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 strategy 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 successfully handle inputs up to two orders of magnitude beyond model context windows and, even for shorter prompts, dramatically outperform the quality of base LLMs and common long-context scaffolds across four diverse long-context tasks, while having comparable (or cheaper) cost per query.

1 Introduction

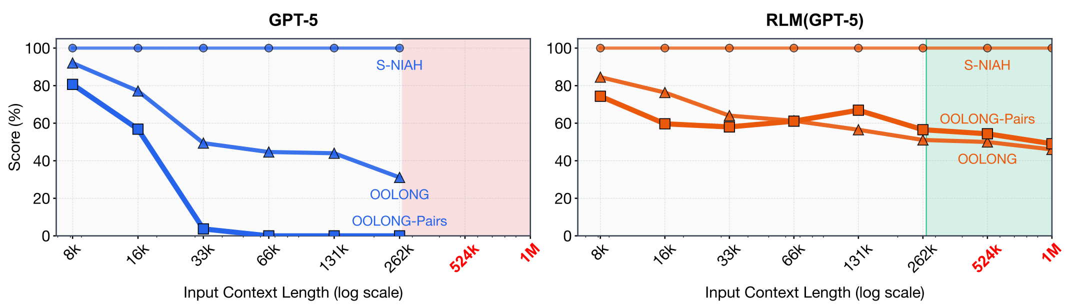


Figure 1: A comparison of GPT-5 and a corresponding RLM on three long-context tasks of increasing complexity: **S-NIAH**, **OOLONG**, and **OOLONG-Pairs**. For each task, we scale the input length from 2^{13} to 2^{18}. GPT-5 performance degrades significantly as a function of both input length and task 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, methods, and benchmarks are in §2.

Despite rapid progress in reasoning and tool use, modern language models still have limited context lengths and, even within these limits, appear to inevitably exhibit *context rot* (Hong et al., 2025), the phenomenon illustrated in the left-hand side of Figure 1 where the quality of even frontier models like GPT-5 degrades quickly as context gets longer. Though we expect context lengths to steadily rise through improvements to training, architecture, and infrastructure, we are interested in *whether it possible to dramatically 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.

We study this question through the lens of scaling inference-time compute. We draw broad inspiration from *out-of-core* algorithms, in which data-processing systems with a small but fast main memory can process far larger datasets by cleverly managing how data is fetched into memory. Inference-time methods for dealing with what are in essence long-context problems are very common, though typically task-specific. One general and increasingly popular inference-time approach in this space is context condensation or compaction (Khattab et al., 2021; Smith, 2025; OpenAI, 2025; Wu et al., 2025), in which the context is repeatedly summarized once it exceeds a length threshold. Unfortunately, compaction is rarely expressive enough for tasks that require dense access

