RETRIEVAL IS CHEAP, SHOW ME THE CODE: EXECUTABLE MULTI-HOP REASONING FOR RETRIEVAL-AUGMENTED GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Retrieval-Augmented Generation (RAG) has become a standard approach for knowledge-intensive question answering, but existing systems remain brittle on multi-hop questions that require intermediate results to persist and constrain later retrieval and reasoning steps. We argue that the central limitation is representational: free-form reasoning leaves state and dependencies implicit in text, forcing the model to repeatedly reconstruct them and making entity drift or inconsistent composition difficult to prevent. We introduce **PyRAG**, a framework that represents multi-hop RAG as an executable intermediate representation. A query-specific Python program invokes retrieval and QA tools, binds intermediate answers to variables, and connects subsequent steps through explicit data dependencies. This representation externalizes reasoning state, enables deterministic composition and inspectable execution traces, and yields substantial gains even before considering refinement mechanisms. Execution further enables compiler-grounded self-repair and execution-driven adaptive retrieval as natural, training-free extensions of the same interface. Experiments on five QA benchmarks (PopQA, HotpotQA, 2WikiMultihopQA, MuSiQue, and Bamboogle) show consistent improvements in both training-free and RL-trained settings. Progressive ablations demonstrate that performance improves as reasoning becomes more explicit: basic execution provides an additional gain over a non-executed structured plan, while the largest aggregate improvement comes from the full execution-enabled pipeline, including adaptive retrieval and compiler-grounded self-repair.

1 INTRODUCTION

Retrieval-Augmented Generation (RAG) (Gao et al., 2023b; Lewis et al., 2020) grounds large language models (LLMs) in external evidence for knowledge-intensive question answering. Yet many questions are inherently multi-hop (Ho et al., 2020; Yang et al., 2018; Trivedi et al., 2022; Press et al., 2023; Shi et al., 2026): answering them requires retrieving multiple facts, preserving intermediate results, and composing them according to their dependencies. For example, answering *"Who is older, Jed Hoyer or John William Henry II?"* requires retrieving and correctly associating two birth dates before comparing them. Thus, the challenge is not retrieval alone, but maintaining reliable state across retrieval and reasoning steps.

Most multi-hop RAG systems represent this computation in free-form language. Chain-of-thought prompting (Wei et al., 2022), iterative retrieve-and-reason methods (Trivedi et al., 2023; Yao et al., 2022; Shao et al., 2023; Shi et al., 2026), and reinforcement-learned search agents (Jin et al., 2025; Song et al., 2025; Zheng et al., 2025; Chen et al., 2025) produce multi-step trajectories, but keep state implicit in the token sequence. Intermediate answers remain embedded in prose, dependencies are inferred rather than enforced, and later steps must reconstruct relevant state from preceding text. This makes entity drift, forgotten constraints, and inconsistent composition difficult to prevent or localize. Increasing the reasoning budget does not directly resolve this representational bottleneck: the model must still encode, recover, and validate its state within an unstructured narrative.

We therefore treat the reasoning representation as a central design choice for multi-hop RAG. We propose an executable intermediate representation in which retrieval and answering are explicit

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

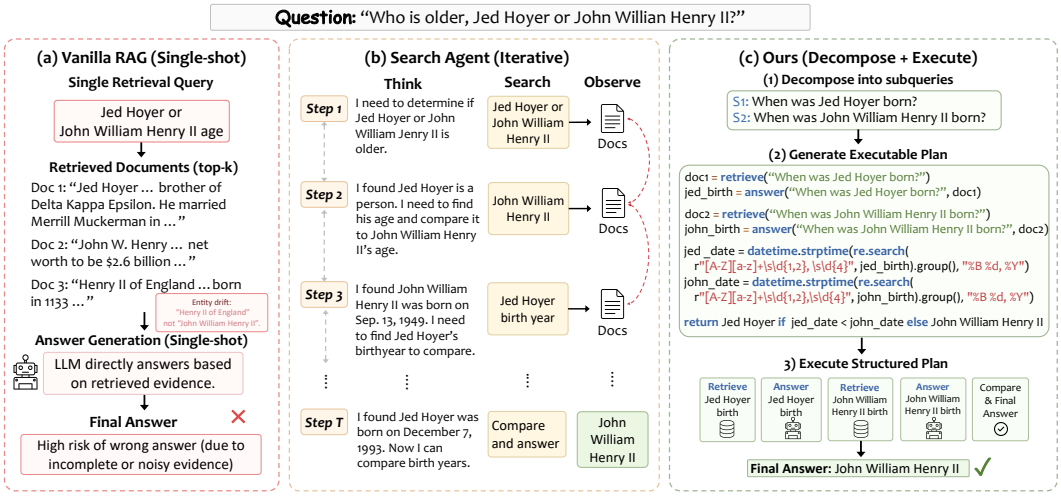


Figure 1: **Reasoning representations for multi-hop RAG.** Vanilla RAG relies on a single retrieval context; Search Agents keep state and dependencies implicit in free-form text; PyRAG executes a program with persistent variables and explicit data flow.

operations, intermediate results are named variables, and downstream steps consume those variables through visible data dependencies. Moving state from prose into an execution environment preserves intermediate values, enforces dependencies, and supports deterministic composition and inspection. It also localizes failures: an incorrect intermediate value or malformed operation can be traced to a specific program step instead of being entangled with an entire textual trajectory.

Unlike prior program-guided reasoning over self-contained inputs (Gao et al., 2023a; Chen et al., 2022; Cheng et al., 2022; Lyu et al., 2023; Pan et al., 2023a;b), open-domain QA must retrieve missing evidence at runtime. An intermediate answer may determine the next entity, relation, or constraint, requiring dynamic retrieval and data-dependent execution rather than merely translating a fixed input into a symbolic calculation.

Based on this view, we introduce PyRAG, which synthesizes and executes a query-specific Python program over two primitives: `retrieve(query, topk)` and `answer(query, docs)`. A Decompose Agent identifies atomic information needs, a Plan Agent compiles them into a program, and an Answer Agent extracts grounded values. Execution binds these values to variables and passes them explicitly to subsequent operations, yielding a persistent, inspectable data-flow computation (Figure 1).

The executable interface also enables compiler-grounded self-repair from runtime errors and selective retrieval expansion when an intermediate step lacks evidence. These are refinement mechanisms built on the primary benefit: making state, dependencies, and composition explicit. Our ablations separate the effects of structure, basic execution, and these execution-time refinements, allowing us to distinguish the value of the representation itself from the additional gains enabled at runtime.

We evaluate PyRAG on five open-domain QA benchmarks in both training-free and RL-trained settings. In the training-free 7B setting, PyRAG improves average Exact Match over Vanilla RAG by 11.8 points and outperforms strong iterative retrieval baselines. Reinforcement learning further improves accuracy while preserving the same executable interface, with gains that transfer across model families and out-of-domain benchmarks. Our contributions are:

- We identify **implicit reasoning state** as a central limitation of multi-hop RAG and formulate retrieval and reasoning as an executable representation with explicit variables, operations, and dependencies.
- We introduce PyRAG, which dynamically synthesizes and executes retrieval programs, with self-repair and adaptive retrieval supported through the same interface.
- Across five benchmarks, we show through progressive ablations and model-specialization analysis that the executable representation and its enabled runtime refinements jointly drive the gains;

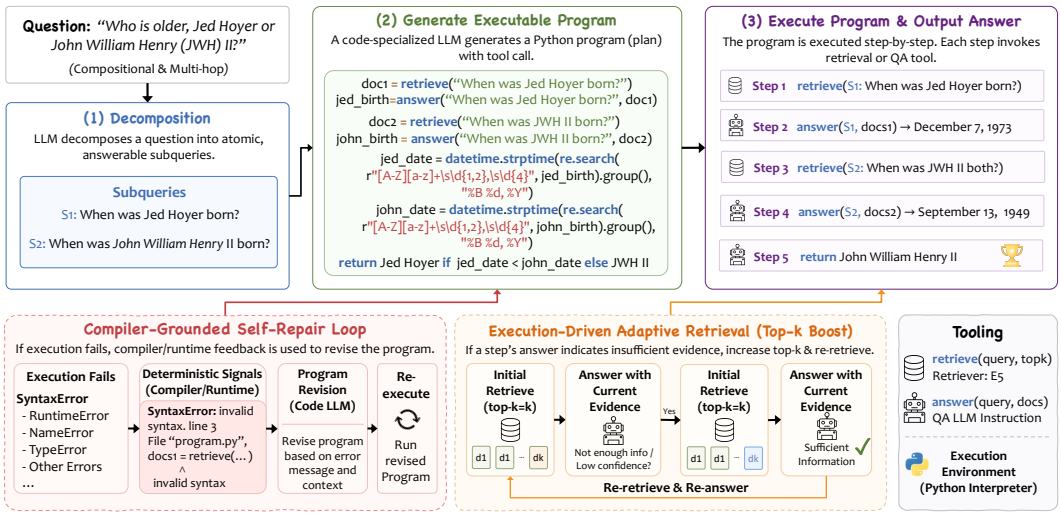


Figure 2: **The PyRAG framework.** Given a multi-hop question, PyRAG proceeds in three stages: (1) Decompose: an LLM breaks the question into atomic, independently answerable sub-queries; (2) Plan: a planning LLM synthesizes an executable Python program over two tool primitives, `retrieve(query, topk)` and `answer(query, docs)`, where intermediate results are bound to variables and composed through explicit data dependencies; (3) Execute: the program is run step-by-step in a Python interpreter, producing an inspectable trace and a grounded final answer. The executable representation additionally supports two optional refinement mechanisms: (A) Compiler-Grounded Self-Repair, which uses runtime exceptions (e.g., `SyntaxError`, `NameError`) as deterministic signals for the planner to revise and re-execute the program; and (B) Execution-Driven Adaptive Retrieval, which boosts the top-k retrieval budget for sub-steps whose answer indicates insufficient evidence. Both mechanisms are training-free and rely on grounded execution feedback rather than LLM self-reflection.

PyRAG improves over Vanilla RAG by 11.8 average EM at 7B, and PyRAG-RL generalizes across Qwen and LLaMA backbones.

2 METHOD

2.1 OVERVIEW

We present PyRAG, a framework that introduces an executable interface for multi-hop RAG, as shown in Figure 2. Instead of representing reasoning as free-form natural language, PyRAG decomposes the problem into a sequence of structured steps and executes them through a program.

Given a question q, PyRAG consists of three components: (1) a **decomposition agent** that breaks q into atomic sub-queries, (2) a **planning agent** that generates an executable program describing the reasoning process, and (3) an **answer agent** that produces answers based on retrieved evidence.

At inference time, the generated program is executed step-by-step, with each step corresponding to a retrieval or question-answering operation. Therefore, we shift multi-hop reasoning from opaque narrative to an explicit, controllable, and verifiable execution process.

Table 1 summarizes how this executable representation differs from single-shot RAG and free-form search agents.

2.2 MOTIVATION: MULTI-HOP QA AS STEP-BY-STEP COMPUTATION

We argue that multi-hop question answering can be naturally viewed as a form of step-by-step computation. Resolving multi-hop queries requires systematically decomposing them into constituent

Table 1: **Comparison of reasoning paradigms.** Search Agents are marked partial on Structured Planning because their plans remain embedded in a reactive thought trace rather than being materialized as an explicit artifact. Their natural-language trajectories have no persistent variables, enforced data flow, or deterministic execution. PyRAG primarily addresses this representational limitation by encoding multi-hop reasoning as an executable program; the resulting execution signals additionally enable grounded reflection.

