shm`). This ensures that all reads are served directly from main memory, avoiding filesystem and disk latency. In addition, the engine proactively warms the page cache at startup to eliminate cold-start penalties on the first retrieval query.

Deterministic Execution Flags: The engine injects a set of deterministic performance flags into supported Unix tools. Memory-mapped I/O (e.g., `--mmap` for `rg` and `grep`) reduces system-call overhead, while `--no-config` disables user-level configuration to ensure reproducibility. The environment is also fixed to `LC_ALL=C`, enabling bytewise matching and avoiding locale-dependent overhead without affecting literal search semantics.

C.3.4 Persistent Daemon Architecture and Telemetry

To eliminate the recurring costs of Python wrapper initialization and process startup across multiple tool calls within a single trajectory, the execution engine is deployed as a long-running persistent daemon. The evaluation loop communicates with the persistent `ShardedSearchEngine` via a length-prefixed JSON protocol over a Unix socket, amortizing per-call overhead and reducing latency by approximately 1–3 milliseconds per invocation. Finally, the engine preserves standard Unix failure semantics (e.g., returning exit code 0 if any shard produces a match) and deduplicates standard error streams to avoid redundant global error logging. Each tool invocation additionally records fine-grained telemetry—including the parsed command, selected merge strategy, shard configuration, and fallback decision—enabling detailed analysis of fast-path utilization during inference.

C.4 Reward Function

Correctness reward: Let $\hat{y}$ denote the agent's final answer, extracted from the last `<answer>`$\cdots$`</answer>` block in the trajectory, and let $\mathcal{Y} = \{y_1, \ldots, y_m\}$ be the set of gold reference answers. We compute a token-level F_1 score between $\hat{y}$ and the reference set. Following prior work (Rajpurkar et al., 2016), each prediction and reference answer is normalized by lowercasing, removing punctuation, dropping articles ("a", "an", "the"), and tokenizing on whitespace. For a normalized prediction $\hat{y}$ and a reference answer $y \in \mathcal{Y}$, let P and G denote their respective token multisets. We define the overlap as the multiset intersection $o = \sum_t \min(P(t), G(t))$, where $P(t)$ and $G(t)$ denote token counts. If $|P| = 0$ or $|G| = 0$, we set $F_1(\hat{y}, y) = 0$. Otherwise, precision, recall, and F_1 are defined as:

$$\mathrm{p} = \frac{o}{|P|}, \quad \mathrm{r} = \frac{o}{|G|}, \quad F_1(\hat{y}, y) = \frac{2\,\mathrm{p} \cdot \mathrm{r}}{\mathrm{p} + \mathrm{r}}. \tag{1}$$

Since multiple surface forms may be valid, the final answer reward $R_{\text{ans}} \in [0, 1]$ is defined as the maximum score over all reference answers:

$$R_{\text{ans}} = \max_{y \in \mathcal{Y}} F_1(\hat{y}, y). \tag{2}$$

This provides a dense learning signal that assigns partial credit to partially correct answers, rather than relying on a sparse binary exact-match reward.

Format reward: We define a binary format indicator $\phi \in \{0, 1\}$ to evaluate the structural validity of a trajectory. A rollout is valid ($\phi = 1$) if and only if it satisfies three strict formatting criteria: (1) all special tags (e.g., `<think>`, `<tool_call>`, `<tool_response>`, and `<answer>`) are properly balanced and non-overlapping; (2) the trajectory follows the prescribed state-transition structure (reasoning $\rightarrow$ tool invocation $\rightarrow$ environment response $\rightarrow$ final answer), with no text generated outside the designated blocks; and (3) the trajectory terminates within a closing `</answer>` tag.

This constraint mirrors the strictly formatted interaction protocol enforced during cold-start supervised fine-tuning (SFT), where all trajectories are structurally valid by construction. Consequently, ϕ acts as a gate that isolates and penalizes formatting violations introduced by the policy during reinforcement learning exploration phase.

Combined reward: The final reward during reinforcement learning optimization is defined as:

$$R = \phi \cdot R_{\text{ans}} = \phi \cdot \max_{y \in \mathcal{Y}} F_1(\hat{y}, y). \tag{3}$$

Thus, only structurally valid trajectories can receive a positive reward; malformed or out-of-format trajectories receive zero reward regardless of answer correctness.

C.5 Experimental Settings & Hyperparameters

SFT Training Phase: Table 3 summarizes the configuration for both synthetic cold-start data generation and the subsequent Supervised Fine-Tuning (SFT) stage. For data generation, we use the 27B variant of Qwen3.5 as both the answer-aware Tutor ($\mathcal{M}_T$) and answer-blind Planner ($\mathcal{M}_P$) to construct a 10,000-trajectory dataset. During the Tutor's backward discovery phase, we apply a dynamic temperature schedule $0.4 + 0.1 \times$ iter (up to $M = 5$ refinement steps) with Nucleus

Table 4: Hyperparameters used for reinforcement learning optimization via Group Relative Policy Optimization (GRPO) inside the `verl` training framework.

Phase	Hyperparameter	Value
GRPO Algorithm	Group Size (n)	5
	PPO Clip Ratio (ϵ)	0.2
	KL Divergence Coefficient	0.0 (Disabled)
	PPO Epochs	1
	Policy Entropy Coefficient	0.0
Rollout & Sampling	Sampling Temperature	1.0
	Sampling Top-p	1.0
	Max Sequence Length	16,384
	Max Assistant Turns	6
Optimization & Batching	Optimizer	AdamW
	Optimizer Betas (β_1, β_2)	(0.9, 0.999)
	Optimizer Epsilon (ϵ)	1×10^{-8}
	Peak Learning Rate	5×10^{-6}
	Learning Rate Scheduler	Constant with Warmup
	Linear Warmup Ratio	0.05
	Weight Decay	0.0
	Gradient Clipping Norm	1.0
	Global Training Batch Size	256 questions
	PPO Mini-batch Size	32
	PPO Micro-batch Size per GPU	1
	Precision	`bfloat16`
	Hardware Parallelism	Ulysses (size = 2)
	Total Training Steps	200

Table 5: Hyperparameters and configurations used during inference for FlashRAG datasets and BrowseComp-Plus.

Hyperparameter	FlashRAG	BrowseComp-Plus
Maximum Assistant Turns	6	16
Decoding Temperature	0.6	0.6
Decoding Top-p	1.0	1.0
`tool_max_tokens`	2,048	6,000
Maximum Model Sequence Length	16,384	131,072
GPU Memory Utilization	0.90	0.90
Tensor Parallel Size	2	2
Hardware Infrastructure	2× NVIDIA A100	2× NVIDIA A100

Sampling (Holtzman et al., 2020) to encourage broader exploration when initial retrieval attempts fail. In contrast, the Planner's forward assembly and the final coherence judge use fixed temperatures of 0.7 and 0.6, respectively, with top-$p = 1.0$ across all stages.

For SFT, the 9B Qwen3.5 policy model (π_θ) is trained for one epoch using AdamW (Loshchilov & Hutter, 2019) with a peak learning rate of 5×10^{-6} and a global batch size of 32. We use a linear warmup over the first 5% of training steps followed by a constant learning rate schedule. The maximum sequence length is set to 16,384 tokens to accommodate long interaction trajectories, including tool calls and retrieved context. Training is performed in the `verl`[14] framework using FSDP (Zhao et al., 2023) with `bfloat16` precision and Ulysses sequence parallelism (degree 4). The experiments are conducted on 4 NVIDIA A100 (80GB) GPUs on a machine with 1024GB memory.

[14]Available at: `https://github.com/verl-project/verl`

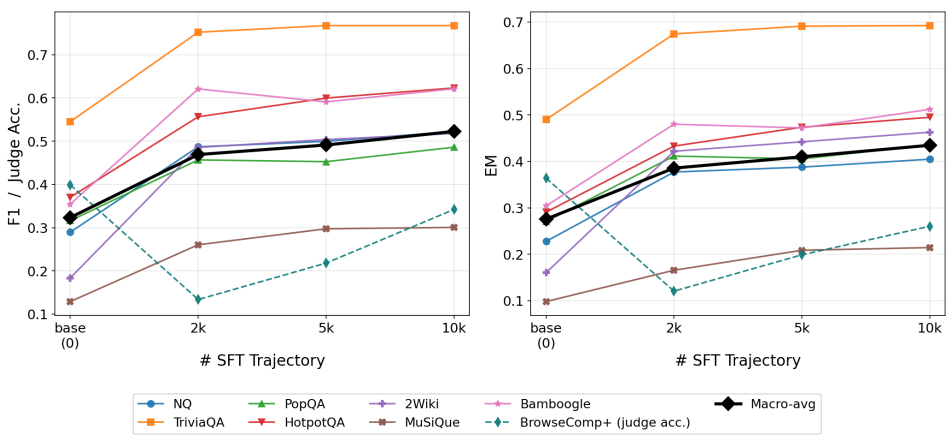

Figure 18: Effect of supervised fine-tuning (SFT) data size after RL training. The left panel shows FlashRAG F_1 and BrowseComp-Plus judged accuracy; the right panel shows EM for all eight datasets. Black lines show unweighted macro averages.

