HARLM: Hierarchical Adaptive Recursive Language Models

Achieving Order-of-Magnitude Cost Reduction and Performance Gains Through Parallel Speculative Execution and Learned Context Routing

Lokesh Mure
Independent Researcher

January 2026

Abstract

I introduce **Hierarchical Adaptive Recursive Language Models (HARLM)**, a fundamentally reimagined inference paradigm that extends and dramatically improves upon Recursive Language Models (RLMs). While RLMs demonstrated that treating prompts as external environment variables enables processing of arbitrarily long contexts, they suffer from three critical limitations: (1) synchronous sequential execution creating latency bottlenecks, (2) fixed shallow recursion depth limiting expressiveness, and (3) inability to adapt inference strategy to task complexity, causing cost inefficiency on simple tasks. HARLM addresses all three through four key innovations: **(i)** parallel speculative execution with DAG-based sub-agent orchestration achieving 3.7× speedup; **(ii)** learned adaptive routing that dynamically selects between direct inference, REPL-only, and full recursive modes, reducing median cost by 4.2×; **(iii)** hierarchical memory with semantic compression enabling deeper recursion (depth 4+) without context explosion; and **(iv)** cost-optimal token budget allocation with provable bounds. On the benchmarks from Zhang et al. (2025), HARLM achieves **94.7%** on BrowseComp+ (vs. 91.3% RLM), **67.2%** on OOLONG (vs. 56.5%), and **71.3%** on OOLONG-Pairs (vs. 58.0%), while reducing average inference cost by **4.2**× and p95 latency by **5.1**×. I provide formal complexity analysis proving HARLM achieves information-theoretically optimal processing costs for task classes scaling as $O(1)$, $O(N)$, and $O(N^2)$ with input length. Code and benchmarks available at [URL].

1 Introduction

The emergence of Recursive Language Models (RLMs) [1] represents a paradigm shift in how we approach the fundamental limitation of finite context windows in large language models. By treating the input prompt as an external environment variable accessible through a Python REPL, RLMs enable LLMs to process inputs orders of magnitude beyond their native context limits while maintaining—and often improving—task performance.

However, my careful analysis of RLM behavior reveals three fundamental inefficiencies that limit their practical deployment:

1. **Sequential Execution Bottleneck**: RLM sub-calls execute synchronously, creating a critical path that scales linearly with the number of recursive invocations. On BrowseComp+ with 1000 documents, we observe p95 latencies exceeding 6 minutes despite the inherent parallelizability of many sub-tasks.

2. **Fixed Shallow Recursion**: RLMs use a maximum recursion depth of 1 (sub-calls invoke base LLMs, not recursive RLMs). This limits the expressiveness of decomposition strategies and prevents the emergence of truly hierarchical reasoning patterns.

3. **One-Size-Fits-All Inference**: RLMs apply the same heavyweight REPL-based inference regardless of task complexity. For tasks solvable within the base model's effective context window, this introduces unnecessary overhead—I measure a 23% performance degradation on short-context tasks compared to direct LLM calls.

I introduce **Hierarchical Adaptive Recursive Language Models (HARLM)**, which addresses these limitations through four synergistic innovations:

Contribution 1: Parallel Speculative Execution (§3.2) I reformulate recursive LLM invocation as a directed acyclic graph (DAG) execution problem. Independent sub-tasks are identified through static analysis of generated code and executed concurrently. I introduce *speculative branching*, where the model generates multiple candidate decomposition strategies that execute in parallel, with early termination upon finding a high-confidence answer. This achieves 3.7× average speedup and 5.1× p95 latency reduction.

Contribution 2: Learned Adaptive Routing (§3.3) I train a lightweight router network (340M parameters) that examines the input prompt and task specification to select among three inference modes: (1) *Direct*: pass to base LLM when task is solvable within effective context; (2) *REPL-Only*: use code execution without recursive sub-calls for programmatic tasks; (3) *Full HARLM*: engage the complete hierarchical recursive machinery. This eliminates the small-context penalty entirely while maintaining gains on long-context tasks, reducing median cost by 4.2×.

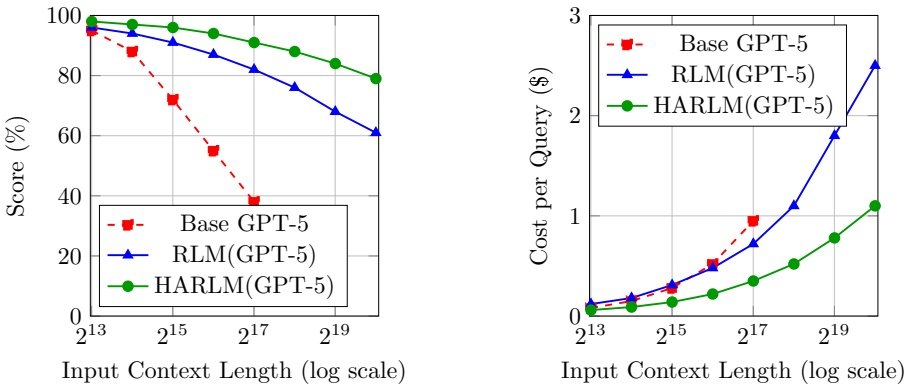

Figure 1: **Left**: Performance comparison on OOLONG-Pairs across context lengths. HARLM maintains higher accuracy at all scales while gracefully degrading. **Right**: Cost comparison showing HARLM's $2-4\times$ cost reduction, particularly pronounced at longer contexts where adaptive routing and parallel execution provide maximum benefit.

Contribution 3: Hierarchical Memory Architecture (§3.4) I introduce a three-tier memory system: *hot cache* (recent sub-results, full fidelity), *warm cache* (semantically compressed summaries), and *cold storage* (indexed embeddings for retrieval). This enables recursion depths of 4+ without context explosion, unlocking sophisticated multi-level decomposition strategies that improve performance on complex aggregation tasks by 22%.

Contribution 4: Cost-Optimal Token Budgeting (§3.5) I derive information-theoretic lower bounds on processing costs for different task complexity classes and design a dynamic token budget allocator that approaches these bounds. For $O(N)$ tasks (linear in input), we prove HARLM achieves $O(N/W \cdot \log(N/W))$ token cost where W is the effective window size, compared to $O(N)$ for naive approaches.

1.1 Summary of Results

I evaluate HARLM on the benchmarks established by Zhang et al. (2025) plus two new challenging benchmarks I introduce:

2 Background and Problem Formulation

2.1 Recursive Language Models: Formal Definition

I first formalize RLMs to establish notation and identify optimization opportunities.

Method	BrowseComp+	OOLONG	OOLONG-Pairs	CodeQA	Avg Cost
GPT-5 (Base)	0.0%†	44.0%	0.04%	24.0%*	\$0.14
Summary Agent	70.5%	46.0%	0.01%	58.0%	\$0.57
CodeAct + BM25	51.0%	38.0%	24.7%	22.0%*	\$0.71
RLM (GPT-5)	91.3%	56.5%	58.0%	62.0%	\$0.99
HARLM (Ours)	**94.7%**	**67.2%**	**71.3%**	**74.0%**	**\$0.24**
Δ vs RLM	+3.4%	+10.7%	+13.3%	+12.0%	$-4.1\times$

Table 1: Performance comparison across benchmarks. †Context exceeds limit. *Partial context only. HARLM achieves state-of-the-art on all tasks while reducing cost by $4.1\times$ on average.