Paradigm	Multi-hop Capability	Interpretability	Structured Planning	Reflection	Executable Interface
Vanilla RAG (Single-shot)	✗	✗	✗	✗	✗
Search Agent (Free-form)	✓	✓	△	△	✗
PyRAG (Executable program)	✓	✓	✓	✓	✓

✓: supported △: partially supported ✗: not supported

sub-problems, computing intermediate results, and ultimately synthesizing these results into a final answer (Trivedi et al., 2023; Yao et al., 2022; Shao et al., 2023; Wei et al., 2022).

This structured process closely aligns with the fundamental principles of programmatic execution: a program defines a sequence of functional operations, maintains intermediate variables, and enforces dependencies between steps (Huang et al., 2024). Code-specialized language models are explicitly trained for such behavior. They are optimized to generate structured programs that decompose tasks, maintain state through variables, and perform consistent step-by-step execution. As a result, they provide a strong inductive bias for multi-hop QA processing.

Motivated by this observation, we cast multi-hop RAG as a program synthesis problem in which the reasoning process is represented as an executable plan. This allows us to directly leverage the step-by-step reasoning capabilities of code models to enable explicit control and verification, rather than relying on such structure to emerge from free-form natural language reasoning.

Our formulation connects program-aided reasoning with multi-hop retrieval. PAL Gao et al. (2023a) and Program-of-Thoughts Chen et al. (2022) primarily use executable programs to perform computation over information supplied in the problem. In multi-hop RAG, some required facts are unavailable when the program is synthesized: intermediate answers obtained during execution must determine subsequent retrieval queries and computations. DSP and DSPy Khattab et al. (2024) provide frameworks for composing and optimizing retrieval and language-model pipelines; our focus is on synthesizing a query-specific executable program that makes these dependencies explicit and preserves intermediate results as runtime state. These perspectives are complementary, and PyRAG could be implemented within such frameworks.

The central design choice is executable semantics rather than Python-specific syntax. In principle, any executable language or intermediate representation that supports tool invocation, persistent state, and explicit data dependencies could serve this role. We adopt Python because of its widespread use in machine learning and its mature ecosystem for integrating retrieval tools, language models, and computation. Our experiments evaluate this Python realization; other execution backends remain an avenue for future investigation.

2.3 PyRAG AGENTS

Decomposition Agent. Given a question q, the decomposition agent produces a sequence of sub-queries $s = [s_1, \ldots, s_n]$, where each sub-query is designed to be answerable with a single retrieval step. This step imposes an explicit structure on the reasoning process, but does not yet define how the steps should be executed or combined.

Answer Agent. The answer agent takes a sub-query and a set of retrieved documents as input and produces a short answer. It is implemented using an instruction-following LLM and is responsible for extracting information from retrieved evidence. It also performs final semantic aggregation when the required composition cannot be expressed more reliably as a deterministic Python operation.

Planning Agent. The planning agent is the core component of PyRAG. Given the original question q and the decomposed sub-queries s, it generates a program π that specifies how to solve the task through a sequence of retrieval and answering operations.

2.4 Executable Planning

The execution tool exposes two API functions:

- `retrieve(query, topk=k)`: returns the top-k relevant documents for the given query, where k can be increased adaptively at execution time (Sec. 2.6).
- `answer(query, docs=None)`: returns a grounded short answer when documents are supplied, or performs semantic synthesis from facts embedded in the query when `docs` is omitted.

The planning agent generates a program that uses these API functions together with variable assignments and native Python operations, including string normalization, comparison, Boolean logic, and arithmetic. Each step retrieves evidence, computes an intermediate answer, and stores the result in a variable. These variables are then reused in subsequent steps.

This formulation makes the reasoning process explicit: instead of implicitly encoding intermediate states in text, the program stores them as variables and connects them through data dependencies. When the required composition has well-defined operational semantics, the program computes the final answer directly; otherwise, it invokes the answer agent in synthesis mode using the explicitly supplied intermediate facts.

2.5 Execution

The generated program π is executed step-by-step. At each step, the system invokes one of the two API functions or evaluates a native Python operation over previously bound values. Tool outputs and computed values are stored for later use.

This execution process yields an execution trace, which records all intermediate queries, retrieved documents, and answers. The trace provides a transparent view of the reasoning process and enables debugging and analysis.

2.6 Execution-Enabled Refinement

The executable representation is useful independently of refinement because it externalizes intermediate state and enforces data dependencies during normal execution. It also creates grounded signals that can be used for targeted refinement when execution fails or evidence is insufficient.

Compiler-Grounded Self-Repair. If the generated program fails to execute due to invalid operations or inconsistent variable usage, the execution environment returns a structured error signal. The planning agent can then revise the program based on this feedback and re-execute it.

Adaptive Retrieval. If an intermediate answer indicates that the available evidence is insufficient, the system can selectively increase the retrieval scope for that step and re-run the corresponding operation. This allows targeted correction without modifying the entire reasoning plan.

These mechanisms are enabled by the executable formulation, but are not required for its core operation. They add targeted recovery without requiring additional training or replacing the explicit program state with free-form self-reflection.

3 Experiments

3.1 Experimental Setup

Benchmarks. We evaluate on five open-domain QA benchmarks spanning single-hop and multi-hop reasoning: PopQA (Mallen et al., 2023), HotpotQA (Yang et al., 2018), 2WikiMultihopQA (Ho et al., 2020), MuSiQue (Trivedi et al., 2022), and Bamboogle (Press et al., 2023).

Table 2: Exact Match (%) of **training-free** methods on five QA benchmarks. All methods use the same setting and identical evaluation splits. Best result within each backbone size is in **bold**; second best is underlined. Δ shows improvement over Vanilla RAG within the same backbone. For each PyRAG row, the backbone named in the block is reused across all three logical roles, which are invoked separately.

Method	PopQA	HotpotQA	2WikiMQA	MuSiQue	Bamboogle	Avg.
Backbone: Qwen2.5-7B-Instruct						
Direct Inference	14.0	18.3	12.6	3.1	12.0	12.0
CoT (Wei et al., 2022)	5.4	9.2	10.8	2.2	23.2	10.2
Vanilla RAG	26.7	28.9	18.9	4.7	16.0	19.0
Self-Ask (Press et al., 2023)	29.4	30.2	21.5	6.8	22.4	22.1
IRCoT (Trivedi et al., 2023)	32.6	32.7	24.8	9.1	24.0	24.6
ITER-RETGEN (Shao et al., 2023)	31.4	32.5	28.9	8.7	29.6	26.2
PyRAG **(ours)**	**33.5**	**34.0**	**33.4**	**11.8**	**41.6**	**30.9**
Δ *vs. Vanilla RAG*	**+6.8**	**+5.1**	**+14.5**	**+7.1**	**+25.6**	**+11.8**
Backbone: Qwen2.5-72B-Instruct						
Direct Inference	19.7	30.6	20.6	5.5	17.6	18.8
CoT (Wei et al., 2022)	24.4	33.2	25.4	9.9	20.0	22.6
Vanilla RAG	33.2	36.8	30.4	10.6	21.6	26.5
Self-Ask (Press et al., 2023)	41.4	48.2	32.5	11.8	26.4	32.1
IRCoT (Trivedi et al., 2023)	44.6	50.9	35.8	14.2	28.0	34.7
ITER-RETGEN (Shao et al., 2023)	43.4	50.5	40.2	13.8	33.6	36.3
PyRAG **(ours)**	**45.5**	**52.0**	**44.4**	**16.9**	**45.6**	**40.9**
Δ *vs. Vanilla RAG*	**+12.3**	**+15.2**	**+14.0**	**+6.3**	**+24.0**	**+14.4**

Exact Match (EM) is used as the primary metric for all benchmarks. HotpotQA serves as the in-domain training set for RL-trained variants; all remaining datasets are evaluated out-of-domain.

Baselines. We compare against the following categories of methods:

Training-free baselines. **Direct Inference** and **CoT** (Wei et al., 2022) require no retrieval. **Vanilla RAG** (Lewis et al., 2020) performs single-step retrieve-then-read. **Self-Ask** (Press et al., 2023) decomposes questions into sub-questions with interleaved retrieval. **IRCoT** (Trivedi et al., 2023) interleaves chain-of-thought reasoning with iterative retrieval. **ITER-RETGEN** (Shao et al., 2023) alternates between retrieval and generation across multiple rounds.

RL-trained baselines. **RAG-SFT** and **RAG-RL** are supervised fine-tuning and reinforcement learning variants of a standard RAG pipeline. **ZEROSEARCH** (Sun et al., 2025), **Search-R1** (Jin et al., 2025) and **StepSearch** (Zheng et al., 2025) are recent RL-based methods that train models to perform adaptive retrieval.

Our methods. PyRAG is our training-free multi-agent framework; **PyRAG-RL** further fine-tunes the framework with reinforcement learning. Unless stated otherwise, all PyRAG variants use Qwen2.5-7B-Instruct as the backbone. The Decompose, Plan, and Answer Agents are distinct logical roles invoked with separate prompts, but reuse the same backbone checkpoint; they are not three independently sized foundation models. For PyRAG-RL, the base backbone is shared and each role has its own LoRA adapter. The only code-specialized configuration is the controlled comparison in Figure 5.

Implementation Details. We follow the retrieval and data setup of Search-R1 (Jin et al., 2025) exactly: an E5-base dense retriever over the Wikipedia 2018 dump (Karpukhin et al., 2020), with the same training splits and evaluation data preprocessing. The default number of retrieved passages per sub-query is k=5. When an `answer()` call returns an insufficient-information response (e.g. "unknown" or "cannot answer"), the runner automatically re-executes the same code with an increased retrieval budget of k=10 for the implicated steps. Unless otherwise stated, results produced under our evaluation protocol are averaged over three independent runs. For external RL baselines in Table 3, we report the values published in the original papers. Additional implementation details, including training, are provided in Appendix E.1.

Table 3: Exact Match (%) of RL-trained methods on four QA benchmarks. All methods are evaluated under the same retrieval setting. Results for all cited methods are taken directly from their original papers. Best result within each backbone size is in **bold**; second best is underlined. † denotes in-domain evaluation; remaining datasets are out-of-domain. For PyRAG-RL, the listed backbone is shared across the three roles, with a separate role-specific LoRA adapter for each.

