

RLM Algorithm: Implementation vs. Paper Comparison

Comparison of our rlm-loop implementation against the original MIT CSAIL paper (arXiv:2512.24601).

Our Algorithm (as implemented)

User Query + Context (file/dir/string)

1. Materialize context → plain Python str
2. Optionally build "context sample" (--no-context-sample disables)
3. Create depth-tracked llm_query(snippet, task) closure (max_depth=3)
4. Initialize REPLEnv with namespace: CONTEXT, FILES, llm_query, FINAL, SHOW_VARS, pre-imported modules
5. [Context Engineer - pre_loop] (if --context-engineer-mode includes pre_loop)
Analyze document sample → produce "document brief" (200-500 words)
Optionally share brief with root LM (--share-brief-with-root)
6. Send to LLM: system prompt (full/compact) + user query [+ context sample]
[+ document brief if share_brief_with_root]

RLM Loop (max_iterations, timeout, token budget)

7. Check timeout / token budget guards
8. LLM responds with markdown
9. Extract ```python blocks (regex)
10. If no code blocks:
 - Check for "FINAL" in text → done
 - Else prompt LLM to write code
11. Execute each code block sequentially
in persistent REPL namespace
llm_query(snippet, task):
 - a. [Context Engineer - per_query]
(if mode includes per_query)
Produce context note from
surrounding text
 - b. Sub-RLM receives:
[document brief] +

```

        [context note] +
        snippet + task
    c. Returns plain text
12. If FINAL() called → done
13. Feed captured print() output back
    as user message: "Output:\n..."
14. Continue loop

```

Return RLMResult(answer, stats, history)

Key files: rlm/rlm.py (orchestrator), rlm/repl.py (REPL environment),
rlm/prompts.py (system prompts), rlm/backends.py (LLM backends)

LLM Role Interaction (Sequence Diagram)

Shows the full interaction between all four LLM roles during a single completion() call with context_engineer_mode="both", share_brief_with_root=True, and verify=True (all features enabled).

```

sequenceDiagram
    actor User
    participant O as RLM Orchestrator
    participant CE as Context Engineer
    participant Root as Root LM
    participant REPL as REPL Environment
    participant Sub as Sub-RLM
    participant V as Verifier

    User->>O: completion(context, query)

    Note over O: _setup_completion()

    O->>O: Create REPL + llm_query closure
    O->>O: Build document sample<br/>(4 × 600-char excerpts)

    rect rgb(240, 248, 255)
        Note over O,CE: Context Engineer - pre_loop<br/>(if mode = pre_loop | both)
        O->>CE: system: CE_PRE_LOOP_PROMPT<br/>user: query + document sample
        CE-->>O: Document brief (200-500 words)
        O->>O: Bind brief to llm_query closure
        Note over O: If share_brief_with_root:<br/>prepend brief to root user prompt
    end

    O->>O: Build messages:<br/>[system_prompt, user_prompt + sample + brief?]

```

```

loop Main iteration loop (max_iterations, timeout, token budget)
  0->>0: Check guards (timeout, token budget)

  0->>Root: messages (full conversation history)
  Root-->>0: Response with ```python``` blocks

  0->>0: Extract code blocks (regex)

  alt No code blocks found
    alt "FINAL" in response text
      Note over 0: Break loop
    else
      0->>0: Append "Please provide Python code"
      Note over 0: Continue loop
    end
  else Code blocks found
    0->>REPL: Execute code blocks sequentially

    opt Code calls llm_query(snippet, task)
      rect rgb(255, 248, 240)
        Note over REPL,CE: Context Engineer - per_query<br/>(if mode = per_query
        REPL->>0: llm_query(snippet, task)
        0->>0: Locate snippet in CONTEXT<br/>(find first 200 chars)
        0->>0: Extract ~500 chars before + after
        0->>CE: system: CE_PER_QUERY_PROMPT<br/>user: task + position + surround
        CE-->>0: Context note (50-150 words)
      end

      rect rgb(240, 255, 240)
        Note over 0,Sub: Sub-RLM call
        0->>Sub: system: SUB_RLM_PROMPT<br/>user: [brief] + [context note] + sn
        Sub-->>0: Plain text analysis
      end
      0-->>REPL: Return sub-RLM response
    end

    REPL-->>0: Execution output (print + final_answer?)

    alt FINAL() called
      Note over 0: Break loop → finalize
    else No FINAL yet
      0->>0: Append output to messages:<br/>"Output:\n{output}"
      Note over 0: Continue loop
    end
  end
end

