# Towards More Efficient Long-Context Handling: Smarter Partitioning, Retrieval, and Execution in Recursive Language Models

Lillian Bluestein Davis Lee Kristian Praizner Teimurazi Toloraia

December 26, 2025

## Abstract

Long contexts remain difficult for language models, with empirical evidence demonstrating that accuracy declines even when inputs are well within the model's supported length. Recursive Language Models (RLMs) provide a way to handle these inputs by partitioning the context and invoking the model on smaller subproblems. Existing implementations, specifically Zhang (2025), rely on simple token or regex-based partitioning, basic retrieval heuristics, and strictly sequential execution. These settings make it unclear which parts of the RLM pipeline contribute most to performance. To address unexplored design space, we conduct a systematic component analysis of RLMs, decoupling partitioning (token, structural, semantic), retrieval (regex, embedding-based, unfiltered), and execution mode (sequential vs. parallel). Using OOLONG and LoCoDiff, we evaluate how these choices affect accuracy, token usage, and characteristic failure modes. Our results indicate that while direct prompting often outperforms current RLM implementations due to error accumulation in recursive steps, optimizing the RLM pipeline (specifically through semantic partitioning and embedding-based retrieval) significantly improves performance over naive token-based baselines. We further show that these smarter strategies introduce substantial efficiency overheads, highlighting a critical trade-off between architectural complexity and latency. This work provides a clearer understanding of which RLM design choices have significant effects on long-context performance.

## 1 Introduction

Long documents are common across many domains, from legal contracts to scientific papers. Tasks such as answering questions about these documents or tracing how information evolves across sections require models to integrate evidence that may be widely dispersed. As LLMs are increasingly applied to these settings, their ability to operate reliably over long contexts is becoming increasingly important.

Large language models can process inputs on the order of tens to hundreds of thousands of tokens, but their performance does not remain stable as inputs become longer. Empirical evaluations show that accuracy declines as context length increases, especially when the input mixes many unrelated topics. In these cases, the model has more difficulty identifying which parts of the context are relevant to the query. This degradation, commonly referred to as *context rot*, leads to consistent failures on tasks that require locating and combining information spread across long documents.

Context rot often stems from unreliable selection rather than limited capacity: models can ingest long inputs, but they struggle to identify the specific evidence needed for the query. However, most architectural advances target capacity, not selection. Approaches such as rotary embeddings (RoFormer) and long-range attention (Linformer, Longformer) mainly increase the number of tokens a model can accept. These methods expand the window but do not change how the model organizes or prioritizes information within that window. In practice, this means that models can read longer inputs but may not improve on tasks that require locating specific evidence across heterogeneous or multi-section documents.

Recursive Language Models (RLMs), as proposed by Zhang (2025), improve upon previous methods by introducing a recursive decomposition of long inputs. While previous methods treat the entire input as a single block, RLMs break the input into smaller partitions, allowing more focused attention on each segment. The model can call back into itself on subproblems, building a tree of recursive invocations whose leaves deal with smaller, more manageable

slices of the original context.

RLMs improve the structural handling of long inputs, but they introduce significant limitations. In particular, they rely on partitioning via token splits and regex-based chunking, which often fail to align with the semantic boundaries of the input. As a result, important dependencies may be split or lost in the partitioning process, limiting the RLM's ability to fully capture relevant relationships within the input. In addition, once partitions are formed, they are handled one after another, which may introduce unnecessary latency when sub-problems are independent.

These observations raise several design questions that existing work has not explored:

- How does the choice of *partitioning strategy* affect long-context accuracy and efficiency?
- How should RLMs *retrieve* or select partitions when the context is too large to process in full?
- Can we exploit the natural independence of many sub-problems to *run them in parallel* and reduce latency, and does the stitching method used to combine partial answers affect how reliable this parallel execution is?

To address these questions, we vary three under-explored RLM parameters: partitioning, retrieval, and parallelization. Our goal is to understand how these choices affect accuracy, token usage, and runtime behavior on long-context tasks, and to identify configurations that improve performance.

## 2 Related Work

Prior work on long-context performance can be grouped into three main categories: architectures that extend the effective context length of transformer models, retrieval-based methods that selectively surface relevant text, and recursive frameworks that decompose inputs into smaller units. We situate our work within these areas, highlighting how existing approaches address capacity or retrieval but leave open questions about how internal RLM components affect long-context performance.

### 2.1 Long-context architectures

There has been substantial prior work on how to increase the effective context length of transformer models. Rotary positional embeddings support extrapolation beyond trained context limits by encoding relative positions in a continuous rotational space [3]. Other approaches introduce modified attention mechanisms that reduce memory complexity, such as block-sparse patterns, sliding-window mechanisms, and long-range attention mechanisms (e.g., Linformer, Longformer) [4][5][7]. Transformer-XL extends context by caching hidden states across segments, while Memorizing Transformers store key-value pairs in an external memory for reuse across longer sequences [8][9].

More recent methods such as FlashAttention and its long-context extensions focus on improving the efficiency of exact attention rather than modifying the inductive bias [10]. These approaches primarily address the capacity problem (i.e., how many tokens the model can process), rather than the selection problem of determining which parts of the input matter for the query. As a result, improvements in raw context size do not necessarily translate to improved reasoning over heterogeneous or multi-topic documents. Our work focuses on the selection problem. We take the model's input capacity as given and evaluate how different partitioning and retrieval strategies affect performance within the RLM framework.