Method	HotpotQA†	2WikiMQA	MuSiQue	Bamboogle	Avg.
Backbone: Qwen2.5-7B-Instruct					
Vanilla RAG	28.9	18.9	4.7	16.0	17.1
RAG-SFT	32.4	22.6	6.8	27.2	22.3
RAG-RL	35.2	34.7	9.6	29.6	27.3
ZEROSEARCH (Sun et al., 2025)	34.6	35.2	18.4	27.8	29.0
Search-R1 (Jin et al., 2025)	37.0	41.4	14.6	36.8	32.4
StepSearch (Zheng et al., 2025)	38.6	36.6	**22.6**	40.0	34.5
PyRAG-RL (ours)	**40.5**	**49.4**	20.7	**46.4**	**39.3**
Backbone: Qwen2.5-3B-Instruct					
Vanilla RAG	25.5	22.6	4.7	8.0	15.2
RAG-SFT	28.9	25.9	6.4	13.6	18.7
RAG-RL	29.7	27.6	7.3	16.8	20.4
ZEROSEARCH (Sun et al., 2025)	27.4	30.0	9.8	11.1	19.6
Search-R1 (Jin et al., 2025)	32.4	31.9	10.3	26.4	25.3
StepSearch (Zheng et al., 2025)	**34.5**	32.0	**17.4**	34.4	29.6
PyRAG-RL (ours)	34.2	**32.5**	16.8	**36.0**	**29.9**
Backbone: Qwen3-4B-Instruct					
Vanilla RAG	27.1	16.7	4.3	15.2	15.8
RAG-SFT	30.5	20.1	6.2	25.6	20.6
RAG-RL	33.2	31.8	8.8	28.0	25.5
PyRAG-RL (ours)	**38.4**	**45.1**	**18.6**	**43.2**	**36.3**
Backbone: LLaMA-3.1-8B-Instruct					
Vanilla RAG	30.3	19.4	6.3	17.6	18.4
RAG-SFT	34.1	23.3	8.5	29.6	23.9
RAG-RL	37.5	35.2	11.4	32.0	29.0
PyRAG-RL (ours)	**43.2**	**50.1**	**22.1**	**48.0**	**40.9**

3.2 Main Results

Table 2 and Table 3 report the main results under training-free and RL-trained settings, respectively.

Training-free results. Under the training-free setting (Table 2), PyRAG consistently outperforms all baselines across both backbone sizes. With Qwen2.5-7B-Instruct, PyRAG achieves an average EM of 30.9, surpassing the strongest baseline ITER-RETGEN by +4.6 points and Vanilla RAG by +11.8 points. Gains are most pronounced on compositional multi-hop benchmarks: +14.5 on 2WikiMQA and +25.6 on Bamboogle relative to Vanilla RAG, datasets specifically designed to stress systems that cannot chain multiple retrieval steps. On PopQA and HotpotQA, PyRAG also achieves the best results (33.5 and 34.0), demonstrating that the structured decompose-plan-answer pipeline does not degrade performance on relatively simpler queries. Scaling to Qwen2.5-72B-Instruct amplifies these trends: PyRAG reaches an average of 40.9, outperforming ITER-RETGEN by +4.6 and delivering the largest single-dataset gain on Bamboogle (+24.0 over Vanilla RAG).

RL-trained results. Table 3 compares PyRAG trained with reinforcement learning (PyRAG-RL) against RL- and SFT-based baselines. With Qwen2.5-7B, PyRAG-RL obtains 39.3 average EM, exceeding Search-R1 by 6.9 points and StepSearch by 4.8, with the best HotpotQA, 2WikiMQA, and Bamboogle scores; StepSearch remains higher on MuSiQue. At 3B, PyRAG-RL achieves

the best average (29.9), narrowly exceeding StepSearch (29.6), and improves over RAG-RL by 9.5 points. It obtains the highest 3B scores on 2WikiMQA (32.5) and Bamboogle (36.0), while StepSearch remains slightly higher on HotpotQA and MuSiQue. Despite using less than half the parameters of the default 7B backbone, the gains remain strong. PyRAG-RL also transfers across model families, reaching 36.3 on Qwen3-4B and 40.9 on LLaMA-3.1-8B, gains of 10.8 and 11.9 over their RAG-RL counterparts. These comparisons should account for supervision differences: StepSearch uses stronger step-level rewards (Zheng et al., 2025), whereas PyRAG primarily changes the reasoning representation. Appendix E.5 reports an LLM-as-a-Judge evaluation that accounts for non-exact surface forms and preserves the same method ranking.

3.3 Ablation Studies of Executable Reasoning

From free-form plans to executable programs. With the backbone, retriever, retrieval budget, and tools fixed, we compare a Free-form Plan that expresses the strategy in natural language and leaves intermediate state in text; a non-executed Structured IR with typed operations, named outputs, and explicit dependency edges; and an Executable IR that encodes the same operations as a Python program with runtime variables and enforced data flow. This progression separates the benefit of making the plan explicit from the additional benefit of enforcing it during execution. Execution-time refinements are evaluated separately in Table 4.

Figure 3 shows that Structured IR raises average EM from 21.3 to 25.8, while full PyRAG reaches 36.3; Bamboogle improves from 16.0 to 24.8 to 41.6. The Executable IR point denotes the full pipeline, including adaptive retrieval and self-repair; Table 4 separates basic execution from these refinements.

Contributions of execution-time refinement. Table 4 decomposes the aggregate Structured-to-Executable gap. With both refinements disabled, Basic Execution improves over Structured IR from 25.8 to 28.5 average EM, isolating a 2.7-point benefit from persistent runtime state and enforced data flow. Its 7.2-point gain over Vanilla RAG additionally includes the structured representation and therefore is not an execution-only comparison.

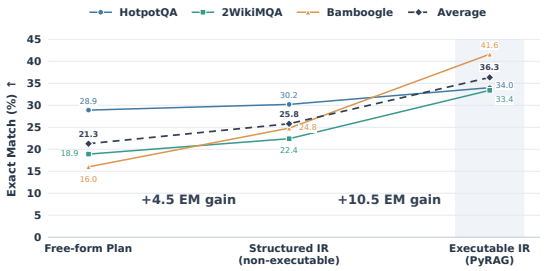


Figure 3: Effect of reasoning representation with backbone, retriever, budget, and tools matched. Executable IR denotes full PyRAG; Table 4 separates its components. Scores are EM (%).

Starting from Basic Execution, adaptive retrieval and self-repair independently raise the average to 31.5 and 32.4; together they reach 36.3, a further 7.8-point gain, showing that the mechanisms are complementary. Because Structured IR has no matched retrieval-expansion or exception-retry mechanisms, the full 10.5-point gap should not be attributed to interpreter execution alone; a stricter isolation would equip the non-executable baseline with both controls.

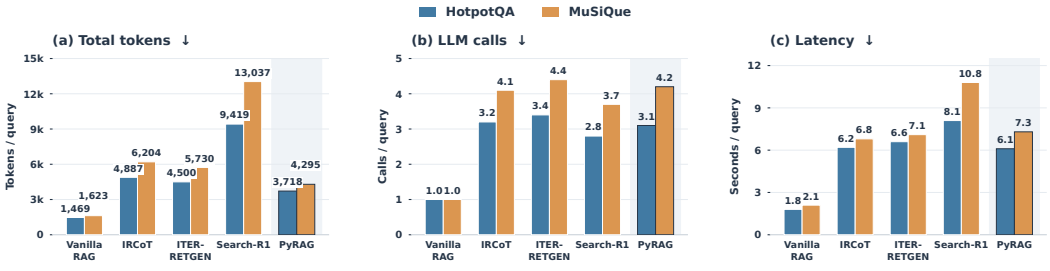


Figure 4: Efficiency on HotpotQA and MuSiQue. PyRAG denotes the Qwen2.5-7B PyRAG-RL system in Table 3.

Table 4: Executable-representation ablation with Qwen2.5-7B-Instruct. Scores are Exact Match (%).

Method	Executable Program	Adaptive Retrieval	Compiler-Grounded Self-Repair	HotpotQA	2WikiMQA	Bamboogle	Avg.
Vanilla RAG	✗	✗	✗	28.9	18.9	16.0	21.3
PyRAG (Basic Execution)	✓	✗	✗	31.5	28.3	25.6	28.5
+ Adaptive Retrieval	✓	✓	✗	32.6	30.6	31.2	31.5
+ Compiler-Grounded Self-Repair	✓	✗	✓	32.3	31.4	33.6	32.4
Full PyRAG	✓	✓	✓	**34.0**	**33.4**	**41.6**	**36.3**

Executable reasoning versus code specialization. We test whether the gains arise from the executable interface or merely from a code-specialized backbone. The instruction-model condition uses Qwen2.5-7B-Instruct for every role, while the code-model condition uses Qwen2.5-Coder-7B-Instruct (Hui et al., 2024); each PyRAG condition still reuses one checkpoint across the Decompose, Plan, and Answer roles.

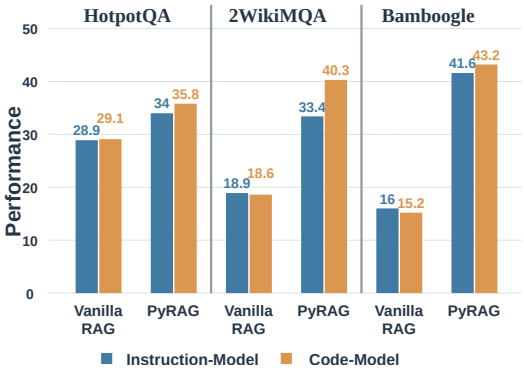


Figure 5: Backbone specialization: Qwen2.5-7B-Instruct versus Qwen2.5-Coder-7B-Instruct. PyRAG reuses the checkpoint across all roles.

Figure 5 shows that code specialization has negligible effect on Vanilla RAG (e.g., 28.9 vs. 29.1 on HotpotQA and 18.9 vs. 18.6 on 2WikiMQA). In contrast, the instruction-model PyRAG already improves HotpotQA, 2WikiMQA, and Bamboogle from 28.9/18.9/16.0 to 34.0/33.4/41.6. The coder backbone then adds 1.8/6.9/2.0 points, indicating that the executable interface drives the primary gain and code specialization is complementary.

3.4 Analysis

Efficiency Analysis Figure 4 compares the Qwen2.5-7B PyRAG-RL system with retrieval baselines. The Decompose and Plan Agents are counted as separate calls, and totals include all `answer()` invocations, self-repair, and adaptive-retrieval re-execution. On HotpotQA, PyRAG-RL averages 3.1 calls, 3,718 tokens, and 6.1 seconds, versus 2.8 calls, 9,419 tokens, and 8.1 seconds for Search-R1. On MuSiQue, the corresponding values are 4.2 calls, 4,295 tokens, and 7.3 seconds, compared with 3.7 calls, 13,037 tokens, and 10.8 seconds. Thus, PyRAG-RL makes slightly more but substantially shorter calls. Relative to Search-R1, this reduces token usage by approximately 61% and 67%, and latency by 25% and 32%, on HotpotQA and MuSiQue, respectively. These measurements correspond to the system reported in Table 3, not a merged-agent variant.

Case Study Qualitative execution traces appear in Appendix G.

4 Conclusion

We presented PyRAG, a framework that addresses the implicit-state bottleneck in multi-hop RAG through an executable reasoning representation. By encoding retrieval and reasoning as a Python program, PyRAG makes intermediate state persistent, data dependencies explicit, and composition executable and inspectable. Across five benchmarks under both training-free and RL-trained settings, PyRAG consistently improves over strong baselines, with the largest gains on compositional multi-hop datasets. Ablations show that structure and basic execution each contribute, while adaptive retrieval and compiler-grounded self-repair provide complementary gains through the shared execution trace. These findings suggest that executable interfaces offer a practical path toward more controllable and inspectable retrieval agents beyond the specific Python realization studied here.

AI USE STATEMENT

We used generative AI tools solely for language editing and polishing of the manuscript. These tools were not used to generate research ideas, formulate the methodology, conduct experiments, analyze results, write code, or draw conclusions. All AI-assisted edits were reviewed and verified by the authors, who take full responsibility for the final content of the paper.

ETHICS STATEMENT

This work uses publicly available question-answering benchmarks and a Wikipedia corpus; it does not involve human subjects or the collection of new personal data. Nevertheless, the underlying corpora and language models may contain biases, outdated information, or incorrect claims, and the system may produce unsupported answers. Consequently, **PyRAG** should not be used for high-stakes decision making without domain-specific validation and human oversight. The execution of model-generated programs also introduces security risks if the interpreter is not isolated. Practical deployments should therefore sandbox execution, restrict filesystem and network access, expose only audited tool interfaces, and validate generated programs before running them. More broadly, automated retrieval and synthesis can be misused to scale misinformation or privacy-invasive information gathering; users are responsible for ensuring that applications comply with applicable privacy, copyright, and data-use requirements.