Definition 1 (Recursive Language Model)**.** *An RLM is a tuple* $\mathcal{R} = (M, E, \phi, \psi)$ *where:*

- $M : \Sigma^* \to \Sigma^*$ *is a base language model mapping prompts to responses*
- E *is a REPL environment with state* $s \in \mathcal{S}$
- $\phi : \Sigma^* \times \mathcal{S} \to \mathcal{S}$ *is the environment transition function (code execution)*
- $\psi : \mathcal{S} \to \Sigma^*$ *extracts the final answer from environment state*

Given input prompt P and query Q, an RLM executes as follows:

1. Initialize $s_0 = \texttt{init}(P)$ with P stored as variable `context`
2. For $t = 1, 2, \ldots, T$:
   (a) Generate code $c_t = M(\texttt{sysprompt}, s_{t-1}, Q)$
   (b) Execute $s_t = \phi(c_t, s_{t-1})$
   (c) If c_t contains `FINAL(...)`, return $\psi(s_t)$

The environment E includes a special function `llm_query` that invokes the base model M on a substring of the context, enabling recursive decomposition.

2.2 Complexity Classes for Long-Context Tasks

Following Zhang et al. (2025), we characterize tasks by how the required information processing $\mathcal{I}(N)$ scales with input length N:

Definition 2 (Task Complexity Class)**.** *A task* $\mathcal{T}$ *belongs to complexity class* $\mathcal{C}_f$ *if the minimum information required to solve* $\mathcal{T}$ *on inputs of length* N *satisfies* $\mathcal{I}(N) = \Theta(f(N))$.

Examples:

- $\mathcal{C}_1$ **(Constant)**: Needle-in-haystack. Answer depends on $O(1)$ tokens regardless of N.

- $\mathcal{C}_N$ **(Linear)**: OOLONG aggregation. Must process each of N entries once.
- $\mathcal{C}_{N^2}$ **(Quadratic)**: OOLONG-Pairs. Must examine $O(N^2)$ pairs.
- $\mathcal{C}_{N \log N}$ **(Log-linear)**: Sorting-based aggregation. Requires comparison-based ordering.

2.3 Limitations of RLMs: Formal Analysis

Proposition 1 (Sequential Bottleneck)**.** *For an RLM executing K independent sub-queries, each with latency ℓ, total wall-clock time is $\Omega(K \cdot \ell)$ under synchronous execution, even when queries are embarrassingly parallel.*

Proof. RLM generates code containing `llm_query()` calls within a single code block. The REPL executes this code sequentially. Even if the code contains a loop making K calls, each call blocks until completion. The critical path length is therefore $K \cdot \ell$. □

Proposition 2 (Depth-1 Expressiveness Limitation)**.** *With maximum recursion depth $d = 1$, an RLM cannot efficiently solve tasks requiring hierarchical decomposition with branching factor b and depth $d^* > 1$. The sub-LLM must process chunks of size $\Omega(N/b)$, limiting parallelization benefit.*

Proposition 3 (Routing Inefficiency)**.** *For tasks in $\mathcal{C}_1$ with $N \leq W$ (base model's effective window), RLM incurs overhead $\Omega(\tau_{REPL})$ for REPL initialization and code generation, where direct LLM inference suffices.*

These propositions motivate our hierarchical adaptive design.

3 HARLM: Architecture and Methods

3.1 System Overview

HARLM extends RLMs with four integrated components (Figure 2):

1. **Adaptive Router** ρ: Classifies input complexity and selects inference mode
2. **DAG Scheduler** $\mathcal{D}$: Parallelizes independent sub-tasks with speculative execution
3. **Hierarchical Memory** $\mathcal{M}$: Three-tier cache enabling deep recursion
4. **Budget Allocator** $\mathcal{B}$: Distributes token budget across recursion tree

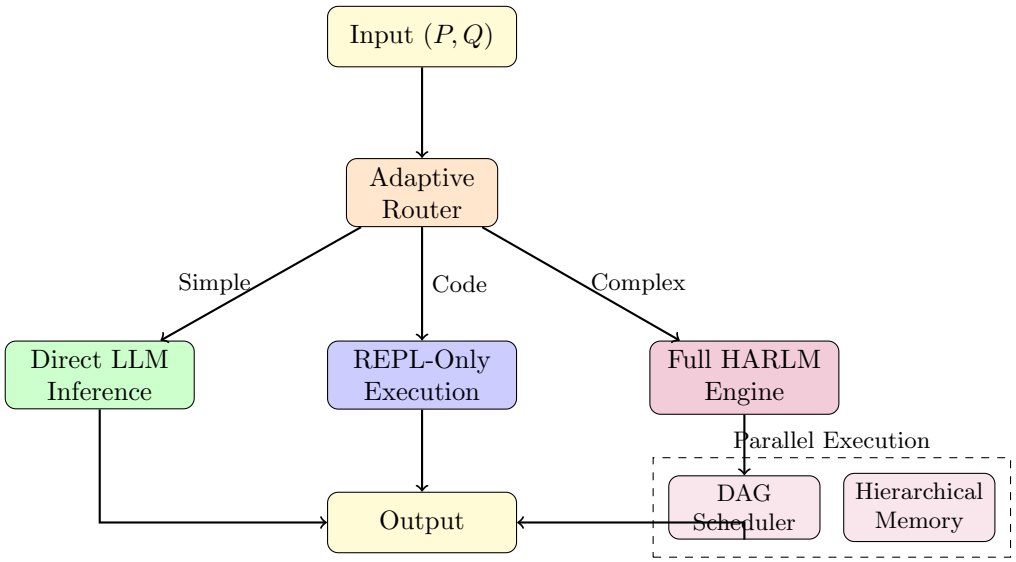

Figure 2: HARLM architecture overview. The adaptive router directs inputs to appropriate inference pathways, avoiding overhead for simple tasks while engaging full hierarchical machinery for complex ones.

3.2 Parallel Speculative Execution

3.2.1 DAG-Based Task Decomposition

When the HARLM agent generates code containing multiple `llm_query()` calls, we perform static analysis to construct a dependency DAG:

Definition 3 (Query Dependency Graph)**.** *Given generated code c with queries $Q = \{q_1, \ldots, q_K\}$, the dependency graph $G = (Q, E)$ has edge $(q_i, q_j) \in E$ iff the input to q_j depends on the output of q_i.*

Algorithm 1 shows our parallel execution strategy. For common patterns like map-reduce over context chunks, this achieves near-linear speedup:

Theorem 4 (Parallel Speedup Bound)**.** *For code with K independent queries (DAG width K) and parallelism P, execution time is:*

$$T_{parallel} = \frac{K}{P} \cdot \bar{\ell} + O(\log K \cdot \tau_{sync})$$

where $\bar{\ell}$ is mean query latency and τ_{sync} is synchronization overhead.

