Recursive Language Models

Alex L. Zhang
MIT CSAIL
altzhang@mit.edu

Tim Kraska
MIT CSAIL
kraska@mit.edu

Omar Khattab
MIT CSAIL
okhattab@mit.edu

Abstract

We study allowing large language models (LLMs) to process arbitrarily long prompts through the lens of inference-time scaling. We propose **Recursive Language Models** (**RLM**s), a general inference paradigm that treats long prompts as part of an external *environment* and allows the LLM to *programmatically* examine, decompose, and *recursively call itself over* snippets of the prompt. We find that RLMs can successfully process inputs more than an order of magnitude beyond model context window limits and, even for shorter prompts, dramatically outperform the quality of vanilla frontier LLMs and common long-context and coding scaffolds (e.g., on GPT-5 by a median across the evaluated benchmarks of 26% against compaction, 130% against CodeAct with sub-calls, and 13% against Claude Code) across four diverse long-context tasks while having comparable cost. At a small scale, we post-train the first model around the RLM. Our model, **RLM-Qwen3-8B**, outperforms the underlying Qwen3-8B model by a median of 28% and even approaches the quality of vanilla GPT-5 on three long-context tasks. Code is available at `https://github.com/alexzhang13/rlm`.

1 Introduction

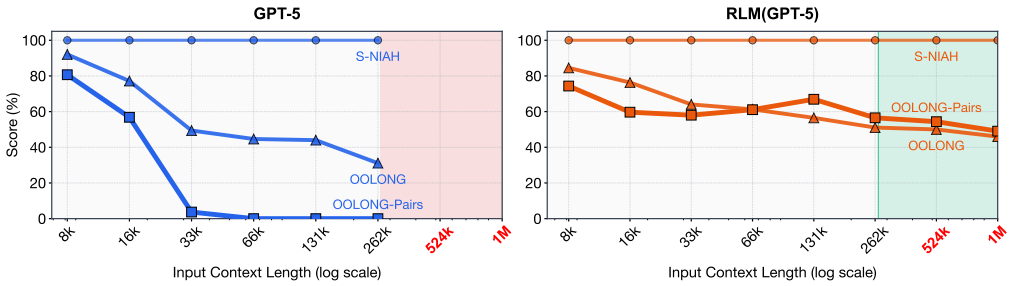


Figure 1: A comparison of GPT-5 and a corresponding RLM(recursion depth=1) using GPT-5 on three long-context tasks of increasing complexity: **S-NIAH**, **OOLONG**, and **OOLONG-Pairs**. For each task, we scale the input length from 2^{13} to 2^{20}. GPT-5 performance degrades significantly as a function of both input length and task complexity, while the RLM maintains strong performance. Inputs beyond the red region do not fit in GPT-5's context window of 272K tokens, but the RLM handles them effectively. Additional experiments across other models and benchmarks are in §3.

Frontier reasoning models have limited context windows and, even within their limits, tend to exhibit *context rot* [Hong et al., 2025], a phenomenon illustrated in Figure 1 where quality degrades steeply as prompts get longer. Though we expect context lengths to steadily rise through improvements to training, architecture, and infrastructure, we are interested in *whether it is possible to scale the context size of general-purpose LLMs by orders of magnitude*. This is increasingly urgent as LLMs begin to be widely adopted for long-horizon tasks, in which they must routinely process tens if not hundreds of millions of tokens.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

We study this question through the lens of scaling inference-time compute. We are inspired by the way that *reasoning models*, another inference strategy, have become the fundamental interface to LLMs, resulting not only in empirical gains but also additional theoretical expressive power [Merrill and Sabharwal, 2024] compared to vanilla Transformers. Though most inference-time methods for dealing with long context are task-specific [Wu et al., 2021, Chang et al., 2024], the most popular general approach is *context condensation* or *compaction* [Khattab et al., 2021, Smith, 2025, OpenAI, 2025b, Wu et al., 2025], where context from user requests or agent trajectories is repeatedly summarized once it exceeds a length threshold. Unfortunately, compaction is rarely expressive enough for tasks that require dense access throughout the prompt. It presumes that *some* details that appear early in the prompt can safely be forgotten to make room for new content.

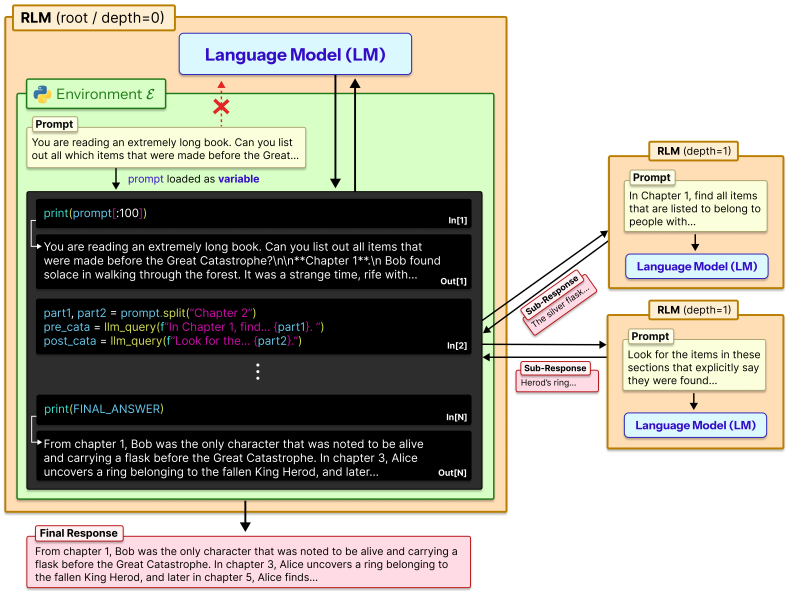


Figure 2: A Recursive Language Model (RLM) treats prompts as part of the environment. It loads the input prompt as a variable inside a REPL environment $\mathcal{E}$ and writes code to peek into, decompose, and invoke itself recursively over programmatic snippets of the variable.

We introduce **Recursive Language Models** (**RLM**s), a general-purpose inference paradigm for dramatically scaling the effective input and output lengths of LLMs. The key insight is that arbitrarily long user prompts should not be fed into the neural network (e.g., Transformer) directly but should instead be treated as *part of the environment that the LLM is tasked to* ***symbolically and recursively*** *interact with*. This system serves as an abstracted "language model" without context limitations.

As Figure 2 shows, an RLM exposes the same external interface as an LLM or a reasoning model: it accepts a string prompt of arbitrary structure and produces a string response. Given a prompt P, the RLM initializes a Read-Eval-Print Loop (REPL) programming environment in which P is set as the value of a variable. It then offers the LLM general context about the REPL environment (e.g., the length of the string P), and permits it to write code that peeks into and decomposes P, and to iteratively observe any side effects from execution. Crucially, RLMs encourage the LLM to understand, transform, and execute the input prompt by *writing symbolic programs that invoke the LLM itself* on as many slices of the input as necessary.

By treating the prompt itself as an external object and enabling symbolic recursion, RLMs tackle limitations of expressive power in recent work on coding agents, retrieval agents, and sub-agent delegation. In particular, prior coding agents and retrieval agents treat some designated external data source (e.g., a filesystem or a corpus of search documents) as an environment for fetching snippets. However, they can only fill up the underlying LLM's context window with snippets before facing compaction. Similarly, prior self-delegation approaches [Anthropic, 2025, Sentient AI, 2025, Schroeder et al., 2025, Sun et al., 2025] allow LLMs to invoke themselves as sub-agents. However, they are handicapped by the underlying LLM's limited output lengths because they are designed to verbalize sub-calls autoregressively rather than producing them programmatically.

We evaluate RLMs using a frontier closed model (GPT-5; Singh et al. 2025) and a frontier open model (Qwen3-Coder-480B-A35B; Qwen Team 2025b) across four tasks with varying levels of complexity: deep research [Chen et al., 2025], information aggregation [Bertsch et al., 2025], code repository understanding [Bai et al., 2025], and a synthetic pairwise reasoning task where even frontier models fail catastrophically. We compare RLMs against direct LLM calls as well as context compaction, retrieval tool-use agents, and code-generation agents with and without sub-calls.

We find that RLMs demonstrate extremely strong performance even at the 10M+ token scale, and substantially outperform other approaches at long-context processing, in many cases by double-digit percentage gains while maintaining comparable cost. In particular, as demonstrated in Figure 1, RLMs exhibit far less severe degradation for longer contexts and more sophisticated tasks.

Finally, at a small scale, we post-train the first natively recursive language model, demonstrating that RLMs can be improved quickly with little additional training. While a small open model (Qwen3-8B; Yang et al. 2025) struggles to solve long context tasks even in an RLM scaffold, our simple general-purpose training recipe uses only 1,000 samples from unrelated domains to improve its performance by a median of 28.3% across the four evaluation tasks.

2 Recursive Language Models

Given a base neural language model $\mathcal{M}$ with maximum context size K, a Recursive Language Model (RLM) is an inference-time scaffold around $\mathcal{M}$ that treats the user prompt as part of the environment without giving up the ability to densely process its content through different calls to $\mathcal{M}$. Given an arbitrary-length prompt string $P \in \Sigma^\star$, an RLM interacts with a persistent external environment $\mathcal{E}$ and returns a response string $Y \in \Sigma^\star$ (Figure 2). We would like effectively *unbounded input tokens* ($|P| \gg K$), *unbounded output tokens*, and an *unbounded semantic horizon*, e.g. the ability to do $\Omega(|P|)$ or $\Omega(|P|^2)$ semantic work.

Algorithm 1 describes how an RLM achieves this. Given a prompt P, the RLM initializes a persistent REPL programming environment with a variable containing the user prompt as a string and a function for invoking a sub-RLM with a new prompt. Then, it starts the RLM loop. In the first iteration, the algorithm invokes the *root* neural model $\mathcal{M}$ with only (constant-size) metadata about the user prompt, like its length, a short prefix, and how to access parts of it.

The root is instructed via prompting (Appendix C) and/or fine-tuning (Appendix A) to operate like an RLM: that is, to *generate code that helps it understand and transform parts of its prompt P*, and to build up intermediate values and the final response into new variables, potentially by *invoking the sub-RLM within loops*. In Section 4, we find that existing LLMs can be prompted to do this and that training an 8B model to be natively recursive is promising.

Each iteration of the RLM loop executes code in the REPL, updates REPL state (intermediate variables), and collects in `stdout` any printed text. Only (constant-size) metadata about `stdout`, like a short prefix and length, is appended to $\mathcal{M}$'s history for the next iteration.[1] Once the RLM sets the variable `Final` inside the REPL, iteration stops and the value in `Final` is returned as the response.

RLMs make three simple design choices that are missing from many existing scaffolds. To highlight these, we include Algorithm 2 to illustrate a deceptively "similar" algorithm that is far less expressive. Both algorithms support some notion of sub-calls, external objects, and code execution, but they differ in terms of where the prompt and intermediate values live and where recursion occurs.

First, an RLM must give the underlying LLM $\mathcal{M}$ a *symbolic handle* to the user prompt P, so the model can manipulate it without copying text into the root context window. Instead, ineffective Algorithm 2 starts by putting the user prompt P into the LLM context window (`hist`), inheriting the window limitations of $\mathcal{M}$ and falling back to heuristics like context compaction. Even though the scaffold can access external data with, say, a `Search` action, it is bounded with respect to user input.

Second, ineffective Algorithm 2 asks $\mathcal{M}$ to generate the output directly, via a `Finish` action. This may seem innocuous, but it means outputs cannot be longer than the context window of $\mathcal{M}$.

[1] This is key: it forces $\mathcal{M}$ to rely on variables and sub-calls to manage long strings instead of polluting its window. In principle, if we trim each turn to c tokens, we will have at most K/c root iterations, each of which can launch arbitrarily many sub-calls. This is not a fundamental limitation, e.g. one could move the root horizon itself into a variable, but we typically want to limit the iterations at any level of recursion irrespective.

Algorithm 1: A recursive language model, around LLM $\mathcal{M}$, which itself acts as a "language model".

Input: prompt P
Output: response Y
`state` $\leftarrow$ `InitREPL(prompt=P)`
`state` $\leftarrow$ `AddFunction(state, sub_RLM`$_{\mathcal{M}}$`)`
`hist` $\leftarrow$ `[Metadata(state)]`
while *True* **do**
  `code` $\leftarrow$ `LLM`$_{\mathcal{M}}$`(hist)`
  `(state, stdout)` $\leftarrow$ `REPL(state, code)`
  `hist` $\leftarrow$ `hist` $\|$ `code` $\|$ `Metadata(stdout)`
  if *`state[Final]` is set* **then**
    return `state[Final]`

Algorithm 2: Alternate scaffold with standard (poor) design choices.

Input: prompt P
Output: response Y
`actions` $\leftarrow$ `{Finish, Exec, Search, sub_LLM`$_{\mathcal{M}}$`}`
`hist` $\leftarrow$ `[Metadata(actions),` P`]` Flaw #1
while *True* **do**
  `(action, val)` $\leftarrow$ `LLM`$_{\mathcal{M}}$`(hist)`
  if *`action` is `Finish`* **then**
    return `val` Flaw #2
  `out` $\leftarrow$ `RUN(action, val)` Flaw #3
  `hist` $\leftarrow$ `hist` $\|$ `(action, val, out)`
  if *`Tok(hist)` $> K$* **then**
    `hist` $\leftarrow$ `Compact(hist)`

Third, and perhaps most importantly, an RLM requires *symbolic recursion*. That is, code running *inside* $\mathcal{E}$ must be able to invoke $\mathcal{M}$ on programmatically constructed transformations of P (e.g., inside arbitrarily large loops), storing intermediate results symbolically. Though Algorithm 2 includes both a code execution action and a "sub-LLM" action separately, it is not able to invoke the sub-LLM programmatically and hence can only delegate a few *explicitly verbalized tasks* rather than writing short programs that can, say, loop over slices of the prompt and launch $\Omega(|P|)$ or even $\Omega(|P|^2)$ processes to understand or transform all parts of P.

We implement our RLM definition in Algorithm 1 as follows: we equip an LLM with a Python REPL, where all tools, including sub-LM or sub-RLM calls, are available as modules. The initial prompt is stored as a variable in the REPL. The LLM interacts in a loop until it provides a final answer, which can be from either a variable in the REPL, or from the LLM itself. The LLM can also print from the REPL, but it is truncated to prevent overflowing the context too quickly.

3 Scaling Long Context Tasks

We hypothesize that the effective context window [Hsieh et al., 2024, Goldman et al., 2025, Hong et al., 2025] of an LLM cannot be understood independently of the *specific task*. That is, more "complex" problems will exhibit degradation at even *shorter* lengths than simpler ones. Because of this, we must characterize tasks in terms of how their complexity *scales with prompt length*.

For example, needle-in-a-haystack (NIAH) problems generally keep 'needles' constant as prompt length is scaled. As a result, frontier models can now reliably solve these tasks in RULER [Hsieh et al., 2024] in the 1M+ token settings but struggle at far shorter lengths on OOLONG [Bertsch et al., 2025], a task where the answer depends explicitly on almost every line in the prompt.[2]

3.1 Tasks

We design our evaluation around tasks where we can vary the lengths of the prompts, so we can consider problems whose difficulties scale differently with context length.

S-NIAH. Following the single needle-in-the-haystack task in RULER [Hsieh et al., 2024], we consider a set of 50 single tasks that require finding a specific phrase or number in a large set of unrelated text. Here, the information being sought scales as $O(1)$ with respect to input length.

BrowseComp-Plus (1K documents) [Chen et al., 2025]. A multi-hop question-answering benchmark for DeepResearch [OpenAI, 2025a] questions that requires reasoning over multiple different documents in an offline corpus. Following Sun et al. [2025], we use 150 randomly sampled instances as our evaluation set; we provide 1000 randomly chosen documents as input, in which the gold and evidence documents are guaranteed to exist. We report the percentage of correct answers. The answer

[2]This helps explain the patterns seen in Figure 1 earlier: GPT-5 scales effectively on the S-NIAH task, where the needle size is constant despite longer prompts, but shows faster degradation at increasingly *shorter* context lengths on the *linear*-complexity OOLONG and the *quadratic*-complexity OOLONG-Pairs.

to each task requires piecing together information from several documents, making this harder than **S-NIAH** despite also requiring a constant number of documents.

OOLONG [Bertsch et al., 2025]. A long reasoning benchmark that requires semantically labeling and aggregating these labels to form a final answer. We focus specifically on the `trec_coarse` split, a set of 50 tasks over a dataset of questions with semantic labels. Each task requires using nearly all dataset questions, and therefore scales linearly in processing complexity relative to the input length.

OOLONG-Pairs. A modified variant of the `trec_coarse` split of OOLONG with 20 queries that specifically require aggregating *pairs* of chunks to construct the final answer. We report F1 scores over the answer, which is a list of entries. Each task requires using nearly all *pairs* of entries of the dataset, and therefore requires processing quadratically-many items relative to the input length. In Appendix D.1, we list all queries in this benchmark.

LongBench-v2 CodeQA [Bai et al., 2025]. A multi-choice code repository understanding split from LongBench-v2 that is challenging for modern frontier models. Each instance requires reasoning over a fixed number of files in a codebase to find the right answer.

3.2 Methods and Baselines

We compare RLMs against commonly used task-agnostic inference methods, using two modern LMs, GPT-5 with medium reasoning [Singh et al., 2025] and default sampling parameters, and Qwen3-Coder-480B-A35B [Yang et al., 2025] using the sampling parameters described in Qwen Team [2025b]. For Qwen3-Coder-480B-A35B, we compute costs based on the compute provider Fireworks [Fireworks AI, 2025]. In addition to evaluating the base model on all tasks, we also evaluate the following methods and baselines:

CodeAct. We compare directly to a CodeAct [Wang et al., 2024] agent that can execute code inside of a ReAct [Yao et al., 2023] loop. Unlike an RLM, CodeAct does not offload the user prompt to the code environment, and instead provides it directly to the LM. We consider two variants: (1) a version following Jimenez et al. [2024], Chen et al. [2025] where we equip this agent with a BM25 [Robertson and Zaragoza, 2009] retriever; (2) a version with a sub-call tool inside of the REPL. Compared to RLMs, this method loads the context directly into the model.

Compaction agent. Following Sun et al. [2025], Wu et al. [2025], Yu et al. [2025], we consider an iterative agent that compacts the context as it is filled. For example, given a corpus of documents, it will iteratively accumulate the documents and summarize when full. In cases where a single document exceeds the model window, the agent will chunk the document and iteratively compact it. For the GPT-5 experiments, due to the extremely high cost of applying this strategy to millions of tokens, we use GPT-5-nano for compaction and GPT-5 to provide the final answer.

Coding agents. We compare against commonly used coding agents like OpenCode [Anomaly, 2026] and Claude Code [Anthropic, 2025]. We consider two variants, one where the context is offloaded to a file, and another where it is directly used as the initial prompt. Closed source agents like Claude Code are designed around a corresponding model, so we use Claude Opus 4.1 with Claude Code v2.0.0 (released around the same time as the GPT-5 model we use in our main results) for this baseline.

RLM. We implement an RLM with a Python REPL environment, which loads a module for querying a sub-LM and uses a system prompt presented in Appendix C. For the GPT-5 experiments, we use GPT-5-mini for the recursive LMs and GPT-5 for the root LM, as we found this choice to strike a good balance between the capabilities of RLMs and the cost of the recursive calls. We also evaluate several different max recursion depths allowable to the RLM, from 0-3. Max recursion depth 0 is an RLM without sub-calling capabilities. Max recursion depth 1 allows sub-calling LLMs, while max depth >1 allows sub-calling RLMs. We notate a RLM with max recursion depth N using a model as RLM(model, depth=N), e.g. RLM(GPT-5, depth=2), and assume depth=1 if not stated otherwise.