REPRODUCIBILITY STATEMENT

We document the agent architecture, tool interfaces, and execution semantics in Section 2, with end-to-end pseudocode in Appendix D. Section 3 specifies the benchmarks, metrics, retrieval corpus, retriever, and evaluation protocol. Appendix E.1 reports the model versions, serving configuration, retry limits, training schedule, GRPO and LoRA hyperparameters, and hardware, while the dataset subsection and Table 5 provide split sizes and domain information. Appendix F reproduces the prompts for all agents and repair procedures verbatim. We also provide retrieval-budget-matched controls in Appendix E.4, the complete LLM-as-a-Judge protocol in Appendix E.5, and qualitative execution traces in Appendix G. Together, these materials specify the components, training settings, prompts, and evaluation procedures needed to reproduce the reported experiments.

REFERENCES

Boyu Chen, Zirui Guo, Zidan Yang, Yuluo Chen, Junze Chen, Zhenghao Liu, Chuan Shi, and Cheng Yang. Pathrag: Pruning graph-based retrieval augmented generation with relational paths. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026.

Mingyang Chen, Linzhuang Sun, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z Pan, Wen Zhang, Huajun Chen, et al. Learning to reason with search for llms via reinforcement learning. *arXiv preprint arXiv:2503.19470*, 2025.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, et al. Binding language models in symbolic languages. *arXiv preprint arXiv:2210.02875*, 2022.

Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In *International conference on machine learning*, 2023a.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, Haofen Wang, et al. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2023b.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Bernal J Gutiérrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. *Advances in neural information processing systems*, 2024.

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/.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, et al. Da-code: Agent data science code generation benchmark for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 13487–13521, 2024.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

Zhengbao Jiang, Frank F Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. Active retrieval augmented generation. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, 2023.

Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan Arik, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning. *arXiv preprint arXiv:2503.09516*, 2025.

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)*, 2020.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp. *arXiv preprint arXiv:2212.14024*, 2022.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. Dspy: Compiling declarative language model calls into self-improving pipelines. 2024.

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 Dai, Jakob Uszkoreit, Quoc Le, and Slav Petrov. Natural questions: A benchmark for question answering research. *Transactions of the Association for Computational Linguistics*, 2019.

Woosuk Kwon. *vLLM: An Efficient Inference Engine for Large Language Models*. PhD thesis, UC Berkeley, 2025.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33, 2020.

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 *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023.

Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP*, 2023a.

Liangming Pan, Xiaobao Wu, Xinyuan Lu, Luu Anh Tuan, William Yang Wang, Min-Yen Kan, and Preslav Nakov. Fact-checking complex claims with program-guided reasoning. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: Long papers)*, 2023b.

Jash Rajesh Parekh, Pengcheng Jiang, and Jiawei Han. Structure-augmented reasoning generation. *arXiv preprint arXiv:2506.08364*, 2025.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP*, 2023.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. Enhancing retrieval-augmented large language models with iterative retrieval-generation synergy. In *Findings of the Association for Computational Linguistics: EMNLP*, 2023.

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

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, 2025.

Jimeng Shi, Sizhe Zhou, Bowen Jin, Wei Hu, Runchu Tian, Shaowen Wang, Giri Narasimhan, and Jiawei Han. Hypercube-based retrieval-augmented generation for scientific question-answering. *arXiv preprint arXiv:2505.19288*, 2025.

Jimeng Shi, Wei Hu, Runchu Tian, Bowen Jin, Wonbin Kweon, SeongKu Kang, Yunfan Kang, Dingqi Ye, Sizhe Zhou, Shaowen Wang, et al. Multicube-rag for multi-hop question answering. *arXiv preprint arXiv:2602.15898*, 2026.

Huatong Song, Jinhao Jiang, Yingqian Min, Jie Chen, Zhipeng Chen, Wayne Xin Zhao, Lei Fang, and Ji-Rong Wen. R1-searcher: Incentivizing the search capability in llms via reinforcement learning. *arXiv preprint arXiv:2503.05592*, 2025.

Hao Sun, Zile Qiao, Jiayan Guo, Xuanbo Fan, Yingyan Hou, Yong Jiang, Pengjun Xie, Yan Zhang, Fei Huang, and Jingren Zhou. Zerosearch: Incentivize the search capability of llms without searching. *arXiv preprint arXiv:2505.04588*, 2025.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo, Yang Wang, Yaming Liang, Xiangyang Ling, Jie Zhou, Shaoliang Cai, and Jin Luo. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=nnVO1PvbTv.

Jiashuo Sun, Pengcheng Jiang, Saizhuo Wang, Jiajun Fan, Heng Wang, Siru Ouyang, Ming Zhong, Yizhu Jiao, Chengsong Huang, Xueqiang Xu, Pengrui Han, Peiran Li, Jiaxin Huang, Ge Liu, Heng Ji, and Jiawei Han. Rethinking the reranker: Boundary-aware evidence selection for robust retrieval-augmented generation, 2026a. URL https://arxiv.org/abs/2602.03689.

Jiashuo Sun, Shixuan Liu, Zhaochen Su, Xianrui Zhong, Pengcheng Jiang, Bowen Jin, Peiran Li, Weijia Shi, and Jiawei Han. GRACE: Generative representation learning via contrastive policy optimization. In *The Fourteenth International Conference on Learning Representations*, 2026b. URL https://openreview.net/forum?id=hs9lwjH1bJ.

Jiashuo Sun, Yixuan Xie, Jimeng Shi, Shaowen Wang, and Jiawei Han. Tasr-rag: Taxonomy-guided structured reasoning for retrieval-augmented generation. *arXiv preprint arXiv:2603.09341*, 2026c.

Jiashuo Sun, Xianrui Zhong, Sizhe Zhou, and Jiawei Han. DynamicRAG: Leveraging outputs of large language model as feedback for dynamic reranking in retrieval-augmented generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2026d. URL https://openreview.net/forum?id=NuCtKoflsV.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Musique: Multihop questions via single-hop question composition. *Transactions of the Association for Computational Linguistics*, 2022.

Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35, 2022.

Junlin Wu, Xianrui Zhong, Jiashuo Sun, Bolian Li, Bowen Jin, Jiawei Han, and Qingkai Zeng. Structure-r1: Dynamically leveraging structural knowledge in llm reasoning through reinforcement learning, 2025. URL https://arxiv.org/abs/2510.15191.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Xuhui Zheng, Kang An, Ziliang Wang, Yuhang Wang, and Yichao Wu. Stepsearch: Igniting llms search ability via step-wise proximal policy optimization. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025.

A LIMITATIONS

While PyRAG demonstrates consistent gains across multi-hop benchmarks, our analysis (3.4) and case studies (Appendix G) reveal several limitations that we believe are informative for future work.

Retrieval recall is the upstream bottleneck. Our failure analysis (Figure 6a) shows that missing retrieval evidence, where the retriever fails to surface gold evidence, accounts for roughly half of all incorrect predictions, making upstream retrieval recall the single largest source of failures. Although our adaptive retrieval mechanism mitigates this for sub-steps where the answer agent explicitly signals insufficient evidence, it cannot recover cases where retrieval silently returns plausible-looking but incorrect documents. Improving retrieval recall, for instance through query reformulation, learned retrievers, or hybrid sparse–dense retrieval, would yield the largest single accuracy gain and is largely orthogonal to PyRAG's contributions at the planning and execution layers.

Answer agents struggle to utilize retrieved context. Although our RL fine-tuning of the Answer Agent yields measurable improvements over the training-free variant, a substantial fraction of remaining failures still trace back to this stage, both as intermediate error propagation in Figure 6a and as the dominant "Unknown Error" mode in Figure 6b, where the program executes successfully but the answer agent cannot ground a response in the retrieved passages. The executable interface successfully isolates failures to a specific stage, but the underlying difficulty, faithfully grounding answers in retrieved evidence and composing them across hops, is not eliminated by current RL objectives. Improving how language models exploit retrieved context, for instance through evidence-grounded training signals, calibrated uncertainty expression, or aggregation-aware objectives, is a key direction for future work. More broadly, explicit variables guarantee data flow between program steps, not the semantic correctness of the values or operations they encode. The appendix failures show that a syntactically valid program can bypass an already bound variable by hard-coding an incorrect entity (Failure F1), pass an "unknown" sentinel into downstream retrieval as ordinary content (Failure F2), or reject a correct intermediate result during final aggregation (Failure F3). Thus, making reasoning executable and inspectable does not by itself make it correct or verified. Our current compiler-grounded self-repair primarily responds to syntax and runtime exceptions and therefore cannot detect many programs that execute legally but implement the wrong reasoning. Closing this gap will require semantic checks over variable use, evidence consistency, and final-answer agreement with intermediate state.

Brittleness of sentinel-based adaptive retrieval. The execution-driven adaptive retrieval mechanism is triggered by string-level matching against sentinel responses such as "unknown" or "cannot answer." Failure F2 illustrates a concrete weakness: when a sentinel value is interpolated into a downstream query as if it were content, retrieval errors silently propagate. A more robust design would replace string sentinels with structured return types or calibrated confidence signals.

Under-decomposition by the planner. Among program errors, the dominant mode is the silent single-retrieve case, where the planner emits syntactically valid code that issues only one `retrieve()` call for a question that requires multiple hops. This bypasses the reasoning chain entirely and is invisible to compiler-grounded self-repair, since no exception is raised. Plan complexity estimation, or an auxiliary objective penalizing under-decomposition, would be a natural complementary signal.

B BROADER IMPACTS

PyRAG aims to improve the factual grounding and interpretability of multi-hop question answering by replacing opaque natural-language reasoning trajectories with executable programs whose intermediate states are inspectable. We see two main positive impacts: (1) the inspectable execution trace lowers the barrier to auditing model behavior in knowledge-intensive applications, where silent reasoning errors or fabricated intermediate facts are otherwise difficult to localize; and (2) the program-execution interface decouples retrieval, computation, and aggregation, allowing deterministic operations (e.g., date arithmetic, boolean conjunction) to be handled outside the language model and reducing a known source of hallucination. At the same time, PyRAG inherits the risks of any

retrieval-augmented system. Because final answers are grounded in retrieved passages, biases, factual errors, or under-representation of certain groups, languages, or domains in the underlying corpus can propagate into outputs while appearing well-supported by an inspectable trace, potentially lending unwarranted credibility to incorrect conclusions. The executable reasoning interface could also, in principle, be repurposed to automate the generation of seemingly evidence-backed but misleading content at scale. Finally, executing model-generated code introduces a standard but non-trivial security surface: deployments must sandbox the Python interpreter and restrict tool APIs to prevent untrusted programs from performing unintended actions. We restrict the runtime in our experiments to the two tool primitives retrieve and answer over a fixed Wikipedia corpus, and we encourage similar isolation in any downstream deployment.