```

```

end

rect rgb(248, 240, 255)
    Note over O,V: Verification (if --verify)
    O-->V: system: VERIFIER_PROMPT<br/>user: answer + first 5000 chars of context
    V-->O: "VERIFIED" or "ISSUES: ..."
    Note over O: Append issues note<br/>if verification fails
end

O-->User: RLMResult(answer, stats, history)

```

Data Flow Between Roles

Shows what data each role receives and produces, and how outputs flow to downstream consumers.

```

flowchart TD
    Q[User Query + Context] --> O[RLM Orchestrator]

    O -->|document sample + query| CE_PRE["Context Engineer<br/>(pre-loop)"]
    CE_PRE -->|document brief| O

    O -->|system prompt + user prompt<br/>+ sample + brief | ROOT[Root LM]
    ROOT -->|Python code| REPL[REPL Environment]
    REPL -->|print output| ROOT

    REPL -->|snippet + task| CE_PQ["Context Engineer<br/>(per-query)"]
    CE_PQ -->|context note| SUB[Sub-RLM]

    O -.->|document brief| SUB
    REPL -->|snippet + task| SUB
    SUB -->|plain text| REPL

    ROOT -->|"FINAL(answer)"| O
    O -->|answer + evidence| V[Verifier]
    V -->|verdict| O
    O -->|RLMResult| U[User]

    style CE_PRE fill:#e8f4fd,stroke:#4a90d9
    style CE_PQ fill:#fff3e0,stroke:#f5a623
    style ROOT fill:#e8f5e9,stroke:#4caf50
    style SUB fill:#e8f5e9,stroke:#81c784
    style V fill:#f3e5f5,stroke:#ab47bc
    style REPL fill:#fff9c4,stroke:#fbc02d

```

Legend

| Arrow | Meaning |
|--------------|---|
| brief | Only if <code>--share-brief-with-root</code> |
| Solid line | Always active when the role is invoked |
| Dashed line | Conditional on <code>context_engineer_mode</code> |

What each role sees

| Role | System Prompt | User Message Contains | Produces |
|-------------------------------------|--|---|--|
| Context Engineer (pre-loop) | Document analysis specialist prompt | Query + 4×600 -char document excerpts | Document brief (200–500 words) |
| Context Engineer (per-query) | Context note specialist prompt | Task + snippet position + ~500 chars before/after | Context note (50–150 words) |
| Root LM | Full strategy prompt (Inspect→Search→Chunk→Synthesize) | Query + document sample [+ brief] | Python code in <code>```python</code> blocks |
| Sub-RLM | Minimal text-analysis prompt | [Brief] + [context note] + snippet + task | Plain text (summary, extraction, etc.) |
| Verifier | Verification prompt | Proposed answer + first 5000 chars of context | “VERIFIED” or “ISSUES: ...” |

Paper’s Algorithm

User Query + Context

1. Store context as `CONTEXT` variable in Python REPL
2. Provide `llm_query(snippet, task)` for sub-RLM calls
3. Root LM receives ONLY the query (context is not inline)
4. Send to LLM: system prompt + query (no context sample)

Code-Observe-Reason Loop

5. LLM generates Python code
6. Execute in REPL sandbox
7. Observe output (truncated)
8. LLM decides next action:
 - More code (inspect/search/chunk)
 - `llm_query()` for semantic analysis
 - `FINAL(answer)` when confident
9. Continue loop