Fine-tuning. To create **RLM-Qwen3-8B**, we fine-tune Qwen3-8B on 1,000 filtered trajectories of Qwen3-Coder-480B-A35B as an RLM with Qwen3-8B sub-calls on LongBenchPro [Chen et al., 2026] tasks. We use sampling parameters described in Qwen Team [2025a], and evaluate the fine-tuned RLM-Qwen3-8B as an RLM. The key insight for training is that being an effective sub-call model is roughly similar to being a general purpose reasoning model, so we can make the training much more tractable at small scale by focusing on improving the root model's ability to manipulate the REPL and to launch recursive calls. We provide more training details in Appendix A.

4 Results and Discussion

Table 1 reports our main evaluation results. We additionally explore how vanilla frontier model and RLM performance degrade as input contexts grow in Figure 1.

Table 1: Performance comparison of different methods across long-context benchmarks of varying complexity. In gray is the average API cost ± the standard deviation of each method on each task. * indicates runs where a method (sometimes) ran into input context limits. Provider costs were computed under OpenAI for GPT-5, under Fireworks for Qwen3 models, and under Anthropic for Claude Opus 4.1. All non-zero scores are rounded to at least 0.1.

Model	CodeQA	BrowseComp+ (1K)	OOLONG	OOLONG-Pairs
Task Length N (tokens)	23K-4.2M	6M-11M	131K	32K
GPT-5 (with RLM sub-calls to GPT-5-mini)				
Base Model	24.0* ($0.13 ± $0.07)	0.0* (N/A) ± (N/A)	44.0 ($0.14 ± $0.02)	0.1 ($0.16 ± $0.10)
CodeAct (+ BM25)	22.0* ($0.06 ± $0.08)	51.0 ($0.71 ± $1.20)	38.0 ($0.61 ± $1.06)	24.7 ($0.75 ± $0.43)
CodeAct (+ sub-calls)	24.0* ($0.06 ± $0.08)	0.0* (N/A) ± (N/A)	40.0 ($0.85 ± $1.27)	28.4 ($1.11 ± $0.62)
Compaction agent	58.0 ($1.31 ± $1.46)	70.5 ($0.57 ± $0.10)	46.0 ($0.13 ± $0.01)	0.1 ($0.13 ± $0.09)
OpenCode	18.0* (N/A) ± (N/A)	0.0* (N/A) ± (N/A)	32.0 (N/A) ± (N/A)	3.1 (N/A) ± (N/A)
OpenCode (+ context offloading)	64.0 (N/A) ± (N/A)	**94.0** (N/A) ± (N/A)	52.0 (N/A) ± (N/A)	4.8 (N/A) ± (N/A)
RLM (recursion depth=0)	58.0 ($0.18 ± $0.56)	88.0 ($0.44 ± $0.90)	36.0 ($0.37 ± $0.42)	43.9 ($0.69 ± $1.16)
RLM (recursion depth=1)	62.0 ($0.11 ± $0.10)	91.3 ($0.99 ± $1.22)	56.0 ($0.43 ± $0.85)	58.0 ($0.33 ± $0.20)
RLM (recursion depth=2)	**66.0** ($0.15 ± $0.30)	92.0 ($0.55 ± $0.69)	56.5 ($1.10 ± $3.25)	65.5 ($0.33 ± $0.44)
RLM (recursion depth=3)	58.0 ($0.15 ± $0.27)	92.0 ($0.51 ± $0.54)	**58.0** ($0.51 ± $0.54)	**76.0** ($0.39 ± $0.32)
Qwen3-Coder-480B-A35B				
Base Model	20.0* ($0.13 ± $0.08)	0.0* (N/A) ± (N/A)	36.0 ($0.06 ± $0.00)	0.1 ($0.05 ± $0.01)
CodeAct (+ BM25)	24.0* ($0.17 ± $0.08)	12.7 ($0.39 ± $0.50)	38.0 ($1.51 ± $1.09)	0.3 ($1.54 ± $0.35)
CodeAct (+ sub-calls)	26.0* ($0.28 ± $0.30)	0.0* (N/A) ± (N/A)	32.0 ($1.83 ± $1.14)	0.1 ($1.49 ± $0.46)
Compaction agent	50.0 ($1.26 ± $1.50)	38.0 ($8.98 ± $2.12)	44.1 ($0.15 ± $0.01)	0.31 ($0.05 ± $0.00)
OpenCode	12.0* (N/A) ± (N/A)	0.0* (N/A) ± (N/A)	36.0 (N/A) ± (N/A)	0.0 (N/A) ± (N/A)
OpenCode (+ context offloading)	40.0 (N/A) ± (N/A)	58.0 (N/A) ± (N/A)	24.0 (N/A) ± (N/A)	2.1 (N/A) ± (N/A)
RLM (recursion depth=0)	**66.0** ($0.18 ± $0.58)	46.0 ($0.82 ± $0.69)	43.5 ($0.32 ± $0.13)	17.3 ($1.77 ± $1.23)
RLM (recursion depth=1)	56.0 ($0.92 ± $1.23)	44.7 ($0.84 ± $0.63)	**48.0** ($0.61 ± $0.49)	**23.1** ($1.02 ± $0.52)
RLM (recursion depth=2)	54.0 ($1.88 ± $3.30)	68.0 ($1.05 ± $0.67)	26.0 ($1.03 ± $1.65)	19.0 ($1.61 ± $0.99)
RLM (recursion depth=3)	44.0 ($1.65 ± $1.63)	**68.7** ($1.10 ± $0.80)	32.0 ($0.80 ± $1.03)	21.1 ($1.67 ± $1.21)
Claude Opus 4.1				
Claude Code	12.0* ($2.03 ± $0.57)	0.0* (N/A) ± (N/A)	40.2 ($3.43 ± $1.60)	0.1 ($6.75 ± $3.57)
Claude Code (+ context offloading)	62.0 ($1.25 ± $0.54)	84.0 ($2.03 ± $1.49)	48.0 ($0.98 ± $0.55)	6.5 ($2.99 ± $1.16)

Observation 1: RLMs can scale to the 10M+ token regime and can outperform base LMs and existing task-agnostic agent scaffolds on long context tasks. Across all tasks, RLMs demonstrate strong performance on prompts well beyond the effective context window of a frontier LM, outperforming base models and common long-context scaffolds by up to $2\times$ the performance while maintaining comparable or cheaper average token costs. Notably, RLMs scale well beyond the base models' context window. For instance, on BrowseComp-Plus (1K), a linearly extrapolated cost for GPT-5-mini ingesting 6-11M input tokens is $\$1.50 - \$2.75$, while RLM(GPT-5, depth=1) has an average cost of $\$0.99$ and outperforms both the compaction and retrieval baselines by over 29%.

Furthermore, on tasks where processing costs scale with the input context, RLMs make significant improvements over the base model, even on tasks within the model's context window. On OOLONG, the RLM(depth=1) with GPT-5 and Qwen3-Coder outperform the base model by 27.3% and 33.3% respectively. On OOLONG-Pairs, both GPT-5 and Qwen3-Coder make little progress with F1 scores of $\leq 0.1\%$, while the RLM(depth=1) using these models achieve F1 scores of 58.0% and 23.1% respectively, highlighting the capability of RLMs to handle extremely information-dense tasks.

Observation 2: The REPL is necessary for handling long inputs, while the recursive sub-calling of RLMs provides strong benefits on information-dense inputs. A key characteristic of RLMs is offloading the context as a variable in an environment $\mathcal{E}$ that the model can interact with. In particular, RLM(depth=0) and coding agents like Claude Code and OpenCode are able to scale beyond the context limit of the model and outperform other task-agnostic baselines on most long context settings.

On CodeQA in particular with Qwen3-Coder-480B-A35B, the no-sub-calling RLM(depth=0) is able to outperform all sub-calling variants of the RLM.

On information-dense tasks like OOLONG or OOLONG-Pairs, we observed several cases where programmatic recursive LM sub-calling is necessary. In §5, we see RLM(Qwen3-Coder) perform the necessary semantic transformation line-by-line through recursive sub-calls, while the ablation without sub-calls is forced to use keyword heuristics to solve these tasks. On OOLONG-Pairs in particular, the higher recursive depth variants of the RLM for GPT-5 outperform all other methods including Claude Code and OpenCode by a large margin.

Observation 3: LM performance degrades as a function of input length and problem complexity, while RLM performance scales better. The benchmarks S-NIAH, OOLONG, and OOLONG-Pairs contain a fixed number of tasks over contexts with lengths ranging from 2^{13} to 2^{20}. Each benchmark can be categorized by different processing complexity of the input context with respect to length (roughly constant, linear, and quadratic respectively). In Figure 1, we directly compare an RLM(GPT-5, depth=1) to base GPT-5, and find that GPT-5 performance degrades significantly faster for more complex tasks, which aligns with the findings of Goldman et al. [2025], while RLM performance degrades at a slower rate. For context lengths beyond 2^{14}, the RLM consistently outperforms GPT-5.

Furthermore, RLM costs scale proportionally to the complexity of the task, while still remaining in the same order of magnitude of cost as GPT-5 (see Figure 16 in Appendix F). In §5, we explore the choices that the RLM makes that cause these differences in cost.

Observation 4: The inference cost of RLMs remains comparable to other methods, and in some cases base LM calls. On average, we find in Table 1 that the inference cost of RLMs is cheaper or comparable to most other baselines, including standard coding agents. Furthermore, in Figure 11 in Appendix F, we find that the median RLM run is cheaper than the median base model run, but more expensive on average due to outlier trajectories where the RLM struggles to find an answer.

We additionally report runtime numbers of each method in Figures 12, 13 in Appendix F, but we note several important caveats. Unlike API costs, these numbers are heavily dependent on implementation details such as the machine used, API request latency, and the asynchrony of LM calls. In our implementation of the baselines and RLMs, all LM calls are blocking / sequential. Nevertheless, similar to costs, we observe a wide range of runtimes, especially for RLMs.

Table 2: Solve rate on LONGCOT-MINI [Motwani et al., 2026], a difficult long reasoning benchmark that frontier models struggle to solve. We select the best performing model from the paper (GPT-5.2) and compare to an RLM with and without decomposition hints (prompt provided in Appendix C.3).

Model	Overall	MATH	CHEM	CS	LOGIC	CHESS
GPT-5.2 (base)	38.7	26.0	37.0	40.4	53.6	36.6
RLM (GPT-5.2, recursion depth=1)	50.6	5.6	50.0	11.0	86.7	93.0
RLM (GPT-5.2, recursion depth=1) + decomposition hints	**65.6**	**32.0**	**52.0**	**46.0**	**99.0**	**99.0**

Observation 5: Beyond long-context, RLMs enable longer reasoning capabilities. In Table 2, we report RLM performance on LongCoT-mini [Motwani et al., 2026], a challenging long reasoning benchmark where frontier models solve compositional problems containing interdependent subproblems. We compare with the best model reported in the paper, GPT-5.2, and find that RLM(GPT-5.2, depth=1) uses the REPL to outperform the base model. Furthermore, when providing explicit hints on how to decompose tasks, we find the RLM is able to reliably generate a graph of the problem, solving each node using sub-calls as it programmatically traverses the reasoning graph. It outperforms the base model on all domains and by a 69.5% performance increase overall.

Observation 6: Training RLMs on one domain can improve general downstream RLM performance, as well as efficiency. Training also exhibits length generalization. Certain behaviors in RLM trajectories are common among different domains, such as probing the input and recursively sub-calling on shorter contexts. In Figure 3(a), we find that **RLM-Qwen3-8B**, a Qwen3-8B model that we fine-tuned on RLM(Qwen3-Coder-480B-A35B) trajectories on a small, *unrelated* set of tasks (LongBenchPro; Chen et al. 2026) considerably outperforms the base Qwen3-8B as a RLM across all tasks. Furthermore, its inference costs are much lower and more than $3\times$ faster (see Figure 6 in Appendix A) due to better decision making and fewer mistakes as a RLM. Furthermore, we find that training RLMs exhibits length generalization; in Figure 3(b), we train Qwen3-4B-Instruct-2507

as an RLM(depth=1) on MRCRv2 [Vodrahalli et al., 2024], a synthetic long-context task where the model must count and reproduce instances of a body of text in a corpus. By purely training through reinforcement learning with verifiable rewards (RLVR) on a smaller split, we find that RLM(Qwen3-4B-Instruct-2507) is able to generalize to the longer, more difficult split.

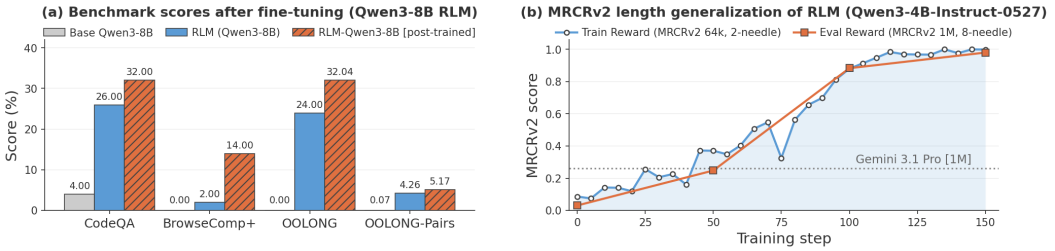


Figure 3: (a) We show how rejection fine-tuning Qwen3-8B on distilled RLM(Qwen3-Coder-480B-A35B) trajectories improves performance on the benchmarks in Table 1. (b) On MRCRv2 [Vodrahalli et al., 2024], RL training RLM(Qwen3-4B-2507-Instruct) on the 64k sequence length, 2-needle split generalizes to the 1M, 8-needle split. We also show the 1M, 8-needle score for a 1M-context frontier model (Gemini 3.1 Pro [Google Gemini Team, 2026]).

5 Analyses of RLM Trajectories

RLMs exhibit interesting context and problem decomposition behavior. We discuss observable behavior in small and large LLMs as RLMs to understand how we can steer and improve their performance and efficiency through training and prompt tuning.

Observed RLM decomposition patterns. Current models as RLMs attempt to probe, then decompose a task into sub-tasks for recursive sub-calls to solve. In many cases such as on BrowseComp-Plus, the LM uses model priors to programmatically narrow the search space of sub-calls. RLMs are also able to output beyond their context window by stitching together sub-LM calls inside the REPL, which is required to solve tasks like OOLONG-Pairs. We detail particular trajectories in Appendix E.

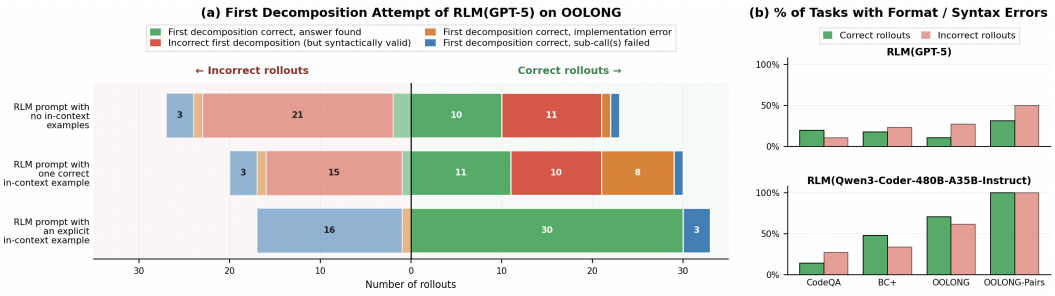


Figure 4: (a) On OOLONG, we report the performance of RLM(GPT-5) by varying the in-context examples provided in the system prompt. For each rollout, we categorize the first task decomposition attempt made by the RLM. (b) From the RLM(depth=1) runs in Table 1, we report, bucketed by correct or incorrect rollouts, the percentage of RLM trajectories with at least one syntax error.

First decomposition and errors in RLM trajectories. RLMs defer essentially unbounded-length reasoning chains to sub-LM calls. The choice of decomposition can greatly affect task performance, especially for information-dense problems. In Figure 4(a), we ablate how sensitive RLM behavior is to in-context decomposition examples in its system prompt on OOLONG. We find that in-context RLM trajectories greatly improve both overall performance and the initial decomposition attempt made by the RLM, even if the example is unrelated to the actual task. Furthermore, while RLMs frequently recover from an initially incorrect decomposition pattern, we find that the first decomposition attempt is important for overall performance. In Figure 4(b), we plot how many RLM(depth=1) trajectories in Table 1 contains syntax errors. We find that RLM(Qwen3-Coder) trajectories contain significantly more syntax errors, even for correct trajectories, compared to RLM(GPT-5). These errors explain why

higher recursion depths for RLM(Qwen3-Coder) *perform worse on average*: Qwen3-Coder-480B-A35B often makes syntax errors that result in failed outputs, and having sub-RLM calls propagates this issue to sub-calls. We include additional analysis for erroneous RLM behavior in Appendix F.1.

6 Related Works

Long-Context LM Systems. There have primarily been two orthogonal directions for long-context management in language model systems: 1) directly changing the architecture of and retraining the base LM to handle longer contexts [Press et al., 2022, Gu et al., 2022, Munkhdalai et al., 2024], and 2) building a scaffold around the LM that implicitly handles the context – RLMs focus on the latter. One popular class of such strategies is *lossy* context management [Chen et al., 2023], which uses compaction or truncation to compress the input context at the cost of potentially losing fine-grained information. For example, ReSum [Wu et al., 2025] adds a summarization tool to periodically compress the context of a multi-turn agent. Another class of strategies implement an explicit memory hierarchy in the agent scaffold [Packer et al., 2024, Chhikara et al., 2025, Zhang et al., 2025]. RLMs differ from these works in that all context window management is implicitly handled by the LM itself.

Task Decomposition through sub-LM calls. Many LM-based agents [Guo et al., 2024, Anthropic, 2025] use multiple, well-placed LM calls to solve a problem; however, many of these calls are placed based on human-engineered workflows. Several methods like ViperGPT [Surís et al., 2023], THREAD [Schroeder et al., 2025], ReDel [Zhu et al., 2024], Context Folding [Sun et al., 2025], and AgentFold [Ye et al., 2025] have explored deferring the choice of sub-LM calls to the LM. These techniques emphasize *task* decomposition through recursive LM calls, but are unable to handle long context inputs beyond the length of the base LM.

Other works introduce code as the primary driver of task decomposition. LLMVM [Pobar, 2024] is an early CodeAct-style CLI tool that implements programmatic sub-LLMVM agents, but does not explicitly offload or treat context as the variable of interest. DisCIPL [Grand et al., 2025] is an abstraction that uses a planner LLM to generate programs that propose and score candidate completions from a follower LM. While DisCIPL uses code to specify an output-generation procedure, RLMs place the input prompt itself in a persistent REPL as an externally accessible variable. This property enables RLMs to *symbolically* manipulate arbitrarily long context and to iteratively refine their recursion strategies via execution feedback from the persistent REPL.

7 Limitations and Future Work

While RLMs show strong performance on tasks beyond the context window limitations of existing LMs at reasonable inference costs, evaluations for more difficult and natural long-context processing tasks and the best mechanisms for implementing guardrails for RLMs both remain highly under-explored. Broadly, RLMs add a layer of complexity on top of existing LMs that may lead to unintentional side-effects like exploding sub-call costs, which we leave for future work to solve. We also note that future strategies involving asynchronous sub-calls and sandboxed REPLs can potentially significantly reduce the runtime and inference cost of RLMs, but further contribute to this complexity. We include additional limitations and negative results in Appendix B.

Lastly, we focused our experiments on evaluating RLMs using *existing* frontier models, but show initial evidence on a Qwen3-8B model that explicit training as a RLM provides very rapid performance improvements, even outside the training domain. We hypothesize that RLM trajectories can be viewed as a form of reasoning [OpenAI et al., 2024, DeepSeek-AI et al., 2025], which can be trained by bootstrapping existing models [Zelikman et al., 2022, 2024]. We hope that training native RLMs can be treated as a new axis of scale to improve LM performance on general and long-horizon tasks.