C Extended Related Work

Multi-Hop Retrieval-Augmented Generation. Multi-hop QA requires chaining evidence across multiple passages, which vanilla RAG Lewis et al. (2020) cannot handle in a single retrieval step. Iterative prompting-based methods interleave retrieval with chain-of-thought (CoT) reasoning Wei et al. (2022) and reasoning-action loops Yao et al. (2022) or decomposed sub-questions Press et al. (2023); Trivedi et al. (2023); Jiang et al. (2023); Shao et al. (2023); Khattab et al. (2022); Shi et al. (2025; 2026). A parallel line of graph-based approaches constructs reasoning structures over retrieved content Edge et al. (2024); Gutiérrez et al. (2024); Chen et al. (2026); Parekh et al. (2025); Sun et al. (2024; 2026d;b); Wu et al. (2025); Sun et al. (2026c;a). More recent work involves training the search policy with reinforcement learning, optimizing the multi-turn retrieval process end-to-end. Search-R1 Jin et al. (2025) extends DeepSeek-R1 Guo et al. (2025) style training to retrieval with outcome-based rewards and retrieved token masking to stabilize multi-turn updates, and R1-Searcher Song et al. (2025) similarly incentivizes search invocation through outcome-based RL. StepSearch Zheng et al. (2025) densifies the RL signal via hop-wise rewards and redundancy penalties. Training-free agentic variants, such as Search-o1 Li et al. (2025), embed retrieval inside o1-style long CoT, distilling retrieved documents before reinjecting them into the reasoning chain. Across these methods, the retrieval-reasoning interaction remains an implicit trajectory shaped by prompts or rewards. **PyRAG** instead represents the full pipeline as an executable program, making intermediate state and dependencies explicit; deterministic runtime feedback is an additional capability of this representation.

C.1 Program-Guided Reasoning

Executable code has proven to be effective for reasoning tasks with well-defined symbolic structure. PAL Gao et al. (2023a) and Program-of-Thoughts Chen et al. (2022) offload numerical reasoning to a Python interpreter, separating planning from deterministic execution. Binder Cheng et al. (2022) extends this to table QA via unified natural-language and SQL commands, Faithful-CoT Lyu et al. (2023) translates questions into symbolic programs that are then executed by an external solver, and Logic-LM Pan et al. (2023a) couples an LLM front-end with symbolic solvers for logical reasoning. ProgramFC Pan et al. (2023b) compiles natural-language claims into Python-style verification, which are then executed by fact-checking modules. These approaches assume that the evidence required for reasoning is available a priori, grounded in self-contained inputs such as tables or closed evidence corpora. A complementary line of work, exemplified by DSPy Khattab et al. (2024), treats LM pipelines as compilable programs and automatically optimizes their prompts and demonstrations via bootstrapped traces; its HotPotQA case study uses a hand-designed two-hop module with fixed structure. DSPy operates at the level of *pipeline construction and prompt optimization*, whereas **PyRAG** prescribes a specific *reasoning representation*—a dynamically generated executable program per query—with execution-grounded self-repair and adaptive retrieval as runtime mechanisms; the two are in principle composable. We do not include DSPy as a direct empirical baseline because the two systems target different layers of the stack and rely on incompatible optimization regimes. DSPy's published HotpotQA pipeline is a hand-designed two-hop module with fixed structure, which cannot adapt to questions of varying hop counts (e.g., MuSiQue's mixture of 2–4 hop queries) without manual redesign. More fundamentally, DSPy's strength comes from teleprompter-based prompt and demonstration bootstrapping over a labeled training set—disabling this optimization reduces DSPy to a standard ReAct prompt, while enabling it makes the comparison incommensurable with our training-free setting and operates at a different granularity (prompt-level optimization) than our

RL-trained variant (policy-level optimization). We instead view PyRAG and DSPy as composable: PyRAG prescribes the per-query reasoning representation, while DSPy could in principle optimize the prompts of PyRAG's individual agents. We leave this integration to future work. **PyRAG** targets a fundamentally different setting, open-domain multi-hop QA, where intermediate answers are unknown at synthesis time and must be dynamically retrieved during execution, with later retrieval queries depending on the results of earlier ones.

D ALGORITHM

Algorithm 1 **PyRAG**

Require: Question q; retriever $\mathcal{R}$; tool APIs `retrieve`$(\cdot, k)$, `answer`$(\cdot, docs = \texttt{None})$; agents $\mathcal{A}_{\text{dec}}, \mathcal{A}_{\text{plan}}, \mathcal{A}_{\text{ans}}$; default top-$k$ k_0, boosted top-k k_1 ($k_1 > k_0$); max repair rounds T; sentinel set $\mathcal{S} = \{\text{"unknown"}, \text{"cannot answer"}, \dots\}$
Ensure: Final answer $\hat{a}$; execution trace τ

// Stage 1: Decomposition
$\mathbf{s} = [s_1, \dots, s_n] \leftarrow \mathcal{A}_{\text{dec}}(q)$ ▷ atomic sub-queries
// Stage 2: Program Synthesis
$\pi \leftarrow \mathcal{A}_{\text{plan}}(q, \mathbf{s})$ ▷ tool calls plus native Python operations
// Stage 3: Execution with grounded refinement
$t \leftarrow 0$, $\tau \leftarrow \emptyset$, env $\leftarrow$ `PythonInterpreter()`
while $t \le T$ **do**
  $\hat{a}, \tau, \text{err} \leftarrow$ EXECUTE$(\pi, \text{env}, \mathcal{R}, \mathcal{A}_{\text{ans}}, k_0, k_1, \mathcal{S})$
  if err $=$ `None` **then**
    break ▷ program executed successfully
  else
    $\pi \leftarrow \mathcal{A}_{\text{plan}}(q, \mathbf{s}, \pi, \text{err})$ ▷ (A) compiler-grounded self-repair
    $t \leftarrow t + 1$
  end if
end while
return $\hat{a}, \tau$

Procedure EXECUTE(π, env, $\mathcal{R}$, $\mathcal{A}_{\text{ans}}$, k_0, k_1, $\mathcal{S}$):
  try:
    for each statement $\ell \in \pi$ **do**
      if ℓ is $v \leftarrow$ `retrieve`$(query, k)$ **then**
        env$[v] \leftarrow \mathcal{R}(query, k)$
      elseif ℓ is $v \leftarrow$ `answer`$(query, docs)$ **then**
        $a \leftarrow \mathcal{A}_{\text{ans}}(query, docs)$
        if $a \in \mathcal{S}$ **then** ▷ (B) execution-driven adaptive retrieval
          $docs' \leftarrow \mathcal{R}(query, k_1)$ ▷ boost top-k for under-evidenced step
          $a \leftarrow \mathcal{A}_{\text{ans}}(query, docs')$
        env$[v] \leftarrow a$
      elseif ℓ is $v \leftarrow$ `answer`$(query)$ **then**
        env$[v] \leftarrow \mathcal{A}_{\text{ans}}(query, \emptyset)$ ▷ optional semantic synthesis
      else ▷ native Python ops (regex, arithmetic, control flow)
        evaluate ℓ in env
      append $(\ell,$ SNAPSHOT(env)$)$ to τ
    return env[final_answer], τ, `None`
  except Exception e: **return** `None`, τ, e

E ADDITIONAL EXPERIMENT

E.1 IMPLEMENT DETAILS

PyRAG is implemented as a three-agent pipeline: a **Decompose Agent** that breaks the input question into atomic sub-queries (JSON list, with up to 3 self-correction retries); a **Plan Agent** that translates

the sub-queries into executable Python code using two primitive functions, `retrieve()` and `answer()`; and an **Answer Agent** that processes each `answer()` call by conditioning on retrieved passages enclosed in structured `<answer>` tags. All inter-agent communication is mediated through a shared `execution_log` that records every retrieval and QA step.

Runtime errors in LLM-generated code trigger a self-repair loop in which the Plan Agent is re-prompted with the failed code and the Python traceback, up to `MAX_FIX_ROUNDS`=3 attempts. Syntax errors detected during code generation are corrected inline within the same generation call, also up to 3 retries.

All language models are served with vLLM (Kwon, 2025). In the default training-free 7B configuration, the Decompose, Plan, and Answer Agents are three logical roles implemented by the same Qwen2.5-7B-Instruct checkpoint (Yang et al., 2024) (tensor parallel size 2). They use distinct role prompts and are invoked separately, but share model weights. For the 72B experiments, all three roles instead share Qwen2.5-72B-Instruct with tensor parallel size 4. The code-specialization ablation in Figure 5 is the only exception: it replaces the shared 7B checkpoint with Qwen2.5-Coder-7B-Instruct for all three roles.

We fine-tune all three roles with GRPO (Shao et al., 2024) using the VERL framework (Sheng et al., 2025). The roles share one frozen base backbone but maintain separate role-specific LoRA adapters (Hu et al., 2022) (rank 64, $\alpha = 32$). Training follows a curriculum-style schedule: we update the Answer adapter first, the Plan adapter second, and the Decompose adapter last; adapters for the other two roles are frozen at each stage. This preserves a single shared foundation model while allowing role specialization. The order is deliberate: the Answer Agent is trained first since it is the terminal step of every reasoning chain and its quality bounds the end-to-end reward; the Plan Agent is trained next on top of a well-calibrated answerer; finally, the Decompose Agent is trained against frozen Plan and Answer adapters. The reward is a weighted combination of EM and F1, $r = 0.7 \cdot \mathrm{F1} + 0.3 \cdot \mathrm{EM}$, computed by executing the full pipeline against gold answers. The Answer adapter uses learning rate $1\mathrm{e}{-6}$, cosine schedule, rollout $n = 8$, batch size 32, and 1 epoch. The Plan and Decompose adapters use batch size 64, rollout $n = 4$, learning rate $3\mathrm{e}{-6}$, KL penalty $\lambda = 0.001$ (low-variance KL), and 2 epochs each. All RL experiments are conducted on a single node of $8\times$A100 80 GB GPUs.

E.2 Datasets

Following the data setup of Search-R1 (Jin et al., 2025), we train on a mixture of Natural Questions (NQ) (Kwiatkowski et al., 2019) and HotpotQA (Yang et al., 2018), yielding 87,925 training examples in total (79,168 single-hop NQ and 8,757 multi-hop HotpotQA). This mixture exposes the model to both single-hop factoid retrieval and compositional multi-hop reasoning during RL fine-tuning, while keeping the training distribution comparable to prior RL-trained RAG baselines for fair comparison.

For evaluation, we use five datasets grouped along two axes: domain (in- vs. out-of-domain relative to training) and hop count (single- vs. multi-hop). The in-domain evaluation set is the held-out split of HotpotQA (7,405). For out-of-domain evaluation, we include one single-hop benchmark, PopQA (Mallen et al., 2023) (14,267), and three multi-hop benchmarks, 2WikiMultiHopQA (Ho et al., 2020) (12,576), MuSiQue (Trivedi et al., 2022) (2,417), and Bamboogle (Press et al., 2023) (125). The multi-hop out-of-domain sets in particular stress-test whether the executable reasoning prior learned by PyRAG transfers across question distributions, hop counts (MuSiQue contains 2–4 hop questions), and compositional patterns unseen during training. Detailed statistics are reported in Table 5. We use Exact Match (EM) as the primary metric throughout, consistent with prior work.