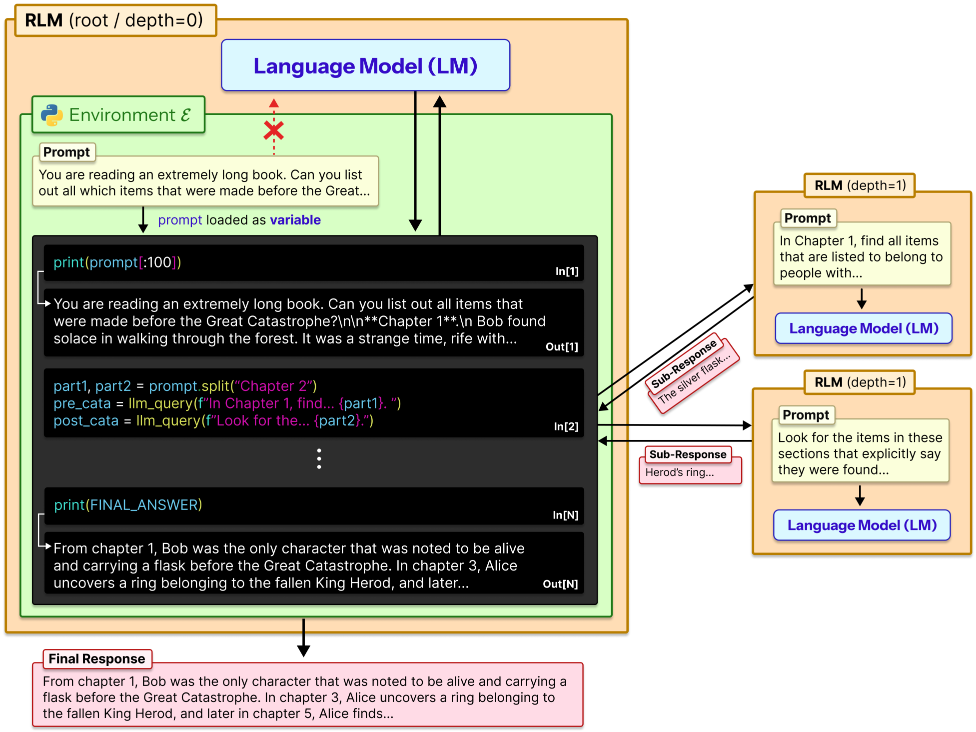


Figure 2: A Recursive Language Model (RLM) treats prompts as part of the environment. It loads the input prompt as a variable inside a Python REPL environment $\mathcal{E}$ and writes code to peek into, decompose, and invoke itself recursively over programmatic snippets of the variable.

to many parts of the prompt, as it presumes in effect that *some* details that appear early in the prompt can safely be forgotten to make room for new content.

We introduce **Recursive Language Models** (**RLM**s), a general-purpose inference paradigm for dramatically scaling the effective input and output lengths of modern LLMs. The key insight is that long 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 can symbolically interact with*.

As Figure 2 illustrates, an RLM exposes the same external interface as an LLM: 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, in the code it produces, to programmatically construct sub-tasks on which they can invoke themselves recursively.

By treating the prompt as an object in the external environment, this simple design of RLMs tackles a foundational limitation in the many prior approaches (Anthropic, 2025; Sentient, 2025; Schroeder et al., 2025; Sun et al., 2025), which focus on recursive decomposition of the *tasks* but cannot allow their input to scale beyond the context window of the underlying LLM.

We evaluate RLMs using a frontier closed model (GPT-5; OpenAI 2025) and a frontier open model (Qwen3-Coder-480B-A35B; Team 2025) across four diverse tasks with varying levels of complexity for 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. We find that RLMs demonstrate extremely strong performance even at the 10M+ token scale, and dramatically outperform all other approaches at long-context processing, in most cases by double-digit percentage gains while maintaining a comparable or lower cost. In particular, as demonstrated in Figure 1 exhibit far less severe degradation for longer contexts and more sophisticated tasks.

2 SCALING LONG CONTEXT TASKS

Recent work (Hsieh et al., 2024; Goldman et al., 2025; Hong et al., 2025) has successfully argued that the *effective* context window of LLMs can often be much shorter than a model's physical maximum number of tokens. Going further, we hypothesize that the effective context window 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, while previous generations of models struggled with NIAH tasks, frontier models can reliably solve these tasks in RULER (Hsieh et al., 2024) even in the 1M+ token settings. Nonetheless, the same models struggle even at shorter lengths on OOLONG (Bertsch et al., 2025), which is a task where the answer depends explicitly on almost every line in the prompt.[1]

2.1 TASKS

Grounded in this intuition, we design our empirical evaluation around tasks where we are able to vary not just the lengths of the prompts, but also consider different scaling patterns for problem complexity. We loosely characterize each task by *information density*, i.e. how much information an agent is required to process to answer the task, and how this scales with different input sizes.

S-NIAH. Following the single needle-in-the-haystack task in RULER (Hsieh et al., 2024), we consider a set of 50 single needle-in-the-haystack tasks that require finding a specific phrase or number in a large set of unrelated text. These tasks require finding a single answer regardless of input size, and as a result scale roughly constant in processing costs with respect to input length.

BrowseComp-Plus (1K documents) (Chen et al., 2025). A multi-hop question-answering benchmark for DeepResearch (OpenAI, 2025) questions that requires reasoning over multiple different documents. The benchmark provides a verified offline corpus of 100K documents that is guaranteed to contain gold, evidence, and hard negative documents for each task. Following Sun et al. (2025), we use 150 randomly sampled tasks as our evaluation set; we provide 1000 randomly chosen documents to the model or agent, in which the gold and evidence documents are guaranteed to exist. We report the percentage of correct answers. The answer to each task requires piecing together information from several documents, making these tasks more complicated than **S-NIAH** despite also requiring a constant number of documents to answer.