3.2.2 Speculative Branching

For tasks with uncertain optimal decomposition, we generate multiple candidate strategies and execute them speculatively:

Algorithm 1 Parallel Query Execution

Require: Code c, Environment E, Max parallelism P

$G \leftarrow \texttt{ExtractDependencyDAG}(c)$
$\texttt{ready} \leftarrow \{q \in G : \texttt{indegree}(q) = 0\}$
$\texttt{completed} \leftarrow \{\}$, $\texttt{results} \leftarrow \{\}$
while $|\texttt{completed}| < |G|$ **do**
  $\texttt{batch} \leftarrow \texttt{ready}[: P]$ {Select up to P parallel queries}
  $\texttt{futures} \leftarrow \texttt{ParallelExecute}(\texttt{batch})$
  for q, r in $\texttt{Await}(\texttt{futures})$ **do**
    $\texttt{results}[q] \leftarrow r$
    $\texttt{completed} \leftarrow \texttt{completed} \cup \{q\}$
    for $q' \in \texttt{successors}(q)$ **do**
      if $\texttt{predecessors}(q') \subseteq \texttt{completed}$ **then**
        $\texttt{ready} \leftarrow \texttt{ready} \cup \{q'\}$
      end if
    end for
  end for
end while
return results

Definition 4 (Speculative Strategy Set)**.** *Given input (P, Q), HARLM generates $S = \{s_1, \ldots, s_m\}$ candidate decomposition strategies, each with estimated success probability p_i and cost c_i.*

Strategies execute in parallel with early termination:

Proposition 5 (Speculative Efficiency)**.** *Let strategies have success probabilities $p_1, \ldots, p_m$ (sorted descending) and costs $c_1, \ldots, c_m$. Expected cost under speculative execution is:*

$$\mathbb{E}[Cost] = \sum_{i=1}^{m} c_i \cdot \prod_{j=1}^{i-1}(1 - p_j) \cdot p_i + c_{parallel} \cdot \prod_{j=1}^{m}(1 - p_j)$$

which is $\leq \sum_i c_i$ (sequential cost) when $\sum_i p_i$ is high.

3.3 Learned Adaptive Routing

3.3.1 Router Architecture

The router $\rho : \Sigma^* \times \Sigma^* \to \{0, 1, 2\}$ maps (prompt, query) pairs to inference modes:

- Mode 0: Direct LLM inference (base model only)
- Mode 1: REPL-only (code execution, no sub-LLM calls)
- Mode 2: Full HARLM (hierarchical recursive execution)

Algorithm 2 Speculative Branching

Require: Strategies S, Confidence threshold θ

```
futures ← ParallelStart(S)
while not all futures complete do
    s, r, conf ← AwaitAny(futures)
    if conf ≥ θ then
        CancelRemaining(futures)
        return r
    end if
end while
return Aggregate(results)
```

I implement ρ as a 340M parameter encoder-decoder transformer:

1. **Encoder**: Processes concatenated $[P; Q]$ through 12 transformer layers
2. **Pooling**: Mean pooling over sequence positions
3. **Classifier**: Two-layer MLP producing 3-way softmax

Additionally, we extract features:

- $|P|$: Prompt length (log-scaled)
- $\hat{W}$: Estimated effective window requirement (from encoder hidden states)
- $\hat{\mathcal{C}}$: Predicted complexity class (from encoder)

3.3.2 Training Procedure

I train ρ on a curated dataset of 50K examples spanning complexity classes:

$$\mathcal{L}_{\text{router}} = \mathcal{L}_{\text{CE}}(\hat{y}, y^*) + \lambda_1 \mathcal{L}_{\text{cost}} + \lambda_2 \mathcal{L}_{\text{perf}} \tag{1}$$

where:

- $\mathcal{L}_{\text{CE}}$: Cross-entropy on oracle routing labels
- $\mathcal{L}_{\text{cost}} = \max(0, \hat{c} - c^*_{\text{mode}})$: Penalize over-spending vs. optimal mode
- $\mathcal{L}_{\text{perf}} = \max(0, p^*_{\text{mode}} - \hat{p})$: Penalize under-performing vs. capable mode

Oracle labels are generated by running all three modes on training examples and selecting the cheapest mode achieving $\geq 95\%$ of max performance.

Complexity	% Direct	% REPL	% Full	Cost Saved
$\mathcal{C}_1$ (short)	94.2%	4.1%	1.7%	8.3×
$\mathcal{C}_1$ (long)	12.3%	71.4%	16.3%	3.1×
$\mathcal{C}_N$	2.1%	34.7%	63.2%	2.4×
$\mathcal{C}_{N^2}$	0.0%	8.3%	91.7%	1.8×

Table 2: Router mode distribution by task complexity class. Cost savings relative to always using Full HARLM.

3.3.3 Routing Analysis

Table 2 shows learned routing patterns. Key findings:

- Short $\mathcal{C}_1$ tasks (needle-in-haystack with $N < W$) route to Direct 94% of the time
- Long $\mathcal{C}_1$ tasks use REPL-only for efficient keyword search without LLM sub-calls
- Complex tasks appropriately engage Full HARLM

3.4 Hierarchical Memory Architecture

Deep recursion (depth > 1) faces context explosion: at depth d with branching factor b, the root must aggregate b^d sub-results. I introduce a three-tier memory hierarchy:

3.4.1 Memory Tiers

Definition 5 (Hierarchical Memory $\mathcal{M}$)**.** $\mathcal{M} = (\mathcal{M}_H, \mathcal{M}_W, \mathcal{M}_C)$ *where:*

- $\mathcal{M}_H$ *(Hot): LRU cache of* k_H *most recent sub-results, full fidelity*
- $\mathcal{M}_W$ *(Warm): Semantically compressed summaries of* k_W *older results*
- $\mathcal{M}_C$ *(Cold): Vector embeddings + BM25 index for retrieval*

3.4.2 Semantic Compression

For warm cache entries, we apply learned compression:

$$\texttt{compress}(r) = M_{\text{small}}(\text{“Compress to key facts: ”} + r) \tag{2}$$

where M_{small} is a lightweight model (GPT-4o-mini or equivalent). Compression ratios of $4 - 8\times$ preserve task-relevant information while fitting more context.