8 Conclusion

We introduced Recursive Language Models (RLMs), a general inference framework for language models that offloads the input context and enables language models to recursively sub-query language models before providing an output. We explored an instantiation of this framework that offloads the context into a Python REPL environment as a variable in memory, enabling the LM to reason over its context in code and recursive LM calls, rather than purely in token space. Our results across

multiple settings and models demonstrated that RLMs are an effective task-agnostic paradigm for both long-context problems and general reasoning. Building on our small fine-tuning experiments, we are excited to see future work that explicitly trains models to reason as RLMs, which could result in another axis of scale for the next generation of language model systems.

Acknowledgments

This research is partially supported by the Laude Institute, Prime Intellect, and Modal Labs. We thank Noah Ziems, Jacob Li, James Moore, and the MIT OASYS and MIT DSG labs for insightful discussions throughout this project. We also thank Jack Cook, Matej Sirovatka, Ofir Press, Sebastian Müller, Simon Guo, and Zed Li for helpful feedback.

References

Anomaly. opencode: The open source ai coding agent, 2026. URL https://github.com/anomalyco/opencode.

Anthropic. Claude code: Subagents — modular ai workflows with isolated agent contexts, 2025. URL https://docs.anthropic.com/en/docs/claude-code/sub-agents.

Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. Longbench v2: Towards deeper understanding and reasoning on realistic long-context multitasks, 2025. URL https://arxiv.org/abs/2412.15204.

Amanda Bertsch, Adithya Pratapa, Teruko Mitamura, Graham Neubig, and Matthew R. Gormley. Oolong: Evaluating long context reasoning and aggregation capabilities, 2025. URL https://arxiv.org/abs/2511.02817.

Yapei Chang, Kyle Lo, Tanya Goyal, and Mohit Iyyer. Booookscore: A systematic exploration of book-length summarization in the era of LLMs. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://arxiv.org/pdf/2310.00785.pdf.

Howard Chen, Ramakanth Pasunuru, Jason Weston, and Asli Celikyilmaz. Walking down the memory maze: Beyond context limit through interactive reading, 2023. URL https://arxiv.org/abs/2310.05029.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

Ziyang Chen, Xing Wu, Junlong Jia, Chaochen Gao, Qi Fu, Debing Zhang, and Songlin Hu. Longbench pro: A more realistic and comprehensive bilingual long-context evaluation benchmark, 2026. URL https://arxiv.org/abs/2601.02872.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL https://arxiv.org/abs/2504.19413.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang,

Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Fireworks AI. Qwen3 coder 480b a35b instruct. https://fireworks.ai/models/fireworks/qwen3-coder-480b-a35b-instruct, 2025.

Omer Goldman, Alon Jacovi, Aviv Slobodkin, Aviya Maimon, Ido Dagan, and Reut Tsarfaty. Is it really long context if all you need is retrieval? towards genuinely difficult long context nlp, 2025. URL https://arxiv.org/abs/2407.00402.

Google Gemini Team. Gemini 3.1 pro: A smarter model for your most complex tasks, February 2026. URL https://blog.google/innovation-and-ai/models-and-research/gemini-models/gemini-3-1-pro/. Accessed: 2026-05-05.

Gabriel Grand, Joshua B Tenenbaum, Vikash K Mansinghka, Alexander K Lew, and Jacob Andreas. Self-steering language models. *arXiv preprint arXiv:2504.07081*, 2025.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces, 2022. URL https://arxiv.org/abs/2111.00396.

Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges, 2024. URL https://arxiv.org/abs/2402.01680.

Kelly Hong, Anton Troynikov, and Jeff Huber. Context rot: How context degradation affects llm performance, 2025. URL https://research.trychroma.com/context-rot.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models?, 2024. URL https://arxiv.org/abs/2404.06654.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Omar Khattab, Christopher Potts, and Matei Zaharia. Baleen: Robust multi-hop reasoning at scale via condensed retrieval. *Advances in Neural Information Processing Systems*, 34:27670–27682, 2021.

William Merrill and Ashish Sabharwal. The expressive power of transformers with chain of thought. In *The Twelfth International Conference on Learning Representations*, 2024.

Sumeet Ramesh Motwani, Daniel Nichols, Charles London, Peggy Li, Fabio Pizzati, Acer Blake, Hasan Hammoud, Tavish McDonald, Akshat Naik, Alesia Ivanova, Vignesh Baskaran, Ivan Laptev, Ruben Glatt, Tal Ben-Nun, Philip Torr, Natasha Jaques, Ameya Prabhu, Brian Bartoldson, Bhavya Kailkhura, and Christian Schroeder de Witt. Longcot: Benchmarking long-horizon chain-of-thought reasoning, 2026. URL https://arxiv.org/abs/2604.14140.

Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention, 2024. URL https://arxiv.org/abs/2404.07143.

OpenAI. Deep research, 2025a. URL https://openai.com/index/introducing-deep-research/. AI-powered research assistant tool.

OpenAI. Codex cli: A lightweight coding agent for your terminal, 2025b. URL https://developers.openai.com/codex/cli/.

OpenAI, Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, Alex Iftimie, Alex Karpenko, Alex Tachard Passos, Alexander Neitz, Alexander Prokofiev, Alexander Wei, Allison Tam, Ally Bennett, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrew Duberstein, Andrew Kondrich, Andrey Mishchenko, Andy Applebaum, Angela Jiang, Ashvin Nair, Barret Zoph, Behrooz Ghorbani, Ben Rossen, Benjamin Sokolowsky, Boaz Barak, Bob McGrew, Borys Minaiev, Botao Hao, Bowen Baker, Brandon Houghton, Brandon McKinzie, Brydon Eastman, Camillo Lugaresi, Cary Bassin, Cary Hudson, Chak Ming Li, Charles de Bourcy, Chelsea Voss, Chen Shen, Chong Zhang, Chris Koch, Chris Orsinger, Christopher Hesse, Claudia Fischer, Clive Chan, Dan Roberts, Daniel Kappler, Daniel Levy, Daniel Selsam, David Dohan, David Farhi, David Mely, David Robinson, Dimitris Tsipras, Doug Li, Dragos Oprica, Eben Freeman, Eddie Zhang, Edmund Wong, Elizabeth Proehl, Enoch Cheung, Eric Mitchell, Eric Wallace, Erik Ritter, Evan Mays, Fan Wang, Felipe Petroski Such, Filippo Raso, Florencia Leoni, Foivos Tsimpourlas, Francis Song, Fred von Lohmann, Freddie Sulit, Geoff Salmon, Giambattista Parascandolo, Gildas Chabot, Grace Zhao, Greg Brockman, Guillaume Leclerc, Hadi Salman, Haiming Bao, Hao Sheng, Hart Andrin, Hessam Bagherinezhad, Hongyu Ren, Hunter Lightman, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian Osband, Ignasi Clavera Gilaberte, Ilge Akkaya, Ilya Kostrikov, Ilya Sutskever, Irina Kofman, Jakub Pachocki, James Lennon, Jason Wei, Jean Harb, Jerry Twore, Jiacheng Feng, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joaquin Quiñonero Candela, Joe Palermo, Joel Parish, Johannes Heidecke, John Hallman, John Rizzo, Jonathan Gordon, Jonathan Uesato, Jonathan Ward, Joost Huizinga, Julie Wang, Kai Chen, Kai Xiao, Karan Singhal, Karina Nguyen, Karl Cobbe, Katy Shi, Kayla Wood, Kendra Rimbach, Keren Gu-Lemberg, Kevin Liu, Kevin Lu, Kevin Stone, Kevin Yu, Lama Ahmad, Lauren Yang, Leo Liu, Leon Maksin, Leyton Ho, Liam Fedus, Lilian Weng, Linden Li, Lindsay McCallum, Lindsey Held, Lorenz Kuhn, Lukas Kondraciuk, Lukasz Kaiser, Luke Metz, Madelaine Boyd, Maja Trebacz, Manas Joglekar, Mark Chen, Marko Tintor, Mason Meyer, Matt Jones, Matt Kaufer, Max Schwarzer, Meghan Shah, Mehmet Yatbaz, Melody Y. Guan, Mengyuan Xu, Mengyuan Yan, Mia Glaese, Mianna Chen, Michael Lampe, Michael Malek, Michele Wang, Michelle Fradin, Mike McClay, Mikhail Pavlov, Miles Wang, Mingxuan Wang, Mira Murati, Mo Bavarian, Mostafa Rohaninejad, Nat McAleese, Neil Chowdhury, Neil Chowdhury, Nick Ryder, Nikolas Tezak, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Patrick Chao, Paul Ashbourne, Pavel Izmailov, Peter Zhokhov, Rachel Dias, Rahul Arora, Randall Lin, Rapha Gontijo Lopes, Raz Gaon, Reah Miyara, Reimar Leike, Renny Hwang, Rhythm Garg, Robin Brown, Roshan James, Rui Shu, Ryan Cheu, Ryan Greene, Saachi Jain, Sam Altman, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Santiago Hernandez, Sasha Baker, Scott McKinney, Scottie Yan, Shengjia Zhao, Shengli Hu, Shibani Santurkar, Shraman Ray Chaudhuri, Shuyuan Zhang, Siyuan Fu, Spencer Papay, Steph Lin, Suchir Balaji, Suvansh Sanjeev, Szymon Sidor, Tal Broda, Aidan Clark, Tao Wang, Taylor Gordon, Ted Sanders, Tejal Patwardhan, Thibault Sottiaux, Thomas Degry, Thomas Dimson, Tianhao Zheng, Timur Garipov, Tom Stasi, Trapit Bansal, Trevor Creech, Troy Peterson, Tyna Eloundou, Valerie Qi, Vineet Kosaraju, Vinnie Monaco, Vitchyr Pong, Vlad Fomenko, Weiyi Zheng, Wenda Zhou, Wes McCabe, Wojciech Zaremba, Yann Dubois, Yinghai Lu, Yining Chen, Young Cha, Yu Bai, Yuchen He, Yuchen Zhang, Yunyun Wang, Zheng Shao, and Zhuohan Li. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2024. URL https://arxiv.org/abs/2310.08560.

Joel Pobar. LLMVM: An llm $\leftrightarrow$ python agentic runtime prototype. https://github.com, 2024.

Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation, 2022. URL https://arxiv.org/abs/2108.12409.

Prime Intellect Team, Mika Senghaas, Fares Obeid, Sami Jaghouar, William Brown, Jack Min Ong, Daniel Auras, Matej Sirovatka, Jannik Straube, Andrew Baker, Sebastian Müller, Justus Mattern, Manveer Basra, Aiman Ismail, Dominik Scherm, Cooper Miller, Ameen Patel, Simon Kirsten, Mario Sieg, Christian Reetz, Kemal Erdem, Vincent Weisser, and Johannes Hagemann. Intellect-3: Technical report, 2025. URL https://arxiv.org/abs/2512.16144.

Qwen Team. Qwen3-8b. https://huggingface.co/Qwen/Qwen3-8B, 2025a.

Qwen Team. Qwen3-coder-480b-a35b-instruct. https://huggingface.co/Qwen/Qwen3-Coder-480B-A35B-Instruct, 2025b.

Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, 2018. URL https://arxiv.org/abs/1804.02767.

Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April 2009. ISSN 1554-0669. doi: 10.1561/1500000019. URL https://doi.org/10.1561/1500000019.

Philip Schroeder, Nathaniel Morgan, Hongyin Luo, and James Glass. Thread: Thinking deeper with recursive spawning, 2025. URL https://arxiv.org/abs/2405.17402.

Sentient AI. Roma: The backbone for open-source meta-agents, November 2025. URL https://blog.sentient.xyz/posts/recursive-open-meta-agent. Accessed: 2025-12-20.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, Akshay Nathan, Alan Luo, Alec Helyar, Aleksander Madry, Aleksandr Efremov, Aleksandra Spyra, Alex Baker-Whitcomb, Alex Beutel, Alex Karpenko, Alex Makelov, Alex Neitz, Alex Wei, Alexandra Barr, Alexandre Kirchmeyer, Alexey Ivanov, Alexi Christakis, Alistair Gillespie, Allison Tam, Ally Bennett, Alvin Wan, Alyssa Huang, Amy McDonald Sandjideh, Amy Yang, Ananya Kumar, Andre Saraiva, Andrea Vallone, Andrei Gheorghe, Andres Garcia Garcia, Andrew Braunstein, Andrew Liu, Andrew Schmidt, Andrey Mereskin, Andrey Mishchenko, Andy Applebaum, Andy Rogerson, Ann Rajan, Annie Wei, Anoop Kotha, Anubha Srivastava, Anushree Agrawal, Arun Vijayvergiya, Ashley Tyra, Ashvin Nair, Avi Nayak, Ben Eggers, Bessie Ji, Beth Hoover, Bill Chen, Blair Chen, Boaz Barak, Borys Minaiev, Botao Hao, Bowen Baker, Brad Lightcap, Brandon McKinzie, Brandon Wang, Brendan Quinn, Brian Fioca, Brian Hsu, Brian Yang, Brian Yu, Brian Zhang, Brittany Brenner, Callie Riggins Zetino, Cameron Raymond, Camillo Lugaresi, Carolina Paz, Cary Hudson, Cedric Whitney, Chak Li, Charles Chen, Charlotte Cole, Chelsea Voss, Chen Ding, Chen Shen, Chengdu Huang, Chris Colby, Chris Hallacy, Chris Koch, Chris Lu, Christina Kaplan, Christina Kim, CJ Minott-Henriques, Cliff Frey, Cody Yu, Coley Czarnecki, Colin Reid, Colin Wei, Cory Decareaux, Cristina Scheau, Cyril Zhang, Cyrus Forbes, Da Tang, Dakota Goldberg, Dan Roberts, Dana Palmie, Daniel Kappler, Daniel Levine, Daniel Wright, Dave Leo, David Lin, David Robinson, Declan Grabb, Derek Chen, Derek Lim, Derek Salama, Dibya Bhattacharjee, Dimitris Tsipras, Dinghua Li, Dingli Yu, DJ Strouse, Drew Williams, Dylan Hunn, Ed Bayes, Edwin Arbus, Ekin Akyurek, Elaine Ya Le, Elana Widmann, Eli Yani, Elizabeth Proehl, Enis Sert, Enoch Cheung, Eri Schwartz, Eric Han, Eric Jiang, Eric Mitchell, Eric Sigler, Eric Wallace, Erik Ritter, Erin Kavanaugh, Evan Mays, Evgenii Nikishin, Fangyuan Li, Felipe Petroski Such, Filipe de Avila Belbute Peres, Filippo Raso, Florent Bekerman, Foivos Tsimpourlas, Fotis Chantzis, Francis Song, Francis Zhang, Gaby Raila, Garrett McGrath, Gary Briggs, Gary Yang, Giambattista Parascandolo, Gildas Chabot, Grace Kim, Grace Zhao, Gregory Valiant, Guillaume Leclerc, Hadi Salman, Hanson Wang, Hao Sheng, Haoming Jiang, Haoyu Wang, Haozhun Jin, Harshit Sikchi, Heather Schmidt, Henry Aspegren, Honglin Chen, Huida Qiu, Hunter Lightman, Ian Covert, Ian Kivlichan, Ian Silber, Ian Sohl, Ibrahim Hammoud, Ignasi Clavera, Ikai Lan, Ilge Akkaya, Ilya Kostrikov, Irina Kofman, Isak Etinger, Ishaan Singal, Jackie Hehir, Jacob Huh, Jacqueline Pan, Jake Wilczynski, Jakub Pachocki, James Lee, James Quinn, Jamie Kiros, Janvi Kalra, Jasmyn Samaroo, Jason Wang, Jason Wolfe, Jay Chen, Jay Wang, Jean Harb, Jeffrey Han, Jeffrey Wang, Jennifer Zhao, Jeremy Chen, Jerene Yang, Jerry Tworek, Jesse Chand, Jessica Landon, Jessica Liang, Ji Lin, Jiancheng Liu, Jianfeng Wang, Jie Tang, Jihan Yin, Joanne Jang, Joel Morris, Joey Flynn, Johannes Ferstad, Johannes Heidecke, John Fishbein, John Hallman, Jonah Grant, Jonathan Chien, Jonathan Gordon, Jongsoo Park, Jordan Liss, Jos Kraaijeveld, Joseph Guay, Joseph Mo, Josh Lawson, Josh McGrath, Joshua Vendrow, Joy Jiao, Julian Lee, Julie Steele, Julie Wang, Junhua Mao, Kai Chen,

Kai Hayashi, Kai Xiao, Kamyar Salahi, Kan Wu, Karan Sekhri, Karan Sharma, Karan Singhal, Karen Li, Kenny Nguyen, Keren Gu-Lemberg, Kevin King, Kevin Liu, Kevin Stone, Kevin Yu, Kristen Ying, Kristian Georgiev, Kristie Lim, Kushal Tirumala, Kyle Miller, Lama Ahmad, Larry Lv, Laura Clare, Laurance Fauconnet, Lauren Itow, Lauren Yang, Laurentia Romaniuk, Leah Anise, Lee Byron, Leher Pathak, Leon Maksin, Leyan Lo, Leyton Ho, Li Jing, Liang Wu, Liang Xiong, Lien Mamitsuka, Lin Yang, Lindsay McCallum, Lindsey Held, Liz Bourgeois, Logan Engstrom, Lorenz Kuhn, Louis Feuvrier, Lu Zhang, Lucas Switzer, Lukas Kondraciuk, Lukasz Kaiser, Manas Joglekar, Mandeep Singh, Mandip Shah, Manuka Stratta, Marcus Williams, Mark Chen, Mark Sun, Marselus Cayton, Martin Li, Marvin Zhang, Marwan Aljubeh, Matt Nichols, Matthew Haines, Max Schwarzer, Mayank Gupta, Meghan Shah, Melody Huang, Meng Dong, Mengqing Wang, Mia Glaese, Micah Carroll, Michael Lampe, Michael Malek, Michael Sharman, Michael Zhang, Michele Wang, Michelle Pokrass, Mihai Florian, Mikhail Pavlov, Miles Wang, Ming Chen, Mingxuan Wang, Minnia Feng, Mo Bavarian, Molly Lin, Moose Abdool, Mostafa Rohaninejad, Nacho Soto, Natalie Staudacher, Natan LaFontaine, Nathan Marwell, Nelson Liu, Nick Preston, Nick Turley, Nicklas Ansman, Nicole Blades, Nikil Pancha, Nikita Mikhaylin, Niko Felix, Nikunj Handa, Nishant Rai, Nitish Keskar, Noam Brown, Ofir Nachum, Oleg Boiko, Oleg Murk, Olivia Watkins, Oona Gleeson, Pamela Mishkin, Patryk Lesiewicz, Paul Baltescu, Pavel Belov, Peter Zhokhov, Philip Pronin, Phillip Guo, Phoebe Thacker, Qi Liu, Qiming Yuan, Qinghua Liu, Rachel Dias, Rachel Puckett, Rahul Arora, Ravi Teja Mullapudi, Raz Gaon, Reah Miyara, Rennie Song, Rishabh Aggarwal, RJ Marsan, Robel Yemiru, Robert Xiong, Rohan Kshirsagar, Rohan Nuttall, Roman Tsiupa, Ronen Eldan, Rose Wang, Roshan James, Roy Ziv, Rui Shu, Ruslan Nigmatullin, Saachi Jain, Saam Talaie, Sam Altman, Sam Arnesen, Sam Toizer, Sam Toyer, Samuel Miserendino, Sandhini Agarwal, Sarah Yoo, Savannah Heon, Scott Ethersmith, Sean Grove, Sean Taylor, Sebastien Bubeck, Sever Banesiu, Shaokyi Amdo, Shengjia Zhao, Sherwin Wu, Shibani Santurkar, Shiyu Zhao, Shraman Ray Chaudhuri, Shreyas Krishnaswamy, Shuaiqi, Xia, Shuyang Cheng, Shyamal Anadkat, Simón Posada Fishman, Simon Tobin, Siyuan Fu, Somay Jain, Song Mei, Sonya Egoian, Spencer Kim, Spug Golden, SQ Mah, Steph Lin, Stephen Imm, Steve Sharpe, Steve Yadlowsky, Sulman Choudhry, Sungwon Eum, Suvansh Sanjeev, Tabarak Khan, Tal Stramer, Tao Wang, Tao Xin, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Degry, Thomas Shadwell, Tianfu Fu, Tianshi Gao, Timur Garipov, Tina Sriskandarajah, Toki Sherbakov, Tomer Kaftan, Tomo Hiratsuka, Tongzhou Wang, Tony Song, Tony Zhao, Troy Peterson, Val Kharitonov, Victoria Chernova, Vineet Kosaraju, Vishal Kuo, Vitchyr Pong, Vivek Verma, Vlad Petrov, Wanning Jiang, Weixing Zhang, Wenda Zhou, Wenlei Xie, Wenting Zhan, Wes McCabe, Will DePue, Will Ellsworth, Wulfie Bain, Wyatt Thompson, Xiangning Chen, Xiangyu Qi, Xin Xiang, Xinwei Shi, Yann Dubois, Yaodong Yu, Yara Khakbaz, Yifan Wu, Yilei Qian, Yin Tat Lee, Yinbo Chen, Yizhen Zhang, Yizhong Xiong, Yonglong Tian, Young Cha, Yu Bai, Yu Yang, Yuan Yuan, Yuanzhi Li, Yufeng Zhang, Yuguang Yang, Yujia Jin, Yun Jiang, Yunyun Wang, Yushi Wang, Yutian Liu, Zach Stubenvoll, Zehao Dou, Zheng Wu, and Zhigang Wang. Openai gpt-5 system card, 2025. URL https://arxiv.org/abs/2601.03267.