OOLONG (Bertsch et al., 2025). A long reasoning benchmark that requires examining and transforming chunks of the input semantically, then aggregating these chunks to form a final answer. We report scoring based on the original paper, which scores numerical answers as $\texttt{score}(\hat{y}) = 0.75^{|y-\hat{y}|}$ and other answers as exact match. We focus specifically on the `trec_coarse` split, which is a set of 50 tasks over a dataset of questions with semantic labels. Each task requires using nearly all entries of the dataset, and therefore scales linearly in processing costs relative to the input length.

OOLONG-Pairs. We manually modify the `trec_coarse` split of OOLONG to include 20 new queries that specifically require aggregating *pairs* of chunks to construct the final answer. In Appendix E.1, we explicitly provide all queries in this benchmark. We report F1 scores over the answer. Each task requires using nearly all *pairs* of entries of the dataset, and therefore scales quadratically in processing costs relative to the input length.

LongBench-v2 CodeQA (Bai et al., 2025). A multi-choice code repository understanding split from LongBench-v2 that is challenging for modern frontier models. We report the score as the percentage of correct answers. Each task requires reasoning over a fixed number of files in a codebase to find the right answer.

[1]This intuition 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.

2.2 METHODS AND BASELINES

We compare RLMs against other commonly used task-agnostic methods. For each of the following methods, we use two contemporary LMs, GPT-5 with medium reasoning (OpenAI, 2025) and default sampling parameters and Qwen3-Coder-480B-A35B (Yang et al., 2025) using the sampling parameters described in Team (2025), chosen to provide results for a commercial and open frontier model respectively. For Qwen3-Coder, we compute costs based on the Fireworks provider (Fireworks, 2025). In addition to evaluating the base model on all tasks, we also evaluate the following methods and baselines:

RLM with REPL. We implement an RLM that loads its context as a string in the memory of a Python REPL environment. The REPL environment also loads in a module that allows it to query a sub-LM inside the environment. The system prompt is fixed across all experiments (see Appendix D). 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 powerful tradeoff between the capabilities of RLMs and the cost of the recursive calls.

RLM with REPL, no sub-calls. We provide an ablation of our method. In it, the REPL environment loads in the context, but is not able to use sub-LM calls. In this setting, the LM can still interact with its context in a REPL environment before providing a final answer.

Summary agent. Following Sun et al. (2025); Wu et al. (2025); Yu et al. (2025), we consider an iterative agent that invokes a summary of the context as it is filled. For example, given a corpus of documents, it will iteratively view the documents and summarize when full. In cases where the provided context exceeds the model window, the agent will chunks the input to fit within the model context window and invoke the same strategy over these chunks. For GPT-5, due to the extremely high cost of handling large token inputs, we use GPT-5-nano for compaction and GPT-5 to provide the final answer.

CodeAct (+ BM25). 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, it does not offload its prompt to the code environment, and instead provides it directly to the LM. Furthermore, following Jimenez et al. (2024); Chen et al. (2025), we equip this agent with a BM25 (Robertson & Zaragoza, 2009) retriever that indexes the input context for tasks where this is appropriate.

3 RESULTS AND DISCUSSION

We focus our main experiments in Table 1 on the benchmarks described in §2.1. Furthermore, we explore how frontier model and RLM performance degrades 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 the method ran into input context limits.