E.3 Failure Analysis

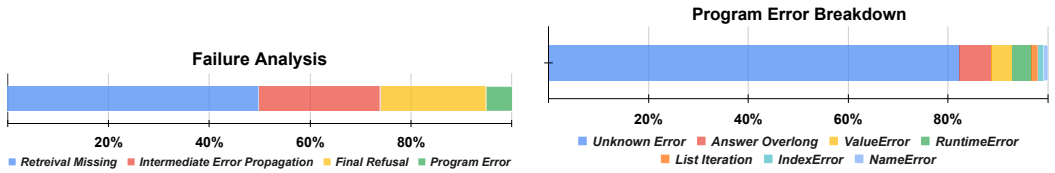
To identify the main error sources in **PyRAG**, we manually categorize 100 randomly sampled incorrect predictions on HotpotQA. As shown in Figure 6a, missing retrieval evidence accounts for roughly half of all failures, making upstream retrieval recall the dominant bottleneck. The next largest category is intermediate error propagation, in which an uncertain sub-answer corrupts downstream steps (Failure F2), followed by final refusals, in which the answer agent declines despite successful program execution. Program errors account for only $\sim$5%, indicating that the planning agent generally produces well-formed executable code. This syntactic reliability should not be

Table 5: Dataset statistics for training and evaluation.

Split	Dataset	#Examples	Task Type	Domain
Train	NQ	79,168	Single-hop	In-domain
	HotpotQA	8,757	Multi-hop	In-domain
	Total	*87,925*	–	–
Eval	HotpotQA	7,405	Multi-hop	In-domain
	PopQA	14,267	Single-hop	Out-of-domain
	2WikiMultiHopQA	12,576	Multi-hop	Out-of-domain
	MuSiQue	2,417	Multi-hop	Out-of-domain
	Bamboogle	125	Multi-hop	Out-of-domain

conflated with semantic correctness. As Failures F1–F3 illustrate, a valid program may bypass a bound variable by hard-coding the wrong entity, propagate an "unknown" value downstream, or produce a final answer that contradicts correct intermediate bindings.

We further characterize program errors using a separate random sample of 100 such cases (Figure 6b). The dominant category is "Unknown Error," in which the program executes without raising an exception, but the answer agent returns a sentinel response (e.g., "unknown") because it cannot compose an answer from the retrieved evidence. This behavior reflects a context-utilization failure rather than a Python-level fault. Genuine Python-level failures (`ValueError`, `RuntimeError`, list-iteration faults, `IndexError`, and `NameError`) together account for less than 20% of program-error cases and are typically traceable to mismatched assumptions about retrieved string formats (e.g., Failure F5).



(a) Overall failure categories.

(b) Breakdown of program-related errors.

Figure 6: Failure analysis on HotpotQA. **Left:** overall failure categories from 100 randomly sampled incorrect predictions; missing retrieval evidence is the dominant error source. **Right:** breakdown from a separate random sample of 100 program-error cases; "Unknown Error" denotes successful execution followed by an insufficient or uncomposed answer rather than a Python exception.

E.4 Retrieval-Budget-Matched Analysis

Adaptive retrieval increases the retrieval budget only for sub-steps with insufficient evidence, raising the possibility that its gains come simply from exposing the model to more documents. To test this alternative explanation, we compare full **PyRAG** against fixed-budget controls on Bamboogle. The controls include both Vanilla RAG and **PyRAG** without adaptive retrieval at k=5 and k=10. Full **PyRAG** starts from k=5 and increases the budget to k=10 only when an `answer()` call returns an insufficient-information response such as "unknown."

Table 6: Retrieval-budget-matched results on Bamboogle with Qwen2.5-7B-Instruct. Full PyRAG outperforms the fixed-k=10 control while retrieving fewer documents and using fewer prompt tokens.

Method	Executable Runtime	Adaptive Retrieval	Retrieval Budget	Avg. Retrieved Docs	Avg. Prompt Tokens	EM
Vanilla RAG	✗	✗	Fixed (k=5)	5.0	1,417	16.0
Vanilla RAG	✗	✗	Fixed (k=10)	10.0	2,615	19.2
PyRAG (w/o Adaptive Retrieval)	✓	✗	Fixed (k=5)	5.0	3,280	33.6
PyRAG (w/o Adaptive Retrieval)	✓	✗	Fixed (k=10)	10.0	4,478	39.2
PyRAG (Full)	✓	✓	Adaptive (5→10 upon "unknown")	6.2	3,527	**41.6**

Increasing a fixed retrieval budget helps but does not reproduce the benefit of adaptive retrieval. For Vanilla RAG, doubling the budget from k=5 to k=10 increases EM by only 3.2 points (16.0 to 19.2), despite doubling the number of retrieved documents. For PyRAG without adaptive retrieval, the same increase raises EM from 33.6 to 39.2, but also increases average prompt length from 3,280 to 4,478 tokens.

Crucially, full PyRAG achieves 41.6 EM, outperforming the fixed-k=10 variant by 2.4 points while retrieving only 6.2 rather than 10.0 documents on average and using 21.2% fewer prompt tokens. Relative to the fixed-k=5 variant, it improves EM by 8.0 points while increasing prompt tokens by only 7.5%. Thus, **the gains cannot be explained by increased retrieval budget alone**. They arise from allocating additional evidence selectively to the sub-steps that exhibit an execution-grounded signal of insufficient information, rather than uniformly expanding the context for every query.

E.5 LLM-as-a-Judge as a Complementary Metric

Exact Match is deliberately strict and can undercount correct short free-form answers that differ from the reference only in wording, aliases, or formatting. We therefore conduct an additional LLM-as-a-Judge evaluation for experiments using Qwen2.5-7B-Instruct, with Qwen2.5-72B-Instruct serving as the judge. Given the question, ground-truth answer, and model response, the judge outputs only "Yes" or "No" to indicate whether the response is semantically correct. We report the percentage of "Yes" decisions as judge accuracy. The complete judging prompt is provided in Appendix F.4.

Table 7: Exact Match and LLM-as-a-Judge accuracy (%) with Qwen2.5-7B-Instruct. Each entry reports **EM / Judge**. The judge is Qwen2.5-72B-Instruct.

Method	HotpotQA	2WikiMQA	MuSiQue	Bamboogle
Vanilla RAG	28.9 / 45.0	18.9 / 23.5	4.7 / 10.0	16.0 / 26.4
RAG-SFT	32.4 / 49.5	22.6 / 27.5	6.8 / 13.5	27.2 / 38.4
RAG-RL	35.2 / 53.5	34.7 / 39.5	9.6 / 17.0	29.6 / 40.8
PyRAG-RL	**40.5 / 60.0**	**49.4 / 56.0**	**20.7 / 31.0**	**46.4 / 58.4**

As shown in Table 7, judge accuracy is consistently higher than EM across all methods and datasets because it recognizes semantically equivalent answers that do not exactly match the reference string. The method ranking is nevertheless unchanged: among the methods evaluated with the same judge and protocol, PyRAG-RL remains best on all four datasets. Its judge accuracy reaches 60.0 on HotpotQA, 56.0 on 2WikiMQA, 31.0 on MuSiQue, and 58.4 on Bamboogle.

We report judge results for Vanilla RAG, RAG-SFT, RAG-RL, and PyRAG-RL because we have their per-instance responses under the same evaluation setting. External methods such as ZEROSEARCH, Search-R1, StepSearch, and ReSearch report aggregate EM scores in their original papers but do not report corresponding LLM-as-a-Judge results. Although their published EM values can be included in our comparison tables, their per-instance responses are unavailable; we therefore cannot independently evaluate them using the same judge and protocol, and do not impute or estimate missing judge scores.

F PROMPTS

This appendix lists the full prompts used by the three PyRAG agents (Decompose, Plan, Answer) together with the two repair templates triggered by compiler-grounded self-repair. Each prompt is reproduced verbatim from our implementation; placeholders in braces (e.g. `{original_query}`) are filled in at runtime. We use three colour codes throughout: **blue** for system prompts that fix an agent's role and output schema, **green** for user-side templates that supply per-question context, and **orange** for repair templates triggered by execution failures.

F.1 DECOMPOSE AGENT

The Decompose Agent maps the original multi-hop question q to a list of atomic, single-hop sub-queries $\mathbf{s} = [s_1, \ldots, s_n]$. The system prompt fixes a strict JSON-list output schema so the result can be parsed deterministically and consumed by the Plan Agent without further post-processing; the user prompt supplies the original question together with a one-shot example that anchors the expected granularity (one search-engine-answerable claim per item). Parsing failures trigger up to three retries before falling back to using the original question as a single-element list.

Decompose Agent — SYSTEM PROMPT

```
You are a question decomposition agent for multi-hop QA.
Break a complex question into a minimal list of atomic single-hop
sub-queries. Each sub-query should be independently answerable by a
search engine. Return a JSON list of strings ONLY. No explanation,
no markdown, no extra text.
```

Decompose Agent — USER TEMPLATE

```
Original question:
{query}

Return a JSON list of atomic sub-queries. Example:
["Who directed Inception?",
 "Who directed Jurassic Park?",
 "When was Christopher Nolan born?"]
```

Figure 7: Prompts used by the Decompose Agent. The system prompt enforces a parseable JSON-list contract; the user template supplies the question and a one-shot example fixing sub-query granularity.

F.2 PLAN AGENT

The Plan Agent is the core of PyRAG: given the original question q and the decomposed sub-queries $\mathbf{s}$, it synthesises an executable Python program over the two externally provided tool primitives `retrieve(query, topk)` and `answer(query, docs)`, together with native Python operations for deterministic composition. The system prompt (Figure 8) codifies the executable interface as a contract: it specifies the exact function signatures; the data-flow discipline that intermediate results must be bound to identifiers and reused rather than re-derived; the safe use of parsing, arithmetic, comparison, and control flow; and the two-part synthesis format used when the final step requires semantic aggregation "Given: <facts>. Answer the question: <original question verbatim>" that prevents an intermediate answer from leaking into the question template. The user prompt instantiates this contract for the specific query and supplies a worked one-shot example demonstrating variable threading and deterministic date comparison across hops.

Plan Agent SYSTEM PROMPT

```
You are a planning agent that writes Python code to answer questions
via RAG.

You have exactly two external tool functions available. You may also
use native Python expressions, control flow, built-ins, and safe
standard-library modules for deterministic computation.

  retrieve(query: str, topk: int = 5) -> List[str]
      Searches a retrieval system and returns a list of relevant
      document strings. Use topk=5 as the default initial budget. The
      runtime may increase it for an under-evidenced step.

  answer(query: str, docs: List[str]) -> str
      Calls an LLM to answer a question given a list of documents.
      Returns ONLY the short text inside the model's <answer>...
      </answer> block (the runtime extracts it). Use that return
      value directly in f-strings -- it is a short phrase, not the
      full model reply.

  answer(query: str) -> str  (no docs -- aggregation / synthesis mode)
      Same as above but with NO documents. The model answers using
      ONLY the information already present in the query string. Use
      this only when the final result requires semantic synthesis
      rather than a deterministic Python operation.

Rules:
- Use retrieve() + answer() step by step to answer each sub-query.
- If you call retrieve() for a sub-question, pass that return value
  into the matching answer() for that sub-question.
- Build on intermediate answers by interpolating them into later
  queries using f-strings.
- Use native Python for operations with well-defined semantics, such
  as string normalization, list construction, boolean conjunction,
  comparison, date/number extraction, and arithmetic. Do not delegate
  such operations to answer() when Python can compute them directly.
- You MAY parse answer() return values with operations such as
  .strip(), .lower(), .split(), indexing, int(), or regex when the
  requested answer has a clear atomic format. Validate the expected
  format before using a parsed value; do not silently guess when a
  required value is missing.
- Imports are limited to safe standard-library modules needed for
  deterministic computation (for example re, math, or datetime).
  Never access the filesystem, network, environment, or subprocesses.
- Variable names MUST be valid Python identifiers: only letters,
  digits, and underscores, NO spaces.
- Every variable you reference MUST be assigned BEFORE it is used.
- If you use a for/if/while block that might conditionally set
  final_answer, initialise final_answer = "" BEFORE the block.
- Before the program terminates, assign the result to final_answer.
  final_answer may be produced directly by a deterministic Python
  expression or by answer(<synthesis question>). Prefer direct Python
  computation for arithmetic, comparison, filtering, and boolean
  logic; do not add an unnecessary LLM call after the result is known.
- If final_answer is produced by answer() in aggregation mode, call it
  with NO docs and use this TWO-PART format:
    "Given: <fact1>, <fact2>, ...
     Answer the question: <ORIGINAL QUESTION>"
  where the first part lists intermediate results as background facts
  and the second part REPEATS the ORIGINAL question VERBATIM after
  "Answer the question:". Do NOT rephrase the original question or
  embed intermediate answers INTO the question itself.
  BAD:  f"Which American director, {name}, who is {nationality},
         hosted ...?"  (answer leaks into the question)
  GOOD: f"Given: {name} hosted the event, {name} is {nationality}.
         Answer the question: Which American director hosted ...?"
- Valid Python 3 only: use ASCII double or single quotes for strings;
  never put backticks around variable names or expressions.
- Return ONLY the Python code, no explanation and no markdown.
```

Figure 8: Plan Agent system prompt. Codifies the executable interface as a contract — function signatures, data-flow discipline, deterministic Python composition, and the optional synthesis format.