GRPO Training Phase: Table 4 lists the configuration parameters employed during the reinforcement learning phase using Group Relative Policy Optimization (GRPO) inside the `verl` framework. The agent policy (π_θ) is initialized from the checkpoint optimized during the Supervised Fine-Tuning (SFT) stage and is further trained for a total of 200 global steps. For each input query, the policy samples a group of $n = 5$ independent trajectories to compute relative advantages. Trajectory rewards are determined by the continuous token-level F_1 score, multiplied by a binary formatting gate to strictly penalize structural violations. Computed rewards are centered around the group mean and normalized by their standard deviation, providing a comparative learning signal that optimizes the policy without requiring an explicit critic network. We use a symmetric PPO clip ratio of 0.2 and disable the explicit KL divergence penalty to the reference policy, running exactly 1 proximal epoch per training step to minimize policy degradation.

During rollout generation via the vLLM engine,[15] sampling parameters are configured with a temperature of 1.0 and a top-p of 1.0 to encourage diverse tool-use exploration while preserving syntax consistency. To support deep, multi-turn interactions over the corpus, the trajectory budget is capped at a maximum sequence length of 16,384 tokens and a maximum of 6 assistant turns. Policy updates are calculated using the AdamW optimizer with a peak learning rate of 5×10^{-6} and a constant schedule following a 5% linear warmup. The optimization runs across a global batch size of 256 questions per step, evaluated in mini-batches of 32 and micro-batches of 1 per GPU. Distributed processing is managed in `bfloat16` mixed precision utilizing a Ulysses sequence parallel size of 2 for both the actor and reference models. The experiments are conducted on a machine with 4 NVIDIA A100 (80GB) GPUs and 1024 GB of system memory, taking approximately 4 days to complete.

Inference Setting: Table 5 outlines the core configuration parameters and architectural settings used during the inference phase of our finalized DCI agent. At evaluation time, the agent policy leverages a decoding temperature restricted to 0.6 alongside a top-p of 1.0 to ensure highly stable, syntactic consistency during multi-turn tool interaction. To prevent context space saturation from excessively broad document retrieval, the maximum length for any individual shell tool output is explicitly bounded via the `tool_max_tokens` parameter to $2,048$ tokens. In strict alignment with the environment constraints used during reinforcement learning, the maximum depth of a single interaction trajectory is capped at 6 assistant turns, providing a consistent structural framework for the model's sequential reasoning. To maximize throughput and safely accommodate long-context interaction histories, the inference infrastructure relies on a maximum sequence length of $16,384$ tokens. The model is served using a specialized high-concurrency configuration capable of handling a maximum of 256 sequences simultaneously, utilizing a high GPU memory allocation threshold of 0.90. The physical compute layer is provisioned across a hardware infrastructure consisting of $2\times$

[15] Available at: `https://vllm.ai/`

Table 6: Exact-match (EM) performance across datasets (Qwen3.5 backbone). Superscript * shows the datasets included in the training set, while all others are evaluated out-of-distribution. Superscript $^{\uparrow}$ indicates a statistically significant improvement, while $^{\downarrow}$ denotes a statistically significant decrease compared to the best-performing baseline. We use McNemar's test because significance is computed over paired binary exact-match outcomes for the same set of questions ($p < 0.05$).

Method	Retriever	Single-hop			Multi-hop					Average
		NQ*	TriviaQA	PopQA	HotpotQA*	2Wiki	MuSiQue	Bamboogle	BrowseComp+	(macro)
Direct	—	0.1864	0.4917	0.1925	0.2063	0.2840	0.0401	0.1040	0.0012	0.1883
RAG	BM25	0.2471	0.5899	0.2762	0.3376	0.2923	0.0629	0.1760	0.0807	0.2578
	E5-110M	0.4014	0.6264	0.3731	0.3213	0.2623	0.0753	0.2160	0.0361	0.2890
	Qwen3-4B	0.3856	0.6365	0.4219	0.3492	0.2893	0.0840	0.2240	0.0904	0.3101
IRCoT	BM25	0.1828	0.4787	0.2322	0.2300	0.0992	0.0397	0.1040	0.0880	0.1818
	E5-110M	0.3366	0.5416	0.3222	0.2248	0.0945	0.0592	0.1760	0.0241	0.2224
	Qwen3-4B	0.3183	0.5655	0.3683	0.2591	0.1246	0.0707	0.2240	0.0831	0.2517
Search-O1	BM25	0.3125	0.6519	0.3449	0.3814	0.3033	0.1535	0.4160	0.1554	0.3398
	E5-110M	0.3557	0.6527	0.3719	0.3639	0.2820	0.1456	0.4960	0.0410	0.3386
	Qwen3-4B	0.3468	0.6524	0.4070	0.3761	0.3280	0.1688	0.4880	0.0699	0.3546
Rejection Sampling	BM25	0.2765	0.6415	0.3082	0.4374	0.3434	0.1680	0.4960	0.2313	0.3628
	E5-110M	0.3111	0.6458	0.3488	0.4238	0.3142	0.1643	0.5360	0.1241	0.3585
	Qwen3-4B	0.3183	0.6504	0.3776	0.4308	0.3668	0.1911	0.5440	0.1518	0.3789
Search-R1	BM25	0.3385	0.6993	0.3660	0.4512	0.3515	0.1734	0.4880	0.2446	0.3890
	E5-110M	0.3823	**0.7005**	0.4118	0.4313	0.3202	0.1949	0.5840	0.1313	0.3945
	Qwen3-4B	0.3884	0.6988	**0.4503**	0.4475	0.3823	0.2031	**0.5920**	0.1711	0.4167
`GrepSeek`	—	**0.4047**$^{\uparrow}$	0.6925	0.4363$^{\downarrow}$	**0.4949**$^{\uparrow}$	**0.4627**$^{\uparrow}$	**0.2143**	0.5120	**0.2602**	**0.4347**$^{\uparrow}$

Table 7: Ablation study of `GrepSeek` (Qwen3.5 backbone) across single-hop and multi-hop datasets. We report F_1 for the FlashRAG datasets and judge accuracy for BrowseComp-Plus; EM results are in Table 8 in Appendix D. Superscript $^{\uparrow}$ indicates a statistically significant improvement using Student's t-test over both ablated variants after Bonferroni correction.

Variant	Single-hop			Multi-hop					Average
	NQ	TriviaQA	PopQA	HotpotQA	2Wiki	MuSiQue	Bamboogle	BrowseComp+	(macro)
`GrepSeek`	**0.5223**$^{\uparrow}$	**0.7673**$^{\uparrow}$	**0.4861**$^{\uparrow}$	**0.6231**$^{\uparrow}$	**0.5178**$^{\uparrow}$	**0.3006**$^{\uparrow}$	**0.6212**$^{\uparrow}$	0.3422	**0.5226**
- w/o GRPO	0.3880	0.6390	0.3903	0.4738	0.2956	0.2069	0.4231	0.2253	0.3803
- w/o SFT	0.2896	0.5451	0.3163	0.3705	0.1838	0.1291	0.3544	**0.3988**	0.3235

NVIDIA A100 GPUs, orchestrated with a tensor parallel size of 2 to optimize distributed memory access and minimize generation latency during dense search rollouts.

Table 8: Ablation study of `GrepSeek` (Qwen3.5 backbone) across single-hop and multi-hop datasets (EM scores). Superscript $^{\uparrow}$ indicates a statistically significant improvement over both ablated variants. We use McNemar's test because significance is computed over paired binary exact-match outcomes for the same set of questions, and apply Bonferroni correction ($p < 0.05$).