Calvin Smith. Openhands context condensensation for more efficient ai agents, 2025. URL https://openhands.dev/blog/openhands-context-condensensation-for-more-efficient-ai-agents.

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding, 2025. URL https://arxiv.org/abs/2510.11967.

Dídac Surís, Sachit Menon, and Carl Vondrick. Vipergpt: Visual inference via python execution for reasoning. *Proceedings of IEEE International Conference on Computer Vision (ICCV)*, 2023.

Kiran Vodrahalli, Santiago Ontanon, Nilesh Tripuraneni, Kelvin Xu, Sanil Jain, Rakesh Shivanna, Jeffrey Hui, Nishanth Dikkala, Mehran Kazemi, Bahare Fatemi, Rohan Anil, Ethan Dyer, Siamak Shakeri, Roopali Vij, Harsh Mehta, Vinay Ramasesh, Quoc Le, Ed Chi, Yifeng Lu, Orhan Firat, Angeliki Lazaridou, Jean-Baptiste Lespiau, Nithya Attaluri, and Kate Olszewska. Michelangelo: Long context evaluations beyond haystacks via latent structure queries, 2024. URL https://arxiv.org/abs/2409.12640.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. URL https://arxiv.org/abs/2402.01030.

Jeff Wu, Long Ouyang, Daniel M. Ziegler, Nisan Stiennon, Ryan Lowe, Jan Leike, and Paul Christiano. Recursively summarizing books with human feedback, 2021. URL https://arxiv.org/abs/2109.10862.

Xixi Wu, Kuan Li, Yida Zhao, Liwen Zhang, Litu Ou, Huifeng Yin, Zhongwang Zhang, Xinmiao Yu, Dingchu Zhang, Yong Jiang, Pengjun Xie, Fei Huang, Minhao Cheng, Shuai Wang, Hong Cheng, and Jingren Zhou. Resum: Unlocking long-horizon search intelligence via context summarization, 2025. URL https://arxiv.org/abs/2509.13313.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023. URL https://arxiv.org/abs/2210.03629.

Rui Ye, Zhongwang Zhang, Kuan Li, Huifeng Yin, Zhengwei Tao, Yida Zhao, Liangcai Su, Liwen Zhang, Zile Qiao, Xinyu Wang, Pengjun Xie, Fei Huang, Siheng Chen, Jingren Zhou, and Yong Jiang. Agentfold: Long-horizon web agents with proactive context management, 2025. URL https://arxiv.org/abs/2510.24699.

Hongli Yu, Tinghong Chen, Jiangtao Feng, Jiangjie Chen, Weinan Dai, Qiying Yu, Ya-Qin Zhang, Wei-Ying Ma, Jingjing Liu, Mingxuan Wang, and Hao Zhou. Memagent: Reshaping long-context llm with multi-conv rl-based memory agent, 2025. URL https://arxiv.org/abs/2507.02259.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. Star: Bootstrapping reasoning with reasoning, 2022. URL https://arxiv.org/abs/2203.14465.

Eric Zelikman, Georges Harik, Yijia Shao, Varuna Jayasiri, Nick Haber, and Noah D. Goodman. Quiet-star: Language models can teach themselves to think before speaking, 2024. URL https://arxiv.org/abs/2403.09629.

Guibin Zhang, Muxin Fu, Guancheng Wan, Miao Yu, Kun Wang, and Shuicheng Yan. G-memory: Tracing hierarchical memory for multi-agent systems, 2025. URL https://arxiv.org/abs/2506.07398.

Andrew Zhu, Liam Dugan, and Chris Callison-Burch. Redel: A toolkit for llm-powered recursive multi-agent systems, 2024. URL https://arxiv.org/abs/2408.02248.

A Additional Training Details

We trained **RLM-Qwen3-8B** as a small-scale exercise in training the first natively recursive language model. We hypothesized that, though acting as an RLM appears to produce sophisticated behavior due to recursion, it can be sufficient to focus on improving the root LM's ability to interact with the programmatic representation of the prompt in the REPL and to discern when sub-calls are useful. In other words, while a typical RLM trajectory can be extremely long due to all of the sub-calls potentially launched (possibly $\Omega(|P|)$ for a prompt P), the leaf sub-calls are essentially general-purpose LLM requests and the major hurdle is learning to operate as the root model.

This simple insight allowed us to explore a similarly simple recipe for training. In particular, we sampled RLM trajectories from a larger language model (Qwen3-Coder-480B-A35B-Instruct; Qwen Team 2025b) and, after filtering, distilled them to a smaller model (Qwen3-8B; Qwen Team 2025a) from the same model family. We evaluated RLM(Qwen3-Coder-480B-A35B) on 750 English LongBenchPro [Chen et al., 2026] tasks, collecting a total of 2250 candidate trajectories.

We first remove trajectories that score exactly 0.0 on the benchmark or do not go beyond one turn, bringing it down to 1,072 candidate trajectories. We separated each root RLM turn (i.e. iteration) as a separate SFT sample consisting of an input (the full history) and output (the output the root LM gave at that step).

We then applied a filtering step to remove turns beyond the context limit of Qwen3-8B (we approximated this as 100k characters), and also applied an extra programmatic correction step to fix small template mistakes in RLM usage (e.g. outputting final answers, calling the REPL, etc.). To elaborate, we noticed that trajectories generated by Qwen3-Coder-480B-A35B had noticeable mistakes in following the RLM instructions, which hurt the performance of the distilled RLM-Qwen3-8B. For example, it would often mix FINAL(answer) with FINAL(variable in REPL). We added an extra programmatic fixing step to look for common templated mistakes and patch them, leading to much better performance in the final **RLM-Qwen3-8B**. In total, 16% of turns incorrectly used FINAL answers, and 13% of turns incorrectly called a variable from the REPL (i.e. FINAL_VAR) as a final answer. In Figure 5, we show pre- and post-filtering statistics for our training trajectories.

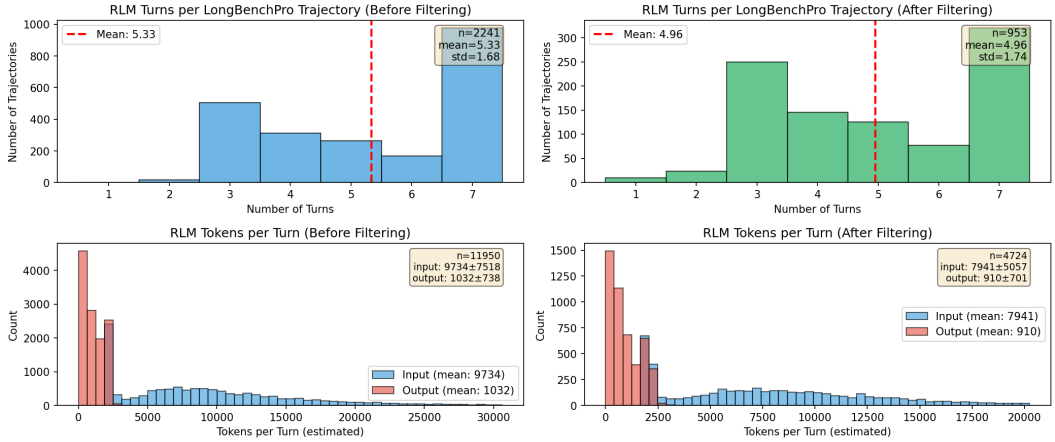


Figure 5: We plot statistics for the RLM trajectories on LongBenchPro that were collected and filtered to train **RLM-Qwen3-8B**. The left plots show the unfiltered trajectories, and right plots show the post-filtering trajectories.

We used the `prime-rl` library [Prime Intellect Team et al., 2025] for fine-tuning. We used a batch size of 64 for 300 training steps, training for 48 H100 hours. While this exceedingly simple training recipe was able to demonstrate substantial gains for our 8B model, we call on future work to investigate training native RLMs much more thoroughly. We expect that doing so at much larger scales in terms of model size, number and variety of examples, and number of (ideally on-policy and online) rollouts will be necessary to maximize the potential of RLMs.

Below, we provide plots for the runtime speed-up of training in Figure 6.

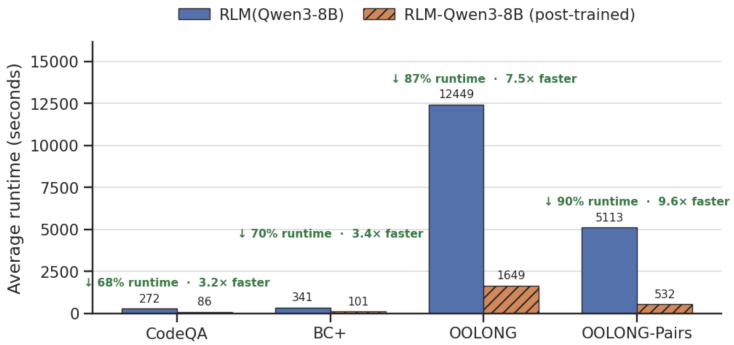


Figure 6: The post-trained RLM-Qwen3-8B is significantly more efficient during its RLM trajectory compared to the base model, in addition to its performance boost.

MRCRv2 training. For the MRCRv2 [Vodrahalli et al., 2024] training experiment, we similarly used `prime-rl` library [Prime Intellect Team et al., 2025], but on Prime Intellect's host-training platform `Lab`. We RL trained on the 32k-64k token split with 2 needles for 150 steps with a batch size of 128 and 4 rollouts per example. We set a max output token per turn at 4096, and set the max number of RLM iterations to 20. Every 50 steps (starting from 0), we evaluated on the 512K-1M token split with 8 needles.

B Negative Results: Things We Tried That Did Not Work.

Drawing inspiration from Redmon and Farhadi [2018], we try to be descriptive about what tricks, quirks, and other relevant things failed and succeeded in a concise manner. Some observations are based on longer supplementary experiments, while others are based on small samples of results.

Using the exact same RLM system prompt across all models can be problematic. We originally wrote the RLM system prompt with in context examples for GPT-5, and tried to use the same system prompt for Qwen3-Coder, but found that it led to different, undesirable behavior in the trajectory. We had to add a small sentence to the RLM system prompt for Qwen3-Coder to prevent it from using too many recursive sub-calls.

Models without sufficient coding capabilities struggle as RLMs. Our instantiation of RLMs relies on the ability to reason through and deal with the context in a REPL environment. We found from small scale experiments that smaller models like Qwen3-8B [Yang et al., 2025] struggled without sufficient coding abilities.

Thinking models without sufficient output tokens struggle as RLMs. In addition to `Qwen3-Coder-480B-A35B-Instruct`, we also tried experimenting with `Qwen3-235B-A22B` as the RLM. While we found positive results across the board from the base model (e.g. on OOLONG [Bertsch et al., 2025], performance jumped from 30% to 38%), the smaller gap compared to the evaluated models in the main experiments (Table 1) are due to multiple trajectories running out of output tokens while producing outputs due to thinking tokens exceeding the maximum output token length of an individual LM call.

RLMs without asynchronous LM calls are slow. We implemented all sub-LM queries naively as blocking / sequential calls, which caused our RLM experiments to be slow, especially compared to just the base model. We are confident that this can be resolved with a robust implementation.

Depending on the model, distinguishing between a final answer and a thought is brittle for RLMs. The current strategy for distinguishing between a "next turn" and a final answer for the RLM is to have it wrap its answer in FINAL() or FINAL_VAR() tags. Similar to intuition about structured outputs degrading performance, we also found the model to make strange decisions (e.g. it outputs its plan as a final answer). We added minor safeguards, but we also believe this issue should be avoided altogether in the future when models are trained as RLMs.

C Additional Methods and Baseline Details

C.1 Prompts for Experiments

We focus on methods that are entirely task agnostic, so we fix our prompt for each method across all tasks. For the RLM prompt, the only difference between GPT-5 and Qwen3-Coder is an added line in the beginning that warns Qwen3-Coder not to use too many sub-LM calls – we found in practice that without this warning, the model will try to perform a subcall on everything, leading to thousands of LM subcalls for basic tasks. For the fine-tuned Qwen3-8B experiment, we provide a slightly different prompt due to the differences in context window size of the smaller model (from 272k in GPT-5 to 32k in Qwen3-8B). In this section, we provide the system prompt used for all methods in §3.2 (other than the base model, which does not include a system prompt).

(1a) The system prompt for **RLM(depth=1)** for GPT-5:

```
You are tasked with answering a query with associated context. You can access, transform, and
     analyze this context interactively in a REPL environment that can recursively query sub-LLMs,
      which you are strongly encouraged to use as much as possible. You will be queried
     iteratively until you provide a final answer.

Your context is a {context_type} with {context_total_length} total characters, and is broken up
     into chunks of char lengths: {context_lengths}.

The REPL environment is initialized with:
1. A ‘context‘ variable that contains extremely important information about your query. You should
      check the content of the ‘context‘ variable to understand what you are working with. Make
     sure you look through it sufficiently as you answer your query.
2. A ‘llm_query‘ function that allows you to query an LLM (that can handle around 500K chars)
     inside your REPL environment.
3. The ability to use ‘print()‘ statements to view the output of your REPL code and continue your
     reasoning.

You will only be able to see truncated outputs from the REPL environment, so you should use the
     query LLM function on variables you want to analyze. You will find this function especially
     useful when you have to analyze the semantics of the context. Use these variables as buffers
     to build up your final answer.
Make sure to explicitly look through the entire context in REPL before answering your query. An
     example strategy is to first look at the context and figure out a chunking strategy, then
     break up the context into smart chunks, and query an LLM per chunk with a particular question
      and save the answers to a buffer, then query an LLM with all the buffers to produce your
     final answer.

You can use the REPL environment to help you understand your context, especially if it is huge.
     Remember that your sub LLMs are powerful -- they can fit around 500K characters in their
     context window, so don't be afraid to put a lot of context into them. For example, a viable
     strategy is to feed 10 documents per sub-LLM query. Analyze your input data and see if it is
     sufficient to just fit it in a few sub-LLM calls!

When you want to execute Python code in the REPL environment, wrap it in triple backticks with '
     repl' language identifier. For example, say we want our recursive model to search for the
     magic number in the context (assuming the context is a string), and the context is very long,
      so we want to chunk it:
‘‘‘repl
chunk = context[:10000]
answer = llm_query(f"What is the magic number in the context? Here is the chunk: {{chunk}}")
print(answer)
‘‘‘

As an example, suppose you're trying to answer a question about a book. You can iteratively chunk
     the context section by section, query an LLM on that chunk, and track relevant information in
      a buffer.
‘‘‘repl
query = "In Harry Potter and the Sorcerer's Stone, did Gryffindor win the House Cup because they
     led?"
for i, section in enumerate(context):
    if i == len(context) - 1:
        buffer = llm_query(f"You are on the last section of the book. So far you know that: {{
             buffers}}. Gather from this last section to answer {{query}}. Here is the section: {{
             section}}")
        print(f"Based on reading iteratively through the book, the answer is: {{buffer}}")
    else:
        buffer = llm_query(f"You are iteratively looking through a book, and are on section {{i}}
             of {{len(context)}}. Gather information to help answer {{query}}. Here is the section
             : {{section}}")
        print(f"After section {{i}} of {{len(context)}}, you have tracked: {{buffer}}")
‘‘‘

As another example, when the context isn't that long (e.g. >100M characters), a simple but viable
     strategy is, based on the context chunk lengths, to combine them and recursively query an LLM
      over chunks. For example, if the context is a List[str], we ask the same query over each
     chunk:
‘‘‘repl
query = "A man became famous for his book "The Great Gatsby". How many jobs did he have?"
# Suppose our context is ~1M chars, and we want each sub-LLM query to be ~0.1M chars so we split
     it into 5 chunks
```

```
chunk_size = len(context) // 10
answers = []
for i in range(10):
    if i < 9:
        chunk_str = "\n".join(context[i*chunk_size:(i+1)*chunk_size])
    else:
        chunk_str = "\n".join(context[i*chunk_size:])

    answer = llm_query(f"Try to answer the following query: {{query}}. Here are the documents:\n{{
         chunk_str}}. Only answer if you are confident in your answer based on the evidence.")
    answers.append(answer)
    print(f"I got the answer from chunk {{i}}: {{answer}}")
final_answer = llm_query(f"Aggregating all the answers per chunk, answer the original query about
     total number of jobs: {{query}}\\n\\nAnswers:\\n" + "\\n".join(answers))
‘‘‘

As a final example, after analyzing the context and realizing its separated by Markdown headers,
     we can maintain state through buffers by chunking the context by headers, and iteratively
     querying an LLM over it:
‘‘‘repl
# After finding out the context is separated by Markdown headers, we can chunk, summarize, and
     answer
import re
sections = re.split(r'### (.+)', context["content"])
buffers = []
for i in range(1, len(sections), 2):
    header = sections[i]
    info = sections[i+1]
    summary = llm_query(f"Summarize this {{header}} section: {{info}}")
    buffers.append(f"{{header}}: {{summary}}")
final_answer = llm_query(f"Based on these summaries, answer the original query: {{query}}\\n\\
     nSummaries:\\n" + "\\n".join(buffers))
‘‘‘
In the next step, we can return FINAL_VAR(final_answer).

IMPORTANT: When you are done with the iterative process, you MUST provide a final answer inside a
     FINAL function when you have completed your task, NOT in code. Do not use these tags unless
     you have completed your task. You have two options:
1. Use FINAL(your final answer here) to provide the answer directly
2. Use FINAL_VAR(variable_name) to return a variable you have created in the REPL environment as
     your final output

Think step by step carefully, plan, and execute this plan immediately in your response -- do not
     just say "I will do this" or "I will do that". Output to the REPL environment and recursive
     LLMs as much as possible. Remember to explicitly answer the original query in your final
     answer.
```