Model	**CodeQA**	**BrowseComp+ (1K)**	**OOLONG**	**OOLONG-Pairs**
Task Length N (tokens)	23K-4.2M	6M-11M	131K	32K
Qwen3-Coder-480B				
Base Model	20.00* (\$0.13 ± \$0.08)	0.00* (N/A) ± (N/A)	36.00 (\$0.06 ± \$0.00)	0.06 (\$0.05 ± \$0.01)
CodeAct (+ BM25)	24.00* (\$0.17 ± \$0.08)	12.66 (\$0.39 ± \$0.50)	38.00 (\$1.51 ± \$1.09)	0.28 (\$1.54 ± \$0.35)
Summary agent	50.00 (\$1.26 ± \$1.50)	38.00 (\$8.98 ± \$2.12)	44.06 (\$0.15 ± \$0.01)	0.31 (\$0.05 ± \$0.00)
RLM	56.00 (\$0.92 ± \$1.23)	44.66 (\$0.84 ± \$0.63)	**48.00** (\$0.61 ± \$0.49)	**23.11** (\$1.02 ± \$0.52)
RLM (no sub-calls)	**66.00** (\$0.18 ± \$0.58)	**46.00** (\$0.82 ± \$0.69)	43.50 (\$0.32 ± \$0.13)	17.34 (\$1.77 ± \$1.23)
GPT-5				
Base Model	24.00* (\$0.13 ± \$0.07)	0.00* (N/A) ± (N/A)	44.00 (\$0.14 ± \$0.02)	0.04 (\$0.16 ± \$0.10)
CodeAct (+ BM25)	22.00* (\$0.06 ± \$0.08)	51.00 (\$0.71 ± \$1.20)	38.00 (\$0.61 ± \$1.06)	24.67 (\$0.75 ± \$0.43)
Summary agent	58.00 (\$1.31 ± \$1.46)	70.47 (\$0.57 ± \$0.10)	46.00 (\$0.13 ± \$0.01)	0.01 (\$0.13 ± \$0.09)
RLM	**62.00** (\$0.11 ± \$0.10)	**91.33** (\$0.99 ± \$1.22)	**56.50** (\$0.43 ± \$0.85)	**58.00** (\$0.33 ± \$0.20)
RLM (no sub-calls)	58.00 (\$0.18 ± \$0.56)	88.00 (\$0.44 ± \$0.90)	36.00 (\$0.37 ± \$0.42)	43.93 (\$0.69 ± \$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 input tasks 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 to the theoretical costs of extending a base model's context window – on BrowseComp-Plus (1K), the cost of GPT-5-mini ingesting 6-11M input tokens is $\$1.50 - \$2.75$, while RLM(GPT-5) has an average cost of $\$0.99$ and outperforms both the summarization 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 on tasks that fit well within the model's context window. On OOLONG, the RLM with GPT-5 and Qwen3-Coder outperform the base model by 28.4% and 33.3% respectively. On OOLONG-Pairs, both GPT-5 and Qwen3-Coder make little progress with F1 scores of $<0.1\%$, while the RLM using these models achieve F1 scores of 58.00% and 23.11% respectively, highlighting the emergent capability of RLMs to handle extremely information-dense tasks.

Observation 2: The REPL environment 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. Even without sub-calling capabilities, our ablation of the RLM is able to scale beyond the context limit of the model, and outperform the base model and other task-agnostic baselines on most long context settings. On the CodeQA and BrowseComp+ tasks with Qwen3-Coder, this ablation is able to outperform the RLM by 17.9% and 3% respectively.

On information-dense tasks like OOLONG or OOLONG-Pairs, we observed several cases where recursive LM sub-calling is necessary. In §3.1, 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. Across all information-dense tasks, RLMs outperform the ablation without sub-calling by 10%-59%.

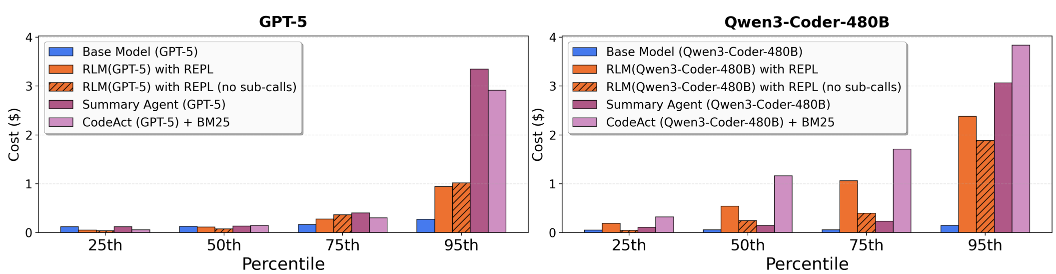


Figure 3: Cost of RLM and baselines described in §2.2 plotted at the 25th, 50th, 75th, and 95th percentile of total API cost. We observe comparable or even lower costs for RLMs at the 50th percentile, but sharp increases at the tail end due to potentially long RLM trajectories.

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 a context with lengths ranging from 2^{13} to 2^{18}. Furthermore, each benchmark can be loosely categorized by different processing costs of the input context with respect to length (roughly constant, linear, and quadratic respectively). In Figure 1, we directly compare an RLM using GPT-5 to base GPT-5 on each task – we find that GPT-5 performance degrades significantly faster for more complex tasks, while RLM performance degrades but at a much slower rate, which aligns with the findings of Goldman et al. (2025). For context lengths beyond 2^{14}, the RLM consistently outperforms GPT-5.