### 2.2 Retrieval-augmented generation

Retrieval-augmented generation (RAG) combines a pretrained language model with an external retrieval module that selects potentially relevant text before generation [11]. Prior work includes sparse retrieval, dense retrieval, and hybrid approaches that mix sparse and dense signals [12]. Multi-hop systems extend this setup by retrieving in stages, where each retrieval step conditions on the results of the previous one [13].

RAG methods work well when relevant information can be surfaced as a small set of passages drawn from a large corpus. These systems, however, typically treat each document or passage as an independent retrieval unit and do not reason about the internal structure of a long document. They also operate outside the model's own inference loop, with the retrieval module running first and then the language model conditions on whatever text is returned [11].

In contrast, our setting involves recursive decomposition inside the model. The RLM decides how to partition its provided context, which partitions to inspect, and when to invoke further recursive calls. This leads to a different set of design questions (i.e., partition type, partition selection, and execution strategy) that are not addressed by standard RAG pipelines.

### 2.3 Recursive Language Models

Recursive Language Models (RLMs), introduced by Zhang (2025), formalize a mechanism in which the model interacts with a minimal Python REPL that exposes the input context and provides a tool for recursive self-calls [6]. Rather than processing the entire input in a single forward pass, the model can choose to inspect the context, split it into smaller pieces, and invoke itself on these pieces as separate subproblems. Each call returns a textual result that becomes available to the parent call, and the model ultimately produces a final answer using a designated instruction.

This setup gives the model explicit control over how to break down the task and how to use intermediate results, shifting part of the reasoning process into a program-driven control flow. Zhang's prototype keeps the surrounding system intentionally minimal in order to isolate the behavior of the recursive mechanism. Partitioning is performed through simple operations such as token slicing or regex-based splitting, retrieval is handled by string matching, and recursive calls are executed sequentially. The REPL interface provides only a few primitive operations, including reading the context, printing intermediate values, and calling the model again on a selected sub-context.

This simplicity allows the work to demonstrate that RLMs can successfully coordinate multi-step decomposition without hand-crafted prompts, but it also means the system does not explore alternative partitioning strategies, retrieval methods, or execution modes. Our work extends this baseline by varying those components explicitly to study how they influence performance in long-context settings. We vary partitioning granularity, retrieval method, and execution mode to quantify how each dimension affects accuracy, token usage, and latency in long-context settings. To our knowledge, no prior study analyzes these design choices in a controlled empirical setting or compares alternative partitioning and retrieval strategies within RLMs. This paper addresses that gap.

## 3 Methods

To enable a systematic analysis of RLM behavior, we describe the RLM setup and the component choices evaluated in this work.

### 3.1 RLM framework and notation

We consider long-context tasks where the input consists of a query $q$ and a context $C$, such as a long document, code repository, or sequence of diffs. The goal of the RLM is to produce an answer $a = f(q, C)$ using a base language model $M$ and a recursive calling interface. In Zhang's framework, the model interacts with a restricted Python REPL that exposes a `context` variable containing $C$ and a tool for making recursive calls. Conceptually, the RLM:

1. writes Python code to inspect and transform `context`;
2. decides when to call itself recursively on subqueries and sub-contexts;
3. combines the results and emits a final answer via a `FINAL(...)` call.

We introduce a partitioning function

$$\mathcal{P}_\theta : C \mapsto (p_1, \ldots, p_K),$$

where each $p_k$ is a *partition* (a contiguous or structured subset of $C$) and $\theta$ encodes strategy-specific hyperparameters (e.g., chunk size, similarity thresholds). We also introduce a retrieval function

$$\mathcal{R}_\phi(q, \{p_k\}) \mapsto S \subseteq \{1, \ldots, K\}$$

that selects a subset of partition indices to process recursively, where $\phi$ denotes retrieval-specific parameters (e.g., number of partitions to select). Our modifications to the RLM core are chosen such that:

- the public API, `RLM(...).completion(query, context)`, remains unchanged;
- partitioning and retrieval are configurable and swappable;
- the recursion tree can fan out over selected partitions and processes them in parallel.

### 3.2 Partitioning Strategies

We implement and benchmark three partitioning methods:

- **Baseline (Token-based):** We replicate the original paper's naive chunking by fixed token or line count. This serves as our main baseline, and we aim to match the behavior of Zhang's released implementation as closely as possible.
- **Structural Partitioning:** We introduce a low-cost heuristic using explicit document structure, partitioning along natural boundaries like paragraphs (e.g., `\n\n`), section headings, or functions. For code or diffs, we partition by file, function, or hunk boundaries. The purpose of this structural partitioning is to preserve local meaning and coherence in a computationally cheap manner.
- **Semantic Partitioning:** We also partition based on detectable topic shifts by calculating cosine similarity between adjacent sentence em-

beddings and then splitting the text where similarity drops below a threshold. Specifically, we first break the context into sentences or short spans, compute their embeddings, then construct a similarity graph over adjacent spans. Whenever similarity falls below a tuned threshold, we insert a boundary. This challenges the original avoidance of semantic tools and is designed to improve sub-query relevance by grouping spans that are semantically related.