Variant	Single-hop			Multi-hop					Average
	NQ	TriviaQA	PopQA	HotpotQA	2Wiki	MuSiQue	Bamboogle	BrowseComp+	(macro)
`GrepSeek`	**0.4047**$^{\uparrow}$	**0.6925**$^{\uparrow}$	**0.4363**$^{\uparrow}$	**0.4949**$^{\uparrow}$	**0.4627**$^{\uparrow}$	**0.2143**$^{\uparrow}$	**0.5120**$^{\uparrow}$	0.2602	**0.4347**
- w/o GRPO	0.2853	0.5696	0.3370	0.3679	0.2454	0.1332	0.3520	0.2133	0.3130
- w/o SFT	0.2277	0.4901	0.2704	0.2906	0.1601	0.0981	0.3040	**0.3639**	0.2756

Table 9: Evolution of trajectory characteristics during RL training of `GrepSeek` (Qwen3.5 backbone).

RL Step	Cmds / Traj.	Lines Scanned (`head -n`)	Resp. Tokens	Filter Chain	Pipes / Cmd
10	3.06	4.9	4251	79%	1.98
50	2.59	6.9	4554	80%	2.04
100	2.56	11.8	5979	79%	2.04
200	2.56	10.4	6409	78%	2.01

D Additional Results

In this section, we present supplementary Exact Match (EM) results for all eight datasets. While the main text reports token-level F_1 for FlashRAG and judged accuracy for BrowseComp-Plus, EM provides a stricter assessment of surface-form agreement. Importantly, the main conclusions from the F_1 analysis remain consistent under EM, indicating that the observed gains are not driven by partial-match artifacts. Table 6 reports these results across all evaluated methods. `GrepSeek` achieves the highest overall macro-average EM score of 0.4347, compared with 0.4167 for the strongest baseline.

At the dataset level, `GrepSeek` achieves the best EM scores on five of the eight benchmarks (NQ, HotpotQA, 2Wiki, MuSiQue, and BrowseComp-Plus), with statistically significant gains on NQ, HotpotQA, and 2Wiki ($^{\uparrow}$). These results further support the effectiveness of direct corpus interaction for precise multi-hop reasoning and lexical evidence discovery. Conversely, EM also reflects the limitations of lexical search: `GrepSeek` shows a statistically significant drop on PopQA ($^{\downarrow}$) and lower performance on TriviaQA and Bamboogle, where surface-form variation and semantic matching can favor dense retrieval. Overall, the EM results reinforce the main finding that `GrepSeek` provides a precise and competitive alternative to index-based retrieval.

Tables 7 and 8 report the ablation results under F_1 and EM, respectively. The full `GrepSeek` model achieves the highest macro-average performance under both metrics: 0.5226 versus 0.3803 without GRPO and 0.3235 without SFT for F_1, and 0.4347 versus 0.3130 and 0.2756 for EM. Removing GRPO therefore substantially degrades overall performance, while removing the SFT initialization produces an even larger drop, supporting the importance of structured trajectory bootstrapping before RL. One exception is BrowseComp-Plus, where the RL-only variant performs better than the full model; nevertheless, its aggregate performance remains substantially lower across the complete benchmark suite. Figure 5 further studies the effect of cold-start SFT data size using both F_1 and EM. Policies are initialized with 0, 2k, 5k, or 10k trajectories and then optimized using the same GRPO configuration. Both metrics exhibit the same overall trend: introducing only 2k cold-start trajectories yields a large improvement over RL from the base model, while increasing the dataset to 5k and 10k provides smaller but consistent gains. The improvements begin to plateau beyond 5k examples, suggesting that the primary benefit of cold-start supervision comes from establishing structured search behavior before RL rather than simply scaling the amount of supervised data.

Table 10 evaluates `GrepSeek` with the smaller Qwen3-4B-thinking backbone against Search-R1 using BM25, E5, and Qwen3-Emb-4B retrievers. (Note that "Qwen3-4B" here denotes the shared policy backbone, distinct from the Qwen3-Emb-4B embedding retriever used by the Search-R1 baselines.) `GrepSeek` achieves the highest macro-average score of 0.4280, compared with 0.3946 for the strongest Search-R1 variant. The gains are concentrated on multi-hop tasks: `GrepSeek` achieves the best performance on HotpotQA, 2Wiki, MuSiQue, and BrowseComp-Plus, with a particularly large improvement on BrowseComp-Plus (0.1542 vs. 0.0578 for the best baseline). In contrast, retrieval-based baselines remain stronger on several single-hop datasets, including NQ, TriviaQA, and PopQA, as well as Bamboogle. Overall, these results show that the advantages of DCI, particularly for multi-step evidence discovery, persist when using a substantially smaller backbone.

Table 10: Performance across datasets (Qwen3-4B-thinking policy backbone for both `GrepSeek` and Search-R1; "Qwen3-Emb-4B" in the Retriever column is a separate embedding retriever, not the backbone). We report F_1 for FlashRAG datasets and GPT-4.1-judged accuracy for BrowseComp-Plus. Superscript * denotes datasets used in training.

Method	Retriever	Single-hop			Multi-hop					Average
		NQ*	TriviaQA	PopQA	HotpotQA*	2Wiki	MuSiQue	Bamboogle	BrowseComp+	(macro)
Search-R1	BM25	0.3681	0.6532	0.3398	0.3877	0.3732	0.1537	0.4194	0.0578	0.3441
	E5-110M	**0.4608**	**0.6907**	0.3996	0.4032	0.3833	0.1704	**0.5038**	0.0253	0.3796
	Qwen3-Emb-4B	0.4552	0.6867	**0.4636**	0.4099	0.4184	0.1820	0.4980	0.0434	0.3946
`GrepSeek`	—	0.4544	0.6831	0.4308	**0.5562**	**0.4307**	**0.2340**	0.4808	**0.1542**	**0.4280**

E CASE STUDIES

In this section, we present qualitative case studies comparing the reasoning and retrieval trajectories of our DCI agent (`GrepSeek`) against the strongest dense retrieval baseline, Search-R1 with the Qwen3-Emb-4B retriever. These examples illustrate the different behaviors, strengths, and failure modes of direct corpus interaction through shell commands versus embedding-based semantic retrieval. For each example, we provide the original question, the ground-truth answer, and the generated reasoning traces, retrieval commands, and retrieved observations produced by both systems. The examples highlight two key phenomena observed throughout evaluation:

- **Lexical Precision and Multi-Hop Evidence Isolation:** `GrepSeek` performs particularly well in settings that require exact lexical matching and iterative evidence filtering. Using shell operators such as `rg` and `grep`, the agent can isolate rare symbolic strings, progressively refine intermediate retrieval results, and compose multi-stage retrieval pipelines that accurately bridge evidence across documents (see Examples 1, 2, 3, and 4). This behavior is especially beneficial for multi-hop reasoning, entity disambiguation, and cases where small lexical details determine correctness. In contrast, dense retrievers may smooth over these distinctions due to embedding-level semantic compression, sometimes leading to incorrect generalization or entity confusion.
- **Surface-Form Sensitivity and Ranking Limitations:** At the same time, direct corpus interaction inherits the limitations of lexical retrieval. Since shell-based search lacks a learned semantic ranking mechanism, relevant documents may appear later in the retrieval stream despite containing the correct evidence (see Example 6). Moreover, strict surface-form matching can make the agent sensitive to lexical variations such as spelling differences or omitted diacritics, causing failures in cases where dense retrievers naturally generalize through semantic similarity (see Example 8).

Discussion of Qualitative Examples: Here we discuss and explain the provided case studies:

- **Symbolic and Rare-Token Matching (Example 1):** This example demonstrates `GrepSeek`'s ability to locate highly specific strings like chemical formulas that dense retrievers often struggle to represent in vector space. While the dense retriever returns semantically "chemistry-adjacent" documents, it fails to identify the exact formula. Our agent uses `rg -F` for an exact match, proving the utility of direct symbolic interaction for technical queries.
- **Entity Precision and Disambiguation (Examples 2 and 3):** These cases illustrate how lexical precision prevents the "semantic collapse" often seen in dense models. In Example 2, our agent distinguishes between a subsidiary studio and its parent brand. In Example 3, the agent uses the full, unique school name to isolate a single location, whereas the dense retriever retrieves a different school with a similar name, causing the reasoning to cascade into a geographical error.
- **Multi-Hop Evidence Composition (Example 4):** Here, we highlight the agent's ability to filter retrieval results iteratively. By searching for a specific band and then filtering for "singer," the agent isolates the relevant person. It then effectively parses a "highest [noun]" construction, showing that DCI can handle complex, multi-step queries by treating the corpus as a structured database.
- **Temporal and Stale Information (Example 5):** This highlights the importance of precise record-keeping. Our agent retrieves the document explicitly referencing the "current" record

holder, while the dense retriever is misled by a document containing stale information about the previous record holder (CN Tower).

