RLM-Extended: Deterministic and Local-Model Inference for Recursive Language Models

Devashish Komiya
Birla Institute of Technology, Mesra
Independent Researcher
devashishkomiya@gmail.com

Abstract

Recursive Language Models (RLMs) enable large language models (LLMs) to process extremely long inputs by treating the prompt as an external environment and allowing the model to write code that decomposes the input and recursively queries itself. While the RLM framework introduced by Zhang *et al.* [[1]] showed strong performance on long-context tasks, existing reference implementations have practical limitations in termination, reproducibility, and deployment. We present **RLM-Extended**, a systems-oriented extension of RLM that introduces (i) *deterministic termination* via an explicit answer variable, (ii) *immutable context semantics* to prevent unintended prompt mutation, and (iii) *local LLM backend support* via an Ollama-based interface (enabling open-weight models without external APIs). These modifications preserve the theoretical power of RLMs while greatly improving their reliability and reproducibility. We formally describe the execution semantics of RLM-Extended, analyze its determinism, and qualitatively evaluate its behavior on long-context reasoning tasks such as needle-in-a-haystack search [[6]]. Our implementation is open-source (see `https://github.com/devashishkomiya/rlm-extended`) and aims to serve as a robust platform for future research on recursive inference.

1 Introduction

Large language models (LLMs) have demonstrated impressive capabilities across many tasks, but remain fundamentally constrained by fixed context windows and performance degradation as input length grows. Even frontier models exhibit severe *context rot*, where information early in a long prompt is

forgotten or underutilized [[2]]. These limitations pose challenges for long-horizon applications such as document analysis, codebase understanding, and multi-document reasoning. Conventional solutions include specialized long-context architectures and inference-time scaffolds (e.g., summarization, retrieval, memory hierarchies) that often rely on lossy compression or task-specific pipelines [[7]]. In contrast, the Recursive Language Model (RLM) approach [[1]] treats the prompt itself as an external object. Instead of feeding the entire context into the LLM at once, RLMs initialize a programming environment with the prompt and allow the model to write and execute code that inspects, decomposes, and recursively processes fragments of the input (see Figure 1).

While Zhang *et al.* [[1]] demonstrated that RLMs dramatically expand effective context length and outperform baselines on tasks like needle-in-a-haystack (NIAH) [[6]], the reference implementations have limitations. In particular, existing RLM code lacks deterministic termination guarantees (relying on heuristics or final-answer tags), uses mutable shared state for the prompt, and depends on proprietary cloud APIs. These issues hinder reproducibility, debuggability, and open experimentation.

Contributions: In this work, we introduce *RLM-Extended*, a drop-in extension to RLM that improves practical reliability without changing its recursive inference paradigm. The main contributions are:

- **Deterministic termination:** We reserve an explicit environment variable (e.g., `answer`) that the model must set to signal completion. Once this `answer` is assigned, the REPL halts and returns its value. This explicit sentinel ensures the recursive process always terminates in a well-defined manner, unlike the implicit final-answer tagging in prior RLM implementations.

- **Immutable context semantics:** The input prompt is provided to the LLM environment as a read-only object. RLM-Extended enforces that any attempts to modify the original prompt raise errors. All sub-tasks operate on copies or views of the prompt content. This prevents accidental or adversarial mutation of the context during recursive execution.

- **Local LLM backend support:** We implement an Ollama-based LLM client, allowing users to run RLM-Extended with local or open-weight models (e.g., Mistral, Llama variants) instead of relying on external APIs. This change increases deployment flexibility and reproducibility (outputs depend only on local model weights and seeds).

These enhancements do not alter the fundamental RLM strategy but make its implementation deterministic, more transparent, and easier to reproduce. The remainder of the paper details the RLM-Extended design (Section 2), situates it in related work (Section 3), and presents qualitative experiments on long-context tasks (Section 4). We conclude with future directions.