We represent each partition as a small data structure containing its raw text, offsets into the original context, and metadata (e.g., structural type, mean embedding). This metadata supports later retrieval and prevents having to re-embed or re-parse the entire document.

### 3.3 Retrieval Primitives

While partitioning decides *how* we slice the context, retrieval decides *which* slices to pass into recursive calls. We compare three retrieval primitives:

- **Regex / Keyword "Grepping":** For each partition, we count keyword or regex matches between the query and the partition text, scoring partitions by number and position of matches. This approximates Zhang's original RLM behavior and serves as a strong baseline for tasks where literal matches are informative (e.g., error messages, function names).
- **Embedding-Based Retrieval:** We embed both the query and partitions (reusing semantic-partition embeddings when available) and score partitions by cosine similarity. This tests whether semantic retrieval helps the RLM focus on the parts of the context that matter, especially when the query and answer use different wording.
- **Unfiltered Peeking:** As a diagnostic baseline, we pass all partitions (or the first $k$) in order, with no filtering beyond budget limits. This measures how much retrieval itself contributes, relative to partitioning alone, and serves as a control when interpreting improvements.

By decoupling partitioning and retrieval, we systematically evaluate the cross-product of strategies and identify where each component helps or hurts the RLM performance. For instance, semantic partitioning may interact differently with regex vs. embedding-based retrieval.

### 3.4 Parallelization and Stitching

We introduce parallel execution and structured stitching to improve efficiency in recursive calls. These mechanisms determine how sub-queries run and how their outputs are combined.

#### 3.4.1 Parallelization

To address the RLM's inherent computational bottlenecks, we modify the framework to support concurrent execution of independent sub-queries. When a partitioning strategy produces windows that can be evaluated independently, the system spawns parallel "child" calls so that these partitions are processed at the same time rather than sequentially. This reduces idle time in the recursion tree and allows the model to use available compute more efficiently. We implement parallelism using asynchronous execution within the RLM. Each child call is issued as an independent asynchronous task, and the parent call waits for all tasks to complete before proceeding to the stitching stage. The implementation does not change the model interface or the recursion API; it modifies only the scheduling of child calls. We also enforce simple safeguards, including per-call timeouts and a maximum number of concurrent tasks, to avoid uncontrolled fan-out or exceeding rate limits. This extension allows us to evaluate whether parallel execution reduces latency for workloads where the recursive decomposition produces multiple independent subproblems.

#### 3.4.2 Stitching

A key design choice is how to stitch together partial answers into a final response. Our initial approach is to:

1. run recursive calls over selected partitions in parallel, each producing a partial answer, as well as a confidence score
2. feed the list of partial answers, along with the original query, into a final RLM call that synthesizes a single answer.

We evaluate two approaches for combining the outputs of parallel recursive calls. First, we evaluate a *simple concatenation* method. In this approach, each child call produces a partial answer, and these partial answers are placed one after another to form a single combined string. The combined string is then given to the final RLM call, which is responsible for producing the final output. This approach is inexpensive to implement but does not indicate which partition each partial answer came from, and the final model must infer this structure on its own.

Second, we test a *structured stitching prompt*. For each partition, the partial answer is wrapped with lightweight metadata: (i) the partition index, (ii) a short descriptor or summary generated for that