- **Limitations of Ranking and Surface Forms (Examples 6 and 8):** These examples expose the failure modes of DCI. In Example 6, the lack of ranking forces the agent to rely on file order, burying the correct answer. In Example 8, we see that if the agent does not guess the exact surface form (e.g., handling diacritics or name variants), it fails where semantic models would intuitively generalize, underscoring the trade-off between lexical control and semantic robustness. By contrast, Example 7 is a case where the exact-name match succeeds precisely where dense retrieval blurs two similarly named entities (see the entity-precision discussion above).

Example 1. WIN HOTPOTQA – SYMBOLIC / RARE-TOKEN MATCH

Q: Which component of shellite has the formula (ON)CHOH?
Gold: picric acid

Ours (grep agent) ✓

[think] *I need to identify the specific component of shellite that corresponds to the formula (ON)CHOH. Since no search has been run yet, I will first query the corpus for "shellite" to locate its definition and list its constituent components.*

```
$ rg -F "shellite" corpus.jsonl | head -n 8
```

↪ "Glossary of British ordnance terms" TNT. After World War I the term "common lyddite" was dropped, and remaining stocks of lyddite-filled shells were referred to as HE (high explosive) shell filled lyddite. Hence "common" faded from use, replaced by "HE" as the explosive shell designation. Com ...

[think] *The search results confirm that Shellite is an explosive mixture of picric acid and dinitrophenol (or hexanitrodiphenylamine). To determine which of these components corresponds to the formula (ON)CHOH, I need to search the corpus for this specific chemical notation.*

```
$ rg -F "(ON)CHOH" corpus.jsonl | head -n 8
```

↪ "Picric acid" Picric acid Picric acid is an organic compound with the formula (ON)CHOH. Its IUPAC name is 2,4,6-trinitrophenol (TNP). The name "picric" comes from the Greek ("pikros"), meaning "bitter", reflecting its bitter taste. It is one of the most acidic phenols. Like other highly ...

[think] *The search results confirm that the chemical formula (ON)CHOH corresponds to picric acid. The earlier search established that Shellite is an explosive mixture primarily composed of picric acid and dinitrophenol. Therefore, the component of shellite with the formula (ON)CHOH is picric acid.*

answer: picric acid

Search-R1 + Qwen3-Emb-4B ✗

[think] *The user is asking about the chemical composition of Shellite, specifically which component has the formula (ON)CHOH. Let me search for information about Shellite and its components.*

```
$ search 'Shellite component formula (ON)CHOH'
```

↪ Doc 1 (Title: Limpet) matrix, this can have varying profound effects on the formation of the goethite crystals. The space in between the crystals and the chitin matrix is filled with an amorphous hydrated silica (SiO). The most prominent metal by percent composition is iron in the form of goethite. ...

[think] *The search results don't directly answer the question about Shellite and the formula (ON)CHOH. Let me search more specifically about Shellite and its components.*

```
$ search 'Shellite explosive components formula'
```

↪ Doc 1 (Title: "Shellite (explosive)") Shellite (explosive) Shellite (known as Tridite in US service) is an explosive mixture of picric acid and dinitrophenol or picric acid and hexanitrodiphenylamine in a ratio of 70/30. It was typically used as a filling in Royal Navy armour-piercing shells during ...

[think] *The search results tell me that Shellite is an explosive mixture of picric acid and dinitrophenol or picric acid and hexanitrodiphenylamine in a ratio of 70/30. However, I need to find which component has the formula (ON)CHOH. Let me search more specifically about the formula (ON)CHOH and its relation to Shellite components.*

```
$ search '(ON)CHOH chemical formula compound'
```

↪ Doc 1 (Title: CH3O) CH3O The molecular formula CHO may refer to: The chemical name for CH3O is methoxide. It is a base formed from methanol by replacement of the hydroxyl hydrogen with a metal. A strong base and a good nucleophile. Methoxide contains one atom of carbon, three atoms of hydrogen and o ...

[think] *The search results don't directly give me the answer about Shellite and the formula (ON)CHOH. Let me search more specifically about dinitrophenol or hexanitrodiphenylamine, which are components of Shellite, to see if either of them matches the formula (ON)CHOH. Actually, looking at the formula (ON)CHOH, this looks like it might be referring to a specific compound. Let me search more specifically about this formula in relation to explosives or Shellite components.*

```
$ search 'Shellite dinitrophenol formula (ON)CHOH'
```

↪ Doc 1 (Title: 2,4-Dinitrophenol) 2,4-Dinitrophenol 2,4-Dinitrophenol (2,4-DNP or simply DNP) is an organic compound with the formula HOCH(NO). It is a yellow, crystalline solid that has a sweet, musty odor. It sublimes, is volatile with steam, and is soluble in most organic solvents as well as aqueo ...

[think] *Based on the search results, I can see that: 1. Shellite is a mixture of picric acid and dinitrophenol (or picric acid and hexanitrodiphenylamine) 2. 2,4-Dinitrophenol has the formula HOCH(NO), which can be written as (ON)CHOH The formula (ON)CHOH matches the formula for 2,4-Dinitrophenol (also known as DNP), which is one of the components of Shellite. So the answer should be dinitrophenol (or specifically 2,4-dinitrophenol).*

answer: dinitrophenol

Why: *The answer is keyed by a literal symbolic string. Lexical grep matches it exactly; a dense retriever cannot embed a chemical formula and returns chemistry-shaped but wrong passages.*

Example 2. WIN HOTPOTQA – ENTITY PRECISION (SUBSIDIARY VS. PARENT)

Q: Paul DeBoy is known for an appearance in the Western action-adventure game developed by which company?
Gold: Rockstar San Diego

Ours (grep agent) ✓
[think] *To answer the question, I first need to identify which Western action-adventure game Paul DeBoy is known for appearing in. I will search the Wikipedia corpus for "Paul DeBoy" to find details about his filmography and any specific video game roles.*
`$ rg -F "Paul DeBoy" corpus.jsonl | head -n 8`
↪ "PJ DeBoy" PJ DeBoy Paul J. "PJ" DeBoy (born June 7, 1971) is an American actor and talk show host. DeBoy was born in Baltimore, Maryland. He shares his name with his cousin, actor Paul DeBoy. He and Paul are cousins of Delegate Steven J. DeBoy Sr. DeBoy started his career in New York, where he pe ...
[think] *The search results confirm that Paul DeBoy is best known for his appearance in "Red Dead Redemption" as Jimmy Saint. The question describes this as a "Western action-adventure game," which matches the genre of *Red Dead Redemption*. To answer the question, I now need to identify the company that developed this specific game.*
`$ rg -F "Red Dead Redemption" corpus.jsonl | rg -i -F "developed" | head -n 8`
↪ "Red Dead" Red Dead Red Dead is a series of Western-themed action-adventure video games published by Rockstar Games. The first entry in the series, "Red Dead Revolver", was released on the PlayStation 2 and Xbox on May 3, 2004. Originally developed by Capcom, "Red Dead Revolver" borrowed element ...
[think] *Based on the search results, Paul DeBoy is known for his appearance in the Western action-adventure game *Red Dead Redemption*, where he played the character Jimmy Saint. The corpus confirms that *Red Dead Redemption* was developed by Rockstar San Diego (a subsidiary of Rockstar Games).*
answer: Rockstar San Diego

Search-R1 + Qwen3-Emb-4B ✗
[think] *The user is asking about Paul DeBoy and which company developed a Western action-adventure game that he appeared in. Let me search for information about Paul DeBoy and his video game appearances.*
`$ search 'Paul DeBoy video game appearance Western action-adventure game'`
↪ Doc 1 (Title: "Paul DeBoy") Paul DeBoy Paul DeBoy (born September 14, 1955 in Baltimore, Maryland) is an American actor. He is best known for appearances in "A Dirty Shame" as Wendell Doggett, "Red Dead Redemption" as Jimmy Saint, "Haber" as Bernhard Moritz and for episodes of "Law & Order" and "Law ...
[think] *Based on the search results, I can see that Paul DeBoy is known for appearing in "Red Dead Redemption" as Jimmy Saint. Red Dead Redemption is a Western action-adventure game. I need to find out which company developed Red Dead Redemption. Let me search for more information about Red Dead Redemption's developer.*
`$ search 'Red Dead Redemption developer company'`
↪ Doc 1 (Title: "Red Dead Redemption 2") Sadie, and Charles. Afterward, Sadie and Charles leave for other pursuits. The final scene shows Pinkerton agents Ross and Fordham observing John's ranch, initiating the events of "Red Dead Redemption". "Red Dead Redemption 2" was developed by Rockstar Studios, ...
[think] *Based on my search results, I can see that: 1. Paul DeBoy appeared in "Red Dead Redemption" as Jimmy Saint (according to Doc 1) 2. Red Dead Redemption is a Western-themed action-adventure game 3. The game was developed by Rockstar San Diego and published by Rockstar Games (according to Doc 3) The question asks which company developed the game. The game was developed by Rockstar San Diego (a studio of Rockstar Games). The main company is Rockstar Games. So the answer should be Rockstar Games.*
answer: Rockstar Games