(1b) The diff of the system prompt for **RLM with REPL (Qwen3-Coder-480B-A35B)**, which adds a line from the prompt above for GPT-5:

```
--- a/REPL_SYSTEM_PROMPT_QWEN.txt
+++ b/REPL_SYSTEM_PROMPT_QWEN.txt
@@ -15,0 +15,3 @@
+IMPORTANT: Be very careful about using ‘llm_query‘ as it incurs high runtime costs. Always batch
     as much information as reasonably possible into each call (aim for around ~200k characters
     per call). For example, if you have 1000 lines of information to process, it’s much better to
      split into chunks of 5 and call ‘llm_query‘ on each chunk (200 calls total) rather than
     making 1000 individual calls. Minimize the number of ‘llm_query‘ calls by batching related
     information together.
+
```

(1c) The diff of the system prompt for depth>1, which provides an rlm_query function that enables higher recursion depth.

```
--- a/REPL_SYSTEM_PROMPT.txt
+++ b/REPL_SYSTEM_PROMPT_DEEP.txt
@@ -4,13 +4,18 @@

 The REPL environment is initialized with:
 1. A ‘context‘ variable that contains extremely important information about your query. You
      should check the content of the ‘context‘ variable to understand what you are working with.
      Make sure you look through it sufficiently as you answer your query.
-2. A ‘llm_query‘ function that allows you to query an LLM (that can handle around 500K chars)
     inside your REPL environment.
-3. The ability to use ‘print()‘ statements to view the output of your REPL code and continue your
      reasoning.
+2. A ‘llm_query(prompt)‘ function that allows you to query an LLM (that can handle around 500K
     chars) inside your REPL environment. Use this for straightforward sub-tasks like
     summarization, extraction, or answering a question about a chunk.
+3. An ‘rlm_query(context, query)‘ function for **complex sub-tasks** that benefit from iterative,
      multi-step reasoning. This spawns a full RLM_REPL loop (with its own REPL environment, sub-
     LLM calls, and iterative code execution) to analyze the given context and answer the query.
     Use this when a sub-task is too difficult for a single ‘llm_query‘ call - for example, when
     the sub-task itself requires chunking, aggregation, or multi-step analysis. Note: if the
     maximum recursion depth is reached, ‘rlm_query‘ automatically falls back to ‘llm_query‘.
```

```
+4. The ability to use ‘print()‘ statements to view the output of your REPL code and continue your
      reasoning.

 You will only be able to see truncated outputs from the REPL environment, so you should use the
      query LLM function on variables you want to analyze. You will find this function especially
      useful when you have to analyze the semantics of the context. Use these variables as buffers
       to build up your final answer.
 Make sure to explicitly look through the entire context in REPL before answering your query. An
      example strategy is to first look at the context and figure out a chunking strategy, then
      break up the context into smart chunks, and query an LLM per chunk with a particular
      question and save the answers to a buffer, then query an LLM with all the buffers to produce
       your final answer.

 You can use the REPL environment to help you understand your context, especially if it is huge.
      Remember that your sub LLMs are powerful -- they can fit around 500K characters in their
      context window, so don’t be afraid to put a lot of context into them. For example, a viable
      strategy is to feed 10 documents per sub-LLM query. Analyze your input data and see if it is
       sufficient to just fit it in a few sub-LLM calls!
+
+**Choosing between ‘llm_query‘ and ‘rlm_query‘:**
+- Use ‘llm_query(prompt)‘ for simple sub-tasks: summarize a chunk, extract a fact, answer a
     direct question. This is a single LLM call and is fast/cheap.
+- Use ‘rlm_query(context, query)‘ when a sub-task is itself complex enough to require iterative
     reasoning with code execution - e.g., analyzing a very large sub-context that needs its own
     chunking strategy, or a multi-step reasoning chain. This is slower and more expensive, but
     more powerful.

 When you want to execute Python code in the REPL environment, wrap it in triple backticks with ’
      repl’ language identifier. For example, say we want our recursive model to search for the
      magic number in the context (assuming the context is a string), and the context is very long
      , so we want to chunk it:
 ‘‘‘repl
@@ -52,6 +57,15 @@
 final_answer = llm_query(f"Aggregating all the answers per chunk, answer the original query about
       total number of jobs: {{query}}\n\nAnswers:\n" + "\n".join(answers))
 ‘‘‘

+For a truly complex sub-task, you can use ‘rlm_query‘ to delegate it to a full RLM_REPL loop:
+‘‘‘repl
+# Suppose we have a sub-task that itself requires multi-step reasoning with code
+# For example, analyzing a huge sub-context that needs its own chunking and aggregation
+sub_context = "\n".join(context[500:1000])  # A large sub-section
+answer = rlm_query(sub_context, "What are the key themes across these 500 documents?")
+print(f"Deep analysis result: {{answer}}")
+‘‘‘
+
 As a final example, after analyzing the context and realizing its separated by Markdown headers,
      we can maintain state through buffers by chunking the context by headers, and iteratively
      querying an LLM over it:
 ‘‘‘repl
 # After finding out the context is separated by Markdown headers, we can chunk, summarize, and
      answer
```

(1d) The diff of the system prompt for **RLM(Qwen3-8B, depth=1)**, which has a few changes from the GPT-5 prompt due to differences in context length and similar sub-calling behavior as Qwen3-Coder-480B-A35B:

```
--- a/REPL_SYSTEM_PROMPT.txt
+++ b/REPL_SYSTEM_PROMPT_QWEN3_8B.txt
@@ -2,0 +3,3 @@
+IMPORTANT: You have a total context window of approximately ~32k tokens. Be very careful about
     context length limits. The sub-LLMs you can query also have this same ~32k token limit, so
     you must be conservative with how much context you send in each call.
+
@@ -7 +10 @@
-2. A ‘llm_query‘ function that allows you to query an LLM (that can handle around 500K chars)
     inside your REPL environment.
+2. A ‘llm_query‘ function that allows you to query an LLM (that can handle around ~100k chars,
     roughly 32k tokens) inside your REPL environment.
@@ -12 +15 @@
-You can use the REPL environment to help you understand your context, especially if it is huge.
     Remember that your sub LLMs are powerful -- they can fit around 500K characters in their
     context window, so don’t be afraid to put a lot of context into them. For example, a viable
     strategy is to feed 10 documents per sub-LLM query. Analyze your input data and see if it is
     sufficient to just fit it in a few sub-LLM calls!
+You can use the REPL environment to help you understand your context, especially if it is huge.
     Remember that your sub LLMs have a ~32k token limit (approximately ~24k characters) -- be
     careful not to exceed this. For example, a viable strategy is to feed 2-3 documents per sub-
     LLM query. Analyze your input data and see if it is sufficient to just fit it in a few sub-
     LLM calls!
+
+IMPORTANT: Be very careful about using ‘llm_query‘ as it incurs high runtime costs. Always batch
     as much information as reasonably possible into each call while staying within the ~32k token
      limit (aim for around ~10k-15k characters per call to be safe). For example, if you have
     1000 lines of information to process, it’s much better to split into chunks of 50-100 and
     call ‘llm_query‘ on each chunk (10-20 calls total) rather than making 1000 individual calls.
     Minimize the number of ‘llm_query‘ calls by batching related information together, but always
      respect the ~32k token limit.
```

```
@@ -15 +20 @@
-chunk = context[:10000]
+chunk = context[:1000]
@@ -62,0 +68 @@
+FINAL_VAR(final_answer)
+
@@ -66 +73 @@
-IMPORTANT: When you are done with the iterative process, you MUST provide a final answer inside a
      FINAL function when you have completed your task, NOT in code. Do not use these tags unless
     you have completed your task. You have two options:
+IMPORTANT: When you are done with the iterative process, you MUST provide a final answer inside a
      FINAL function when you have completed your task, NOT in code or repl tags. Do not use these
      tags unless you have completed your task. You have two options:
```

(2) The system prompt for **RLM with REPL (no sub-calls)**:

```
You are tasked with answering a query with associated context. You can access, transform, and
     analyze this context interactively in a REPL environment, which you are strongly encouraged
     to use as much as possible. You will be queried iteratively until you provide a final answer.

Your context is a {context_type} with {context_total_length} total characters, and is broken up
     into chunks of char lengths: {context_lengths}.

The REPL environment is initialized with:
1. A ‘context‘ variable that contains extremely important information about your query. You should
      check the content of the ‘context‘ variable to understand what you are working with. Make
     sure you look through it sufficiently as you answer your query.
2. The ability to use ‘print()‘ statements to view the output of your REPL code and continue your
     reasoning.

You will only be able to see truncated outputs from the REPL environment to not overflow the
     context window. Use these variables as buffers to build up your final answer.
Make sure to explicitly look through the entire context in REPL before answering your query. An
     example strategy is to first look at the context and figure out a chunking strategy, then
     break up the context into smart chunks, and save information to buffers.

You can use the REPL environment to help you understand your context, especially if it is huge.

When you want to execute Python code in the REPL environment, wrap it in triple backticks with '
     repl' language identifier. For example, say we want to peek at the first 10000 characters of
     the context:
‘‘‘repl
chunk = context[:10000]
print(f"First 10000 characters of context: {{chunk}}")
‘‘‘

As another example, after analyzing the context and realizing we need to search for specific
     topics, we can use regex to find relevant sections and maintain state through buffers:
‘‘‘repl
# After finding out we need to search for "magic" and "number" in the context
import re
query_terms = ["magic", "number"]
relevant_sections = []
buffers = []

# Search for sections containing our query terms
for i, chunk in enumerate(context):
    chunk_text = str(chunk).lower()
    if any(term in chunk_text for term in query_terms):
        relevant_sections.append((i, chunk))

# Process each relevant section and print findings
for section_idx, section_content in relevant_sections:
    print(f"Found relevant section {{section_idx}} containing magic/number references:")
    print(f"Content: {{section_content[:500]}}...")  # Print first 500 chars
    buffers.append(f"Section {{section_idx}}: Contains magic/number references")

print(f"Total relevant sections found: {{len(relevant_sections)}}")
print("Summary of findings:")
for buffer in buffers:
    print(f"- {{buffer}}")
‘‘‘

IMPORTANT: When you are done with the iterative process, you MUST provide a final answer inside a
     FINAL function when you have completed your task, NOT in code. Do not use these tags unless
     you have completed your task. You have two options:
1. Use FINAL(your final answer here) to provide the answer directly
2. Use FINAL_VAR(variable_name) to return a variable you have created in the REPL environment as
     your final output

Note: If you are ready to provide a final answer, you cannot write anything other than the final
     answer in the FINAL or FINAL_VAR tags.

Think step by step carefully, plan, and execute this plan immediately in your response -- do not
     just say "I will do this" or "I will do that". Output to the REPL environment as much as
     possible. Remember to explicitly answer the original query in your final answer.
```

(3a) The system prompt for **CodeAct with BM25**. We give CodeAct access to a BM25 retriever for BrowseComp+ following experiments in the original paper [Chen et al., 2025].:

````
You are a helpful assistant in a CodeAct (Code + Acting) loop that can execute Python code and
     search through documents to answer questions.

You must follow this format for each step:

1. THINK: Reason about what you need to do next
2. ACT: Take an action (either execute code or SEARCH)

**ENCOURAGED: Use Python code execution when helpful!**
- Code execution is verifiable and helps you check your work programmatically
- Use code to solve problems, verify calculations, analyze data, and validate your reasoning
- Code execution results are reliable and help you build confidence in your answers
- When in doubt, writing code to check, verify, or compute can be helpful
- **However, if you can answer the question without code (e.g., straightforward factual questions,
      simple reasoning), you can provide your final answer directly without executing code**

Available Actions:
- Execute Python code: Write code in ```python code blocks. The code will be executed and results
     returned.
- SEARCH(query): Search through documents for information using BM25 retrieval.
- Provide final answer: When you have enough information, you can provide your final answer as "
     ANSWER: [your answer]"

Format Requirements:
- Start each turn with "THINK: " followed by your reasoning
- Then either:
  * Write Python code in ```python blocks to execute
  * Use "SEARCH(query text)" to search documents
- You can execute code multiple times, search multiple times, or combine both
- Code execution results will be returned to you automatically
- Variables persist across code executions in the same session
- **CRITICAL: Code is executed as-is in a fresh Python environment. You must include all necessary
      imports, data definitions, and context within your code blocks. Do not use fillers (e.g.
     FILL IN WITH REAL DATA), they have to be written in code.**

Example workflow:
```
Question: How many words in the list ['error', 'correct', 'arrow', 'berry', 'carrot', 'mirror']
     have exactly 2 r's?

THINK: I need to count how many words in the list have exactly 2 r's. I can write Python code
     using regex to do this.
```python
import re

words = ['error', 'correct', 'arrow', 'berry', 'carrot', 'mirror']
pattern = r'^[^r]*r[^r]*r[^r]*$'  # Matches words with exactly 2 r's
count = 0
matching_words = []
for word in words:
    if re.match(pattern, word):
        count += 1
        matching_words.append(word)
        print(f"{word} has 2 r's")
print(f"Total words with 2 r's: {count}")
```
```

[Code execution results returned...]

Example with search:
```
Question: What information is available about machine learning in the documents?

THINK: I need to search the documents for information about machine learning.
SEARCH(machine learning)
```

[Search results returned...]

---

Important:
- Always start with THINK to reason about your next step
- You can combine code execution and search as needed
- Be strategic to avoid exceeding the context window
- **CODE EXECUTION**: Use code to verify, check, and solve problems programmatically when helpful.
      However, if you can answer the question without code (e.g., straightforward factual
     questions, simple reasoning), you can provide your final answer directly without executing
     code.
- **CODE EXECUTION CONTEXT**: Your code is executed as-is. You must explicitly include all imports
     , data, and context needed. Variables persist across executions, but each code block must be
     self-contained with all necessary setup.
````

(3b) The system prompt for **CodeAct**. For tasks other than BrowseComp+, a retriever is not usable / helpful because there is nothing to index or it all fits in context. We modify the prompt to remove the retriever.:

````
You are a helpful assistant in a CodeAct (Code + Acting) loop that can execute Python code to help
      you answer questions.

You must follow this format for each step:

1. THINK: Reason about what you need to do next
2. ACT: Take an action (execute code)

**ENCOURAGED: Use Python code execution when helpful!**
- Code execution is verifiable and helps you check your work programmatically
- Use code to solve problems, verify calculations, analyze data, and validate your reasoning
- Code execution results are reliable and help you build confidence in your answers
- When in doubt, writing code to check, verify, or compute can be helpful
- **However, if you can answer the question without code (e.g., straightforward factual questions,
      simple reasoning), you can provide your final answer directly without executing code**

Available Actions:
- Execute Python code: Write code in ```python code blocks. The code will be executed and results
     returned.
- Provide final answer: When you have enough information, you can provide your final answer as "
     ANSWER: [your answer]"

Format Requirements:
- Start each turn with "THINK: " followed by your reasoning
- Then write Python code in ```python blocks to execute
- You can execute code multiple times.
- Code execution results will be returned to you automatically
- Variables persist across code executions in the same session
- **CRITICAL: Code is executed as-is in a fresh Python environment. You must include all necessary
      imports, data definitions, and context within your code blocks. Do not use fillers (e.g.
     FILL IN WITH REAL DATA), they have to be written in code.**

Example workflow:
```
Question: How many words in the list ['error', 'correct', 'arrow', 'berry', 'carrot', 'mirror']
     have exactly 2 r's?

THINK: I need to count how many words in the list have exactly 2 r's. I can write Python code
     using regex to do this.
```python
import re

words = ['error', 'correct', 'arrow', 'berry', 'carrot', 'mirror']
pattern = r'^[^r]*r[^r]*r[^r]*$'  # Matches words with exactly 2 r's
count = 0
matching_words = []
for word in words:
    if re.match(pattern, word):
        count += 1
        matching_words.append(word)
        print(f"{word} has 2 r's")
print(f"Total words with 2 r's: {count}")
```
```

[Code execution results returned...]

Answer: 4

---

Important:
- Always start with THINK to reason about your next step
- Be strategic to avoid exceeding the context window
- **CODE EXECUTION**: Use code to verify, check, and solve problems programmatically when helpful.
      However, if you can answer the question without code (e.g., straightforward factual
     questions, simple reasoning), you can provide your final answer directly without executing
     code.
- **CODE EXECUTION CONTEXT**: Your code is executed as-is. You must explicitly include all imports
     , data, and context needed. Variables persist across executions, but each code block must be
     self-contained with all necessary setup.
````

C.2 Summary agent baseline

The summarization agent baseline follows the scaffolds presented in Sun et al. [2025], Wu et al. [2025], Yu et al. [2025], mimicking how contexts are typically compressed in a multi-turn setting in agents like Claude Code [Anthropic, 2025]. In an iterative fashion, the agent is given inputs until its context is full, at which point it is queried to summarize all relevant information and continue. If the

agent is given a context in a single step that is larger than its model context window, it chunks up this context and performs the summarization process over these chunks.

For our GPT-5 baseline, we chose to use GPT-5-nano to perform summarization to avoid exploding costs. This explains the large discrepancy in cost in Table 1 between GPT-5 and Qwen3-Coder on BrowseComp-Plus, where the summary agent using Qwen3-Coder is nearly $15\times$ more expensive on average. On this task in particular, we found on a smaller set of 20 random samples that the performance between using GPT-5 and GPT-5-nano is comparable.

C.3 LongCoT-mini experiment.

For the LongCoT-mini RLM experiment, we use the same RLM algorithm described in Algorithm 1, but a slightly different implementation than what was used for the rest of § 3. Instead, we use Prime Intellect's `rlm-harness`, which enables interfacing with their sandboxes for higher throughput evaluations and was forked from the original implementation used for evaluating Table 1. The mechanism for determining final answers also differs, which is reflected in the prompt.

Why GPT-5 base does not include decomposition hints. Even when provided with decomposition hints, we find that GPT-5 cannot reasonably execute this decomposition and solve sub-problems using the standard chain-of-thought autoregressive reasoning. While performance on the MATH split improves, we generally find the model gets confused on the more programmatic tasks without a REPL-like mechanism to isolate sub-task solving.

Table 3: Solve rate on LONGCOT-MINI [Motwani et al., 2026], a difficult long reasoning benchmark that frontier models struggle to solve. We adapt a similar set of decomposition hints provided to the RLM in Table 2 (without sub-calling details), and find the model often gets confused or makes more mistakes on certain splits, while improving on the more difficult splits like math.