Furthermore, RLM costs scale proportionally to the the complexity of the task, while still remaining in the same order of magnitude of cost as GPT-5 (see Figure 9 in Appendix C). In §3.1, we explore what choices the RLM makes in these settings that causes these differences in cost. Lastly, in this setting, we also observe that the base LM outperforms RLM in the small input context regime. By construction, an RLM has strictly more representation capacity than an LM: the choice of an environment that calls the root LM is equivalent to the base LM; in practice, however, we observe that

RLM performance is slightly worse on smaller input lengths, suggesting a tradeoff point between when to use a base LM and when to use an RLM.

Observation 4: The inference cost of RLMs remain comparable to a base model call but are high variance due to differences in trajectory lengths. RLMs iteratively interact with their context until they find a suitable answer, leading to large differences in iteration length depending on task complexity. In Figure 3, we plot the quartile costs for each method across all experiments in Table 1 excluding BrowseComp-Plus (1K), as the base models cannot fit any of these tasks in context. For GPT-5, the median RLM run is cheaper than the median base model run, but many outlier RLM runs are significantly more expensive than any base model query. However, compared to the summarization baseline which ingests the entire input context, RLMs are up to $3\times$ cheaper while maintaining stronger performance across all tasks because the model is able to selectively view context.

We additionally report runtime numbers of each method in Figures 5, 6 in Appendix C, 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.

Observation 5: RLMs are a model-agnostic inference strategy, but different models exhibit different overall decisions on context management and sub-calling. While GPT-5 and Qwen3-Coder-480B both exhibit strong performance as RLMs relative to their base model and other baselines, they also exhibit different performance and behavior across all tasks. On BrowseComp-Plus in particular, RLM(GPT-5) nearly solves all tasks while RLM(Qwen3-Coder) struggles to solve half.

We note that the RLM system prompt is fixed for each model across all experiments and is not tuned for any particular benchmark. Between GPT-5 and Qwen3-Coder, the only difference in the prompt is an extra line in the RLM(Qwen3-Coder) prompt warning against using too many subcalls (see Appendix D). We provide an explicit example of this difference in example B.3, where RLM(Qwen3-Coder) performs the semantic transformation in OOLONG as a separate sub-LM call per line while GPT-5 is conservative about sub-querying LMs.

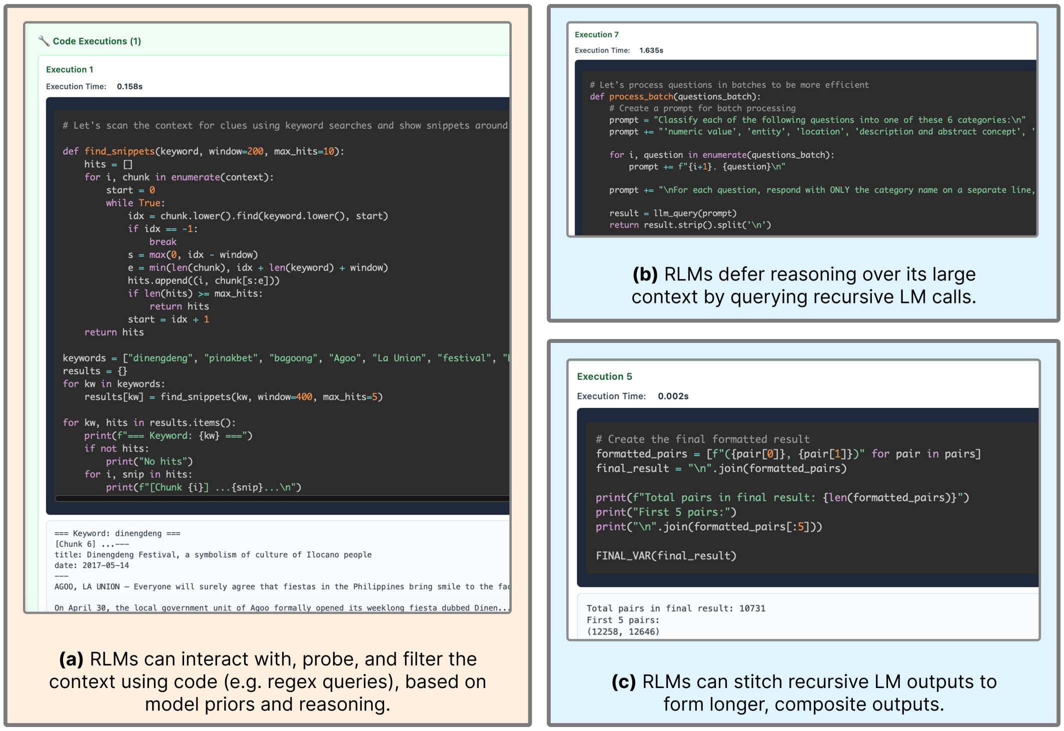
3.1 Emergent Patterns in RLM Trajectories