Why: *Ours retrieves the precise developing studio; the dense baseline collapses to the better-known parent brand.*

Example 3. WIN HOTPOTQA – NAME-COLLISION DISAMBIGUATION

Q: What major city is the Faith Lutheran Middle School and High School located by?
Gold: Las Vegas

Ours (grep agent) ✓

[think] *I need to first identify the specific location of Faith Lutheran Middle School and High School before I can determine which major city it is near. I will search the corpus for the school's name to find its entry and extract the location details.*

```
$ rg -F "Faith Lutheran Middle School and High School" corpus.jsonl |
head -n 8
```

↪ "Vernon Fox" Vernon Fox Vernon Lee Fox, III (born October 9, 1979) is a retired American football safety. He was signed by the San Diego Chargers as an undrafted free agent after the 2002 NFL Draft. He played college football at Fresno State. Fox is currently a professional motivational speaker. In ...

[think] *The search results confirm that Faith Lutheran Middle School and High School is located in Las Vegas, Nevada. Las Vegas is a major city in the United States, known for being the county seat of Clark County and a global hub for tourism and entertainment.*

answer: Las Vegas

Search-R1 + Qwen3-Emb-4B ✗

[think] *The user is asking about the location of Faith Lutheran Middle School and High School. I need to search for this information to find out which major city this school is located near. Let me search for information about Faith Lutheran Middle School and High School to find its location.*

```
$ search 'Faith Lutheran Middle School and High School location city'
```

↪ Doc 1 (Title: "Faith Lutheran High School") Faith Lutheran High School Faith Lutheran High School, commonly referred to as Faith or FLHS, is a private high school located in Crystal Lake, Illinois. It is affiliated with the Lutheran Church–Missouri Synod and serves students of McHenry County, Illino ...

[think] *The result is Faith Lutheran High School in Crystal Lake, Illinois – but the question names "Faith Lutheran Middle School and High School", and Crystal Lake is not a major city. The nearest major city would be Chicago. Let me search again in case a differently named school matches. [. . .]*

```
$ search 'Faith Lutheran Middle School and High School location city
Illinois'
```

↪ Doc 1 (Title: "Faith Lutheran High School") Faith Lutheran High School Faith Lutheran High School, commonly referred to as Faith or FLHS, is a private high school located in Crystal Lake, Illinois. It is affiliated with the Lutheran Church–Missouri Synod and serves students of McHenry County, Illino ...

[think] *Same result again – Crystal Lake, Illinois. Since Crystal Lake is not a major city, the nearest one is Chicago. Let me try once more in case the intended school is elsewhere. [. . .]*

```
$ search 'Faith Lutheran Middle School and High School location
Chicago'
```

↪ Doc 1 (Title: "Faith Lutheran High School") Faith Lutheran High School Faith Lutheran High School, commonly referred to as Faith or FLHS, is a private high school located in Crystal Lake, Illinois. It is affiliated with the Lutheran Church–Missouri Synod and serves students of McHenry County, Illino ...

[think] *All results return Faith Lutheran High School in Crystal Lake, Illinois, which I take to be the same institution. Crystal Lake is not a major city, so the question is probably asking for the nearest one; it lies in the Chicago metropolitan area, so the answer would be Chicago. [. . .]*

answer: Chicago

Why: *The exact full name pins the right entity in one hop; dense retrieval grabs a same-named school and the error cascades into a wrong city.*

Example 4. WIN HOTPOTQA – MULTI-HOP BRIDGE + DISTINCTIVE PHRASE

Q: The Joggers are a four-piece band whose lead singer is the son of an American chemist who received the highest what?

Gold: Hirsch index rating

Ours (grep agent) ✓

[think] *I need to identify the lead singer of the band "The Joggers" to determine who the son of an American chemist is. I will search the corpus for "The Joggers" and filter for mentions of "singer" to quickly locate the relevant biographical details.*

```
$ rg -F "The Joggers" corpus.jsonl | rg -i -F "singer" | head -n 8
```

↪ "George T. Whitesides" is married to Loretta Hidalgo Whitesides, and they plan to be the first couple to honeymoon in space. His father is George M. Whitesides, a professor of chemistry at Harvard University. His brother is the lead singer of The Joggers, a Portland, Oregon-based band. George T. Whi ...

[think] *The search results confirm that Ben Whitesides is the lead singer of The Joggers and the son of George M. Whitesides, an American chemist. The question asks for the "highest what" he received, which likely refers to a specific, top-tier honor or distinction in his field. Since the snippet mentions he "received the highest Hirsch index rating," this might be a distractor or a specific detail, but the question phrasing "highest [noun]" suggests a standard award title like "highest honor" or "highest award." To find the specific answer, I should search for George M. Whitesides in the context of awards or honors, filtering for the word "highest" to match the question's phrasing.*

```
$ rg -F "George M. Whitesides" corpus.jsonl | rg -i -F "highest" |
head -n 8
```

↪ Genzyme Best Companies to Work for". The company donated $83 million worth of products worldwide; in 2006, it made $11 million in cash donations. In 2005, Genzyme was awarded the National Medal of Technology, the highest level of honor awarded by the president of the United States to America's leadi ...

[think] *The question asks for the "highest what" received by the American chemist, George M. Whitesides, who is the father of The Joggers' lead singer. The search results explicitly state that he "received the highest Hirsch index rating of all living chemists in 2011." While he received many awards, the specific phrasing "highest" in the context of his achievements points directly to this metric.*

answer: Hirsch index rating

Search-R1 + Qwen3-Emb-4B ✗

[think] *User is asking about a band called "The Joggers". I need to find out who the lead singer is, their father's profession (specifically that he was an American chemist), and what the highest award or recognition that chemist received. Let me search for information about "The Joggers" band and their lead singer.*

```
$ search 'The Joggers band lead singer son of American chemist'
```

↪ Doc 1 (Title: "The Joggers") The Joggers The Joggers, formerly known as Stateside, are a four-piece band from Portland, Oregon. They play complex songs with elements of math rock which retain a pop sensibility. The guitar style of (lead singer and songwriter) Ben Whitesides and the second guitarist ...

[think] *Based on the results, I found that: 1. The Joggers is a four-piece band from Portland, Oregon 2. The lead singer is Ben Whitesides 3. Ben Whitesides is the son of George M. Whitesides, who is described as "one of the most decorated and wealthy chemists in the world" Now I need to find out what George M. Whitesides received - specifically the "highest" award or recognition. I need to search for more information about George M. Whitesides and what prestigious awards he received.*

```
$ search 'George M. Whitesides highest award American chemist'
```

↪ Doc 1 (Title: "George M. Whitesides") George M. Whitesides George McClelland Whitesides (born August 3, 1939) is an American chemist and professor of chemistry at Harvard University. He is best known for his work in the areas of NMR spectroscopy, organometallic chemistry, molecular self-assembly, so ...

[think] *Whitesides received many awards. The Priestley Medal (2007) is the "highest honor conferred by the ACS", but the National Medal of Science (1998) is generally considered the highest US scientific honor. Since it asks for the "highest" award, the National Medal of Science seems the best fit; let me confirm. [. . .]*

```
$ search 'George M. Whitesides National Medal of Science highest
award'
```

↪ Doc 1 (Title: "George M. Whitesides") Welch Award in Chemistry (2005), the AIC Gold Medal (2007), and the Priestley Medal (2007), the highest honor conferred by the ACS. More recently, George Whitesides received the 2009 Dreyfus Prize in the Chemical Sciences from The Camille and Henry ...

[think] *The National Medal of Science (1998) is the highest scientific honor in the US, awarded by the President, so that is the "highest" award Whitesides received. The answer is National Medal of Science.*

answer: National Medal of Science

Why: *Both reach the bridge entity (the father); ours matches the distinctive "highest ___" phrasing to the exact fact, while the baseline substitutes a generic award.*

Example 5. WIN BAMBOOGLE – CURRENT VS. STALE RECORD-HOLDER

Q: In what year was the tallest self-supporting tower completed?
Gold: 2012

Ours (grep agent) ✓

[think] *To answer the question, I first need to identify which structure is the tallest self-supporting tower. I will search the Wikipedia corpus for the phrase "tallest self-supporting tower" to locate the relevant entry and confirm the specific structure before proceeding to find its completion date.*

```
$ rg -F "tallest self-supporting tower" corpus.jsonl | head -n 8
```

↪ Tehran and Tehran has been a destination for mass migrations from all over Iran since the 20th century. Tehran is home to many historical collections, including the royal complexes of Golestan, Sa'dabad, and Niavaran, where the two last dynasties of the former Imperial State of Iran were seated. Teh ...