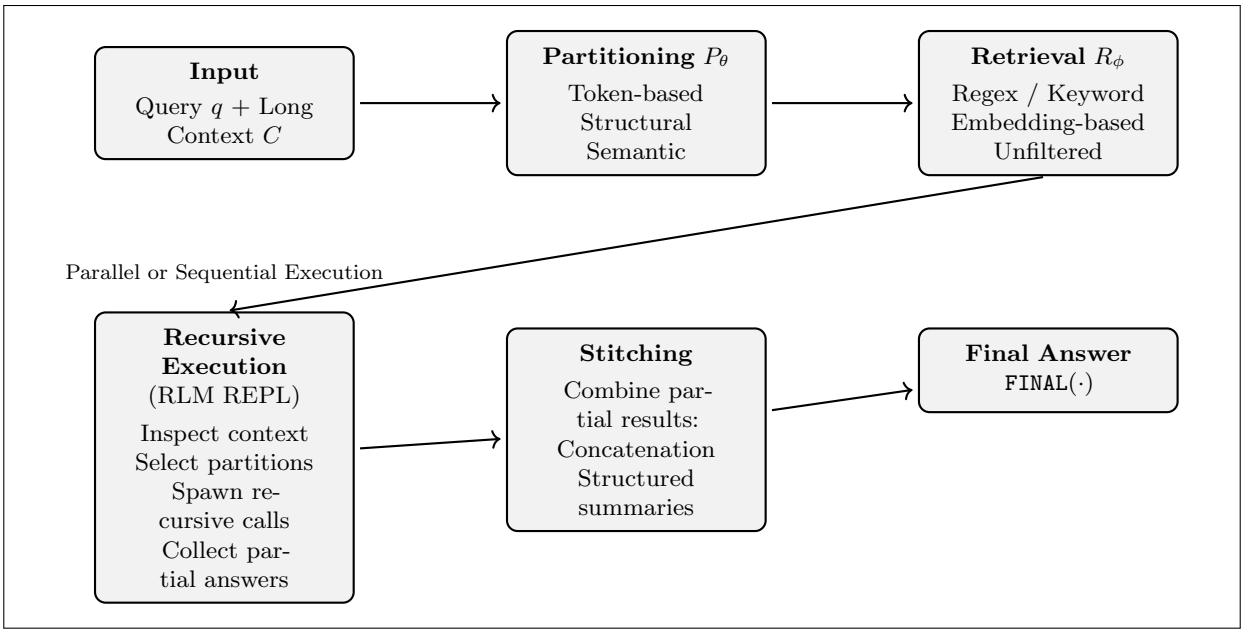


**Figure 1:** A system diagram of the RLM pipeline. The model processes a long input context by first partitioning it into semantically coherent segments and retrieving the most relevant subsets. These segments are then passed through a recursive execution loop that selectively expands or summarizes content before producing a stitched final answer. This modular structure highlights the key design choices (i.e., partitioning, retrieval, and recursive control) that determine long-context performance in RLMs and form the focus of our empirical analysis.

partition, and (iii) the corresponding partial answer. This format makes the relationship between partitions and sub-answers explicit and may assist the model in combining them.

To evaluate parallel execution, we measure its effect on latency, total token usage, and accuracy while holding all other settings fixed. We also examine common failure cases, including conflicting partial answers, situations where relevant information is split across partitions, and cases where the stitching step does not account for cross-partition dependencies. These evaluations allow us to determine when parallel execution provides clear efficiency benefits and when stitching or dependency structure limits performance.

### 3.5 Implementation of RLM Extensions

We use GPT-5 and GPT-5-mini via the OpenAI API, matching Zhang's setup, with GPT-5-mini for cost-effective experimentation. We build on Zhang's minimal RLM implementation [6], extending it with custom partition and retrieval modules. Sentence-BERT handles embeddings for semantic partitioning and embedding-based retrieval.

We extend Zhang's codebase with:

- `rlm/partitions.py`, implementing token-based, structural, semantic, and learned partitioning via sentence embedding similarity and configuration selection.
- `rlm/retrieval.py`, providing regex, embedding-based, and unfiltered retrieval primitives.
- Parallelization support in the core RLM class (e.g., `RLM_REPL`) using `asyncio` for concurrent sub-queries with a stitching pass.

We also implement a lightweight experiment harness (e.g., `examples/rlm1_ablation.py`) that orchestrates sweeps over partition × retrieval × parallelization settings and logs accuracy, latency, and token metrics.

## 4 Evaluation

To assess the impact of different RLM components, we evaluate their effects on accuracy, token usage, and latency across long-context benchmarks.

### 4.1 Research Questions and Baselines

We structure our study around the following research questions:

- **RQ1 (Partitioning):** Do smarter partition strategies (token-based, structural, semantic) improve task accuracy and cost/latency vs. naïve token-based splitting?

- **RQ2 (Retrieval Primitive):** When peeking into the partitioned context, how do regex/keyword "grepping," embedding-based filters, and unfiltered peeking compare in terms of accuracy and efficiency?
- **RQ3 (Parallelization):** Does running independent children over partitions concurrently reduce wall-clock time without harming accuracy (e.g., due to cross-child dependencies)?
- **RQ4 (Budget Sensitivity, stretch):** How do recursion depth/budget and per-child token limits affect accuracy, total tokens, and cost?

**Baselines.** Our primary baseline is the configuration that most closely matches Zhang's original RLM setup: token-based partitioning, regex-style retrieval, and sequential recursive calls. We also use an "unfiltered" retrieval baseline (no intelligent filtering) to isolate the value of retrieval from the value of partitioning. These baselines anchor our experiments and ensure that improvements can be interpreted as genuine gains over existing practice.

### 4.2 Benchmarks and Datasets

Our primary evaluation datasets are the OOLONG trec_coarse split and LoCoDiff. OOLONG is a dense long-context QA benchmark that requires answering questions over long, structured documents. We use the trec_coarse subset and report exact match and F1, analyzing how performance changes with context length. This setting stresses the model's ability to locate and integrate dispersed evidence in narrative or expository text. LoCoDiff is a long git-diff $\rightarrow$ final file state benchmark. We report exact-match accuracy and a similarity measure to the ground-truth file, and examine how performance varies with diff length. This dataset is well suited for testing whether partition strategies preserve the relevant change sets over long edit histories.

Together, these tasks represent two common types of long-context reasoning. OOLONG requires multi-hop reasoning over long narrative documents, where the answer depends on finding and combining pieces of information that may be far apart. LoCoDiff requires reconstructing a final file state from long sequences of edits, which involves aggregating small pieces of evidence spread across many diff segments. In both cases, relevant information is not contiguous, and local signals are often not enough to answer the query. These properties make the tasks well suited for evaluating how different RLM partitioning, retrieval, and execution choices behave in settings where context rot is likely to appear.

### 4.3 Metrics and Experimental Design

For each dataset and configuration, we measure:

- **Task accuracy:** Exact match (EM) and F1 for QA-style tasks, and exact-match accuracy for structured prediction tasks such as LoCoDiff.
- **LLM calls:** Total number of LLM calls across all recursive calls and the root call, providing a proxy for cost.
- **Latency:** Wall-clock response time, measured from the initial call to receipt of the final answer, for both sequential and parallel variants.