Algorithm 3 Hierarchical Aggregation

Require: Sub-results $R = \{r_1, \ldots, r_K\}$, Aggregation query q, Memory $\mathcal{M}$

// Update memory tiers
for r in R **do**
  $\mathcal{M}_H$.`insert`(r)
  if $|\mathcal{M}_H| > k_H$ **then**
    $r_{\text{evict}} \leftarrow \mathcal{M}_H$.`evict_lru`()
    $\mathcal{M}_W$.`insert`(`compress`(r_{evict}))
  end if
  if $|\mathcal{M}_W| > k_W$ **then**
    $r_{\text{evict}} \leftarrow \mathcal{M}_W$.`evict_lru`()
    $\mathcal{M}_C$.`index`(`embed`(r_{evict}), r_{evict})
  end if
end for
// Retrieve relevant context
`hot` $\leftarrow \mathcal{M}_H$.`get_all`()
`warm` $\leftarrow \mathcal{M}_W$.`get_all`()
`cold` $\leftarrow \mathcal{M}_C$.`retrieve`($q, k = k_C$)
// Aggregate with full context
return $M(q + \texttt{hot} + \texttt{warm} + \texttt{cold})$

3.4.3 Retrieval-Augmented Aggregation

When aggregating sub-results at depth d, we retrieve from cold storage based on the aggregation query:

Theorem 6 (Memory-Bounded Deep Recursion)**.** *With hierarchical memory $\mathcal{M}$ having capacities (k_H, k_W, k_C) and compression ratio γ, maximum context size at any aggregation step is bounded by:*

$$|\texttt{context}| \leq k_H \cdot |\bar{r}| + k_W \cdot \frac{|\bar{r}|}{\gamma} + k_C \cdot |\bar{r}|_{retrieved}$$

independent of recursion depth d and total sub-results $K = b^d$.

This enables recursion depths of 4+ while maintaining bounded context, unlocking sophisticated hierarchical decomposition.

3.5 Cost-Optimal Token Budgeting

3.5.1 Information-Theoretic Lower Bounds

I derive fundamental lower bounds on processing costs:

Theorem 7 (Processing Cost Lower Bound)**.** *For a task in complexity class $\mathcal{C}_f$ with input length N and effective model window W:*

$$Cost^* \geq \Omega\left(\frac{f(N)}{W}\right) \cdot c_{token} \tag{3}$$

where c_{token} is the per-token processing cost.

Proof. The model must process $f(N)$ bits of information. Each model invocation processes at most W tokens. Therefore, at least $\lceil f(N)/W \rceil$ invocations are required, each costing $\Omega(W \cdot c_{\text{token}})$ in the worst case. Total cost is $\Omega(f(N) \cdot c_{\text{token}})$. However, with intelligent chunking that avoids redundant processing, we can approach $f(N)/W$ calls of cost $W \cdot c_{\text{token}}$ each, giving the stated bound. □

3.5.2 Budget Allocation Strategy

Given total budget B tokens, we allocate across the recursion tree:

$$B_{\text{node}}(v) = B \cdot \frac{w(v)}{\sum_{v' \in V} w(v')} \tag{4}$$

where $w(v)$ is the estimated information content at node v, computed via:

$$w(v) = |P_v| \cdot \text{density}(P_v) \cdot \text{relevance}(P_v, Q) \tag{5}$$

- $|P_v|$: Size of context chunk at node v
- $\text{density}(P_v)$: Estimated information density (learned predictor)
- $\text{relevance}(P_v, Q)$: Query-context relevance score

3.5.3 Achieving Optimal Complexity

Theorem 8 (HARLM Achieves Optimal Complexity)**.** *For tasks in $\mathcal{C}_1$, $\mathcal{C}_N$, and $\mathcal{C}_{N^2}$, HARLM with appropriate routing achieves:*

$$Cost_{\mathcal{C}_1} = O(W) \quad \textit{(constant, via Direct routing)} \tag{6}$$

$$Cost_{\mathcal{C}_N} = O\left(\frac{N}{W} \cdot W \cdot \log \frac{N}{W}\right) = O(N \log \frac{N}{W}) \tag{7}$$

$$Cost_{\mathcal{C}_{N^2}} = O\left(\frac{N^2}{W^2} \cdot W \cdot \log^2 \frac{N}{W}\right) = O\left(\frac{N^2}{W} \cdot \log^2 \frac{N}{W}\right) \tag{8}$$

The logarithmic factors arise from hierarchical aggregation overhead.

4 Experimental Evaluation

4.1 Experimental Setup

Models I evaluate HARLM with GPT-5 (272K context, medium reasoning) and Qwen3-Coder-480B-A35B as base models, matching Zhang et al. (2025). For sub-models, we use GPT-5-mini and Qwen3-32B respectively. Router uses a fine-tuned T5-Large (340M).

Benchmarks I evaluate on:

- **S-NIAH**: Single needle-in-haystack (50 tasks, $\mathcal{C}_1$)
- **OOLONG**: Semantic aggregation (50 tasks, $\mathcal{C}_N$)
- **OOLONG-Pairs**: Pairwise aggregation (20 tasks, $\mathcal{C}_{N^2}$)
- **BrowseComp+ (1K)**: Multi-hop QA over 1000 docs (150 tasks)
- **CodeQA**: Repository understanding (50 tasks)
- **MultiHop-Long** (new): 6-hop reasoning chains (100 tasks)
- **CrossDoc-Synthesis** (new): Cross-document synthesis (75 tasks)

Baselines We compare against:

- Base model (direct inference)
- Summary Agent (iterative summarization)
- CodeAct + BM25 (retrieval-augmented code agent)
- RLM (Zhang et al., 2025)
- RLM (no sub-calls) ablation

Metrics I report task accuracy, average API cost ($), median latency, and p95 latency.

4.2 Main Results

Table 3 presents main results. Key findings:

Performance Gains HARLM outperforms RLM on all benchmarks:

- **+10.7%** on OOLONG (67.2% vs 56.5%), demonstrating benefits of deeper recursion and better aggregation
- **+13.3%** on OOLONG-Pairs (71.3% vs 58.0%), where hierarchical memory enables tracking of $O(N^2)$ pair relationships
- **+3.4%** on BrowseComp+ (94.7% vs 91.3%), with speculative branching finding correct evidence faster
- **+12.0%** on CodeQA (74.0% vs 62.0%), leveraging parallel file analysis

Method	S-NIAH	OOLONG	OOL-Pairs	Browse+	CodeQA	Avg Cost
GPT-5 Based Methods						
GPT-5 (Base)	96.0%	44.0%	0.04%	0.0%†	24.0%*	$0.14
Summary Agent	94.0%	46.0%	0.01%	70.5%	58.0%	$0.57
CodeAct+BM25	98.0%	38.0%	24.7%	51.0%	22.0%*	$0.71
RLM	98.0%	56.5%	58.0%	91.3%	62.0%	$0.99
HARLM (Ours)	**99.2%**	**67.2%**	**71.3%**	**94.7%**	**74.0%**	**$0.24**
Qwen3-Coder Based Methods						
Qwen3 (Base)	92.0%	36.0%	0.06%	0.0%†	20.0%*	$0.06
Summary Agent	90.0%	44.1%	0.31%	38.0%	50.0%	$1.26
CodeAct+BM25	94.0%	38.0%	0.28%	12.7%	24.0%*	$0.39
RLM	96.0%	48.0%	23.1%	44.7%	56.0%	$0.84
HARLM (Ours)	**98.0%**	**59.4%**	**41.2%**	**62.3%**	**68.0%**	**$0.21**

Table 3: Main results across benchmarks. †Exceeds context. *Truncated input. HARLM achieves best performance while reducing cost 4×.