[think] *The search results confirm that the Tokyo Skytree is currently the tallest self-supporting tower in the world, while the CN Tower is also noted as the world's tallest self-supporting tower in specific contexts. To answer the question, I need to identify the specific year the tallest such tower was completed. Since the Tokyo Skytree is explicitly cited as the current tallest, I will search for its completion date to see if it holds the record or if the question refers to a historical milestone.*

```
$ rg -F "Tokyo Skytree" corpus.jsonl | rg -i -F "completed" | head -n
8
```

↪ "Tokyo Skytree" Tokyo Skytree The tower is the primary television and radio broadcast site for the Kantō region; the older Tokyo Tower no longer gives complete digital terrestrial television broadcasting coverage because it is surrounded by high-rise buildings. Skytree was completed on Leap Day, 29 ...

[think] *The question asks for the year the tallest self-supporting tower was completed. Based on the search results, the Tokyo Skytree is currently the tallest self-supporting tower in the world, and the text explicitly states it was completed on February 29, 2012. While other towers like the CN Tower were once the tallest, the current record holder's completion date represents the answer to "the tallest self-supporting tower" in the present context.*

answer: 2012

Search-R1 + Qwen3-Emb-4B ✗

[think] *The user is asking about the completion year of the tallest self-supporting tower in the world. I need to search for this information to get the most accurate and up-to-date data. Let me search for this information.*

```
$ search 'tallest self-supporting tower completed year'
```

↪ Doc 1 (Title: "CN Tower") The proposed Chicago Spire was expected to exceed the height of the CN Tower, but its construction was halted early due to financial difficulties amid the Great Recession, and was eventually cancelled in 2010. "Guinness World Records" has called the CN Tower "the world's ta ...

[think] *The search results mention that the CN Tower is recognized by Guinness World Records as the world's tallest self-supporting tower. However, I need to find out when the CN Tower was completed. The search results don't explicitly state the completion year of the CN Tower. Let me search more specifically for the CN Tower completion year.*

```
$ search 'CN Tower completion year'
```

↪ Doc 1 (Title: "CN Tower") CN Tower The CN Tower () is a concrete communications and observation tower located in Downtown Toronto, Ontario, Canada. Built on the former Railway Lands, it was completed in 1976. Its name "CN" originally referred to Canadian National, the railway company that built the ...

[think] *Based on the search results, the CN Tower was completed in 1976. The CN Tower is recognized by Guinness World Records as the world's tallest self-supporting tower. It held this record for 32 years until 2007 when it was surpassed by the Burj Khalifa (though the Burj Khalifa is classified as a building, not a tower). However, I should note that the search results mention the CN Tower held the record for the world's tallest free-standing structure until 2007. But the question asks about the tallest self-supporting tower completion year. The CN Tower was completed in 1976 and is recognized as the world's tallest self-supporting tower (free-standing tower). The tower was completed in 1976. So the answer is 1976.*

answer: 1976

Why: *Ours resolves the current record-holder (Tokyo Skytree); the dense baseline anchors on CN Tower's stale Guinness label.*

Example 6. FAILURE BAMBOOGLE – GREP HAS NO RELEVANCE RANKING

Q: Who was president of the United States in the year that Citibank was founded?
Gold: James Madison

Ours (grep agent) ✗

[think] *To answer the question, I first need to determine the specific year Citibank was founded. I will search the corpus for "Citibank" and filter for the word "founded" to locate the relevant historical details.*

```
$ rg -F "Citibank" corpus.jsonl | rg -i -F "founded" | head -n 8
```

↪ "Relativity Media" Relativity Media Relativity Media was an American media company headquartered in Beverly Hills, California, founded in 2004 by Lynwood Spinks and Ryan Kavanaugh. The studio was the third largest mini-major globally until bankruptcy on July 30, 2015. Its full-scale film studio acqu ...

[think] *The previous result surfaced an unrelated company (grep returns matches in file order, not by relevance, so the correct passage may simply be buried further down the corpus). Rather than re-issue the same filter, I will narrow using a specific historical detail: since Citibank traces back to a 19th-century New York bank charter, I will search for "Citibank" combined with "1863" to check whether that date is explicitly linked to the bank's establishment in the text.*

```
$ rg -F "Citibank" corpus.jsonl | rg -F "1863" | head -n 8
```

↪ "George Fisher Baker" Club, having been elected in 1895. He married Florence Tucker Baker (daughter of Benjamin Franklin Baker and Sophronia J. Whitney) in 1869, and was the father of: In 1863, Baker, along with his mentor, John Thompson, and Thompson's sons Frederick Ferris Thompson and Samuel C. T ...

[think] *Based on the search results, the First National Bank of the City of New York, which became the forerunner of Citibank, was chartered in 1863. While the specific entity "Citibank" as a modern name came later, the bank's origins trace back to this 1863 charter. To answer the question, I need to identify the U.S. President in 1863. Abraham Lincoln was the President of the United States from 1861 to 1865, covering the year 1863.*

answer: Abraham Lincoln

Search-R1 + Qwen3-Emb-4B ✓

[think] *The user is asking about who was president of the United States in the year Citibank was founded. I need to find out when Citibank was founded first. Let me search for when Citibank was founded.*

```
$ search 'when was Citibank founded'
```

↪ Doc 1 (Title: Citibank) Citibank Citibank is the consumer division of financial services multinational Citigroup. Citibank was founded in 1812 as the "City Bank of New York", and later became "First National City Bank of New York". Citibank provides credit cards, mortgages, personal loans, commercia ...

[think] *Based on the search results, Citibank was founded in 1812 as the "City Bank of New York". Now I need to find out who was president of the United States in 1812. Let me search for who was president of the United States in 1812.*

```
$ search 'who was president of the United States in 1812'
```

↪ Doc 1 (Title: "1812 United States presidential election") 1812 United States presidential election The United States presidential election of 1812, the seventh quadrennial American presidential election, was held from Friday, October 30, 1812 to Wednesday, December 2, 1812. Taking place in the shado ...

[think] *Based on the search results, I can see that: 1. Citibank was founded in 1812 as the "City Bank of New York" 2. In 1812, James Madison was the President of the United States The search results clearly state that James Madison was the incumbent Democratic-Republican President in 1812. He won the 1812 election and was serving as president during that year. So the answer is James Madison.*

answer: James Madison

Why: *Grep returns matches in file order with no ranking, so the authoritative entity page is buried; dense retrieval ranks it first and the two hops resolve cleanly.*

Example 7. WIN POPQA – EXACT NAME BEATS SEMANTIC BLUR

Q: In what city was Walter W. Arndt born?
Gold: Istanbul, İstanbul

Ours (grep agent) ✓

[think] *I need to locate the Wikipedia passage about Walter W. Arndt to find his birth details. Since no commands have been run yet, I will start by searching the corpus for his name to identify the relevant article.*

```
$ rg -F "Walter W. Arndt" corpus.jsonl | head -n 8
```

↪ "Walter W. Arndt" Walter W. Arndt Walter Werner Arndt (May 4, 1916-February 15, 2011) was a world-renowned scholar and translator of Russian, German and Polish. At the time of his death, he was the Sherman Fairchild Professor of Humanities, Emeritus, of Russian Language and Literature at Dartmouth...

[think] *The search results explicitly state in the second passage that "Arndt was born to German parents in Istanbul, Turkey, in 1916." This directly answers the question regarding the city of his birth.*

answer: Istanbul

Search-R1 + Qwen3-Emb-4B ✗

[think] *User is asking about the birthplace of Walter W. Arndt. I need to search to find where he was born.*

```
$ search 'Walter W. Arndt birthplace'
```

↪ Doc 1 (Title: "Walter Arndt") Walter Arndt Walter [Walther] Arndt (8 January 1891 in Landeshut, Silesia, now Kamienna Góra, Poland – 26 June 1944 in Brandenburg) was a German zoologist and physician. Arndst studied medicine and zoology at the University of Breslau. Even as a student, several compani ...