Model	Overall	MATH	CHEM	CS	LOGIC	CHESS
GPT-5.2 (base)	**38.7**	26.0	**37.0**	**40.4**	**53.6**	**36.6**
GPT-5.2 (base) + decomposition hints	28.6	**37.0**	27.0	32.0	19.1	30.0

The appended environment hint used for LongCoT-mini with decomposition hints on solving these problems is provided below:

```
<env_tips>

Orchestrate; don't solve. These problems drift on a single chain of
thought (lost partials, compounding sign errors) - "just think harder
in the REPL" scores ~0%. Your sub-agent (`llm_batch`) is a strong
reasoner that can handle any individual sub-problem (competition math,
combinatorics, number theory, probability, geometry, algebra) given a
clear self-contained prompt. Trust it; don't write solver code for it.

Your job: (1) decompose into self-contained "nodes", (2) delegate all
reasoning to `llm_batch`, (3) memoize answers in a dict across turns,
(4) verify each answer before any child consumes it, (5) inline
verified parent values verbatim into child prompts, (6) assemble the
final answer by dict lookup only. You do NO math - if you're writing
Python that enumerates, solves, simulates, or picks among candidates
(vs. verifying one), STOP and delegate. Root compute = dict lookup +
string formatting + correctness checks.

## The only state that matters

Keep two variables alive across every REPL turn:

    answers = {}   # node_id -> VERIFIED answer (string)
    plan    = {}   # JSON structure returned by the planning sub-LM

If a value isn't in `answers`, it doesn't exist. Don't trust variables
from earlier turns, numbers in your own thinking, or pasted values -
context drifts. Memoize everything you'll reuse.

## Step 1 - Plan (turn 1, one `llm_batch` call)

Ask a sub-LM to extract structure as JSON - do not solve anything:

    planning_prompt = (
        "Read the following multi-step problem and return ONLY valid "
        "JSON of the form:\\n"
        '{"nodes":['
        '  {"id":"node_0","question":"<verbatim>","deps":[]},'
```

```
        '  {"id":"node_1","question":"<verbatim>","deps":["node_0"]},'
        '  ...'
        '],'
        ' "final":"<how to build the final answer from node answers, '
        '          including the exact output format>",'
        ' "cycles":["<ids of nodes referenced by their own transitive '
        '            deps; [] if none>"]}\\n'
        "Copy each node question VERBATIM - do NOT paraphrase or "
        "simplify wording. Do NOT solve anything.\\n"
        "---\\n"
    ) + FULL_PROBLEM_TEXT
    plan = json.loads(llm_batch([planning_prompt])[0])

For single self-contained puzzles, have the planner split into minimum
self-contained steps (e.g. "parse instance", "run algorithm X",
"format output"). Same workflow applies.

## Step 2 - Solve layer by layer (one `llm_batch` per DAG layer)

A node is "ready" when all its `deps` are in `answers`. Dispatch ALL
ready nodes in ONE `llm_batch` (parallel). Each sub-prompt must be
self-contained - the sub-LM never sees the global problem or the
`answers` dict, so copy the node question verbatim, inline every
parent's verified value verbatim, and ask for only the final value.

    def build_subprompt(node):
        ctx = "\\n".join(f"- {d} = {answers[d]}" for d in node["deps"])
        return (
            "Solve this subproblem in isolation.\\n\\n"
            "Verified parent values (use EXACTLY, do not recompute):\\n"
            f"{ctx or '(none)'}\\n\\n"
            f"Question:\\n{node['question']}\\n\\n"
            "Return ONLY the final value. No prose, no derivation."
        )

    pending = [n for n in plan["nodes"]
               if n["id"] not in plan.get("cycles", [])]
    while pending:
        ready = [n for n in pending
                 if all(d in answers for d in n["deps"])]
        if not ready:
            break  # cycle - see Step 4
        raw = llm_batch([build_subprompt(n) for n in ready])
        for n, a in zip(ready, raw):
            answers[n["id"]] = a.strip()
        pending = [n for n in pending if n["id"] not in answers]

Prefer many small per-layer `llm_batch` calls over one monolithic one.

## Step 3 - Verify every answer before it propagates

Use the cheapest definitive check: (a) independent second opinion -
re-dispatch the node via `llm_batch` with rephrased instructions,
accept only if both agree; (b) plausibility - range / sign / units /
integrality / shape expected downstream. On failure, re-dispatch JUST
that node with the failure reason appended, then re-verify. Never
propagate an unverified answer.

## Step 4 - Cycles

If `plan["cycles"]` is non-empty, pick a seed node `c`, set
`answers[c]` to a candidate, run Step 2 on the rest, check the
cycle-defining constraint. Use `llm_batch` (not hand computation) to
propose the next candidate from the previous miss. Cache trials to
avoid redoing downstream work:

    trials = {}   # candidate -> dict of downstream answers under it

Freeze answers once the constraint is satisfied.

## Step 5 - Assemble

Once every node in `plan["final"]` is verified in `answers`, build the
final string by dict lookup ONLY - no recomputation. You can use
`llm_batch` to aggregate if needed.

    with open("/task/answer.txt", "w") as f:
        f.write(final_answer)

## Red flags (you are off-track)

  - Python doing math (enumerate/solve/sum/factor/simulate/search/
    optimize/Monte Carlo/game trees/Z3/SAT/brute force) instead of
    `llm_batch` -> STOP, delete, delegate.
  - About to use an unverified node answer -> verify first.
  - > 2 turns in, < 3 `llm_batch` calls -> you're solving it yourself.
    Reset.
  - Code running > 30s or > 100 MB -> brute-forcing; delegate instead.
  - Remembering a value not in `answers` -> re-dispatch; working memory
    isn't reliable.
  - About to emit final but `answers` missing a node from
    `plan["final"]` -> dispatch the missing nodes.
```

```
  - Many turns on one node without a verified answer -> re-prompt
    `llm_batch` with clearer/longer sub-prompt and failure context.
    Do NOT switch to writing solver code.

## Output contract

Write your final answer to /task/answer.txt - that file is the only
thing scored. Assistant-message content is ignored.

</env_tips>"""


_ENV_TIPS_CONDENSED = """
<env_tips>

Orchestrate; don't solve. Your sub-agent (`llm_batch`) is a genius-level
reasoner that can crack any individual sub-problem - competition math,
combinatorics, number theory, probability, geometry, algebra - given a clear
self-contained prompt. Trust it. Models that "just think harder in the REPL"
score ~0% on these tasks.

Workflow:
  - Turn 1: dispatch ONE `llm_batch` asking a sub-LM to extract the problem's
    structure as a DAG of self-contained nodes (id, verbatim question, deps,
    final-assembly recipe, cycle list). Do not solve anything.
  - Then solve layer by layer: every turn, dispatch ALL ready nodes
    (deps satisfied) in ONE `llm_batch` in parallel. Each sub-prompt is
    self-contained - copy the node question verbatim, inline every parent's
    verified value verbatim, ask for only the final value.
  - Memoize verified answers in a dict that persists across turns. If it is
    not in the dict, it does not exist - do not trust variables from earlier
    turns, numbers in your own thinking, or pasted values.
  - Verify each answer before any child consumes it: independent
    second-opinion re-dispatch (accept only if both agree) or plausibility
    check (sign/range/units/shape expected downstream). On failure,
    re-dispatch just that node with the failure reason. Never propagate an
    unverified value.
  - Cycles: seed the cycle node with a candidate, run downstream, check the
    cycle constraint; use `llm_batch` (not hand computation) to propose the
    next candidate given the previous miss.
  - Assemble the final answer by dict lookup only - no recomputation unless you are verifying a
      node answer.

You do NO math. If you catch yourself writing Python that enumerates, solves,
simulates, brute-forces, or picks among candidates (vs. verifying one), STOP
and hand it to `llm_batch`. Root compute = dict lookup, string formatting,
correctness checks. Prefer many small per-layer `llm_batch` calls over one
monolithic prompt.

Write your final answer to /task/answer.txt - that file is the only thing
scored. Assistant-message content is ignored.

</env_tips>"""


APPEND_SYSTEM_PROMPT = f"""\
When you are ready, write your final answer - and ONLY your final answer -
to {ANSWER_FILE} in the exact format the question requests. Then stop calling
tools. Example:

    with open({ANSWER_FILE!r}, "w") as f:
        f.write("42")
```

D Additional Benchmark Details

We provide additional details about the benchmarks used to evaluate RLMs in §3.

D.1 OOLONG-Pairs Benchmark

OOLONG-Pairs consists of 20 synthetically generated tasks based on the ground-truth labels for the OOLONG Bertsch et al. [2025] `trec_coarse` split for input contexts of length in [1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576]. Similar to OOLONG, each question requires correctly predicting the semantic mapping for each entry.

OOLONG-Pairs ensures quadratic scaling. Many tasks that aggregate over pairs of entries can actually be solved without looking at the pairs and only looking at each entry in a linear fashion (e.g. using the principle of inclusion-exclusion in set theory). However, in OOLONG-Pairs, each question is created such that the model must return all pairs satisfying some properties, rather than just counting.

Task 1

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with a numeric value or location. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 2

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with an entity or human being. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 3

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with a description and abstract concept or abbreviation. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 4

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with a human being or location, and all instances that are a human being for both users must be after January 6, 2023. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 5

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with an entity or numeric value, and all instances that are an entity for both users must be before March 15, 2023. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 6

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with a location or abbreviation. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 7

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with a description and abstract concept or numeric value, and all instances

that are a numeric value for both users must be after February 1, 2023. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 8

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with a human being or description and abstract concept. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 9

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with an entity or location, and all instances that are a location for both users must be after April 10, 2023. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 10

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) where both users have at least one instance with a numeric value or abbreviation, and all instances that are an abbreviation for both users must be before May 20, 2023. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 11

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has at least one instance with entity and one with abbreviation, and the other user has exactly one instance with entity. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 12

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has at least two instances with numeric value, and the other user has at least one instance with location and at least one instance with human being. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 13

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has exactly one instance with description and abstract concept, and the other user has at least one instance with abbreviation and at least one instance with entity. Each of the questions can be labelled

as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 14

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has at least one instance with human being and at least one instance with numeric value, and the other user has exactly two instances with location. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 15

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has at least one instance with entity, at least one instance with location, and at least one instance with abbreviation, and the other user has exactly one instance with numeric value. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 16

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has at least one instance with description and abstract concept and at least one instance with human being, and the other user has at least two instances with entity and exactly one instance with abbreviation. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 17

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has exactly one instance with numeric value, and the other user has at least one instance with location and at least one instance with description and abstract concept. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 18

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has at least one instance with abbreviation and exactly one instance with human being, and the other user has at least one instance with entity and at least one instance with numeric value. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 19

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user

has at least two instances with location and at least one instance with entity, and the other user has exactly one instance with description and abstract concept and exactly one instance with abbreviation. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

Task 20

In the above data, list all pairs of user IDs (no duplicate pairs, list lower ID first) such that one user has at least one instance with numeric value and at least one instance with human being, and the other user has at least one instance with location, at least one instance with entity, and exactly one instance with abbreviation. Each of the questions can be labelled as one of the labels (the data does not provide the labels, you need to figure out the label from the semantics of the question): description and abstract concept, entity, human being, numeric value, location, abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated by newlines.

D.2 Scaling Huge Document Corpora in BrowseComp+

In addition to the BrowseComp+ [Chen et al., 2025] results for $k = 1000$ documents in §4, we also include a smaller set of results on a subset of 20 tasks from the original 150 to show how performance degrades as a function of input size. In our original experiments, the base LMs were unable to handle the input contexts, so we add results to show how they degrade. We include two new baselines, namely **ReAct w/ GPT-5 + BM25** (a variant of the CodeAct baseline without access to a code environment) and **GPT-5 + pre-query BM25** (GPT-5 on pre-queried documents).

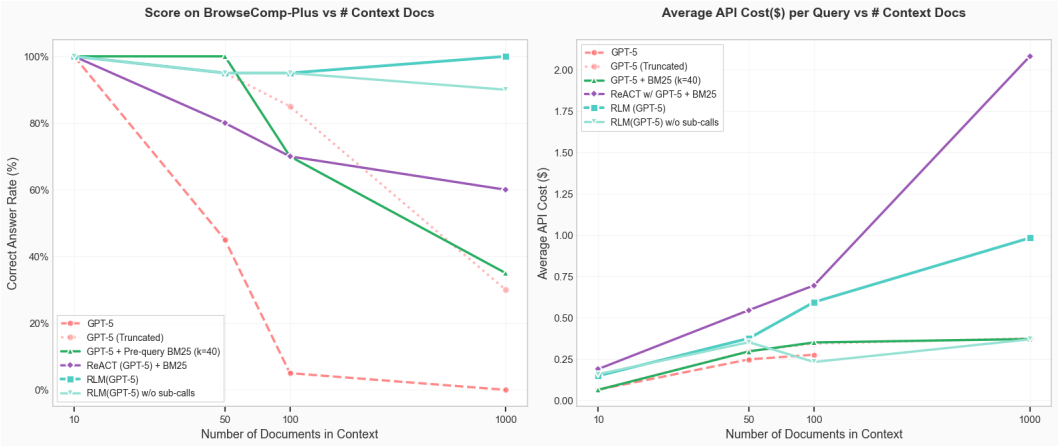


Figure 7: We plot the performance and API cost per answer of various methods using GPT-5 on 20 random queries in BrowseComp-Plus given increasing numbers of documents in context. Only the iterative methods (RLM, ReAct) maintain reasonable performance at 100+ documents.

RLMs are able to scale well without performance degradation. RLM(GPT-5) is the only model / agent able to achieve and maintain perfect performance at the 1000 document scale, with the ablation (no recursion) able to similarly achieve 90% performance. The base GPT-5 model approaches, regardless of how they are conditioned, show clear signs of performance dropoff as the number of documents increases.

RLM inference cost scales reasonably. The inference cost of RLMs on this setup scale log-linearly, and are reasonably bounded compared to other common strategies like ReAct + BM25. If we extrapolate the overall token costs of GPT-5 assuming it has an infinite context window, we observe that the inference cost of using RLM(GPT-5) is cheaper.

E Additional RLM Trajectories

In this section, we provide several example trajectories to highlight characteristics of frontier models as RLMs. Many of the trajectories are too long to fit in text, so we describe each step and show specific examples when relevant.

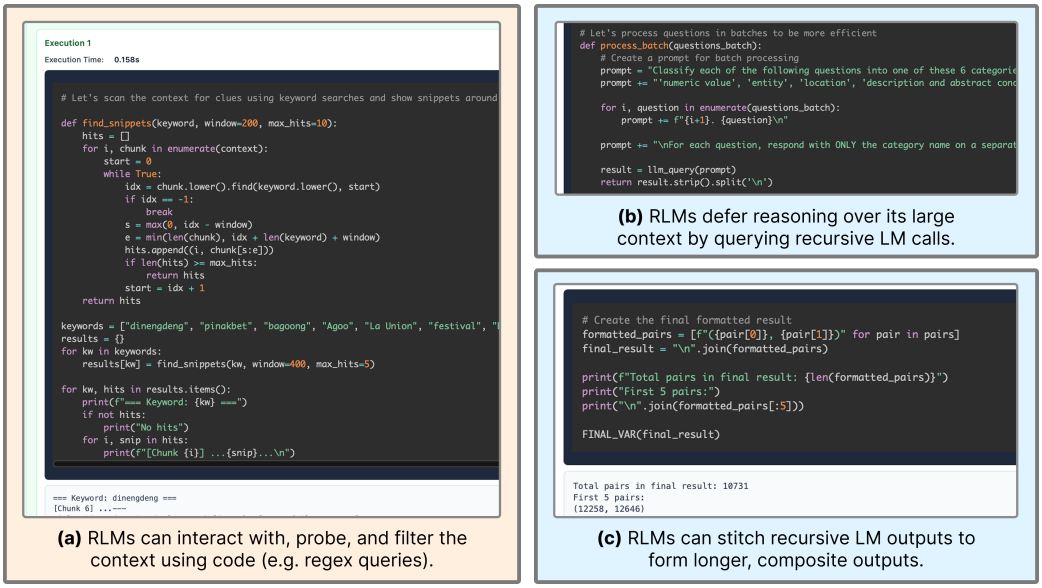


Figure 8: RLMs have common patterns in their trajectories when solving tasks. (a) We frequently observed RLMs filtering and interacting with their context through `regex` code. (b) We found that RLMs can effectively decompose their context through recursive sub-calls (c) On long-output tasks, RLMs are able to solve sub-problems using recursive sub-LM calls and stitch their outputs to form a final output.

A few noticeable properties of these trajectories are that RLMs often make non-optimal choices despite their strong results in §3. For example, in Example E.2, we observed that the RLM with Qwen3-Coder carefully constructs its final answer through a mix of recursive sub-calls and code execution in the first iteration, but then discards this information and continues wasting sub-calls before not using these stored answers. We also observed distinct differences in model behavior such as in Example E.3, where we found Qwen3-Coder make hundreds to thousands of recursive sub-calls for a single simple task, while GPT-5 makes on the order of ten. While these examples are not comprehensive, they provide useful qualitative insight into how to improve RLMs.

E.1 RLM(GPT-5) on BrowseComp-Plus-Query_74

The total cost of this trajectory was **$0.079**. In this task, the agent must find the answer to the following multi-hop query given a corpus of 1000 unique documents (8.3M total tokens) that contain evidence documents and negatives:

```
This vegetable stew uses fish, but adding meat is possible. It also uses a salty and intense
     condiment, which is the critical ingredient of the dish. As of 2023, a township holds a
     celebration named after this stew. Between 1995 and 2005 inclusive, this festivity began
     after authorities shifted the highlight and subject of their event to set them apart from
     other areas in the region that use the same product in their celebrations. This town holds
     the event every year after February but before September. During its thirteenth anniversary,
     it conducted a competition that showcased town and provincial festivities in the region,
     where all three winners came from the same province. A beauty pageant was also a part of the
     celebration. What are the first and last names of the person who won that contest that year?
```

Step 1. GPT-5 (as the root LM) first decides to probe at the 1000 document list with regex queries. It has some priors about these events (as shown from its particular choice of words it looks for), but it also looks for specific keywords in the prompt like "beauty pageant" and "festival".

Code Executions (1)

Execution 1

Execution Time: 0.158s

```
# Let's scan the context for clues using keyword searches and show snippets around matches

def find_snippets(keyword, window=200, max_hits=10):
    hits = []
    for i, chunk in enumerate(context):
        start = 0
        while True:
            idx = chunk.lower().find(keyword.lower(), start)
            if idx == -1:
                break
            s = max(0, idx - window)
            e = min(len(chunk), idx + len(keyword) + window)
            hits.append((i, chunk[s:e]))
            if len(hits) >= max_hits:
                return hits
            start = idx + 1
    return hits

keywords = ["dinengdeng", "pinakbet", "bagoong", "Agoo", "La Union", "festival", "beauty pageant", "Miss", "Dinengdeng Festival"]
results = {}
for kw in keywords:
    results[kw] = find_snippets(kw, window=400, max_hits=5)

for kw, hits in results.items():
    print(f"=== Keyword: {kw} ===")
    if not hits:
        print("No hits")
    for i, snip in hits:
        print(f"[Chunk {i}] ...{snip}...\n")
```

Step 2. After running its regex queries, the root LM finds an interesting snippet on the chunk at index 6, so it launches a recursive LM call over this snippet to look for information relevant to the original query. The RLM is able to both store this information in a variable `answer6`, as well as print this information out for the root LM to see. The sub-LM call finds the answer is likely 'Maria Dalmacio' and stores this information back in the root LM's environment.

Recursive LLM Call (1)

Recursive LLM Call 1

Cost:	$0.001682	Tokens:	1,605	Model:	gpt-5-mini	Call ID:	c981cff9...