Even without explicit training, RLMs exhibit interesting context management and problem decomposition behavior. We select several examples of snippets from RLM trajectories to understand how they solve long context problems and where they can improve. We discuss particular examples of interesting behavior here, with additional examples in Appendix B.

Filtering input information using code execution based on model priors. A key intuition for why the RLM abstraction can maintain strong performance on huge inputs without exploding costs is the LM's ability to filter input context without explicitly seeing it. Furthermore, model priors enable the RLM to narrow the search space and process fewer input tokens. As an example, in Figure 4a, we observed RLM(GPT-5) using `regex` queries search for chunks containing keywords in the original prompt (e.g. "festival") and phrases it has a prior about (e.g. "La Union"). Across most trajectories, a common strategy we observed was probing the context by printing a few lines back to the root LM, then filtering based on its observations.

Chunking and recursively sub-calling LMs. RLMs defer essentially unbounded-length reasoning chains to sub-(R)LM calls. The choice of decomposition can greatly affect task performance, especially for information-dense problems. In our experiments, we did not observe complicated partitioning strategies beyond uniform chunking or keyword searches. In Figure 4b, RLM(Qwen3-Coder) chunks by newline in a 1000+ line context from OOLONG.

Answer verification through sub-LM calls with small contexts. We observed several instances of answer verification made by RLMs through sub-LM calls. Some of these strategies implicitly avoid context rot by using sub-LMs to perform verification (see example B.1), while others solely use code execution to programmatically verify answers are correct. In some instances, however, the answer verification is redundant and significantly increases the cost per task — in example B.3, we observed a trajectory on OOLONG where the model tries to reproduce its correct answer more than five times before choosing the incorrect answer in the end.



(a) RLMs can interact with, probe, and filter the context using code (e.g. regex queries), based on model priors and reasoning.

(b) RLMs defer reasoning over its large context by querying recursive LM calls.

(c) RLMs can stitch recursive LM outputs to form longer, composite outputs.

Figure 4: RLMs have common patterns in their trajectories when solving tasks. (a) We frequently observed RLMs filtering and interacting with their context through code like `regex` queries. (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.

Passing recursive LM outputs through variables for long output tasks. RLMs are able to produce essentially unbounded tokens well beyond the limit of the base LM by returning variables in the REPL as output. Through the REPL, the RLM can iteratively construct these variables as a mixture of programmatic and sub-(R)LM output calls. We observed this strategy used heavily in OOLONG-Pairs trajectories, where the RLM stored the output of sub-LM calls over the input in variables and stitched them together to form a final answer (see Figure 4c).

4 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, which uses summarization or truncation to compress the input context at the cost of potentially losing fine-grained information. For example, MemWalker (Chen et al., 2023) constructs a tree-like data structure of the input that the LM can navigate when answering long context questions. ReSum (Wu et al., 2025) is another work that 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 are different from prior work 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), DisCIPL (Grand 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. RLMs, on the other hand, are enabled by an extremely simple intuition (i.e., placing the prompt as part of the external environment) to *symbolically* manipulate arbitrarily long strings and to iteratively refine their recursion via execution feedback from the persistent REPL environment.

5 Limitations and Future Work

While RLMs show strong performance on tasks beyond the context window limitations of existing LMs at reasonable inference costs, the optimal mechanism for implementing RLMs remains underexplored. We focused on synchronous sub-calls inside of a Python REPL environment, but we note that alternative strategies involving asynchronous sub-calls and sandboxed REPLs can potentially significantly reduce the runtime and inference cost of RLMs. Furthermore, we chose to use a max recursion depth of one (i.e. sub-calls are LMs); while we found strong performance on existing long-context benchmarks, we believe that future work should investigate deeper layers of recursion.