Plan Agent — USER TEMPLATE

```
Original question:
{original_query}

Sub-queries to resolve:
{sub_queries}

Reference example (do NOT copy, write code for the actual question
above):
{CODE_EXAMPLE}

Now write the Python code for the original question.
Assign the final result to final_answer. Compute it directly in Python
when the required operation is deterministic; otherwise use:
final_answer = answer(f"Given: <facts>.
                        Answer the question: {original_query}")
```

Plan Agent — ONE-SHOT EXAMPLE (CODE_EXAMPLE)

```
# Example for: "Who was born earlier, the director of
# Inception or Jurassic Park?"
import re

docs1 = retrieve("Who directed Inception?", topk=5)
director1 = answer("Who directed Inception?", docs1)

docs2 = retrieve("Who directed Jurassic Park?", topk=5)
director2 = answer("Who directed Jurassic Park?", docs2)

docs3 = retrieve(f"When was {director1} born?", topk=5)
birth1 = answer(f"When was {director1} born?", docs3)

docs4 = retrieve(f"When was {director2} born?", topk=5)
birth2 = answer(f"When was {director2} born?", docs4)

def extract_year(text):
    match = re.search(r"\b\d{4}\b", text)
    if match is None:
        raise ValueError(f"No four-digit year in: {text}")
    return int(match.group())

year1 = extract_year(birth1)
year2 = extract_year(birth2)
final_answer = director1 if year1 < year2 else director2
```

Figure 9: Plan Agent user-side context. Top: the per-question template filled with the original query, the decomposed sub-queries, and the one-shot `CODE_EXAMPLE`. Bottom: the example itself, demonstrating variable threading, validated parsing, and deterministic date comparison.

F.2.1 SELF-REPAIR TEMPLATES

PyRAG's compiler-grounded self-repair operates at two granularities. Syntax-level feedback (Figure 11) is triggered when the model's output fails to compile under `compile(..., 'exec')`; the failed snippet and the parser's error location are returned to the same generation call, up to three retries. Runtime-level feedback (Figure 10) is triggered after a successful compile when the program raises a Python exception during execution; the original question, the failing program, and the traceback are surfaced back to the Plan Agent, which produces a corrected program for the runtime to re-execute. Both templates re-iterate the contract conditions most commonly responsible

for failure (uninitialised `final_answer`, unchecked parsing, and missing `docs` arguments) so that repair is grounded in deterministic compiler signals rather than the model's self-judgement.

Plan Agent — RUNTIME-ERROR REPAIR TEMPLATE

```
The following Python code was generated to answer the question but
raised a runtime error. Fix the code so it runs correctly. Return
ONLY the corrected Python code.

=== Original question ===
{original_query}

=== Failing code ===
{failed_code}

=== Runtime error ===
{error_msg}

=== Fix instructions ===
- Parsing short answer() values is allowed when required for a
  deterministic operation, but validate regex matches, indices, and
  conversions before using them.
- Every variable you reference must be explicitly assigned earlier.
- If retrieve() was called, pass its docs into answer(); do not use
  answer(..., []) for that step.
- Initialise final_answer = "" before any loop/conditional that
  might set it.
- Ensure the successful program assigns its result to final_answer.
  Preserve deterministic Python computation when the answer is an
  arithmetic, comparison, filtering, or boolean result. If semantic
  synthesis is required, use final_answer = answer(<synthesis
    question>)
  with NO docs and the Given/Answer-the-question format.
```

Figure 10: Runtime-level self-repair template. Triggered when an executed program raises a Python exception; the traceback and the original question are surfaced back as deterministic, grounded feedback signals.

Plan Agent — SYNTAX-ERROR REPAIR TEMPLATE

```
---
Your previous output is NOT valid Python (syntax error).
Details: {error_detail}

Fix it. Use only ASCII ' or " for strings; never wrap variable names
in backticks; output code only.

Previous attempt:
```python
{failed_code}
```
Return ONLY corrected Python code.
```

Figure 11: Syntax-level self-repair template. Triggered when the generated code fails to compile; the parser's error location is fed back to the same generation call.

F.3 ANSWER AGENT

The Answer Agent is invoked once per `answer()` call in the executed program and operates in two distinct modes that share an identical `<redacted_thinking>` / `<answer>` output schema. In evidence mode (Figure 12), the agent receives a sub-query together with retrieved passages, must answer using only those passages, and must cite each used passage inline as "`Doc [i]`". The schema fixes type-matching between the question and the answer span (a who question must return a name, not a date) and reserves the literal token "`unknown`" as a sentinel for under-evidenced steps, which directly drives the execution-driven adaptive retrieval mechanism described in Section 2.6 of the main text. In aggregation mode (Figure 13), used only when the program requires semantic rather than deterministic composition, no documents are supplied; the prompt instead relies on the two-part Given: ... Answer the question: ... template emitted by the Plan Agent, and the agent composes the final answer from the supplied facts. The aggregation prompt explicitly forbids yes/no responses to wh-questions, addressing the failure mode in which the model otherwise treats the synthesis call as fact verification. Retrieved documents are formatted as `[Doc 1]\n<text>\n\n[Doc 2]\n<text>\n\n...` so that inline citation indices are unambiguous.

Answer Agent — EVIDENCE MODE (WITH DOCUMENTS)

```
You are given a question and retrieved documents.
You MUST answer using ONLY information from the retrieved documents.
Even for yes/no questions, decide yes or no by reasoning from facts
in the documents.

Output format (STRICT):
<thinking> ... </thinking>
<answer> ... </answer>

Evidence citation rule:
- Whenever you use evidence from the documents in your reasoning,
  you MUST cite it inline as Doc [i] (matching the document indices
  shown in the retrieved block, e.g. [Doc 1] -> Doc [1]).
- Only cite documents that are actually relevant.
- Keep <thinking> concise (1-3 sentences).

Answer rules:
- The <answer> should be a short phrase, preferably taken directly
  from the documents when possible.
- Match the answer TYPE to the QUESTION:
    WHO / which person / born first / earlier
        -> a person's NAME in <answer>, not only a date;
    WHEN -> date or time;
    yes/no -> exactly yes / no / unknown when the documents do not
              support a definite answer.
- Do NOT output anything outside <thinking> and <answer>.

Example (do NOT copy the content, only follow the style):
<thinking>Doc [1] states that Future Ted serves as the
narrator, and Doc [4] confirms the voice actor.</thinking>
<answer> Ted Mosby </answer>
```

Figure 12: Answer Agent system prompt — evidence mode. Used when at least one retrieved passage is supplied. The schema fixes question-type matching and reserves "`unknown`" as the sentinel that drives adaptive retrieval.

Answer Agent — AGGREGATION MODE (NO DOCUMENTS)

```
There are NO retrieved documents. The question text itself contains
background facts (after 'Given:') and the actual question to answer
(after 'Answer the question:').
You MUST use the provided facts to answer the ACTUAL QUESTION.

CRITICAL: Your job is to ANSWER the question, NOT to confirm whether
the facts are true.
- If the question asks WHO / WHICH person -> reply with a person's
    NAME.
- If the question asks WHEN -> reply with a date or time.
- If the question asks WHERE -> reply with a place.
- ONLY answer yes/no when the question is explicitly a yes/no question
  (e.g. 'Are both ...?', 'Is it true ...?').
- NEVER answer 'yes' or 'no' to a WHO/WHICH/WHEN/WHERE question.

Output format (STRICT):
<thinking> ... </thinking>
<answer> ... </answer>

- In <thinking>, identify the actual question type and
  combine the given facts to produce the answer (1-2 sentences).
- The <answer> must directly answer the question -- a name, date,
  place, etc. -- NOT 'yes' or 'no' unless the question is truly
  yes/no.
- Do NOT output anything outside <thinking> and <answer>.
```

Figure 13: Answer Agent system prompt — aggregation mode. Used when the program chooses a final semantic-synthesis call with no `docs`. Forbids yes/no responses to wh-questions, eliminating the failure mode where the synthesis call collapses into fact verification.

F.4 LLM-AS-A-JUDGE PROMPT

We use the following prompt with Qwen2.5-72B-Instruct to evaluate whether a model response is semantically equivalent to the reference answer. The judge is instructed to accept harmless surface-form variation while rejecting responses that omit essential information or introduce contradictions. Its output is parsed as a binary "Yes" or "No" decision.

LLM-as-a-Judge — SEMANTIC CORRECTNESS PROMPT

```
You are a strict evaluator of short question-answering responses.

Determine whether the candidate response is semantically correct for
the question with respect to the reference answer.

Evaluation rules:
- Output exactly one word: Yes or No.
- Answer Yes when the candidate expresses the same answer as the
  reference, including equivalent aliases, abbreviations, paraphrases,
  or harmless differences in formatting.
- Extra explanatory text is acceptable only when the answer remains
  unambiguous and does not contradict the reference.
- For answers containing multiple required items, all essential items
  must be present.
- Answer No if the candidate names the wrong entity, omits essential
  information, contradicts the reference, or says that the answer is
  unknown or cannot be determined.
- Do not provide reasoning or any text other than Yes or No.
```

LLM-as-a-Judge (continued)

```
Question:
{question}

Reference answer:
{reference_answer}

Candidate response:
{model_response}

Is the candidate response semantically correct?
```

Usage. This prompt is used for the complementary LLM-as-a-Judge evaluation reported in Appendix E.5.

G Case Study

G.1 Successful Executions

Case A Correct

Question: 2014 S/S is the debut album of a South Korean boy group that was formed by who?
Gold: `YG Entertainment` **Predicted:** `YG Entertainment`

```
# Step 1-2: identify the boy group
ctx_grp = retrieve("2014 S/S debut South Korean boy group", topk=5)
group   = answer("Which group is the album by?", ctx_grp)
# group = "WINNER"

# Step 3-4: the next query *uses* the bound variable `group`
ctx_co  = retrieve(f"Which company formed {group}?", topk=5)
company = answer(f"Which company formed {group}?", ctx_co)
# company = "YG Entertainment"

# Step 5: final aggregation is itself an answer(...) call whose query
#         is built from the two intermediate variables
final_answer = answer(
    f"Given: {group} is a South Korean boy group; "
    f"{company} formed {group}. "
    f"Answer the question: 2014 S/S is the debut album "
    f"of a South Korean boy group that was formed by who?"
)
# final_answer -> "YG Entertainment"   [ok]
```

Takeaway. A representative correct example. Variables produced at one step are explicitly consumed by subsequent calls through string interpolation.