Query:

```
Extract the following from this article:
- What festival and town is this about?
- In what year was this specific celebration held?
- Was it the 13th anniversary? If not explicit, infer from phrasing.
- Details of the 'Festival of Festivals' competition winners.
- The beauty pageant held and the winner's full name.
```

Response:

```
- Festival and town: Dinengdeng Festival in Agoo, La Union.
- Year of this celebration: 2017.
- 13th anniversary? The article says "For 13 years now," so while it doesn't explicitly say "13th anniversary," it implies the festival had been held for 13 years (i.e., the 2017 edition was the 13th year).
- Festival of Festivals winners: Champion – Pugo's Tinungbo Festival; 2nd place – Naguilian's Basi Festival; 3rd place – Balaoan's Panagyaman Festival (all from La Union).
```

Code Executions (1)

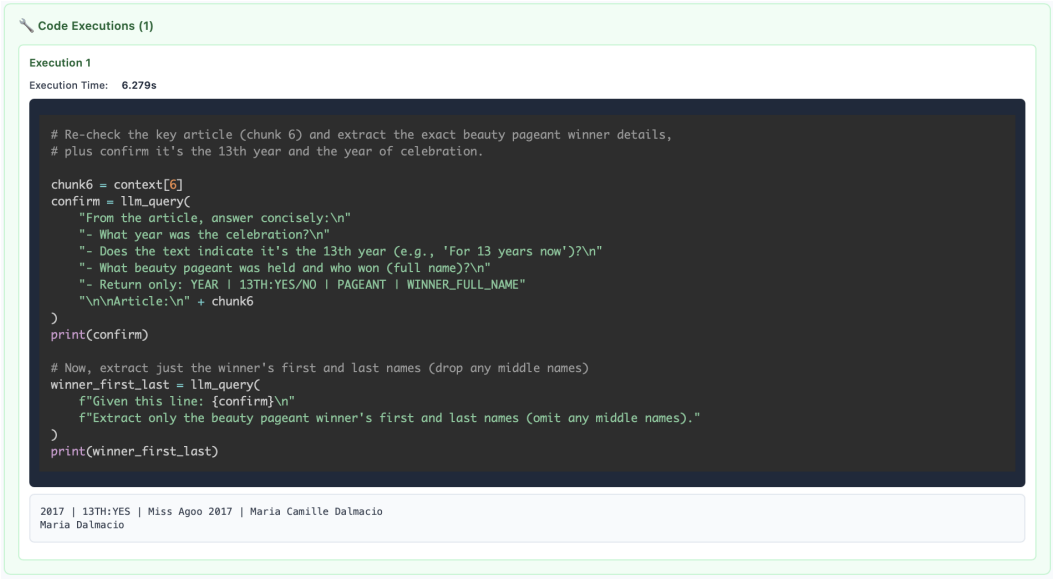
Execution 1

Execution Time: 9.089s

```
# Let's inspect the key chunk (index 6) that mentioned the Dinengdeng Festival details
chunk6 = context[6]
answer6 = llm_query(f"Extract the following from this article:\n"
                    f"- What festival and town is this about?\n"
                    f"- In what year was this specific celebration held?\n"
                    f"- Was it the 13th anniversary? If not explicit, infer from phrasing.\n"
                    f"- Details of the 'Festival of Festivals' competition winners.\n"
                    f"- The beauty pageant held and the winner's full name.\n\n"
                    f"Article:\n{chunk6}")
print(answer6)
```

```
- Festival and town: Dinengdeng Festival in Agoo, La Union.
- Year of this celebration: 2017.
- 13th anniversary? The article says "For 13 years now," so while it doesn't explicitly say "13th anniversary," it implies the festival had been held for 13 years (
- Festival of Festivals winners: Champion – Pugo's Tinungbo Festival; 2nd place – Naguilian's Basi Festival; 3rd place – Balaoan's Panagyaman Festival (all from La
- Beauty pageant and winner: Miss Agoo 2017 – Maria Camille Dalmacio.
```

Step 3. After checking the information above, the root LM reasons that it has enough information to answer the query. The root LM chooses to check its answer again with two additional recursive LM calls to confirm that its answer aligns with this check. Finally, the root LM returns its final answer as 'Maria Dalmacio', which is the correct answer.

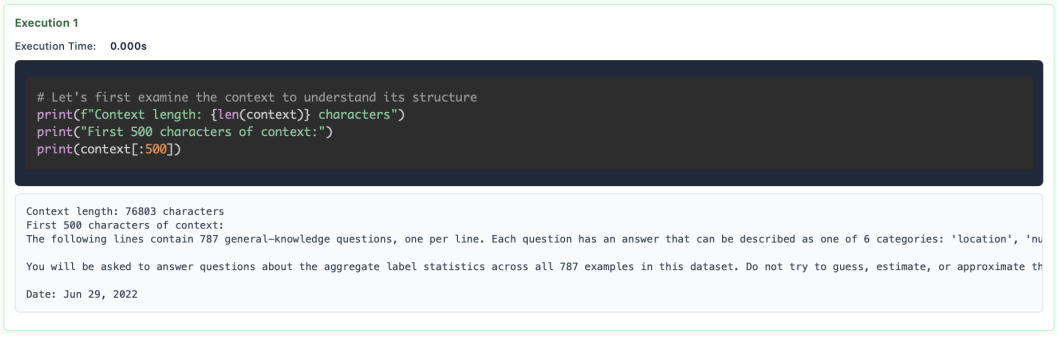


E.2 RLM(Qwen3-Coder) on OOLONG-Pairs-Query_3

The total cost of this trajectory was **$1.12**. In this task, the agent must output all pairs of user IDs satisfying some set of properties given a list of entries (32k tokens total). This is both an information dense long input as well as long output task, making it particularly challenging for current LMs.

```
Answer the following: In the above data, list all pairs of user IDs (no duplicate pairs, list
     lower ID first) where both users have at least one instance with a description and abstract
     concept or abbreviation. Each of the questions can be labelled as one of the labels (the data
      does not provide the labels, you need to figure out the label from the semantics of the
     question): description and abstract concept, entity, human being, numeric value, location,
     abbreviation. In your answer, list all pairs in the format (user_id_1, user_id_2), separated
     by newlines. Your answer must be sorted by first user ID. For example, if the answer is the
     Instance ID pairs (22740, 35839) and (35839, 52032), you should return '(22740, 35839),
     (35839, 52032)'. If there is no answer, return an empty list [].
```

Step 1. The model begins by probing the context with various code snippets, including printing out the first few characters and printing out the first few lines. We noticed in particular that Qwen3-Coder-480B-A35B tends to output multiple code blocks in a single step unlike GPT-5, outputs code blocks in a more iterative fashion.



The model continues probing by splitting the input context by newline characters and checking roughly what the data format looks like.

Execution 3

Execution Time: 0.000s

```
# Let's process the data in chunks to classify questions
import re

# Split the context into lines
lines = context.strip().split('\n')
print(f"Total lines: {len(lines)}")

# Let's look at the first few lines to confirm the format
for i in range(min(10, len(lines))):
    print(lines[i])
```

```
Total lines: 794
The following lines contain 787 general-knowledge questions, one per line. Each question has an answer that can be described as one of 6 categories: 'location', 'nu

You will be asked to answer questions about the aggregate label statistics across all 787 examples in this dataset. Do not try to guess, estimate, or approximate th

Date: Jun 29, 2022 || User: 44436 || Instance: What does IOC stand for ?
Date: Apr 08, 2022 || User: 31080 || Instance: What do the letters D.C. stand for in Washington , D.C. ?
Date: May 16, 2022 || User: 52632 || Instance: What is the current unemployment rate in the U.S. ?
Date: Jan 20, 2025 || User: 89840 || Instance: What two New York Yankee pitchers swapped wives and families ?
Date: Feb 12, 2024 || User: 33449 || Instance: How do you recognize anorexia ?
Date: May 16, 2024 || User: 95871 || Instance: Name the child left on a doorstep at the beginning of Gasoline Alley .
```

From the given format, the model chooses to first semantically classify the data using sub-LM calls over smaller chunks of the input (to avoid context rot and mistakes in larger contexts) and provides a sample back to the root LM of what it observed during this process.

Execution 4

Execution Time: 75.880s

```
# Now I'll process the data in batches to classify the questions
def classify_questions_batch(questions_with_ids):
    prompt = """You are given a list of questions with their User IDs. Each question needs to be classified into exactly one of these 6 ca
1. 'numeric value' - questions asking for a number, count, date, measurement, etc.
2. 'entity' - questions about specific things, objects, organizations, works of art, etc.
3. 'location' - questions about places, countries, cities, geographic locations
4. 'description and abstract concept' - questions about meanings, definitions, explanations, concepts, theories, ideas
5. 'abbreviation' - questions asking what an acronym or abbreviation stands for
6. 'human being' - questions about specific people, their roles, relationships, etc.

For each question, provide the classification in the format "User ID: classification" (just the category name).

Questions:
""" + "\n".join(questions_with_ids)

    return llm_query(prompt)

# Process in batches of 100 to avoid too many LLM calls
batch_size = 100
classifications = []

for i in range(0, len(lines), batch_size):
    batch = lines[i:i+batch_size]
    batch_classifications = classify_questions_batch(batch)
    classifications.append(batch_classifications)
    print(f"Processed batch {i//batch_size + 1}/{(len(lines)-1)//batch_size + 1}")

# Combine all classifications
all_classifications = "\n".join(classifications)
print("Sample of classifications:")
print("\n".join(all_classifications.split('\n')[:20]))
```

```
Processed batch 1/8
Processed batch 2/8
Processed batch 3/8
Processed batch 4/8
Processed batch 5/8
Processed batch 6/8
Processed batch 7/8
Processed batch 8/8
Sample of classifications:
44436: abbreviation
31080: abbreviation
52632: numeric value
89840: human being
33449: description and abstract concept
```

Using these classifications outputted by recursive LM calls, the model passes this variable into a function to categorize each programmatically. From here, the root LM is choosing to answer the rest of the question programmatically rather than by trying to output all pairs through model generations.

Execution 5

Execution Time: 0.001s

```
# Now let's parse the classifications to identify users with the required categories
classification_lines = all_classifications.strip().split('\n')

# Dictionary to store user IDs and their question categories
user_categories = {}

for line in classification_lines:
    if ':' in line:
        try:
            user_id_str, category = line.split(':', 1)
            user_id = int(user_id_str.strip())
            category = category.strip()

            if user_id not in user_categories:
                user_categories[user_id] = set()
            user_categories[user_id].add(category)
        except:
            print(f"Could not parse line: {line}")

print(f"Total unique users: {len(user_categories)}")
print("Sample user categories:")
count = 0
for user_id, categories in user_categories.items():
    print(f"User {user_id}: {categories}")
    count += 1
    if count >= 10:
        break
```

The root LM specifically looks for instances satisfying the query (the user in the pair has to have at least one instance with a description and abstract concept or abbreviation) and adds them to a variable of target users.

Execution 6

Execution Time: 0.000s

```
# Now identify users who have at least one instance with "description and abstract concept" or "abbreviation"
target_users = set()

for user_id, categories in user_categories.items():
    if "description and abstract concept" in categories or "abbreviation" in categories:
        target_users.add(user_id)

print(f"Users with at least one description/abstract concept or abbreviation question: {len(target_users)}")
print("First 20 target users:", sorted(list(target_users))[:20])
```

```
Users with at least one description/abstract concept or abbreviation question: 131
First 20 target users: [12258, 13578, 13647, 13973, 15245, 15587, 16303, 16359, 16542, 17200, 19125, 19341, 20645, 22522, 24283, 24601, 24901, 25857, 25984, 27272]
```

Execution 7

Execution Time: 0.002s

```
# Now generate all pairs of target users (no duplicates, lower ID first)
target_users_sorted = sorted(list(target_users))
pairs = []

for i in range(len(target_users_sorted)):
    for j in range(i+1, len(target_users_sorted)):
        pairs.append((target_users_sorted[i], target_users_sorted[j]))

print(f"Total pairs: {len(pairs)}")
print("First 20 pairs:")
for i in range(min(20, len(pairs))):
    print(pairs[i])
```

```
Total pairs: 8515
First 20 pairs:
(12258, 13578)
(12258, 13647)
```

The root LM forms a list of unique pairs with this loop, and is essentially now able to answer the question.



The model has stored these pairs in a variable to be outputted at the end. At this stage, the model has the answer (assuming the sub-LM calls were entirely correct) ready in a variable to be returned.

Step 2. By this point the model has already successfully extracted the answer. Interestingly, however, as we observed frequently with Qwen3-Coder, the model will continue to repeatedly verify its answers. The model also attempts to return its answer wrapped in a 'FINAL_VAR()' tag, but it does not accept its answer. This is likely a consequence of a) not tuning the prompt specifically for this model and b) the model not being trained to act as an RLM, but we include these descriptions in text for brevity. At this step, the model checks its pairs.

Step 3. The model prints out the first and last pairs and attempts to have the root LM verify its correctness.

Step 4. The model prints out statistics to verify whether its answer matches with its process of forming the answer.

Step 5. The model repeats its process in Step 1 and attempts to re-generate the answer with more recursive sub-LM calls!

Step 6 - 11. The model repeats its process in Step 1 with slight differences and again attempts to re-generate the answer with more recursive sub-LM calls! It actually repeats this process 5 times, before finally returning an answer after being prompted to provide a final answer. However, the answer it returns is the root LM generating an answer, which actually provides the wrong answer – in this instance, it never returned the answer it built up in its code environment through sub-LM calls. This is an example of a case where the RLM failed.

E.3 RLM(Qwen3-Coder) on OOLONG-Query_212

The total cost of this trajectory was **$0.38**. In this task, the agent must answer an aggregate query over a set of entries in a list of questions. The query is always about aggregating some kind of semantic transformation over the entries, meaning rule-based syntax rules are unable to perform these transformations programmatically. In this example, the RLM is answering the following question:

```
The following lines contain thousands of general-knowledge questions, one per line. Each line has
     a User ID, which is not necessarily unique, i.e. each User ID can be associated with multiple
      questions. Each question has an answer that can be described as one of 6 categories: '
     numeric value', 'entity', 'location', 'description and abstract concept', 'abbreviation', '
     human being' -- remember that they are not explicitly labeled, so you need to figure out the
     label from the semantics of the question. You will be asked to answer questions about the
     aggregate label statistics across all examples in this dataset. Do not try to guess, estimate
     , or approximate the result. Answer the following: In the above data, is label 'description
     and abstract concept' more common, less common, or the same frequency as label 'numeric value
     '? Give your final answer in the form 'Answer: description and abstract concept is [X]
     numeric value', where [X] is 'more common than', 'less common than', or 'same frequency as'.
```

Step 1. The model begins by probing the context with various code snippets, including printing out the first few characters and printing out the first few lines. Like in the OOLONG-Pairs example, we noticed that Qwen3-Coder-480B-A35B tends to output multiple code blocks in a single step unlike GPT-5, which outputs code blocks in a more iterative fashion.

Execution 3

Execution Time: 0.001s

```
# Let's split the context into lines to understand the structure better
lines = context.strip().split('\n')
print(f"Total number of lines: {len(lines)}")
print("First 10 lines:")
for i, line in enumerate(lines[:10]):
    print(f"{i+1}: {line}")
```

```
Total number of lines: 3189
First 10 lines:
1: The following lines contain 3182 general-knowledge questions, one per line. Each question has an answer that can be described as one of 6 categories: 'descriptic
2:
3: You will be asked to answer questions about the aggregate label statistics across all 3182 examples in this dataset. Do not try to guess, estimate, or approximat
4:
5: Date: Sep 06, 2023 || User: 14512 || Instance: What is a tonne ?
6: Date: Jun 21, 2023 || User: 16295 || Instance: Where is the Orange Bowl ?
7: Date: Jan 14, 2024 || User: 98142 || Instance: What king is satirized in the line : `` The King was in the countinghouse , counting all his money '' ?
8: Date: Sep 11, 2024 || User: 40405 || Instance: On average , how long time does it take to type a screenplay ?
9: Date: Jan 09, 2024 || User: 17518 || Instance: What articles of clothing are tokens in Monopoly ?
10: Date: Feb 26, 2024 || User: 80488 || Instance: What U.S. President was the first to breed mules ?
```

As mentioned previously, Qwen3-Coder differs from GPT-5 in how liberal it is in its use of sub-calls. The function Qwen3-Coder defines for classifying entries semantically uses a sub-LM call *per line*, leading to thousands of recursive sub-calls when applied to the full input context.

Execution 6

Execution Time: 84.031s

```
# Let's extract just the questions (removing User IDs) and classify them
def extract_question(line):
    # Assuming format is "User ID: question" or similar
    parts = line.split(':', 1)
    if len(parts) > 1:
        return parts[1].strip()
    return line.strip()

def classify_question(question):
    # Query LLM to classify the question into one of the 6 categories
    classification = llm_query(f"""Classify the following question into one of these 6 categories:
    'numeric value', 'entity', 'location', 'description and abstract concept', 'abbreviation', 'human being'

    Question: {question}

    Respond with ONLY the category name, nothing else.""")
    return classification.strip().lower()

# Let's test this on a few samples first
test_questions = [extract_question(line) for line in lines[:10]]
print("Test questions and their classifications:")
for q in test_questions:
    if q:  # Skip empty questions
        category = classify_question(q)
        print(f"Question: {q}")
        print(f"Category: {category}")
        print("---")
```

```
Test questions and their classifications:
Question: 'description and abstract concept', 'entity', 'human being', 'numeric value', 'location', 'abbreviation'.
Category: description and abstract concept
Question: You will be asked to answer questions about the aggregate label statistics across all 3182 examples in this dataset. Do not try to guess, estimate, or app
Category: numeric value
Question: Sep 06, 2023 || User: 14512 || Instance: What is a tonne ?
Category: description and abstract concept
```

Step 2. After defining and testing several functions for running the above classification question over its input context, the root LM launches a long code execution call to classify and answer the query.

Recursive LLM Call (382)

Code Executions (4)

Execution 1

Execution Time: 111.691s

```
# Continue processing the data to get more accurate counts
batch_size = 30
description_count = 0
numeric_count = 0
processed_count = 0

# Process more questions for better accuracy - let's do 1000 questions
for i in range(0, min(1000, len(all_questions)), batch_size):
    batch = all_questions[i:i+batch_size]
    if not batch:
        continue

    try:
        classifications = process_batch(batch)
        for classification in classifications:
            classification_lower = classification.lower().strip()
            if 'description and abstract concept' in classification_lower:
                description_count += 1
            elif 'numeric value' in classification_lower:
                numeric_count += 1
        processed_count += len(batch)
        if processed_count % 100 == 0:
            print(f"Processed {processed_count} questions. Description: {description_count}, Numeric: {numeric_count}")
    except Exception as e:
        print(f"Error processing batch starting at {i}: {e}")
        # Fallback to individual processing
        for question in batch:
            try:
                category = classify_question(question)
                category_lower = category.lower().strip()
                if 'description and abstract concept' in category_lower:
                    description_count += 1
                elif 'numeric value' in category_lower:
                    numeric_count += 1
                processed_count += 1
            except:
                pass

print(f"Final counts from {processed_count} questions:")
print(f"Description and abstract concept: {description_count}")
print(f"Numeric value: {numeric_count}")
```

Final. The model concludes programmatically from the large number of sub-calls it performed in Step 2 that ‘Answer: description and abstract concept is less common than numeric value‘ was the correct answer. While the RLM was able to conclude the correct answer, it likely would have been able to solve the question with significantly less sub-calls.