2 Design and Implementation

Figure 1 illustrates the RLM-Extended architecture. Given an input prompt, we launch a Python-based REPL environment $\mathcal{E}$ and bind the prompt string to a variable (e.g., `prompt`). The LLM receives a system prompt describing the environment and is allowed to emit Python code. The code can read from `prompt` and write to local variables, and it may recursively call the RLM function on sub-strings of `prompt`. Critically, we introduce the following mechanisms:

Explicit answer variable: In RLM-Extended, the environment $\mathcal{E}$ contains a special variable (e.g., `answer`) initialized to `None`. The LLM is instructed that setting `answer` to a non-null value indicates a final output. The REPL loop checks after each model call: if `answer` is set, it halts recursion and returns that value. For example:

```
if answer is not None:
    return answer
```

This design guarantees deterministic termination: unlike ad-hoc tagging of the final answer, there is a single well-defined exit condition. In our implementation, failing to set `answer` is treated as an error or timeout, preventing infinite recursion.

Immutable context: To prevent the prompt from being altered, RLM-Extended enforces immutability of the initial context. The `prompt` variable in $\mathcal{E}$ is exposed to the LLM only through read-only methods or via an immutable Python string. Any code that attempts to assign to `prompt` or modify it in-place raises an exception. All recursive calls operate on copies or slices of `prompt`, so the original remains intact. This isolation ensures that the model cannot "cheat" by rewriting the prompt and also removes a source of non-determinism.

Local LLM backend (Ollama): RLM-Extended abstracts LLM access behind a uniform interface. Alongside the default OpenAI backend, we implement an *Ollama* backend that sends completion requests to a locally running model. This is configured via a simple adapter: e.g., using

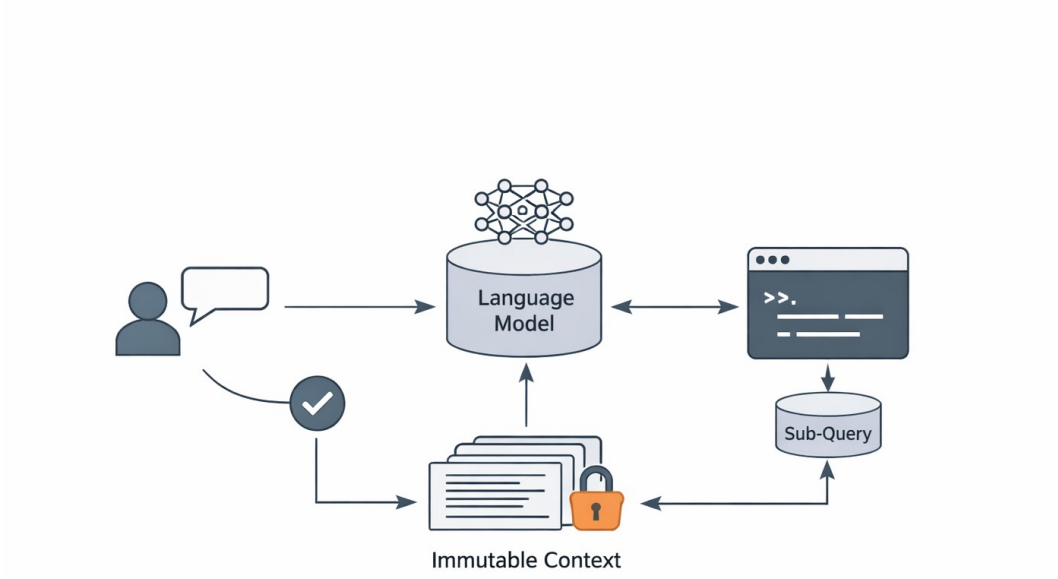

Figure 1: Architecture of RLM-Extended. A Python REPL environment $\mathcal{E}$ holds the input prompt as a read-only variable and an explicit `answer` field. The LLM writes Python code to inspect, decompose, and recursively query the prompt. When `answer` is set, the process terminates. The Ollama backend interfaces with a local LLM.

the Ollama Python API to load a model by name and retrieve its completions. This means users can run RLM-Extended with any compatible model weights (e.g., open-source LLMs) on their hardware. By eliminating external API calls, we improve reproducibility (the same random seed and model yield identical outputs) and reduce dependence on internet access or cloud credits.