Lastly, we focused our experiments on evaluating RLMs using *existing* frontier models. Explicitly training models to be used as RLMs (e.g. as root or sub-LMs) could provide additional performance improvements – as we found in §3.1, current models are inefficient decision makers over their context. 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 frontier models (Zelikman et al., 2022; 2024).

6 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. 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. 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 Matej Sirovatka, Ofir Press, Sebastian Müller, Simon Guo, and Zed Li for helpful feedback.

References

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`.

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.

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. 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.

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.

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

OpenAI. Gpt-5 system card. Online; August 7, 2025, 2025. URL https://openai.com/blog/gpt-5-system-card/.

OpenAI. Codex cli: A lightweight coding agent for your terminal, 2025. 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`.

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`.

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

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. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 11888–11898, 2023.

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

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`.

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 andsen 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. *arXiv preprint arXiv:2408.02248*, 2024.

A NEGATIVE RESULTS: THINGS WE TRIED THAT DID NOT WORK.

Drawing inspiration from Redmon & 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.

B 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 (we also provide the raw trajectories and a visualizer in our codebase), so we describe each step and show specific examples when relevant.

A few noticeable properties of these trajectories are that RLMs often make non-optimal choices despite their strong results in §2. For example, in Example B.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 B.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.

B.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 pagent" and "festival".

Code Executions (1)

Execution 1

Execution Time: 0.158s

```python
# 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.

Code Executions (1)

Execution 1

Execution Time: 6.279s

```
# Re-check the key article (chunk 6) and extract the exact beauty pageant winner details,
# plus confirm it's the 13th year and the year of celebration.

chunk6 = context[6]
confirm = llm_query(
    "From the article, answer concisely:\n"
    "- What year was the celebration?\n"
    "- Does the text indicate it's the 13th year (e.g., 'For 13 years now')?\n"
    "- What beauty pageant was held and who won (full name)?\n"
    "- Return only: YEAR | 13TH:YES/NO | PAGEANT | WINNER_FULL_NAME"
    "\n\nArticle:\n" + chunk6
)
print(confirm)

# Now, extract just the winner's first and last names (drop any middle names)
winner_first_last = llm_query(
    f"Given this line: {confirm}\n"
    f"Extract only the beauty pageant winner's first and last names (omit any middle names)."
)
print(winner_first_last)
```

```
2017 | 13TH:YES | Miss Agoo 2017 | Maria Camille Dalmacio
Maria Dalmacio
```

B.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, which makes outputs in a more iterative fashion.

Execution 1

Execution Time: **0.000s**

```python
# Let's first examine the context to understand its structure
print(f"Context length: {len(context)} characters")
print("First 500 characters of context:")
print(context[:500])
```

```
Context length: 76803 characters
First 500 characters of context:
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
```

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**

```python
# 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

```python
# 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 generaetions.

Execution 5

Execution Time: 0.001s

```python
# 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 abstraction 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.

Execution 8

Execution Time: 0.002s

```
# Format the final answer as required
if pairs:
    formatted_pairs = [f"({pair[0]}, {pair[1]})" for pair in pairs]
    final_result = "\n".join(formatted_pairs)
    print("Final result (first 50 lines):")
    print("\n".join(formatted_pairs[:50]))
else:
    final_result = "[]"
```

```
Final result (first 50 lines):
(12258, 13578)
(12258, 13647)
(12258, 13973)
(12258, 15245)
(12258, 15587)
(12258, 16303)
(12258, 16359)
(12258, 16542)
(12258, 17200)
(12258, 19125)
```

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 difference 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.

B.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 makes outputs in a more iterative fashion.

Execution 3

Execution Time: **0.001s**

```python
# 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**

```python
# 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):
```

```
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.

B.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.

Recursive LLM Call (11)

Code Executions (1)