Return final answer

Side-by-Side Comparison

| Aspect | Paper | Our Implementation | Delta |
|---------------------------------|---|--|---|
| Initial context exposure | Root LM sees only the query; CONTEXT is opaque until code runs | Context sample included by default; <code>--no-context-sample</code> disables it | Aligned — can now match paper's opaque-context design via flag |
| llm_query signature | <code>llm_query(snippet, task)</code> — two args | <code>llm_query(snippet, task)</code> — two args | Match |
| Recursive depth | Depth-1 only (root → sub-RLM, no deeper) | Configurable <code>max_depth=3</code> (supports multi-level recursion) | We're ahead of the paper here |
| Sub-RLM model | Smaller/cheaper model for sub-calls (e.g., GPT-5-mini when root is GPT-5) | <code>sub-rlm-model</code> param exists but defaults to same model as <code>--model</code> ; per-role backends/models via <code>--config</code> YAML | Paper's design is more cost-efficient; config file enables it |

| Aspect | Paper | Our Implementation | Delta |
|------------------------------------|--|--|---|
| System prompt for sub-calls | Not detailed | Minimal task-focused system prompt (text analysis role) | Aligned — sub-calls now receive focused guidance |
| Termination | FINAL() + max iterations + timeouts | FINAL() + max iterations + <code>--timeout</code> + <code>--max-token-budget</code> | Match |
| Default model | Not applicable (paper uses specific models per experiment) | <code>--model</code> is required (no defaults) | Aligned — user must choose explicitly |
| Code extraction | Not specified in detail | Regex: <code>```python\n(.*)```</code> — only explicitly tagged Python blocks | Reasonable; ignores non-Python fenced blocks |
| Structured output | Not discussed (implicitly uses free-form text) | <code>LLMBackend.supports_structured_output</code> property (default <code>False</code>); <code>StructuredResponse</code> dataclass and <code>STRUCTURED_RESPONSE_SCHEMA</code> for JSON-schema mode; <code>CompletionResult.structured</code> carries parsed response when available | Extension beyond the paper; interface only — no backends implement it yet |
| Sandbox isolation | Rootless container (security delegated to runtime) | Safe builtins whitelist (no <code>__import__</code> , <code>open</code> , <code>exec</code>) + container delegation | We add in-process restrictions on top of container isolation |
| Output truncation | Mentioned (first ~10K chars) | 10,000 char cap per execution | Match |
| Multi-file support | Not discussed | <code>FILES</code> dict + <code>CompositeContext</code> for directories | Extension beyond the paper |

| Aspect | Paper | Our Implementation | Delta |
|------------------------------|--|---|---|
| Context loading | Context as string | Memory-mapped files via LazyContext , CompositeContext for multi-file | Engineering refinement beyond the paper |
| Async execution | Identified as future work | Stubs exist (acompletion) but delegate to sync | Neither implements true async |
| Native post-training | RLM-Qwen3-8B (28.3% improvement over base) | Prompt-only, no fine-tuning | Paper has a training component we don't |
| Cost tracking | Discussed (comparable median, high variance tail) | RLMStats tracks tokens, iterations, sub-RLM calls; --max-token-budget enforces limits | Aligned — cost enforcement now available |
| Verification strategy | Some models perform redundant verification sub-calls | --verify flag enables explicit verification sub-call on the final answer | Paper observes this as emergent behavior; we formalize it as opt-in |
| Sub-RLM context | Sub-RLM receives only snippet + task (no document context) | Context-engineer role can provide document brief and per-query context notes to sub-RLM calls | Extension beyond the paper; opt-in via --context-engineer-mode |

LLM Roles

The RLM pattern uses up to four distinct LLM roles with different responsibilities:

Role Comparison

| Aspect | Root LM | Sub-RLM | Verifier | Context Engineer |
|---------------|---|--|---|--|
| Depth | 0 | 1+ (up to <code>max_depth</code>) | 0 (post-processing) | 0 (pre-processing / per-query) |
| System prompt | Full strategy prompt (~120 lines) or compact (~25 lines) | Minimal task-focused prompt | Verification focused prompt | Pre-loop: document analysis prompt; Per-query: context note prompt |
| Conversation | Multi-turn (code → output → code → ... → FINAL) | Single-turn (one prompt, one response) | Single-turn | Single-turn (per invocation) |
| Produces | Python code in <code>```python</code> blocks | Plain text (summary, extraction, analysis) | VERIFIED or ISSUES: ... | Document brief (pre-loop) or context note (per-query) |
| Has access to | <code>CONTEXT</code> , <code>FILES</code> , <code>llm_query()</code> , <code>FINAL()</code> , <code>SHOW_VARS()</code> , pre-imported modules | Only the snippet passed by root + optional brief/note | Proposed answer + evidence | Document sample (pre-loop) or snippet + surrounding text (per-query) |
| Model | <code>--model</code> | <code>--sub-rlm-model</code> (defaults to <code>--model</code>) | Config only (defaults to sub-RLM model) | Config only (defaults to sub-RLM model) |
| Backend | <code>--backend</code> or config | Config only (defaults to root's) | Config only (defaults to root's) | Config only (defaults to root's) |
| Custom prompt | Via <code>--config</code> <code>roles.root.system_prompt</code> | Via <code>--config</code> <code>roles.sub_rlm.system_prompt</code> | Via <code>--config</code> <code>roles.verifier.system_prompt</code> | Via <code>--config</code> <code>roles.context_engineer.system_prompt</code> (pre-loop) and <code>per_query_prompt</code> |

Root LM Responsibilities

The root LM writes Python code to explore the document programmatically. It never sees the full document inline (unless `--no-context-sample` is disabled). Instead, it:

1. Receives the user query (and optionally a document sample)
2. Writes exploration code to inspect structure, search for patterns, extract chunks
3. Delegates **semantic** analysis to the sub-RLM via `llm_query(snippet, task)`
4. Aggregates sub-RLM results and calls `FINAL(answer)` when confident

Sub-RLM Responsibilities

The sub-RLM processes individual text chunks. It:

1. Receives a snippet of text and a task description
2. Returns plain-text results (summaries, extractions, classifications)
3. Has no access to the REPL, `CONTEXT`, or any tools — only the snippet it was given
4. Does not generate code

Emergent Strategies from the Paper

The paper (arXiv:2512.24601) and the accompanying blog post identify five emergent strategies that models discover when given the RLM environment:

| Strategy | Description | Example Code |
|-------------------------------|---|---|
| Peeking | Inspect document structure and size | <code>print(CONTEXT[:1000]), print(len(CONTEXT))</code> |
| Grepping | Keyword/regex search to locate relevant regions | <code>re.findall(r'pattern', CONTEXT, re.IGNORECASE)</code> |
| Partition+Map | Chunk the document and fan out <code>llm_query()</code> calls | Split <code>CONTEXT</code> into N chunks, call <code>llm_query()</code> on each |
| Summarization | Hierarchical summarization via sub-RLM calls | Summarize chunks, then summarize the summaries |
| Long-input/long-output | Direct extraction for structured outputs | Extract lists, tables, or formatted data from large regions |

Our prompts map these emergent strategies to a prescriptive four-phase model:

| Our Phase | Paper Strategies Covered |
|------------------------|---|
| Inspect | Peeking |
| Search | Grepping |
| Chunk + Analyze | Partition+Map, Summarization |
| Synthesize | Long-input/long-output, final aggregation |

The paper observes that models discover these strategies naturally with minimal prompting. Our prompts are more prescriptive (explicit chunk size guidance, specific function examples), which improves consistency but may constrain creative exploration.

Context Engineer (Extension Beyond Paper)

The paper’s sub-RLM calls operate in complete isolation — each `llm_query(snippet, task)` gives the sub-RLM only the snippet and task, with no knowledge of the document’s type, structure, terminology, or where the snippet sits. This limits sub-RLM accuracy for domain-specific documents (legal contracts, research papers, technical specs).

Our context-engineer role addresses this limitation with two configurable intervention points:

| Mode | When | LLM Calls | What it does |
|------------------|---|-----------|--|
| off | Never (default) | 0 | Current behavior, matches paper |
| pre_loop | Before main loop | 1 | Produces document brief from sample |
| per_query | Before each <code>llm_query()</code> | N | Adds surrounding context + position metadata |
| both | Both points | 1 + N | Brief + per-query enhancement |

Pre-loop analysis (`_run_pre_loop_analysis`): Feeds a document sample to the CE backend and receives a document brief (200-500 words) covering document type, key terminology, structure, important entities, and writing conventions. The brief is then prepended to every sub-RLM call, and optionally shared with the root LM (`--share-brief-with-root`).