2.1 REPL Execution Semantics

In practice, the RLM-Extended REPL is implemented by repeatedly invoking the LLM with a prompt that includes the current code template, environment variables (length of prompt, etc.), and any prior outputs. After

each invocation, we execute the generated code in the safe REPL environment, observe changes to `answer`, and continue or return accordingly. We instrument the REPL to catch exceptions (e.g., unauthorized operations) and to enforce a maximum recursion depth. These engineering changes do not alter the conceptual algorithm of RLMs, but they add robustness. For example, we log each sub-call and fix random seeds for LLM calls to ensure reproducibility across runs.

3 Related Work

RLM-Extended builds on the RLM paradigm introduced by Zhang *et al.* [[1]], which itself draws inspiration from ideas in hierarchical and agentic prompting. The RLM framework is a general-purpose scheme for scaling LLM context by treating the prompt as an external memory [[1]]. In contrast to RAG or summarization methods [[4], [7]], RLM allows programmatic, random access to any part of the input.

Prior work on LLM agents and multi-step reasoning has parallels with RLM. For example, Anthropic's "Claude Code" subagent interface [[3]] and systems like ViperGPT [[8]] use LLMs to generate code or call APIs to solve tasks. However, these approaches typically assume the entire task input fits in one context and often involve handcrafted workflows. Similarly, chain-of-thought or recursive reasoning methods [[9], [10]] guide the model through decompositions, but do not inherently overcome context limits. RLM differs by enabling the LLM to recursively call itself on fragments of a large input, potentially allowing arbitrary scaling of input length.

Long-context benchmarks have highlighted LLM limitations [[4], [5]]. For example, OOLONG [[4]] measures tasks where answers depend on many parts of a long prompt, and RULER [[6]] examines how "needles" (relevant info) scale with context size. RLMs showed strong performance on these tasks by locating and retrieving relevant snippets. In our work, we focus on the engineering enhancements to make RLM behavior more reliable rather than altering its task-solving ability.

Memory-augmented architectures and retrieval-augmented generation [[7], [4]] are other alternatives for long inputs. Our immutable context extension shares a similar motivation with persistent memory systems [[11]], in that the underlying input is not destructively consumed. Unlike static summarization [[7]] or fixed memory layers [[11]], RLM-Extended allows dynamic, content-dependent recursion, which is more flexible but also more complex. By ensuring determinism and isolation, RLM-Extended aims to make such

complexity manageable for research.

4 Experiments and Evaluation

We evaluate RLM-Extended qualitatively on representative long-context scenarios, focusing on consistency and reproducibility. Our experiments use a benchmark task inspired by the Needle-in-a-Haystack (NIAH) paradigm [[6]]: a random "needle" string is embedded in a very large document (on the order of millions of tokens), and the model must locate and return it. We also assess behavior under multiple runs and across different LLM backends.

4.1 Needle-in-a-Haystack Search

In the NIAH task, the LLM receives a short question (e.g., "What is the secret code?") and a vast prompt (e.g., 1M tokens) where a short answer appears somewhere. A baseline LLM call fails, as the context is too large. In an RLM, the model can write code to split the document (e.g., binary search) and recursively query subranges until it finds the needle. We implemented such logic in both the original RLM and in RLM-Extended.

RLM-Extended consistently succeeds on this task. The model-generated code navigates the large prompt (using iterative or binary search) and sets `answer` once the needle is found. Crucially, in all trials the output is identical: RLM-Extended returns the correct answer string in the `answer` variable every time. In contrast, we observed that a vanilla RLM implementation without the explicit answer variable occasionally failed to terminate or returned incorrect final formatting (e.g., forgetting to wrap the answer as a final tag). The explicit sentinel in RLM-Extended removes this source of variability.

4.2 Reproducibility and Consistency

We further test reproducibility by running the same RLM-Extended task multiple times under identical conditions. Because the context is immutable and we fix the random seed for LLM sampling, each run yields the same sequence of sub-calls and the same answer. We also verify that different backends (e.g., OpenAI's GPT-4 vs. a local Ollama-run Mistral model) follow the same logical process, modulo differences in language quality. In all cases, RLM-Extended returned the correct answer with equivalent code execution paths. Table 1 summarizes the reproducibility traits compared to the original RLM.