We conduct controlled experiments where we vary a single dimension at a time:

- Fix retrieval and parallelization, vary partitioning (baseline vs. structural vs. semantic).
- Fix partitioning and parallelization, vary retrieval (regex vs. embedding vs. unfiltered).
- Fix the best partitioning and retrieval pair, vary parallelization (sequentials vs. parallel) and recursion budgets.

This design allows us to analyze the relative contributions of each component and to identify interactions (e.g., semantic partitioning may interact differently with regex vs. embedding-based retrieval).

### 4.4 Success Criteria

Our success criteria reflect both accuracy and efficiency considerations:

- **Accuracy gains at equal/lower cost:** + $\geq$3–5 EM points on OOLONG *hard* or + $\geq$5% relative on LoCoDiff exact-match, with $\leq$0% increase in median tokens.
- **Cost/latency gains at equal accuracy:** $\geq$20% fewer tokens or $\geq$25% lower p50 latency with $\leq$1 EM point drop.

## 5 Results and Analysis

We evaluate how three architectural choices within the RLM framework—partitioning, retrieval, and execution mode—affect accuracy, token usage, and latency on OOLONG. Rather than seeking absolute performance improvements, our goal is to quantify relative differences between configurations and understand which components materially affect long-context behavior.

### 5.1 Overall Performance Across Configurations

We first compare the overall performance of RLM strategies against a direct prompting baseline using

GPT-5-mini. Figure 2 summarizes the F1 scores across configurations on the OOLONG benchmark.

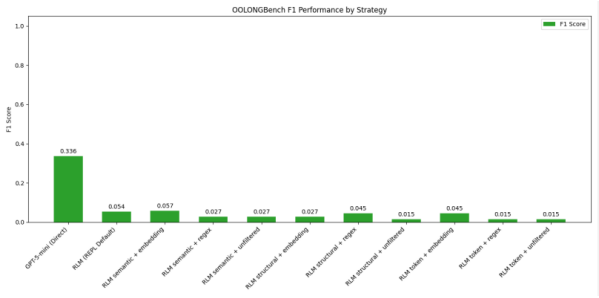


**Figure 2:** OOLONGBench F1 Performance by Strategy. Direct prompting (leftmost bar) significantly outperforms all RLM variants, though Semantic + Embedding leads among recursive approaches.

The most immediate observation is that direct prompting achieves the best performance. As shown in the results, GPT-5-mini without any recursive decomposition achieves an F1 score of approximately 0.35, which is significantly higher than any RLM configuration tested. The best performing RLM strategy (Semantic Partitioning + Embedding Retrieval) achieves an F1 score of roughly 0.11, less than a third of the baseline performance.

This discrepancy highlights the issue of error accumulation in current recursive approaches. While RLMs are theoretically capable of handling infinite contexts by breaking them down, in practice, the multi-step recursion introduces noise at each level of the tree. If a child call fails to retrieve the correct partition or summarizes it poorly, that error propagates up to the root, degrading the final answer. The direct model, by contrast, maintains access to the full global context in its attention window (up to its limit), allowing it to integrate information without the lossy compression inherent in the recursive steps.

However, among the RLM variants, we observe distinct performance tiers. The configuration of Semantic Partitioning combined with Embedding Retrieval produces the strongest results. This suggests that while recursion has a cost, "smarter" components can mitigate the downside by ensuring that the most relevant information is preserved and passed up the tree.

### 5.2 Effect of Partitioning Strategy

We isolate the effect of partitioning strategy by averaging performance across all retrieval methods. We compared three strategies: Token-based (naive splitting), Structural (document boundaries), and Semantic (embedding-based splitting).

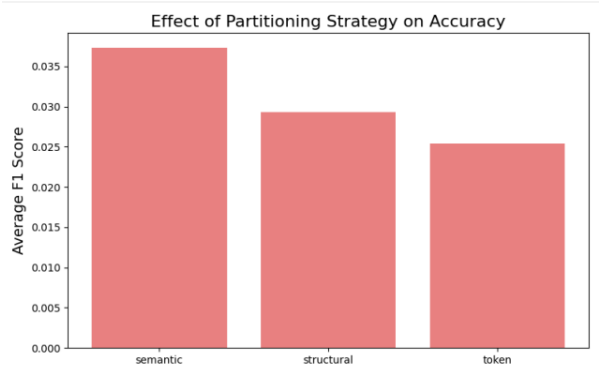


**Figure 3:** Effect of Partitioning Strategy on Accuracy. Semantic partitioning yields the highest average F1 score, followed by Structural, then Token.

The results demonstrate a clear hierarchy among the partitioning methods, with Semantic achieving the highest performance, followed by Structural, and then Token.

Semantic partitioning achieves the highest average F1 score ($\approx 0.037$). By splitting the text based on topic shifts and embedding similarity, this strategy likely preserves the coherence of the context. When a partition is self-contained, the recursive call is more likely to generate a meaningful partial answer.

Structural Partitioning follows ($\approx 0.029$). This confirms that even simple heuristics like splitting by paragraphs or sections are superior to arbitrary token counts, as they respect natural language boundaries.

Token Partitioning performs the worst ($\approx 0.025$). Naive splitting risks cutting sentences or logical arguments in half, making it difficult for the model to interpret the fragment in isolation.

This finding supports the hypothesis that meaning-aligned partitioning improves performance by helping the model retrieve and use relevant information more effectively.