Per-query enhancement (`_enhance_per_query`): Before each `llm_query()`, locates the snippet within the full `CONTEXT`, extracts ~500 chars of surrounding text, and calls the CE backend to produce a context note (50-150 words) about position, terminology references, and relevant background. The note is prepended to the sub-RLM prompt.

The context-engineer defaults cascade from the sub-RLM role (both are “helper” roles), so configuring `sub_rlm` in the YAML config automatically applies to `context_engineer` unless overridden.

Verification (Emergent Behavior)

The paper observes that some models perform redundant “verification” sub-calls — re-checking their own intermediate results before calling `FINAL()`. This behavior is emergent and not prompted for. Our implementation offers an explicit `--verify` flag that formalizes this pattern as an opt-in post-processing step.

Remaining Divergences

1. Context Sample in Initial Prompt (our addition, now optional)

Our approach (`_build_context_sample()`): By default, sends evenly-spaced 500-char excerpts from the document in the first message. Can be disabled with `--no-context-sample` to match the paper’s opaque-context design.

Paper’s approach: Root LM only knows `CONTEXT` exists. It must write code to inspect it.

Trade-off: The default gives the LLM a head start (knows format, size, structure immediately) but adds tokens to the initial prompt. Use `--no-context-sample` for the paper’s strict “context as environment” paradigm.

2. Multi-level Recursion (our extension)

Paper: Depth-1 only (root \rightarrow sub-RLM, no deeper).

Ours: Configurable `max_depth=3` (supports multi-level recursion).

This is an intentional extension beyond the paper’s design.

Emergent Strategy Comparison

Both the paper and our prompts describe the same four-phase strategy:

| Phase | Paper | Our Prompts |
|----------------|---|--|
| Inspect | Examine context structure, size, format | If sample provided, study it; otherwise <code>print(CONTEXT[:1000])</code> |

| Phase | Paper | Our Prompts |
|-------------------|--|---|
| Search | Regex/keyword filtering to narrow focus | “Start BROAD: <code>re.findall(pattern, CONTEXT)</code> with <code>re.IGNORECASE</code> ” |
| Chunk | Uniform or semantic partitioning | “Extract sections: <code>CONTEXT[start:end]</code> (aim for 1000-5000 chars)” |
| Synthesize | Aggregate sub-RLM results + <code>FINAL()</code> | “Combine and call <code>FINAL(answer)</code> ” |

Our prompts also include guidance on when to use `llm_query()` (semantic tasks) vs Python directly (structural tasks). The paper observes these strategies as emergent behavior that models discover with minimal prompting; our prompts are more prescriptive to improve consistency.

Future Refinement Candidates

Actionable items identified by this comparison:

- **Default to a cheaper model for sub-RLM calls** — leverage `--sub-rlm-model` param with a smaller default
- **Evaluate prompt prescriptiveness** — test whether our detailed prompts help or constrain the LLM compared to the paper’s minimal approach
- **Native post-training** — the paper fine-tuned RLM-Qwen3-8B (28.3% improvement over base); we rely on prompt-only guidance
- **True async execution** — replace sync-delegating `acompletion` stubs with genuine async REPL execution
- **Context-engineer token budget** — CE calls currently draw from the same `max_token_budget`; per-query mode (N calls) can exhaust budget faster; consider separate budgets or adaptive call limits
- **Context-engineer snippet location** — `_enhance_per_query()` uses `context_str.find(snippet[:200])` which can fail if the root LM transforms the snippet; more robust matching (fuzzy, n-gram) could improve hit rate
- **Structured output backend implementations** — `LLMBackend ABC` now defines `supports_structured_output` and `StructuredResponse/STRUCTURED_RESPONSE_SCHEMA`; concrete backends (Anthropic, OpenAI-compatible) need to implement JSON-schema response mode to replace regex-based code extraction