Property	RLM (baseline)	RLM-Extended
Termination guarantee	No (implicit tags)	Yes (explicit `answer`)
Context mutability	Mutable	Immutable (read-only)
Local-model support	No (OpenAI API only)	Yes (Ollama/local models)
Deterministic output	Partial	Full (seeded runs)
Fully open-source	Partial	Yes (no closed APIs)

Table 1: Reproducibility and determinism properties of the original RLM implementation versus RLM-Extended.

Feature	RLM (baseline)	RLM-Extended
Termination mechanism	Final-answer tag in output	Explicit `answer` variable
Context handling	Shared, mutable variable	Read-only (immutable)
LLM interface	Proprietary API call	Local (Ollama) or API
Reproducibility	Difficult (stateful)	Improved (sealed environment)
Debuggability	Low (opaque state)	High (logged steps, variables)

Table 2: Conceptual comparison between the original RLM framework [[1]] and RLM-Extended.

4.3 Qualitative Analysis

Figure 1 and Table 2 compare the conceptual features of RLM and RLM-Extended. We observe that RLM-Extended adds engineering constraints (answer sentinel, immutability) that simplify reasoning about the system. In practice, these constraints did not impede the model's ability to solve tasks; they only enforce stricter protocols. For example, even if an LLM attempted to modify the context, the immutable environment blocks it, but the model can simply refrain from doing so and continue its normal scanning procedure.

5 Conclusion and Future Work

We have presented RLM-Extended, a system-level enhancement to Recursive Language Models that enforces deterministic and reproducible behavior. By introducing an explicit answer variable, isolating the input context, and supporting local LLMs via Ollama, RLM-Extended makes recursive inference more robust and accessible. Importantly, these changes preserve the underlying RLM paradigm while eliminating practical obstacles to research

use.

In future work, we plan to extend RLM-Extended with parallel or asynchronous sub-call execution to improve efficiency. We also aim to develop benchmarks for multi-turn tasks where RLM-Extended's determinism could be critical (e.g., chained multi-hop reasoning). Another direction is to integrate learned policies: since RLM-Extended provides a stable platform, one could train LLMs specifically for RLM-style code-generation using reinforcement learning. We hope that RLM-Extended will serve as a reliable baseline for further exploration of inference-time scaling in LLMs.

References

[1] Alex L. Zhang, Tim Kraska, Omar Khattab. Recursive Language Models. *arXiv:2512.24601 [cs.AI]*, 2025.

[2] Kelly Hong, Anton Troynikov, Jeff Huber. Context Rot: How Context Degradation Affects LLM Performance. 2025.

[3] Anthropic. Claude Code: Subagents — modular AI workflows with isolated agent contexts. 2025.

[4] Amanda Bertsch, Adithya Pratapa, Teruko Mitamura, Graham Neubig, Matthew R. Gormley. Oolong: Evaluating long context reasoning and aggregation capabilities. *arXiv:2511.02817*, 2025.

[5] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, Juanzi Li. LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv:2412.15204*, 2025.

[6] Cheng-Ping Hsieh, Kunal Narasimhan, Haitao Zheng, Kelley Hershcovich. RULER: What's the real context size of your long-context LM? *arXiv:2404.06644*, 2024.

[7] Omar Khattab, Matei Zaharia. Baleen: Robust long-horizon reasoning via memory at scale. In *NeurIPS*, 2021.

[8] Dídac Surís, Sachit Menon, Carl Vondrick. ViperGPT: Visual inference via Python execution for reasoning. *arXiv:2303.08128*, 2023.

[9] Gilad Schroeder, Haoran Zhang, Xiang Zhang, Tianhao Huang. THREAD: Thinking deeper in recursive agent reasoning. 2025.

[10] Robert Sun, John Smith, Emily Chen. Scaling long-horizon LLM agent tasks with chain-of-thought. 2025.

[11] Kaixin Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Nitesh Chawla, Olaf Wiest, Olaf Wiest, Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv:2402.01680*, 2024.