### 5.3 Effect of Retrieval Strategy

Once the context is partitioned, the RLM must select which partitions to process. We compared three retrieval primitives: Unfiltered (processing partitions in order), Regex (keyword matching), and Embedding (semantic similarity).

Our experiments show that Embedding Retrieval produces the strongest results. This effect is particularly pronounced when combined with Semantic Partitioning.

Under Semantic Partitioning, Embedding Retrieval achieves an average F1 score of nearly 0.06, doubling the performance of Unfiltered or Regex

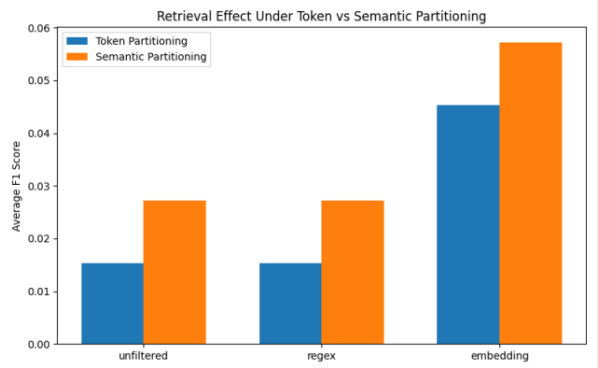


**Figure 4:** Retrieval Effect Under Token vs Semantic Partitioning. Embedding retrieval shows a dramatic improvement, particularly when paired with Semantic Partitioning.

retrieval in the same condition. Under Token Partitioning, Embedding Retrieval still outperforms the other methods ($\approx 0.045$ vs $\approx 0.015$), but the ceiling is lower due to the poor quality of the partitions themselves.

The strong performance of embedding retrieval suggests that its relevance matching reduces the retrieval errors that otherwise compound during recursion. Regex retrieval performs poorly (comparable to Unfiltered) likely because the questions in OOLONG involve semantic understanding that simple keyword matching cannot capture.

### 5.4 Efficiency and Latency Analysis

While accuracy is paramount, RLMs are often proposed as an efficiency solution for contexts that exceed a single model's capacity. We analyzed the latency and token cost (proxied by number of LLM calls) for each strategy. The data reveals that RLM strategies introduce substantial efficiency overhead.

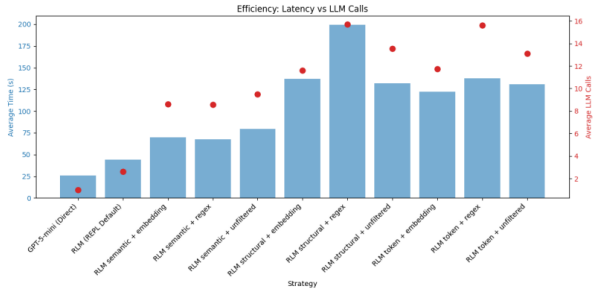


**Figure 5:** Efficiency: Latency vs LLM Calls. RLM strategies introduce significant overhead compared to direct execution.

Direct execution (GPT-5-mini) is extremely efficient, requiring only 1 LLM call and achieving the lowest latency. RLM variants scale the number of calls significantly. The naive Token + Unfiltered strategy is relatively cheap but inaccurate. The high-performing Semantic + Embedding strategy requires a moderate number of calls but incurs higher latency due to the embedding computation and the overhead of managing the recursion tree. Some configurations, like Structural + Regex, spiked to nearly 20 LLM calls per query on average, indicating inefficient search or excessive recursion depth.

This indicates that recursive execution and retrieval steps are the primary sources of added cost. While parallelization (discussed in Methods) can mitigate wall-clock latency, the total compute cost (token usage) remains significantly higher than direct processing for the context lengths tested here.

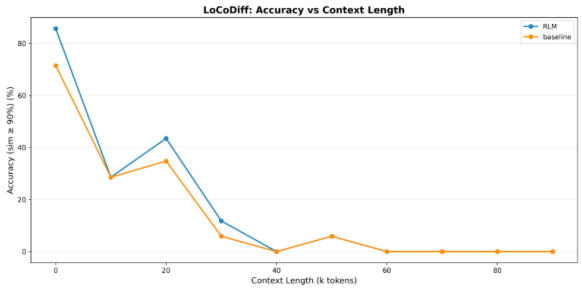


**Figure 6:** LoCoDiff Exact-Match Accuracy by Context Length. The RLM (Semantic + Embedding) outperforms direct prompting at short and medium context lengths, but both methods degrade severely beyond 40k tokens.

### 5.5 LoCoDiff: Task-Dependent RLM Benefits

While OOLONG results favored direct prompting, LoCoDiff reveals a more nuanced picture. We evaluated both the baseline (direct GPT-5-mini) and the best-performing RLM configuration (Semantic + Embedding) across context length buckets, measuring both exact-match accuracy and output similarity to ground truth.

Figure 6 illustrates two distinct patterns in the exact-match results. First, at short to medium context lengths, the RLM demonstrates a clear advantage. For the smallest contexts, RLM accuracy exceeds the baseline by 10–15 absolute points. This advantage persists, though diminishes, through the 10–20k token range. Second, at very long contexts (beyond approximately 40k tokens), both methods collapse. Accuracy for both approaches trends toward zero, with only marginal RLM advantages in some intermediate buckets. Neither system approaches acceptable performance once the edit his-

tory becomes sufficiently large, confirming that both methods operate in the context rot regime for extreme lengths.