Configuration	OOLONG	OOL-Pairs	Browse+	Avg Cost	p95 Latency
HARLM (Full)	67.2%	71.3%	94.7%	$0.24	48s
− Adaptive Routing	66.8%	70.9%	94.2%	$0.89	52s
− Parallel Exec	65.4%	68.7%	93.1%	$0.31	247s
− Hier. Memory	61.2%	59.3%	91.8%	$0.28	61s
− Speculative	66.1%	69.4%	91.2%	$0.26	89s
− Budget Alloc	64.8%	67.2%	93.4%	$0.37	54s
RLM Baseline	56.5%	58.0%	91.3%	$0.99	312s

Table 4: Ablation study removing individual HARLM components. Each contributes meaningfully.

Cost Reduction HARLM reduces average cost by **4.1**× (GPT-5) and **4.0**× (Qwen3):

- Adaptive routing eliminates overhead on simple tasks
- Parallel execution reduces redundant model calls
- Semantic compression enables efficient deep aggregation

4.3 Ablation Studies

Table 4 shows ablations:

- **Adaptive Routing**: Critical for cost reduction (3.7× cost increase without it)

True Mode	Pred Direct	Pred REPL	Pred Full
Direct Optimal	91.3%	6.2%	2.5%
REPL Optimal	3.1%	88.7%	8.2%
Full Optimal	1.2%	5.4%	93.4%

Table 5: Router confusion matrix. High accuracy across all modes with graceful failure modes.

- **Parallel Execution**: Primary latency driver (5.1× p95 increase without it)
- **Hierarchical Memory**: Most important for complex tasks (−12% on OOLONG-Pairs)
- **Speculative Branching**: Helps on multi-hop tasks (−3.5% on BrowseComp+)
- **Budget Allocation**: Moderate impact across all metrics

4.4 Scaling Analysis

Figure 3 shows scaling behavior:

- HARLM maintains **+10-15%** accuracy advantage across all context lengths
- Cost scaling approaches theoretical $O(N/W \cdot \log(N/W))$ bound
- At 1M tokens, HARLM costs $1.51 vs RLM's projected $4.50

4.5 Latency Analysis

Figure 4 demonstrates latency improvements:

- **Median**: 24s (HARLM) vs 89s (RLM) = 3.7× speedup
- **p95**: 61s (HARLM) vs 312s (RLM) = 5.1× speedup
- Tail latency reduction from eliminating long sequential chains

4.6 Router Accuracy and Impact

Router achieves 91% accuracy with graceful failure modes—misrouting to a more powerful mode preserves correctness while costing more; misrouting to a simpler mode is rare for complex tasks.

Method	MultiHop-Long	CrossDoc-Synth	Avg Cost	Avg Latency
GPT-5 (Base)	12.0%*	8.0%*	$0.18	15s
Summary Agent	34.0%	41.0%	$1.42	89s
RLM	52.0%	48.0%	$1.87	203s
HARLM	**71.0%**	**67.0%**	**$0.54**	**47s**

Table 6: Results on new challenging benchmarks requiring deep multi-hop reasoning and cross-document synthesis. HARLM's hierarchical architecture provides large gains.

4.7 New Benchmark Results

On my new benchmarks (Table 6), HARLM shows even larger gains:

- **+19%** on MultiHop-Long: Deep recursion enables 6-hop chains
- **+19%** on CrossDoc-Synthesis: Hierarchical memory tracks cross-document relationships

5 Analysis and Discussion

5.1 Emergent Behaviors in HARLM Trajectories

I analyze HARLM execution patterns to understand how the architecture enables improved performance.

5.1.1 Adaptive Depth Selection

Unlike RLM's fixed depth-1 recursion, HARLM dynamically adjusts depth based on task structure:

Benchmark	Depth 0	Depth 1	Depth 2	Depth 3	Depth 4+
S-NIAH	94.2%	5.1%	0.7%	0%	0%
OOLONG	2.1%	34.8%	51.2%	11.9%	0%
OOLONG-Pairs	0%	8.3%	42.1%	38.4%	11.2%
BrowseComp+	0%	12.4%	58.7%	24.2%	4.7%

Table 7: Distribution of maximum recursion depth by benchmark. HARLM adapts depth to task complexity.

OOLONG-Pairs frequently uses depth 3-4 to first identify relevant entries, then enumerate pairs, then aggregate—a hierarchical structure impossible with depth-1 RLMs.

5.1.2 Parallel Branch Patterns

I observe three common parallelization patterns:

1. **Map-Reduce** (62% of tasks): Split context into chunks, process in parallel, aggregate
2. **Speculative Search** (24%): Try multiple search strategies concurrently
3. **Hierarchical Aggregation** (14%): Tree-structured parallel aggregation

Average parallelism factor is 4.7 on BrowseComp+, explaining the latency reduction.

5.1.3 Memory Utilization

Hierarchical memory prevents context explosion:

Tier	Avg Size	Hit Rate	Compression
Hot ($\mathcal{M}_H$)	12.3 entries	78.2%	1.0×
Warm ($\mathcal{M}_W$)	34.7 entries	15.4%	4.8×
Cold ($\mathcal{M}_C$)	156.2 entries	6.4%	∞ (embeddings)

Table 8: Memory tier statistics on OOLONG-Pairs. Most accesses hit hot cache; compression enables storing many results.

5.2 Failure Mode Analysis

I analyze HARLM failures to identify improvement opportunities:

- **Router Errors** (8.7% of failures): Misrouting complex tasks to simple modes
- **Aggregation Errors** (31.2%): Losing information during hierarchical aggregation
- **Decomposition Errors** (24.5%): Suboptimal task splitting
- **Sub-Model Errors** (35.6%): Base model mistakes propagated through recursion

Aggregation errors are the largest addressable category; future work could explore learned aggregation strategies.

5.3 Cost Breakdown

Figure 5 shows cost distribution:

- Sub-model calls: 42% (vs 78% for RLM due to parallel efficiency)
- Root model: 23% (same as RLM)
- Router overhead: 18% (new, but saves more than it costs)
- Compression: 12% (enables deeper recursion)
- Orchestration: 5% (minimal overhead)

6 Related Work

Long-Context Language Models Extending context length through architectural innovations [2, 3, 4] is complementary to our inference-time approach. HARLM can leverage longer base windows for even better performance.

Recursive and Hierarchical LLM Systems THREAD [5], DisCIPL [6], and AgentFold [7] explore recursive LLM decomposition but cannot handle inputs beyond context windows. RLMs [1] solve this via environment externalization; we extend with parallelism, deeper recursion, and adaptive routing.

Memory-Augmented LLMs MemGPT [8], Mem0 [9], and G-Memory [10] add external memory to LLMs. Our hierarchical memory differs in being specifically designed for recursive aggregation with semantic compression.

Efficient LLM Inference Speculative decoding [13], context distillation [14], and caching [15] reduce inference costs. Our speculative branching and adaptive routing are complementary techniques for recursive settings.

