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. Build "context sample" (evenly-spaced excerpts showing size + structure)
3. Create depth-tracked llm_query() closure (max_depth=3)
4. Initialize REPLEnv with namespace: CONTEXT, FILES, llm_query, FINAL,
   FINAL_VAR, SHOW_VARS, pre-imported modules


5. Send to LLM: system prompt (full/compact) + user query + context sample


   Iteration Loop (max_iterations=10)

  6. LLM responds with markdown
  7. Extract ```python blocks (regex)
  8. If no code blocks:
       - Check for "FINAL" in text → done
       - Else prompt LLM to write code
  9. Execute each code block sequentially
       in persistent REPL namespace
 10. If FINAL()/FINAL_VAR() called → done
 11. Feed captured print() output back
       as user message: "Output:\n..."
 12. 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)

Paper's Algorithm

```
User Query + Context


1. Store context as CONTEXT variable in Python REPL
2. Provide llm_query(snippet, task) for recursive sub-LM 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 (evenly-spaced excerpts) included in initial message	We leak structure upfront — helpful but diverges from the paper's "context as environment" principle

Aspect	Paper	Our Implementation	Delta
`llm_query` signature	`llm_query(snippet, task)` — two args	`llm_query(prompt)` — one arg (user builds the prompt)	Paper separates context snippet from task; ours conflates them
Recursive depth	Depth-1 only (root → sub-LM, no deeper)	Configurable `max_depth=3` (supports multi-level recursion)	We're ahead of the paper here
Sub-LM model	Smaller/cheaper model for sub-calls (e.g., GPT-5-mini when root is GPT-5)	`recursive_model` param exists but defaults to same model as root	Paper's design is more cost-efficient
System prompt for sub-calls	Not detailed	No system prompt — bare user message only	Consistent but sub-calls might benefit from minimal guidance
Termination	`FINAL()` + max iterations + timeouts	`FINAL()` + `FINAL_VAR()` + `final_` prefix fallback + max iterations	We have more termination paths; `FINAL_VAR` is partially broken
Code extraction	Not specified in detail	Regex: ```` ```python\n(.*?)``` ```` — only explicitly tagged Python blocks	Reasonable; ignores non-Python fenced blocks

Aspect	Paper	Our Implementation	Delta
Sandbox isolation	Rootless container (security delegated to runtime)	Safe builtins whitelist (no `__import__`, `open`, `exec`) + 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	`FILES` dict + `CompositeContext` for directories	Extension beyond the paper
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, recursive calls	We track but don't enforce cost limits
Verification strategy	Some models perform redundant verification sub-calls	Not prompted for; up to the LLM	Paper observes this as emergent behavior

Key Divergences

1. Context Sample in Initial Prompt (our addition)

Our approach (`_build_context_sample()`): Sends evenly-spaced 500-char excerpts from the document (beginning, ~25%, ~50%, ~75%, end) in the first message.

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

Trade-off: Our approach gives the LLM a head start (knows the format, size, structure immediately) but adds tokens to the initial prompt and partially undermines the "context as environment" paradigm. The paper argues the power of RLM is that the root LM's context window never grows beyond query + REPL output.

2. Single-arg vs Two-arg `llm_query`

Paper: `llm_query(snippet, task)` — cleanly separates the context chunk from the instruction.

Ours: `llm_query(prompt)` — the LLM must construct a single prompt string containing both the snippet and the task.

Impact: The two-arg version is more structured and could allow the system to format sub-calls more consistently (e.g., always putting the snippet in a designated section).

3. `FINAL_VAR` Is Broken

Per CLAUDE.md: "`_final_var` is a no-op; the fallback logic checks for `final_` prefixed variables in the namespace instead." This means `FINAL_VAR("my_result")` doesn't actually look up `my_result` — it falls back to scanning for variables named `final_*`.

4. No Cost/Timeout Guards

The paper mentions cost limits and timeouts as practical safeguards. Our implementation only has `max_iterations` (default 10) and `max_depth` (default 3). No wall-clock timeout or token budget enforcement.

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	"Study the document sample… understand format"
Search	Regex/keyword filtering to narrow focus	"Start BROAD: `re.findall(pattern, CONTEXT)` with `re.IGNORECASE`"
Chunk	Uniform or semantic partitioning	"Extract sections: `CONTEXT[start:end]` (aim for 1000-5000 chars)"
Synthesize	Aggregate sub-LM results + FINAL()	"Combine and call `FINAL(answer)`"

Our prompts are more prescriptive (explicit chunk size guidance, specific function examples). The paper observes these strategies as emergent behavior that models discover with minimal prompting.

Future Refinement Candidates

Actionable items identified by this comparison:

- **Fix `FINAL_VAR` implementation** — make `_final_var` actually look up the named variable
- **Make context sample optional** — consider removing or gating behind a flag to match the paper's opaque-context design
- **Two-arg `llm_query(snippet, task)` signature** — align with the paper for cleaner sub-call semantics
- **Add cost/timeout guards** — wall-clock timeout and token budget enforcement
- **Default to a cheaper model for recursive sub-calls** — leverage `recursive_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