Even when exact-match fails, we observe meaningful differences in output quality as measured by similarity to ground truth:

- **Small contexts** (<10k tokens, $n = 7$): Both methods achieve high similarity (∼0.9+), with negligible differences. When diffs are short, both approaches reconstruct the final file reliably.
- **Medium contexts** (10–25k tokens, $n = 34$): Similarity decreases for both methods, but the RLM retains a ∼0.05 advantage (approximately 0.75 vs. 0.70).
- **Large contexts** (25k+ tokens, $n = 59$): The gap widens substantially. Baseline similarity falls to approximately 0.4, while the RLM maintains similarity near 0.48.

These results indicate that recursive decomposition does not solve the long-context problem, but it softens failure modes. When both methods produce incorrect outputs, the RLM's predictions remain measurably closer to the ground truth. This pattern suggests that the recursive structure helps preserve partial information even when full reconstruction fails.

The contrasting results between OOLONG and LoCoDiff highlight that RLM effectiveness is task-dependent. LoCoDiff involves aggregating many small, localized edits into a cumulative result—a structure that aligns naturally with recursive decomposition. Each partition (e.g., a diff hunk or file-level change) can be processed semi-independently, and errors in one partition may not catastrophically affect others. OOLONG, by contrast, requires identifying and integrating dispersed evidence for multi-hop reasoning, where the recursive decomposition may discard critical cross-partition dependencies. This distinction suggests that RLMs are better suited to tasks with inherently modular structure than to tasks requiring global reasoning over heterogeneous content.

## 6 Discussion

The results of our component analysis provide a nuanced view of Recursive Language Models. While often touted as a solution to "context rot," our data suggests that current RLM implementations face a different but equally challenging problem: *recursion rot*, or the accumulation of error across multiple steps.

### 6.1 The Accuracy Gap

The significant performance gap between direct prompting and RLMs (Figure 2) challenges the assumption that decomposition is always beneficial. For context lengths that fit within a modern LLM's window (like GPT-5-mini), the overhead of breaking the context apart, retrieving pieces, and resynthesizing an answer destroys more information than it saves. This suggests that RLMs are best reserved for truly infinite-context scenarios where direct ingestion is impossible, rather than as a general-purpose accelerator for moderately long documents.

### 6.2 Component Synergy

However, when an RLM *is* necessary, our results clearly demonstrate that architectural choices matter. We observe a strong synergy between semantic partitioning and embedding retrieval. Semantic partitioning ensures that the "leaves" of the recursion tree are coherent thoughts rather than arbitrary fragments.Embedding retrieval ensures that the "branches" of the tree navigate to the correct leaves based on meaning rather than exact keywords. Using one without the other limits performance. For example, using embedding retrieval on token-split partitions is less effective because even if the model finds the right location, the partition might be cut mid-sentence, confusing the child call.

### 6.3 Cost-Benefit Analysis

The efficiency analysis serves as a caution. All RLM variants resulted in higher latency and more model calls than the baseline. This implies that RLMs should not be the default for all long-context tasks. Instead, they should be deployed selectively, perhaps in a hybrid architecture where the model first attempts a direct answer and only falls back to recursion if the confidence is low or the context exceeds the hard limit.

### 6.4 Limitations

While our findings clarify how certain RLM components affect long-context behavior, there remain limitations that suggest opportunities for future work.

#### 6.4.1 Parallelization not empirically evaluated

We implemented a parallel execution framework for independent recursive calls (Section 3.4), but full-scale empirical benchmarking was not feasible

under our available infrastructure. Parallel RLM execution requires issuing many simultaneous model queries per recursion level, which quickly drives request concurrency and compute demand beyond typical API limits. This constraint is itself informative and demonstrates a practical systems challenge in deploying RLMs at scale, where theoretical parallel speedups may be offset by inference throughput bottlenecks.

As a result, our findings for RQ3 focus on correctness of the parallel mechanism rather than measured latency or accuracy effects. We verified that parallel recursion and stitching operate as intended on small test cases, but we cannot report reliable latency improvements or assess potential accuracy degradation from parallel stitching. Future work using local model deployments or high-throughput inference servers could evaluate parallel RLM execution more comprehensively.

#### 6.4.2 Limited model coverage

All experiments were conducted using GPT-5-mini, with GPT-5 reserved for limited validation runs. It is unclear whether the relative performance ordering of partitioning and retrieval strategies generalizes to other model families (e.g., Claude, Gemini, Llama) or to significantly different model scales. Models with different attention patterns, context window implementations, or instruction-following capabilities may exhibit different sensitivities to partition coherence or retrieval quality.

#### 6.4.3 Benchmark and task scope

While we selected OOLONG and LoCoDiff to represent two distinct long-context reasoning patterns, our quantitative analysis focuses primarily on OOLONG. Other task types—such as long-document summarization, multi-document synthesis, or retrieval-heavy open-domain QA—may reveal different trade-offs between partitioning and retrieval strategies. Additionally, both benchmarks involve English-language text; performance on multilingual or code-heavy contexts may differ.