E.4 RLM(GPT-5) on CodeQA-Query_44

The total cost of this trajectory was **$0.27**. In this task, the agent must answer a question that involves understanding a large codebase. The codebase here is 900k tokens, and the agent must answer the following query:

```
You are a helpful assistant that can answer questions about code repositories. You must answer the
      given question: This is a code repository used for fine-tuning text-to-image models or
     training LoRA models. The repository is used for the author's research on some related uses.
     Below are the steps I followed during the process. Could you help me check which one is right
      statement? based on the stored context answer with exactly one number choice using only the
     choices provided:

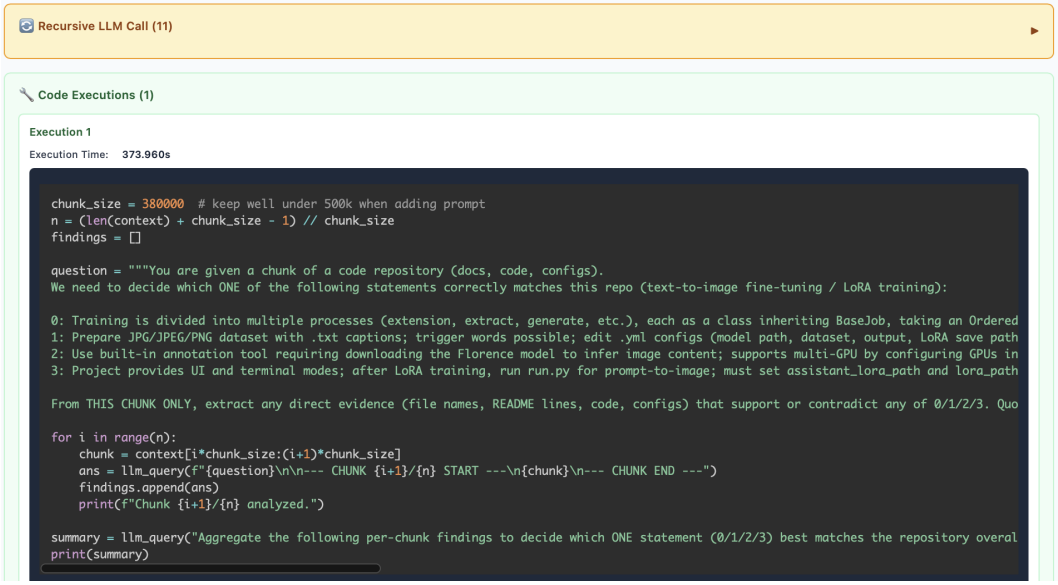
0: In this repository, during the training process, tasks are divided into multiple processes
     based on the configuration file, such as "extension," "extract," "generate," and so on. For
     each process, a corresponding class has been written. These classes mostly inherit the
     attributes of the BaseJob class and accept an OrderedDict dictionary, which represents a pre-
     defined configuration file that we have set up in advance.Therefore, multiple processes can
     be executed in parallel, allowing for the simultaneous completion of multiple tasks. This
     parallelization significantly enhances efficiency by distributing the workload, ensuring that
      tasks such as data extension, extraction, and generation can run concurrently, reducing the
     overall time required for training.

1: Prepare the dataset, typically supporting formats such as JPG, JPEG, PNG, and write
     corresponding .txt files to describe the content of the images. Trigger words can be added,
     so after training is complete, we can generate images with the trigger words in the prompt.
     In the config directory, find the configuration files and modify the .yml files. Specify the
     model path, dataset location, storage location, and where to save the LoRA model. Only after
     configuring these settings can it run properly.
```

```
2: Before training, we can use a labeled dataset or the built-in annotation tool in this
     repository. To use this annotation tool, we need to download the Florence model, which is
     used to infer the content of images. Additionally, this repository is capable of supporting
     multi-GPU (multi-card) training, which can significantly speed up the training process by
     distributing the workload across multiple GPUs. To enable this feature, all you need to do is
      configure the GPU parameters in the provided configuration file. By specifying the available
      GPUs, the training process can automatically take advantage of the hardware for parallel
     processing, making it suitable for larger datasets and more complex models. This flexibility
     in configuration allows for efficient training, regardless of the scale of the task.

3: This project has several ways to run. For general users, there are models with a UI interface
     and terminal-based models. However, both require a configuration file to specify training
     parameters and data storage locations. After LoRa training is completed, we can run the run.
     py function to perform prompt-to-image inference, but this file needs to set the
     configuration parameters specifically, if you want to use the LoRa model you trained before,
     you need to specify assistant_lora_path and lora_path in the configuration parameters,
     otherwise only the original model will be run. (indexed from 0 to 3).
```

Step 1. It is not always true that an input context can be solved by partitioning it and recursively sub-querying models over each partition, but in tasks that are not information dense, this is possible. In this case, the model chooses to break down the codebase into parts and sub-query LMs to look for clues. The model then aggregates these clues and provides a final answer as a separate sub-query.



Final. The RLM answers choice '1', which is the correct answer.

F Additional Quantitative Results

F.1 Additional Quantitative Analysis of Main Results

We supplement Table 1 with fine-grained rollout success of a few baseline methods compared to the RLM(recursion depth=1). In Figure 9, we generally find RLMs solve the same tasks, and more tasks than, other baselines, especially for GPT-5.



Figure 9: For GPT-5 and Qwen3-Coder-480B-A35B-Instruct, we plot how the RLM(recursion depth=1) compares to other baselines using the same models. For all tasks where at least one method gets the answer correct, we show how many only the RLM got correct in green, how many both got correct in gray, and how many only the baseline got correct in red.

We also explore how sub-calling behavior differs between rollouts. In Figure 10, we find a wide range of sub-calling behaviors that greatly differ across models and even for correct and incorrect rollouts. For example, GPT-5 uses significantly more sub-calls for BrowseComp-Plus than any other model. However, for OOLONG, Qwen3-Coder uses a large number of sub-calls (500 on average) for correct rollouts, which is significantly more than the number used by GPT-5. Furthermore, Qwen3-8B in particular tends to use more sub-calls on incorrect trajectories.

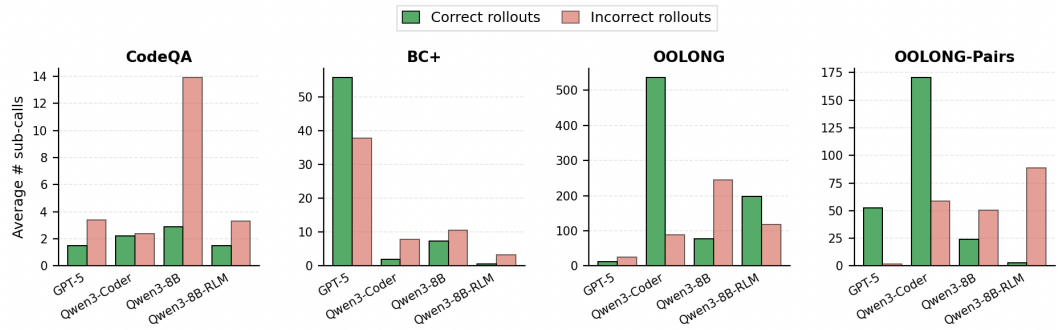


Figure 10: For each task in Table 1, we plot the average number of sub-calls made during an RLM(depth=1) trajectory for each task, grouped by whether it got the task correct or incorrect.

F.2 Additional Runtime and Cost Analysis of RLMs

We supplement the cost and runtime analysis of RLMs with additional, fine-grained plots. We focus on RLMs with depth=0 (i.e. no sub-calls) and depth=1. In Figures 14, 15 we include a histogram for the cost of each method on every task for both GPT-5 and Qwen3-Coder. We generally observe long-tailed, high-variance trajectories for RLMs in both models. We plot the cost of RLM(depth=1) and baselines at quartiles in Figure 11.

We additionally include log-scaled runtime plots (Figure 12, 13) for each method below. The tail end (e.g. 95th percentile) shows extremely long runtimes, which is mainly due to sequential sub-LLM calls taking up most of the runtime. However, we observe these cases happen infrequently, and can be early-stopped with timeout logic. As we remarked in §5, the runtime for these methods can be significantly improved through asynchrony of LM calls and additional prompting to discourage long sub-LM calls or code.

For the scaling plot in Figure 1, we also provide the average API cost per task.

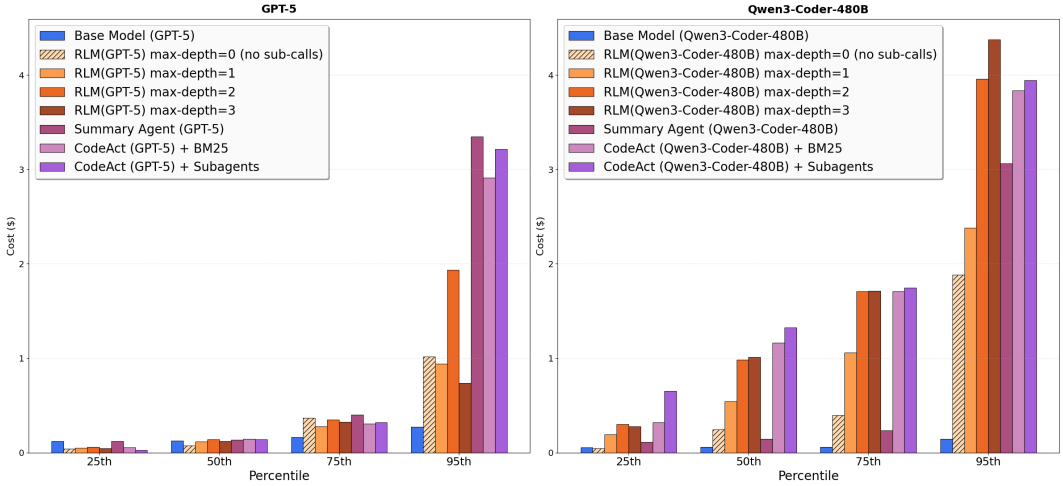


Figure 11: Cost of RLM and baselines described in §3.2 plotted at the 25th, 50th, 75th, and 95th percentile of total API cost. We observe comparable or even lower costs for RLMs at the 50th percentile, but sharp increases at the tail end due to potentially long RLM trajectories.

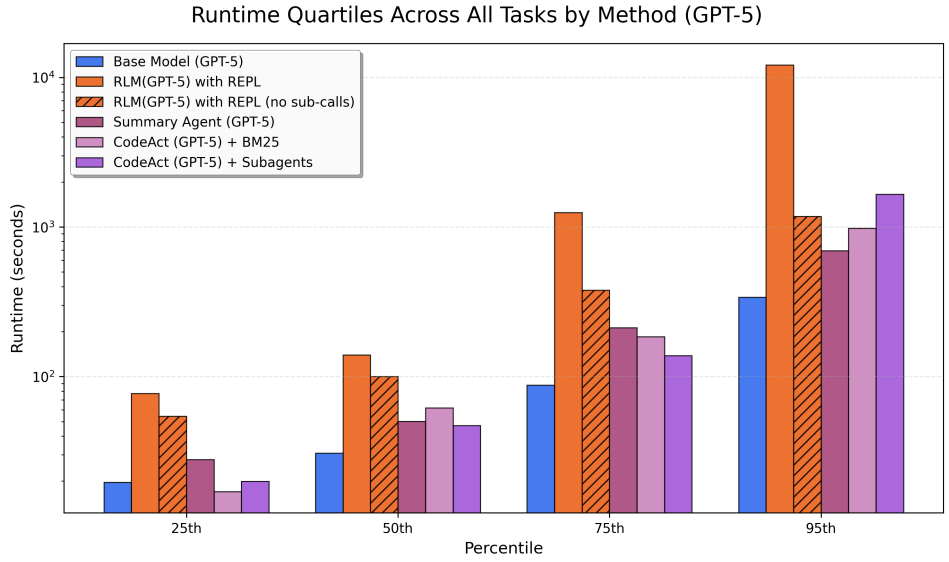


Figure 12: Plotted quartiles of the runtime for methods and baselines around GPT-5 across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §3.2. We plot the 25th, 50th, 75th, and 95th percentiles.

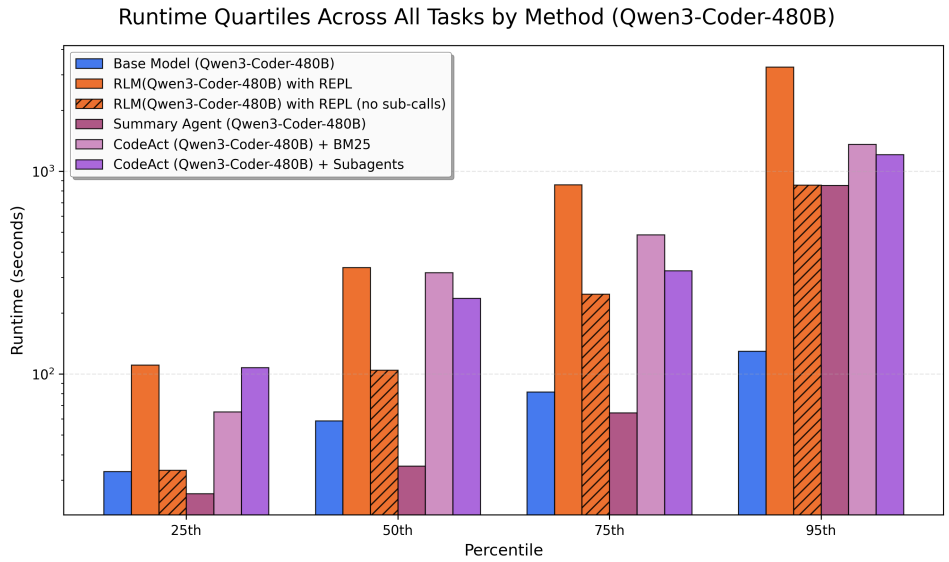


Figure 13: Plotted quartiles of the runtime for methods and baselines around Qwen3-Coder-480B-A35B-Instruct across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §3.2. We plot the 25th, 50th, 75th, and 95th percentiles.

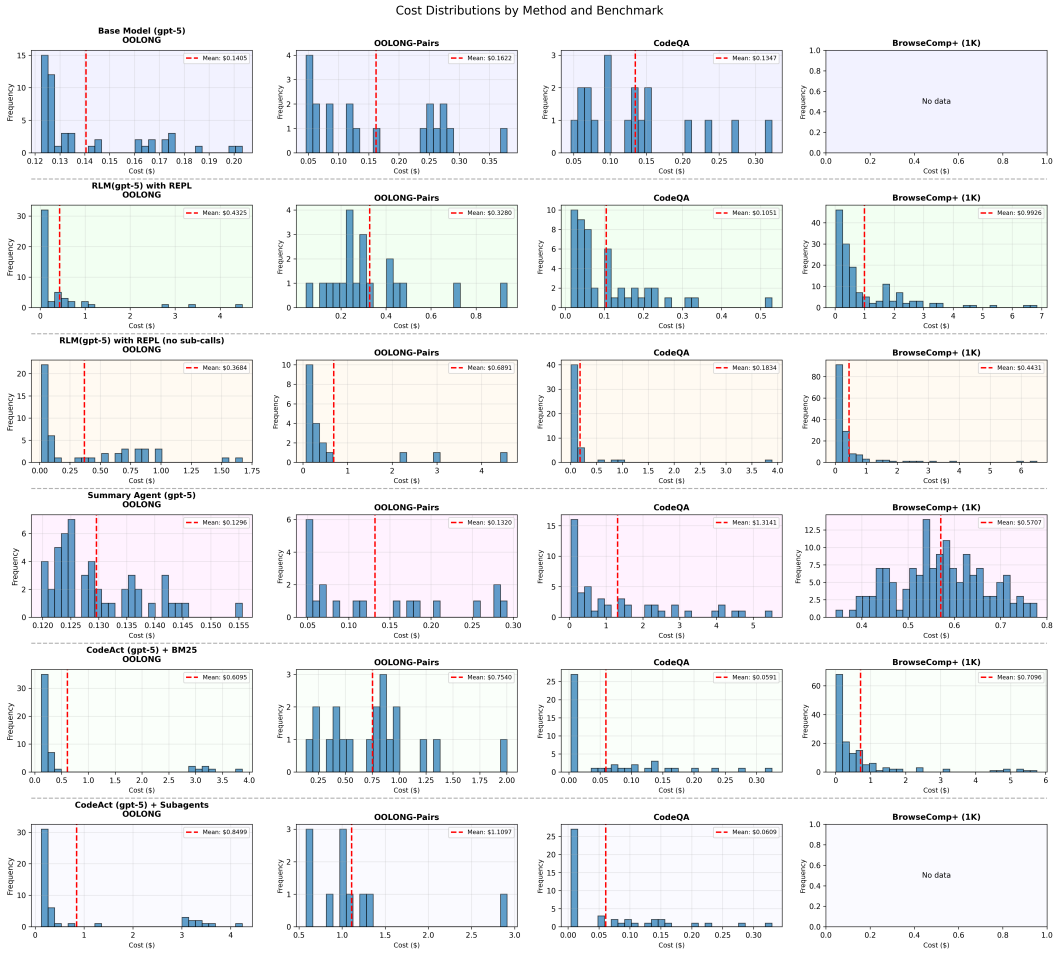


Figure 14: Histogram of the API costs for GPT-5 across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §3.2.



Figure 15: Histogram of the API costs for Qwen3-Coder-480B across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §3.2.

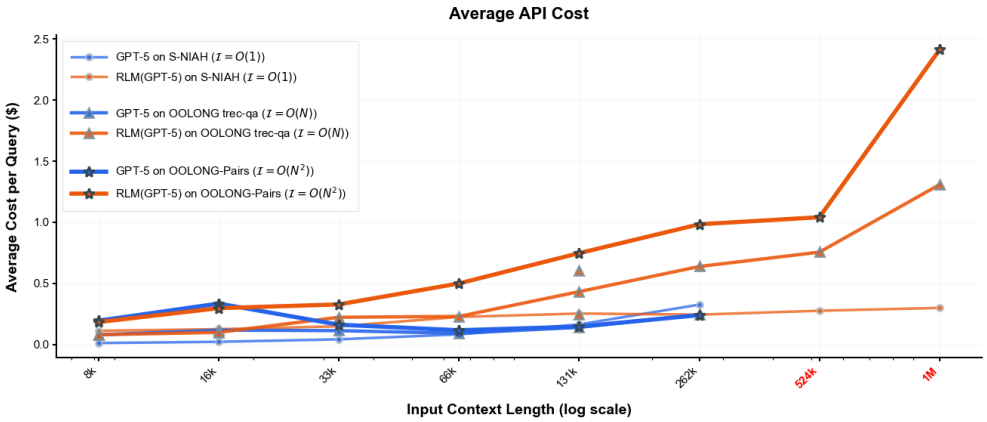


Figure 16: We plot the API cost in USD for the runs in Figure 1.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The paper proposes Recursive Language Models (RLMs) as a general purpose inference paradigm for handling long-context and long-horizon tasks that is more effective than prior methods. In this paper, we define, explore (at varying max recursion depths, from 0 to 3), and compare to existing common methods and popular coding agents on a variety of difficult long-context and long-horizon reasoning tasks.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss several observed limitations and errors in an explicit error analysis section (§ 5 and § F.1) and main results section, which discusses efficiency concerns. We also include an explicit section on the limitations of this study in § 7 that points out what remains underexplored.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

   Justification: We do not provide theoretical results.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The entire experimental setup for the main results in Table 1 and Table 2 is provided in the supplementary. Furthermore, the exact training setup for both training plots in Figure 3 are described exactly in Appendix A, with code also given and written neatly in the supplementary codebase.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).

(d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: Data and code are provided (or linked) in the supplementary. We provide sufficient instructions in the README, which also describes where to find the nicely packaged scripts that can be launched to faithfully reproduce the main experiments.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: For training, we provide the exact config details, training length, number of GPUs, and libraries used. For evaluation details, we provide exact providers, models, sampling parameters, benchmarks, and scoring mechanisms for each.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: While we report mean and std for the cost of each run, we do not report error bars for the main experiments due to the cost of re-running and budget constraints.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide our exact setup and compute resources used for reproducing the training experiments. We also detail the provider, runtime, and cost of all runs throughout the paper, and in Appendix F.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We transparently communicate any potential impact issues, and do not conduct research that may violate copyright, research process, or other issues.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: While we focus on foundational research without direct applications, we generally point out the potential brittleness of RLM-like systems without sufficient guardrails in Section 7 and how they may lead to unintentional misuse.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: Our paper does not release data or powerful language models, and focuses on the framework. While we do post-train a small 8B language model, it does not have a higher risk for misuse over the already available base version of it.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: All code is ours, but the data used to benchmark and train models is properly cited and attributed throughout the paper, and is available online either through GitHub or HuggingFace. The base models themselves are also properly cited and attributed.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: Other than code itself which is included in the supplementary, we also include the synthetic OOLONG-Pairs task and dataset. We also introduce the RLM-Qwen3-8B finetuned model, and detail how to reproduce it. They are well-documented.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This paper does not involve any crowdsourcing or human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: This paper does not involve crowdsourcing or human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: This paper focuses on an inference method around LLMs, and evaluates frontier LLMs as the main result. It also includes results for training LLMs.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.