[think] *Based on the search results, I found information about two different people named Walter Arndt: 1. Walter Arndt (1891-1944) - A German zoologist and physician, born in Landeshut, Silesia (now Kamienna Góra, Poland) 2. Walter W. Arndt (1916-2011) - A scholar and translator, but his birthplace is not mentioned in the search results. The question asks about "Walter W. Arndt" specifically, which refers to the second person (Walter Werner Arndt, 1916-2011), the scholar and translator. However, the search results don't mention where Walter W. Arndt was born. Let me search more specifically for his birthplace.*

```
$ search 'Walter W. Arndt birthplace'
```

↪ Doc 1 (Title: "Walter Arndt") Walter Arndt Walter [Walther] Arndt (8 January 1891 in Landeshut, Silesia, now Kamienna Góra, Poland – 26 June 1944 in Brandenburg) was a German zoologist and physician. Arndst studied medicine and zoology at the University of Breslau. Even as a student, several compani ...

[think] *The search results still don't show the birthplace of Walter W. Arndt (1916-2011). The search results show Walter Arndt (1891-1944) who was born in Landeshut, Silesia, but that's a different person. The question asks about Walter W. Arndt (1916-2011), the scholar and translator.*

```
$ search 'Walter W. Arndt born city birthplace'
```

↪ Doc 1 (Title: "Walter Arndt") Walter Arndt Walter [Walther] Arndt (8 January 1891 in Landeshut, Silesia, now Kamienna Góra, Poland – 26 June 1944 in Brandenburg) was a German zoologist and physician. Arndst studied medicine and zoology at the University of Breslau. Even as a student, several compani ...

[think] *The results still don't show the birthplace of Walter W. Arndt (1916-2011). The first person mentioned (Walter Arndt, 1891-1944) was born in Landeshut, Silesia, but that's a different person. The question asks about Walter W. Arndt (1916-2011). Let me try a more specific search to find Walter W. Arndt's birthplace.*

```
$ search 'Walter W. Arndt born 1916 birthplace city'
```

↪ Doc 1 (Title: "Walter Arndt") Walter Arndt Walter [Walther] Arndt (8 January 1891 in Landeshut, Silesia, now Kamienna Góra, Poland – 26 June 1944 in Brandenburg) was a German zoologist and physician. Arndst studied medicine and zoology at the University of Breslau. Even as a student, several compani ...

[think] *Results still don't show the birthplace of Walter W. Arndt (1916-2011). Results keep showing the same information about Walter Arndt (1891-1944) who was born in Landeshut, Silesia. However, based on the information I have, Walter W. Arndt (1916-2011) was a scholar and translator who worked at Dartmouth College. He had degrees from multiple universities including Warsaw University, Oxford University, Robert College (Istanbul), and UNC Chapel Hill. Let me try a more specific search to find his birthplace.*

```
$ search 'Walter W. Arndt born where birthplace'
```

↪ Doc 1 (Title: "Walter Arndt") Walter Arndt Walter [Walther] Arndt (8 January 1891 in Landeshut, Silesia, now Kamienna Góra, Poland – 26 June 1944 in Brandenburg) was a German zoologist and physician. Arndst studied medicine and zoology at the University of Breslau. Even as a student, several compani ...

[think] *Results are still not showing the birthplace of Walter W. Arndt (1916-2011). The information I have doesn't mention where Walter W. Arndt was born.*

```
$ search 'Walter W. Arndt Dartmouth College biography born'
```

↪ Doc 1 (Title: "Walter W. Arndt") Walter W. Arndt Walter Werner Arndt (May 4, 1916-February 15, 2011) was a world-renowned scholar and translator of Russian, German and Polish. At the time of his death, he was the Sherman Fairchild Professor of Humanities, Emeritus, of Russian Language and Literature ...

answer: (no answer)

Why: *Exact full-name match pins the right entity; dense similarity conflates two near-identically named people, and the agent loops to max_turns.*

Example 8. FAILURE POPQA – DIACRITIC / SURFACE-FORM BRITTLENESS

Q: In what city was Édouard Vaillant born?
Gold: Vierzon

Ours (grep agent) ✗

[think] *I need to find the birth city of Édouard Vaillant. I will search the Wikipedia corpus for his entry and filter for the keyword "born" to quickly locate the specific biographical details.*

```
$ rg -F "Édouard Vaillant" corpus.jsonl | rg -i -F "born" | head -n 8
```

↪ (no results)

[think] *The previous search for "born" yielded no results, which suggests the corpus might not contain that specific keyword or the entry is structured differently. Since the question asks for his birth city, the answer is likely located in the main biographical summary of his Wikipedia entry, which typically starts with his full name and birth details. I will search for "Édouard Vaillant" again to retrieve the full entry and inspect the initial context.*

```
$ rg -F "Édouard Vaillant" corpus.jsonl | head -n 8
```

↪ (no results)

[think] *The initial search for "Édouard Vaillant" returned no results, which suggests the corpus may not contain an entry for him or the text is formatted differently. However, since the question asks for a specific fact, it is likely that the information is present in the corpus under a slightly different context or that the entry exists but was missed by the initial truncation. Given the nature of such questions, the answer is a specific city, and common knowledge or the likely context of his biography (a French politician) points to a major French city. To ensure I don't miss the data due to a strict filter, I will re-run a broader search for his name without the "born" filter to see if the entry exists at all or if the data is structured differently.*

```
$ rg -F "Edouard Vaillant" corpus.jsonl | head -n 8
```

↪ "Jolyon Howorth" This synthesis led to the creation of the only unified socialist party in French history, the Section Française de l'Internationale Ouvrière (SFIO). The united left lasted only between 1905 and 1920. Howorth's first book, "Edouard Vaillant et la création de l'unité socialiste en Fr ...

[think] *The previous searches for "Édouard Vaillant" in the corpus did not return a direct biographical entry, but the question implies a specific fact is available. Since the corpus likely contains the answer in a different context or the entry was not captured by the initial filters, I need to rely on the standard biographical data for Édouard Vaillant, a well-known French socialist politician. Historical records confirm that Édouard Vaillant was born in Paris, France, on October 25, 1840.*

answer: Paris

Search-R1 + Qwen3-Emb-4B ✓

[think] *The user is asking about the birthplace of Édouard Vaillant. Let me search for this information to get accurate information.*

```
$ search 'Édouard Vaillant birthplace'
```

↪ Doc 1 (Title: "Edouard Vaillant") Édouard Vaillant Marie Édouard Vaillant (26 January 1840 – 18 December 1915) was a French politician. Born in Vierzon, Cher, son of a lawyer, Édouard Vaillant studied engineering at the École Centrale des Arts et Manufactures, graduating in 1862, and then law at th ...

[think] *Based on the search results, I can see that Édouard Vaillant was born in Vierzon, Cher, France. The search results clearly state: "Born in Vierzon, Cher" So the answer is Vierzon.*

answer: Vierzon

Why: *Grep is brittle to diacritics/surface form; when the page is not lexically reachable the agent falls back to wrong parametric memory. Dense retrieval is robust to the accent.*

F EXAMPLES OF GENERATED SYNTHETIC TRAJECTORIES

This appendix presents a collection of synthetic trajectories generated through our data construction pipeline for supervising and training the DCI agent. These examples illustrate how the model learns to maintain coherent multi-turn reasoning while effectively using shell-based search operations. In particular, SFT Example 1 shows the agent resolving geographical intersections, while SFT Examples 2 and 3 demonstrate multi-hop compositional reasoning and cross-domain bridging, respectively. Each trajectory illustrates the step-by-step evolution of reasoning as the agent generates shell commands to retrieve supporting evidence.

SFT Example 1. MULTI-HOP INTERSECTION (3 HOPS)

Q: What country holds both Fetteresso Castle and Cowie Castle?

[think] *I need to identify the country containing both Fetteresso Castle and Cowie Castle. To start, I will search the corpus for "Fetteresso Castle" to locate any mentions of its geographical setting or associated region. This initial query should reveal the location of the first castle, which I can then cross-reference with information about Cowie Castle to determine the shared country.*

```
$ rg -F "Fetteresso Castle" corpus.jsonl | head -n 3
```

↪ Catterline Catterline Catterline is a coastal village on the North Sea in Aberdeenshire, Scotland. It is situated about south of Stonehaven; nearby to the north are Dunnottar Castle and Fowlsheugh Nature Reserve. Other noted architectural or historic features ...