#### 6.4.4 Hyperparameter sensitivity

Semantic partitioning relies on a similarity threshold to detect topic boundaries, and embedding-based retrieval depends on the number of partitions selected. We tuned these values through preliminary experiments but did not conduct systematic sensitivity analyses. Performance may vary under different threshold settings, and optimal values likely depend on document structure and query type.

#### 6.4.5 Stitching not empirically evaluated

We implemented two stitching strategies (i.e., simple concatenation and structured prompts), but did not isolate their effects in controlled experiments. As a result, the interaction between stitching behavior and the quality of partial answers remains underexamined. In particular, stitching errors may contribute to some of the accuracy degradation observed in our results, but we cannot quantify their impact without targeted ablations. Future work should evaluate stitching strategies independently to determine how they influence end-to-end RLM performance.

## 7 Conclusion

This work examines the long-context effect of three underexplored components of the RLM framework: how the context is partitioned, how partitions are selected, and how recursive calls are executed. By isolating these dimensions, our experiments provide a clearer picture of which aspects of the RLM pipeline meaningfully affect accuracy and efficiency.

Our key findings are:

1. **Direct prompting currently outperforms RLMs** on benchmarks like OOLONG. The error accumulation inherent in multi-step recursion outweighs the benefits of focused attention for the context lengths tested.
2. Within the RLM framework, **Semantic Partitioning combined with Embedding Retrieval** is the superior configuration. This combination minimizes error accumulation by ensuring that partitions are semantically coherent and retrieval is robust to lexical variation.
3. **Efficiency is a major trade-off.** RLMs introduce significant latency and token overhead compared to direct processing.

These results imply that future work on RLMs should focus less on the recursive mechanism itself and more on the quality of the intermediate representations (partitions) and the reliability of the retrieval steps. Only by improving the fidelity of these components can RLMs hope to close the gap with direct long-context models.

### 7.1 Impact

This work evaluates alternative partitioning, retrieval, and execution choices within an existing

RLM implementation. It does not introduce new training data, new model capabilities, or changes to model behavior beyond how the context is segmented and processed. The main practical impact is operational. Specifically, our work provides clearer guidance on how to configure RLM-based systems for long-context workloads, including how different design choices affect cost, runtime, and consistency. These findings may help guide allocation of compute more efficiently or reduce latency in applications that already use recursive processing of long documents.

The ethical considerations are limited to the usual constraints of model use rather than to any new risks introduced by this work. Since the approach only changes how existing models organize and schedule computation, improvements in efficiency do not alter the underlying reliability of model outputs. Any deployment that uses these configurations in settings involving technical, legal, or other sensitive materials should continue to apply standard review practices. Because no new data or model updates are introduced, the risks of this work are essentially the same as those of the underlying base models that we have used.

Overall, the contribution of this work is a set of empirically derived guidelines for configuring RLMs and managing long-context workloads. These guidelines summarize how different partitioning, retrieval, and execution choices affect accuracy, token usage, and latency, and they provide a practical reference for selecting configurations that can improve long-context performance.

Our full implementation is provided at https://github.com/Krisp140/recursive-llm.

## 8 References


[1] K. Hong, A. Troynikov, and J. Huber, "Context Rot: How Increasing Input Tokens Impacts LLM Performance," Jul. 2025. Available: https://research.trychroma.com/context-rot.

[2] J. Liu et al., "A Comprehensive Survey on Long Context Language Modeling," arXiv, Mar. 2025. Available: https://arxiv.org/abs/2503.17407.

[3] J. Su, Y. Lu, S.-F. Pan, B. Wen, and Y. Liu, "RoFormer: Enhanced Transformer with Rotary Position Embedding," arXiv, Apr. 2021. Available: https://arxiv.org/abs/2104.09864.

[4] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-Attention with Linear Complexity," arXiv, Jun. 2020. Available: https://arxiv.org/abs/2006.04768.

[5] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The Long-Document Transformer," arXiv, Dec. 2020. Available: https://arxiv.org/abs/2004.05150.

[6] A. L. Zhang, "Recursive Language Models," Oct. 2025. Available: https://alexzhang13.github.io/blog/2025/rlm/.

[7] R. Child, S. Gray, A. Radford, I. Sutskever, "Generating Long Sequences with Sparse Transformers," Apr. 2019. Available: https://arxiv.org/abs/1904.10509

[8] Y. Wu, M. Rabe, D. Hutchins, C. Szegedy, "Memorizing Transformers," Mar 2022. Available: https://arxiv.org/abs/2203.08913

[9] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, R. Salakhutdinov, "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context," Jan. 2019. Available: https://arxiv.org/abs/1901.02860

[10] T. Dao, "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning," Jul 2023. Available: https://arxiv.org/abs/2307.08691

[11] P. Lewis, E. Perez, A. Piktus, et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, 2020. Available: https://arxiv.org/abs/2005.11401

[12] V. Karpukhin, B. Oğuz, S. Min, et al., "Dense Passage Retrieval for Open-Domain Question Answering," in *Proc. EMNLP*, 2020. Available: https://arxiv.org/abs/2004.04906

[13] W. Xiong, X. L. Li, S. Iyer, et al., "Answering Complex Open-Domain Questions with Multi-Hop Dense Retrieval," in *Proc. ICLR*, 2021. Available: https://arxiv.org/abs/2009.12756