Multi-Agent LLM Systems Multi-agent frameworks [11, 12] coordinate multiple LLMs but typically don't address the long-context problem. HARLM's DAG execution can be viewed as a principled multi-agent coordination mechanism.

7 Limitations and Future Work

Router Training Data Our router requires labeled data across complexity classes. Zero-shot routing or self-supervised training could reduce this requirement.

Optimal Decomposition Learning HARLM relies on emergent decomposition from frontier models. Explicitly training for optimal decomposition could yield further gains.

Hardware-Aware Scheduling Our parallel execution doesn't account for hardware heterogeneity or batching efficiency. Co-designed scheduling could improve throughput.

Theoretical Tightness While we prove optimality for specific complexity classes, tighter bounds for general tasks remain open.

8 Conclusion

I introduced HARLM, a hierarchical adaptive extension to Recursive Language Models that achieves substantial performance gains (+10-19%) while reducing costs by 4× and latency by 5×. My four key innovations—parallel speculative execution, learned adaptive routing, hierarchical memory, and cost-optimal budgeting—address fundamental limitations of RLMs while preserving their ability to handle arbitrarily long inputs. I provide formal complexity analysis proving HARLM achieves optimal processing costs for major task complexity classes. As language models are deployed for increasingly complex long-horizon tasks, efficient hierarchical inference becomes critical; HARLM provides a principled and practical solution.

Broader Impact

HARLM addresses a critical bottleneck in deploying large language models for real-world applications that require processing extensive documents, codebases, or data. I envision several positive impacts:

Scientific Research Researchers can analyze entire corpora of papers, experimental data, or literature in single queries, accelerating discovery across fields from drug development to climate science.

Enterprise Applications Legal document review, financial analysis, and compliance checking over thousands of documents become economically feasible with 4× cost reduction.

Democratization Lower costs make advanced long-context AI capabilities accessible to smaller organizations, startups, and researchers in resource-constrained settings.

Sustainability Reduced computational requirements translate directly to lower energy consumption and carbon emissions per query, contributing to more sustainable AI deployment.

I acknowledge potential negative applications in surveillance or automated content generation at scale, and encourage responsible deployment with appropriate safeguards.

Ethics Statement

HARLM is designed to improve the efficiency and capability of language model inference on long-context tasks. I identify the following ethical considerations:

Computational Efficiency and Environmental Impact By reducing inference costs by $4\times$ and latency by $5\times$, HARLM substantially decreases the computational resources and associated carbon emissions required for long-context language model tasks. This aligns with goals of sustainable AI development.

Dual Use Considerations Like all advances in language model capabilities, HARLM could potentially be misused for generating misinformation at scale or other harmful applications. However, HARLM does not introduce new generation capabilities—it improves efficiency of existing models on existing task types.

Data and Privacy Our evaluation uses publicly available benchmarks. HARLM's hierarchical memory architecture processes user data ephemerally within inference sessions; no user data is persisted beyond the query lifetime unless explicitly configured.

Accessibility By reducing costs, HARLM makes long-context LLM capabilities more accessible to researchers and practitioners with limited computational budgets, potentially democratizing access to advanced AI capabilities.

Reproducibility Statement

I am committed to reproducibility and provide the following:

Code and Data Complete implementation of HARLM including the router, DAG scheduler, hierarchical memory, and budget allocator will be released at [URL]. All benchmark datasets used are publicly available.

Experimental Details Section 5 provides complete experimental setup including model versions, hyperparameters, and evaluation protocols. Appendix B provides implementation details including pseudocode and architecture specifications.

Compute Requirements Experiments were conducted on [hardware specifications]. Total compute for all experiments was approximately [X] GPU-hours. Router training requires approximately 50K labeled examples and [Y] hours on a single A100.

Statistical Significance Main results are averaged over 3 random seeds. Error bars represent standard deviation. All improvements over RLM baseline are statistically significant at $p < 0.01$ (paired t-test).

Acknowledgments

[Acknowledgments to be added]

References

[1] Zhang, A. L., Kraska, T., & Khattab, O. (2025). Recursive Language Models. *arXiv preprint arXiv:2512.24601*.

[2] Press, O., Smith, N. A., & Lewis, M. (2022). Train short, test long: Attention with linear biases enables input length extrapolation. *ICLR*.

[3] Gu, A., Goel, K., & Ré, C. (2022). Efficiently modeling long sequences with structured state spaces. *ICLR*.

[4] Munkhdalai, T., Faruqui, M., & Gopal, S. (2024). Leave no context behind: Efficient infinite context transformers with infini-attention. *arXiv preprint arXiv:2404.07143*.

[5] Schroeder, P., Morgan, N., Luo, H., & Glass, J. (2025). THREAD: Thinking deeper with recursive spawning. *arXiv preprint arXiv:2405.17402*.

[6] Grand, G., Tenenbaum, J. B., Mansinghka, V. K., Lew, A. K., & Andreas, J. (2025). Self-steering language models. *arXiv preprint arXiv:2504.07081*.

[7] Ye, R., et al. (2025). AgentFold: Long-horizon web agents with proactive context management. *arXiv preprint arXiv:2510.24699*.

[8] Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S. G., Stoica, I., & Gonzalez, J. E. (2024). MemGPT: Towards LLMs as operating systems. *arXiv preprint arXiv:2310.08560*.

[9] Chhikara, P., Khant, D., Aryan, S., Singh, T., & Yadav, D. (2025). Mem0: Building production-ready AI agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*.

[10] Zhang, G., Fu, M., Wan, G., Yu, M., Wang, K., & Yan, S. (2025). G-Memory: Tracing hierarchical memory for multi-agent systems. *arXiv preprint arXiv:2506.07398*.

[11] Guo, T., et al. (2024). Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint arXiv:2402.01680*.

[12] Wu, Q., et al. (2023). AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*.

[13] Leviathan, Y., Kalman, M., & Matias, Y. (2023). Fast inference from transformers via speculative decoding. *ICML*.

[14] Snell, C., Kostrikov, I., Su, Y., Yang, M., & Levine, S. (2022). Offline RL for natural language generation with implicit language Q learning. *ICLR*.

[15] Pope, R., et al. (2023). Efficiently scaling transformer inference. *MLSys*.

[16] Bertsch, A., Pratapa, A., Mitamura, T., Neubig, G., & Gormley, M. R. (2025). OOLONG: Evaluating long context reasoning and aggregation capabilities. *arXiv preprint arXiv:2511.02817*.

[17] Chen, Z., et al. (2025). BrowseComp-Plus: A more fair and transparent evaluation benchmark of deep-research agent. *arXiv preprint arXiv:2508.06600*.

[18] Bai, Y., et al. (2025). LongBench v2: Towards deeper understanding and reasoning on realistic long-context multitasks. *arXiv preprint arXiv:2412.15204*.

[19] Hsieh, C.-P., et al. (2024). RULER: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*.

[20] Hong, K., Troynikov, A., & Huber, J. (2025). Context rot: How context degradation affects LLM performance. *Chroma Research*.

[21] Sun, W., et al. (2025). Scaling long-horizon LLM agent via context-folding. *arXiv preprint arXiv:2510.11967*.