Case B Correct

Question: Which other Mexican Formula One race car driver has held the podium besides the Force India driver born in 1990?
Gold: `Pedro Rodríguez` **Predicted:** `Pedro Rodríguez`

```
# Step 1-2: a known Mexican F1 podium driver
ctx_a   = retrieve("Mexican F1 driver who held the podium", topk=5)
```

Case B (continued)

```
known   = answer("Who is the Mexican F1 podium driver?", ctx_a)
# known = "Sergio P\'erez"

# Step 3-4: try to identify the Force India 1990-born driver
ctx_b   = retrieve("Force India driver born in 1990", topk=5)
exclude = answer("Which Force India driver was born in 1990?", ctx_b)
# exclude = "unknown"      [warn] insufficient evidence
```

```
# Step 3-4: executor reissues a broader retrieval (adaptive re-
    retrieval)
ctx_b   = retrieve("Force India driver born in 1990", topk=10)
exclude = answer("Which Force India driver was born in 1990?", ctx_b)
# exclude = "Pedro Rodr\'iguez and Sergio P\'erez"
```

```
# Step 5: the final aggregation is again an answer(...) call,
#         whose query templates in the previously bound variables
final_answer = answer(
    f"Given: {known} is a Mexican F1 podium driver; "
    f"the Mexican F1 podium drivers are {exclude}. "
    f"Answer the question: Which other Mexican Formula One race car
    driver "
    f"has held the podium besides the Force India driver born in 1990?
    "
)
# final_answer -> "Pedro Rodr\'iguez"   [ok]
```

Takeaway. When Step 4 returns the sentinel `"unknown"`, execution-guided refinement triggers a broader re-retrieval. The plan structure is preserved; only the under-evidenced sub-step is repaired.

Case C Correct

Question: Are Freakonomics and In the Realm of the Hackers both American documentaries?
Gold: `no` **Predicted:** `No`

```
films      = ["Freakonomics", "In the Realm of the Hackers"]
predicates = ["is a documentary", "is American"]

# Two predicates per film -> grid of 4 yes/no probes
flags = {f: {} for f in films}
for film in films:
    for pred in predicates:
        ctx               = retrieve(f"Is {film} {pred}?", topk=5)
        flags[film][pred] = answer(f"Is {film} {pred}?", ctx)

# flags = {
#   "Freakonomics":              {"is a documentary": "no",  "is
    American": "yes"},
#   "In the Realm of the Hackers":{"is a documentary": "yes", "is
    American": "no"},
# }

# Boolean structure handled in Python -- not by the answer agent
def qualifies(d):
    return all(v.lower() == "yes" for v in d.values())

final_answer = "yes" if all(qualifies(flags[f]) for f in films) else "
    no"
```

Case C (continued)

```
# final_answer -> "no"   (Freakonomics fails the documentary check;
#                         In the Realm of the Hackers fails the
    American check)   [ok]
```

Takeaway. The plan reduces the question to a grid of yes/no probes whose conjunction is decided by Python's `all`; the answer agent never performs the multi-clause logical operation.

Case D Correct

Question: How old was Virginia Bruce when she starred in Let Freedom Ring?
Gold: `29` **Predicted:** `29`

```
import re

# Two retrievals supply the numeric inputs
ctx_film = retrieve("In what year did Virginia Bruce star in Let
    Freedom Ring?", topk=5)
year_str = answer("In what year did Virginia Bruce star in Let Freedom
     Ring?", ctx_film)
# year_str = "1939"

ctx_born = retrieve("In what year was Virginia Bruce born?", topk=5)
born_str = answer("In what year was Virginia Bruce born?", ctx_born)
# born_str = "1910"

# Cast text -> int and compute the answer in Python; no further LLM
    call needed
film_match = re.search(r"\d{4}", year_str)
born_match = re.search(r"\d{4}", born_str)
if film_match is None or born_match is None:
    raise ValueError("Expected a four-digit year in both answers")
year_film = int(film_match.group())
year_born = int(born_match.group())
final_answer = year_film - year_born
# final_answer -> 29   [ok]
```

Takeaway. The retrieved year strings are converted to integers and subtracted deterministically, moving arithmetic outside the language model.

G.2 Representative Failures

Failure F1 Incorrect

Question: Aside from the Apple Remote, what other device can control the program Apple Remote was originally designed to interact with?
Gold: `keyboard function keys` **Predicted:** `iPod Touch, iPad` ✗

```
# Step 1-2: correctly identify the program
ctx_p   = retrieve("Program the Apple Remote was designed for", topk
    =5)
program = answer("What program is Apple Remote designed for?", ctx_p)
# program = "Front Row"   [ok]
```

Failure F1 (continued)

```
# Step 3-4: planner SHOULD use `program` here, but silently rebinds to
    "iTunes"
ctx_d  = retrieve("What other devices can control iTunes?", topk=5)
   # [warn] entity drift
device = answer("What other devices can control iTunes?", ctx_d)
# device = "iPod Touch, iPad"
```

```
# Step 5: final answer(...) propagates the corrupted variable
final_answer = answer(
    f"Given: Apple Remote was designed for {program}; "
    f"other devices that can control {program} include {device}. "
    f"Answer the question: Aside from the Apple Remote, what other
    device "
    f"can control the program Apple Remote was originally designed to
    interact with?"
)
# final_answer -> "iPod Touch, iPad"   [wrong]
```

Diagnosis. The planner hard-codes `"iTunes"` instead of reusing `program`. The executable trace localizes the entity drift to a single planning step.

Failure F2 HotpotQA id=47 Incorrect

Question: According to the 2001 census, what was the population of the city in which Kirton End is located?

Gold: `35,124` **Predicted:** `12,790` ✗

```
# Step 1-2: locality lookup succeeds
ctx_loc  = retrieve("Where is Kirton End located?", topk=5)
locality = answer("Where is Kirton End located?", ctx_loc)
# locality = "Kirton, Lincolnshire"

# Step 3-4: the next hop (city containing the locality) FAILS
ctx_city = retrieve(f"In what city is {locality}?", topk=10)
city     = answer(f"In what city is {locality}?", ctx_city)
# city = "unknown"   [warn]
```

```
# Step 5-6: sentinel value "unknown" is interpolated as a *content*
   string
ctx_pop = retrieve(f"Population of {city} in the 2001 census", topk=5)
       # [warn]
pop     = answer(f"Population of {city} in the 2001 census?", ctx_pop)
# pop = "12,790"   (population of an unrelated locality)
```

```
# Step 7: final answer(...) silently consumes the corrupted hop
final_answer = answer(
    f"Given: Kirton End is located in {locality}; "
    f"the containing city is {city}; "
    f"the 2001 census population of {city} was {pop}. "
    f"Answer the question: According to the 2001 census, what was the
    population "
    f"of the city in which Kirton End is located?"
)
# final_answer -> "12,790"   [wrong]
```

Failure F2 (continued)

Diagnosis. The sentinel `"unknown"` is treated as content and propagated downstream. Such hops should branch to a guarded fallback rather than continue the data-flow chain.

Failure F3 Incorrect

Question: Which writer was from England, Henry Roth or Robert Erskine Childers?
Gold: `Robert Erskine Childers` **Predicted:** `Neither` ✗

```
# Step 1-2: Henry Roth -- correctly resolved
ctx_h = retrieve("Is Henry Roth from England?", topk=5)
henry = answer("Is Henry Roth from England?", ctx_h)
# henry = "no"    [ok]

# Step 3-4: Robert Erskine Childers -- correctly resolved
ctx_c    = retrieve("Is Robert Erskine Childers from England?", topk
    =5)
childers = answer("Is Robert Erskine Childers from England?", ctx_c)
# childers = "yes"   [ok]
```

```
# Step 5: variables are correct, but the final answer(...) call
#         returns a value that contradicts childers == "yes"
final_answer = answer(
    f"Given: Henry Roth from England = {henry}; "
    f"Robert Erskine Childers from England = {childers}. "
    f"Answer the question: Which writer was from England, Henry Roth
    or "
    f"Robert Erskine Childers?"
)
# final_answer -> "Neither"   [warn] contradicts childers == "yes"
```

Diagnosis. Retrieval and variable binding are correct, but the final semantic aggregation contradicts `childers == "yes"`. The trace isolates the answer agent as the failure point.

Failure F4 Incorrect

Question: Are Northwestern University and Middlebury College both private schools?
Gold: `yes` **Predicted:** `No` ✗

```
schools = ["Northwestern University", "Middlebury College"]
flags   = {}
# Per-school predicate check -- both correctly resolved
for s in schools:
    ctx      = retrieve(f"Is {s} a private school?", topk=5)
    flags[s] = answer(f"Is {s} a private school?", ctx)
# flags = {"Northwestern University": "yes", "Middlebury College": "
    yes"}   [ok]
```

```
# Final aggregation delegates the conjunction to the answer agent.
# The model's thinking flips both variables to "public" and returns "
    No".
final_answer = answer(
    f"Given: Northwestern University is {flags[schools[0]]}, "
    f"Middlebury College is {flags[schools[1]]}. "
    f"Answer the question: Are Northwestern University and Middlebury
    College "
```

Failure F4 (continued)

```
    f"both private schools?"
)
# final_answer -> "No"   [warn] contradicts flags == {...: "yes", ...:
    "yes"}
```

```
final_answer = "yes" if all(v.lower() == "yes" for v in flags.values()
    ) else "no"
```

Diagnosis. Both flags are `"yes"`, but the answer agent flips the conjunction. Replacing semantic aggregation with Python's `all(...)` eliminates the failure.

Failure F5 Incorrect

Question: Which of Aaron Goodwin's clients was born on May 31, 1984?

Gold: `Nate Robinson` **Predicted:** `None` ✗

```
ctx_cli = retrieve("Who are Aaron Goodwin's clients?", topk=5)
clients = answer("List Aaron Goodwin's clients.", ctx_cli)
# clients is a *string*: "LeBron James, Dwight Howard, Chris Webber,
    ..."
```

```
# The plan iterates `clients` directly. In Python, iterating a `str`
# yields one *character* per step -- not one client.
births = {}
for name in clients:                    # [warn] name is "L", then "e
    ", "B", "r", ...
    ctx           = retrieve(f"When is the birthdate of {name}?", topk
    =5)
    births[name]  = answer(f"When is the birthdate of {name}?", ctx)
# Trace explodes: 400+ retrieve+answer calls, each on a single
    character.
# Most return "unknown"; a few accidentally hit "August 6, 1992", "
    October 13, 1992", ...
```

```
# Final aggregation cannot recover; no character matches "May 31,
    1984"
final_answer = answer(
    f"Given: the queried client birthdates are {births}. "
    f"Answer the question: Which of Aaron Goodwin's clients was born
    on May 31, 1984?"
)
# final_answer -> "None"   [wrong]   (correct answer: Nate Robinson)
```

```
clients = [c.strip() for c in clients.split(",")]
```

Diagnosis. `clients` is a comma-joined string, so the loop iterates over characters. Splitting it into a list before the loop fixes the type confusion and prevents the retrieval explosion.