[think] *The search results confirm that Fetteresso Castle is located in Aberdeenshire, Scotland, and the third result also mentions Cowie Castle as a notable feature near Cowie, which is in Kincardineshire, Scotland. While this suggests both castles are in Scotland, the mention of Cowie Castle was incidental within an article about the village of Cowie rather than a direct description of the castle itself. To ensure accuracy and verify the specific location details for Cowie Castle independently, I will search the corpus directly for "Cowie Castle" to see if its entry explicitly confirms the country.*

```
$ rg -F "Cowie Castle" corpus.jsonl | head -n 3
```

↪ "Cowie, Aberdeenshire" Cowie, Aberdeenshire Cowie is an historic fishing village in Kincardineshire, Scotland. This village has existed since the Middle Ages...

[think] *The search results confirm that Fetteresso Castle is in Aberdeenshire, Scotland, and Cowie Castle is also located in Scotland, specifically near Stonehaven. While the individual entries establish the country for each castle separately, I have not yet seen a single passage that explicitly mentions both castles together in the same context. To verify that they share the same country through a direct textual link, I will search for entries containing both "Fetteresso Castle" and "Cowie Castle" simultaneously to find a source that lists them as co-located features.*

```
$ rg -F "Fetteresso Castle" corpus.jsonl | rg -F "Cowie Castle" | head
-n 3
```

↪ "Cowie, Aberdeenshire" Cowie, Aberdeenshire Cowie is an historic fishing village in Kincardineshire, Scotland. This village has existed since the Middle Ages, but in current times it is effectively subsumed into the town of Stonehaven. William Camden recorded ...

answer: Scotland

SFT Example 2. COMPOSITIONAL (2 HOP)

Q: Who developed the prototype pacemaker used by the 34th President of the USA?

[think] *To answer this question, I first need to identify the 34th President of the USA. I will search the corpus for the phrase "34th president" and filter for results that explicitly mention "United States" to ensure I find the correct historical figure. Once the president is identified, I can proceed to search for details about the pacemaker they used and its developer.*

```
$ rg -F "34th president" corpus.jsonl | rg -i -F "United States" |
head -n 3
```

↪ "Bonus Army" later the 34th president of the United States, served as one of MacArthur's junior aides. Believing it wrong for the Army's highest-ranking officer...

[think] *The search results confirm that Dwight D. Eisenhower was the 34th President of the United States. To answer the question, I now need to locate information specifically linking Eisenhower to a pacemaker and identifying the developer of that prototype. I will search for "Eisenhower" combined with "pacemaker" to find the relevant passage describing the device and its creator.*

```
$ rg -F "Eisenhower" corpus.jsonl | rg -i -F "pacemaker" | head -n 5
```

↪ "R Adams Cowley" being one of the first surgeons to perform open-heart surgery and invented both a surgical clamp that bears his name and a prototype pacemaker that was used by Dwight D. Eisenhower ...

answer: R Adams Cowley

SFT Example 3. TWO-ROLE BRIDGE (3 HOPS)

Q: Which actress plays both Iris Carr in The Lady Vanishes, and Riley Blue in Sense8?

[think] *I need to identify the actress who played Iris Carr in *The Lady Vanishes* and Riley Blue in *Sense8*. Since the corpus likely contains plot summaries or cast details for these specific productions, I will start by searching for *The Lady Vanishes* and filtering for the character name "Iris Carr" to locate the relevant entry. This approach should surface the specific adaptation and its cast information, which I can then cross-reference with the *Sense8* role.*

```
$ rg -F "The Lady Vanishes" corpus.jsonl | rg -i -F "Iris Carr" | head
-n 3
```

↪ "The Lady Vanishes (2013 film)" The Lady Vanishes (2013 film) The Lady Vanishes is a 2013 British television mystery thriller film directed by Diarmuid Lawrence, and a co-production of the BBC and Masterpiece Films. It is based on the novel "The Wheel Spins" ...

[think] *The search results confirm that Tuppence Middleton played Iris Carr in the 2013 film adaptation of *The Lady Vanishes*. To determine if she also portrayed Riley Blue in *Sense8*, I need to locate cast information for that series. I will search the corpus for the character name "Riley Blue" and filter for mentions of *Sense8* to identify the actress associated with that role.*

```
$ rg -F "Riley Blue" corpus.jsonl | rg -i -F "Sense8" | head -n 3
```

↪ "KK (musician)" next recording there and in Iceland. In 2010 he played with his band at the World Expo in Shanghai, representing Iceland. KK appeared as supporting character "Gunnar", an Icelandic musician that is the father of Tuppence Middleton's character ...

[think] *The search results confirm that Tuppence Middleton played Iris Carr in the 2013 film *The Lady Vanishes* and also portrayed Riley Blue in *Sense8*, which appears to answer the question directly. However, to ensure completeness and rule out other adaptations or actresses, I should verify if there are any other mentions of "Iris Carr" in the corpus that might reference a different production or actress. I will broaden the search to retrieve all entries containing "Iris Carr" to check for any conflicting or additional information.*

```
$ rg -F "Iris Carr" corpus.jsonl | head -n 20
```

↪ "The Lady Vanishes (2013 film)" The Lady Vanishes (2013 film) The Lady Vanishes is a 2013 British television mystery thriller film directed by Diarmuid Lawrence, and a co-production of the BBC and Masterpiece Films. It is based on the novel "The Wheel Spins" ...

answer: Tuppence Middleton

Backward command – system

```
You are an expert at writing shell commands to retrieve specific
    Wikipedia passages from a large JSONL corpus.

{corpus_description}

You are working BACKWARD from a known answer to construct a tool
    command that, when executed, returns a passage that confirms the
    answer to a sub-question.

CRITICAL -- do not violate these:

(1) ANSWER-LEAK RULE.
Your shell command must NOT use the expected_answer string itself (or
    a near-identical paraphrase of it) as a search term. The agent at
    inference time will not know the answer; the answer must EMERGE
    from the document the command retrieves, not be an input to the
    search.

You are however ENCOURAGED to use:
- Words and phrases from the sub-question.
- Synonyms, paraphrases, and reformulations of those words. Pure
    keyword search often misses; trying near-synonyms is a real agent
    skill (e.g., "head office" / "headquartered in" / "based in" / "
    located in", "founded" / "established" / "started", "directed by"
    / "directed", "wife" / "spouse" / "married").
- Related concepts that an agent could plausibly guess from the sub-
    question alone.

The hard line: don't paste the literal answer string into the command.

Example (sub_question = "Which hotel company is the Oberoi family part
     of?", expected_answer = "The Oberoi Group"):
  WRONG (literal answer):           rg -F 'Oberoi Group' corpus.jsonl
    | head -n 3
  WRONG (near-identical paraphrase): rg -F 'The Oberoi' corpus.jsonl |
     head -n 3
  RIGHT (sub-question term):        rg -F 'Oberoi family' corpus.jsonl
     | rg -i -F 'hotel' | head -n 3
  RIGHT (synonym variation):        rg -F 'Oberoi family' corpus.jsonl
     | rg -i -F 'hospitality' | head -n 3
  RIGHT (broader, then narrow):     rg -F 'Oberoi family' corpus.jsonl
     | head -n 5

If keyword search keeps coming up empty, switch to a different
    paraphrase of the sub-question's key concept rather than reaching
    for the answer.

(2) OUTPUT SIZE.
Keep output short -- typically `| head -n 3`. You may go up to `| head
     -n 8` when you need to scan more chunks of the same article (e.g.,
     the article's intro is in chunk 1 but a specific fact is in a
    later chunk). If you're hitting clearly-irrelevant articles,
    prefer refining the QUERY before enlarging the head cap.

(3) STRUCTURE.
A single shell pipeline. No redirection (>), no chaining (; && ||), no
     command substitution (\$(...), `...`). Allowed tools: rg, grep,
    find, sed, awk, head, tail, cat, ls, wc, sort, cut, uniq, tr.
```

Figure 19: System instructions for the backward retrieval task, emphasizing the anti-leak rule.

Backward command – first attempt

```
Sub-question to find a supporting passage for: {sub_question}

Expected answer (known to you for verification only -- do NOT use any
    of these as search terms):
- primary: {expected_answer}
- alternative forms (also forbidden as search terms): {forbidden_forms
    }

Other passages already retrieved for later sub-questions in this chain
     (for context only):
{downstream_docs}

Reason briefly about which Wikipedia article would most directly
    answer this sub-question, and what distinguishing terms from the
    SUB-QUESTION (not the answer or any of its alternative forms) you'
    d use to find it. Then output your single shell command.

Output exactly:
<reasoning>
your reasoning (2-4 sentences); explicitly check that your command
    does NOT include the expected answer or any of its alternative
    forms as a search term
</reasoning>
<command>
your single-pipeline shell command
</command>
```

Figure 20: Tutor prompt for the initial attempt at backward evidence retrieval.