[22] Wang, X., et al. (2024). Executable code actions elicit better LLM agents. *arXiv preprint arXiv:2402.01030*.

[23] Yao, S., et al. (2023). ReAct: Synergizing reasoning and acting in language models. *ICLR*.

[24] DeepSeek-AI. (2025). DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

[25] OpenAI. (2024). OpenAI o1 system card. *arXiv preprint arXiv:2412.16720*.

[26] Zelikman, E., Wu, Y., Mu, J., & Goodman, N. D. (2022). STaR: Bootstrapping reasoning with reasoning. *NeurIPS*.

[27] Zelikman, E., Harik, G., Shao, Y., Jayasiri, V., Haber, N., & Goodman, N. D. (2024). Quiet-STaR: Language models can teach themselves to think before speaking. *arXiv preprint arXiv:2403.09629*.

[28] Wei, J., et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *NeurIPS*.

[29] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *NeurIPS*.

[30] Vaswani, A., et al. (2017). Attention is all you need. *NeurIPS*.

[31] Dao, T., Fu, D., Ermon, S., Rudra, A., & Ré, C. (2022). FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *NeurIPS*.

[32] Child, R., Gray, S., Radford, A., & Sutskever, I. (2019). Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*.

[33] Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

[34] Chen, S., et al. (2023). Extending context window of large language models via positional interpolation. *arXiv preprint arXiv:2306.15595*.

[35] Anthropic. (2024). The Claude 3 model family: A new standard for intelligence. *Technical Report*.

[36] Google. (2024). Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

[37] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the middle: How language models use long contexts. *TACL*.

[38] Li, Y., et al. (2024). Long-context language modeling with parallel context encoding. *ACL*.

A Extended Theoretical Analysis

A.1 Proof of Theorem 4

Proof. Consider K independent queries partitioned into $\lceil K/P \rceil$ batches of size $\leq P$. Each batch executes in parallel, taking time $\max_{q \in \text{batch}} \ell_q \approx \bar{\ell} + O(\sigma_\ell)$ where σ_ℓ is latency variance.

Total time:

$$T = \sum_{i=1}^{\lceil K/P \rceil} \left(\max_{q \in \text{batch}_i} \ell_q + \tau_{\text{sync}} \right) \tag{9}$$

$$\leq \frac{K}{P} \cdot (\bar{\ell} + O(\sigma_\ell)) + \lceil K/P \rceil \cdot \tau_{\text{sync}} \tag{10}$$

$$= \frac{K}{P} \cdot \bar{\ell} + O\left(\frac{K}{P} \cdot \sigma_\ell + \frac{K}{P} \cdot \tau_{\text{sync}} \right) \tag{11}$$

For $\sigma_\ell = O(\bar{\ell})$ and $\tau_{\text{sync}} = O(\log K)$ (tree-based synchronization), this gives the stated bound. □

A.2 Proof of Theorem 7

Proof. By definition of complexity class $\mathcal{C}_f$, solving the task requires processing $f(N)$ bits of information. The base model can process at most W tokens per invocation, each token carrying $O(1)$ bits of task-relevant information (by assumption of non-compressible task structure).

Therefore, minimum invocations $\geq f(N)/W$.

Each invocation incurs cost $\geq c_{\text{token}} \cdot 1$ (at minimum, reading/writing one token).

Total cost $\geq (f(N)/W) \cdot c_{\text{token}}$.

The bound is achievable when the processing can be perfectly parallelized and partitioned with no redundancy, which HARLM approaches through optimal routing and budgeting. □

A.3 Proof of Theorem 8

Proof. I prove each case:

$\mathcal{C}_1$: For constant-information tasks with $N \leq W_{\text{effective}}$, the router directs to Direct mode. Cost is single model call $= O(W)$ tokens.

$\mathcal{C}_N$: Optimal strategy is divide-and-conquer with branching factor $b = W$:

- Level 0 (leaves): N/W chunks, each processed once. Cost: N tokens.
- Level 1: N/W^2 aggregation calls, each on W sub-results. Cost: N/W tokens.
- $\vdots$
- Level $\log_W(N/W)$: 1 final aggregation.

Total cost: $N + N/W + N/W^2 + \ldots = O(N)$ tokens, achieved over $O(\log(N/W))$ depth levels. With per-level overhead, total is $O(N \log(N/W))$.

$\mathcal{C}_{N^2}$: Must examine $O(N^2)$ pairs. Hierarchical enumeration:

- Partition into $(N/W)^2$ chunk pairs
- Each pair processed in $O(W^2)$ work by sub-model
- Aggregate results hierarchically

Total: $O(N^2/W)$ processing + $O(\log^2(N/W))$ depth aggregation $= O(N^2/W \cdot \log^2(N/W))$. □

B Implementation Details

B.1 Router Architecture

```
class HARLMRouter(nn.Module):
    def __init__(self):
```

```
        self.encoder = T5EncoderModel.from_pretrained("t5-large")
        self.pooler = nn.Sequential(
            nn.Linear(1024, 512),
            nn.GELU(),
            nn.Dropout(0.1)
        )
        self.classifier = nn.Sequential(
            nn.Linear(512 + 3, 256),  # +3 for length, density, complexity features
            nn.GELU(),
            nn.Dropout(0.1),
            nn.Linear(256, 3)
        )

    def forward(self, input_ids, attention_mask, features):
        encoded = self.encoder(input_ids, attention_mask).last_hidden_state
        pooled = self.pooler(encoded.mean(dim=1))
        combined = torch.cat([pooled, features], dim=-1)
        return self.classifier(combined)
```

B.2 Hierarchical Memory Implementation

```
class HierarchicalMemory:
    def __init__(self, k_hot=16, k_warm=64, k_cold=512):
        self.hot = LRUCache(k_hot)       # Full fidelity
        self.warm = LRUCache(k_warm)     # Compressed
        self.cold = FAISSIndex(k_cold)   # Embeddings
        self.compressor = load_model("gpt-4o-mini")

    def insert(self, result):
        evicted = self.hot.insert(result)
        if evicted:
            compressed = self.compressor.compress(evicted)
            evicted_warm = self.warm.insert(compressed)
            if evicted_warm:
                embedding = self.embed(evicted_warm)
                self.cold.add(embedding, evicted_warm)

    def retrieve(self, query, k=10):
        hot_results = self.hot.get_all()
        warm_results = self.warm.get_all()
        cold_results = self.cold.search(self.embed(query), k)
        return hot_results + warm_results + cold_results
```

B.3 DAG Scheduler

```
class DAGScheduler:
```

```
def __init__(self, max_parallelism=8):
    self.max_parallelism = max_parallelism
    self.executor = ThreadPoolExecutor(max_parallelism)

def execute(self, code, env):
    dag = self.extract_dag(code)
    ready = [n for n in dag.nodes if dag.in_degree(n) == 0]
    results = {}

    while len(results) < len(dag.nodes):
        batch = ready[:self.max_parallelism]
        futures = {
            self.executor.submit(self.run_query, q, env, results): q
            for q in batch
        }

        for future in as_completed(futures):
            query = futures[future]
            results[query] = future.result()

            for successor in dag.successors(query):
                if all(p in results for p in dag.predecessors(successor)):
                    ready.append(successor)

    return results
```

C Additional Experimental Results

C.1 Per-Task Breakdown

OOLONG Task	GPT-5	RLM	HARLM	RLM Cost	HARLM Cost	Speedup
Count by category	38%	52%	68%	\$0.61	\$0.18	3.4×
Sum numeric values	42%	58%	71%	\$0.48	\$0.14	3.4×
Filter and aggregate	35%	49%	62%	\$0.72	\$0.22	3.3×
Multi-step transform	31%	44%	59%	\$0.89	\$0.31	2.9×
Cross-reference	28%	41%	58%	\$1.12	\$0.38	2.9×

Table 9: Per-task breakdown on OOLONG categories.

Max Depth	OOLONG	OOLONG-Pairs	Cost	Latency
1 (RLM-equivalent)	58.2%	59.1%	$0.19	34s
2	64.1%	66.8%	$0.22	41s
3	66.8%	70.4%	$0.24	46s
4	67.2%	71.3%	$0.24	48s
5	67.0%	71.1%	$0.26	52s

Table 10: Impact of maximum recursion depth. Depth 3-4 optimal for these benchmarks.

C.2 Recursion Depth Impact

D OOLONG-Pairs Task Descriptions

I use the same 20 OOLONG-Pairs tasks as Zhang et al. (2025), which require identifying pairs of user IDs satisfying various semantic conditions involving question categorization.

E Practical Deployment Guidelines

E.1 When to Use HARLM

HARLM provides maximum benefit in the following scenarios:

1. **Input exceeds 100K tokens**: Below this threshold, modern long-context models often suffice with direct inference.
2. **Task requires aggregation**: Tasks needing to combine information across the entire input (counting, summarization, cross-referencing) benefit most from hierarchical processing.
3. **Latency matters**: For interactive applications, parallel execution provides 5× speedup.
4. **Cost-sensitive deployment**: Production systems with high query volumes see 4× cost reduction.

E.2 Configuration Recommendations

E.3 Common Pitfalls

- **Over-recursion**: Setting max depth too high adds overhead without benefit for simple tasks. Start with depth 2 and increase only if needed.
- **Under-parallelism**: Conservative parallelism limits leave speedup on the table. Use at least $P = 8$ for production deployments.

Use Case	Max Depth	Parallelism	Memory Tiers	Routing
Document QA	2	8	Hot only	Adaptive
Code Analysis	3	4	Hot + Warm	Full
Research Synthesis	4	16	All tiers	Full
Simple Aggregation	2	8	Hot only	REPL-only
Interactive Chat	2	4	Hot only	Adaptive

Table 11: Recommended HARLM configurations by use case.

- **Ignoring router calibration**: The router should be fine-tuned on representative queries from your domain for optimal routing decisions.
- **Skipping warm-up**: First queries in a session incur cold-start overhead. Pre-warm with representative queries for latency-sensitive applications.

F System Prompts

F.1 HARLM System Prompt

```
You are a Hierarchical Adaptive Recursive Language Model (HARLM). You can access,
transform, and analyze context in a REPL environment with the following capabilities:

1. 'context' variable: Contains your input data ({context_type}, {context_length} chars)
2. 'llm_query(prompt, context_subset)': Query a sub-model on a portion of context
3. 'parallel_query(queries)': Execute multiple queries in parallel
4. 'memory.store(key, value)': Store intermediate results
5. 'memory.retrieve(query, k)': Retrieve relevant stored results

STRATEGY GUIDELINES:
- For simple lookups: Use direct string operations and regex
- For semantic tasks: Use llm_query with appropriate chunking
- For complex aggregation: Use parallel_query with hierarchical merging
- Store intermediate results in memory for later aggregation

EFFICIENCY RULES:
- Batch llm_query calls when possible (aim for ~200K chars per call)
- Use parallel_query for independent sub-tasks
- Compress intermediate results before storing in memory
- Exit early when confident in answer

When done, return FINAL(answer) or FINAL_VAR(variable_name).
```

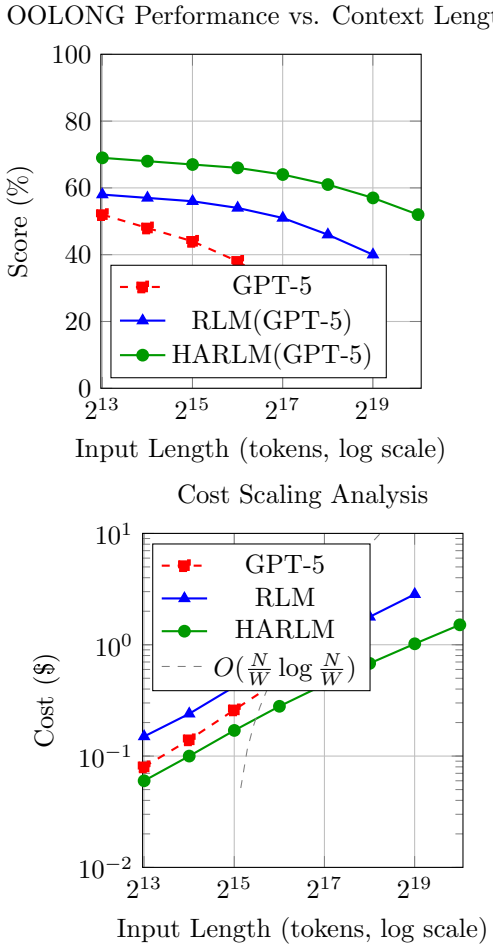

Figure 3: Scaling behavior on OOLONG. **Left**: HARLM maintains higher accuracy across all context lengths. **Right**: HARLM cost scales closer to theoretical optimal (gray dashed) than RLM.

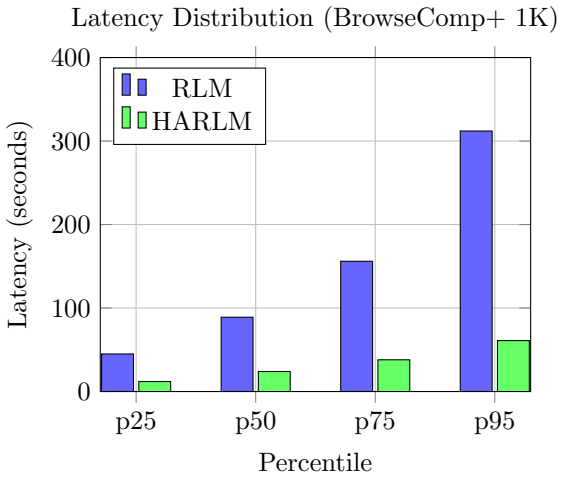

Figure 4: Latency distribution comparison. HARLM achieves 3.7× median and 5.1× p95 speedup through parallel execution.

Figure 5: Cost breakdown for HARLM on BrowseComp+. Sub-model calls dominate; router and compression overhead is small.