Execution 1

Execution Time: 373.960s

```
chunk_size = 380000  # keep well under 500k when adding prompt
n = (len(context) + chunk_size - 1) // chunk_size
findings = []

question = """You are given a chunk of a code repository (docs, code, configs).
We need to decide which ONE of the following statements correctly matches this repo (text-to-image fine-tuning / LoRA training):

0: Training is divided into multiple processes (extension, extract, generate, etc.), each as a class inheriting BaseJob, taking an Ordered
1: Prepare JPG/JPEG/PNG dataset with .txt captions; trigger words possible; edit .yml configs (model path, dataset, output, LoRA save path
2: Use built-in annotation tool requiring downloading the Florence model to infer image content; supports multi-GPU by configuring GPUs in
3: Project provides UI and terminal modes; after LoRA training, run run.py for prompt-to-image; must set assistant_lora_path and lora_path

From THIS CHUNK ONLY, extract any direct evidence (file names, README lines, code, configs) that support or contradict any of 0/1/2/3. Quo

for i in range(n):
    chunk = context[i*chunk_size:(i+1)*chunk_size]
    ans = llm_query(f"{question}\n\n--- CHUNK {i+1}/{n} START ---\n{chunk}\n--- CHUNK END ---")
    findings.append(ans)
    print(f"Chunk {i+1}/{n} analyzed.")

summary = llm_query("Aggregate the following per-chunk findings to decide which ONE statement (0/1/2/3) best matches the repository overal
print(summary)
```

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

C ADDITIONAL RUNTIME AND COST ANALYSIS OF RLMS

We supplement the cost and runtime analysis of RLMs with additional, fine-grained plots. In Figures 7, 8 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 additionally include log-scaled runtime plots for each method below. As we remarked in §3.1, 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.

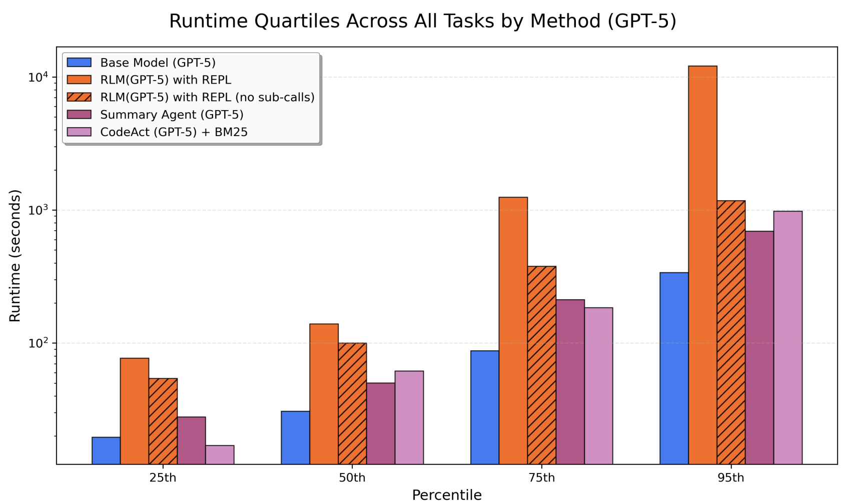


Figure 5: Plotted quartiles of the runtime GPT-5 across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §2.2. We plot the 25th, 50th, 75th, and 95th percentiles.

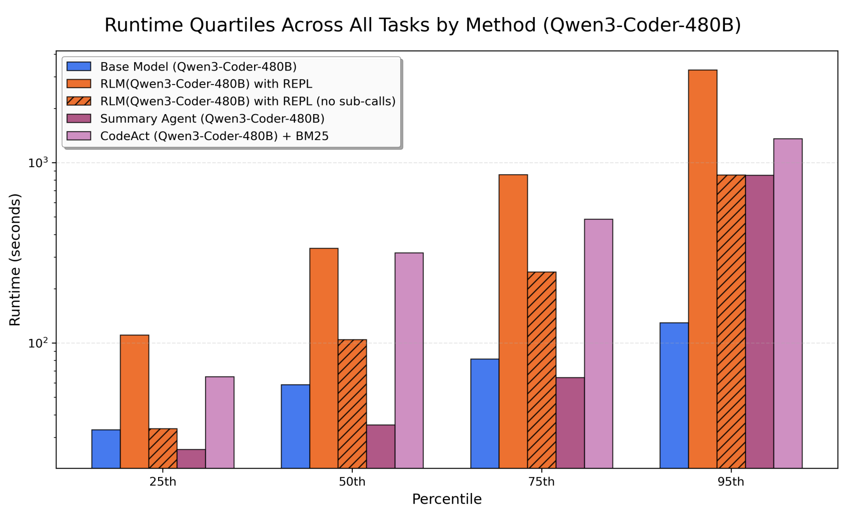


Figure 6: Plotted quartiles of the runtime Qwen3-Coder-480B across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §2.2. We plot the 25th, 50th, 75th, and 95th percentiles.

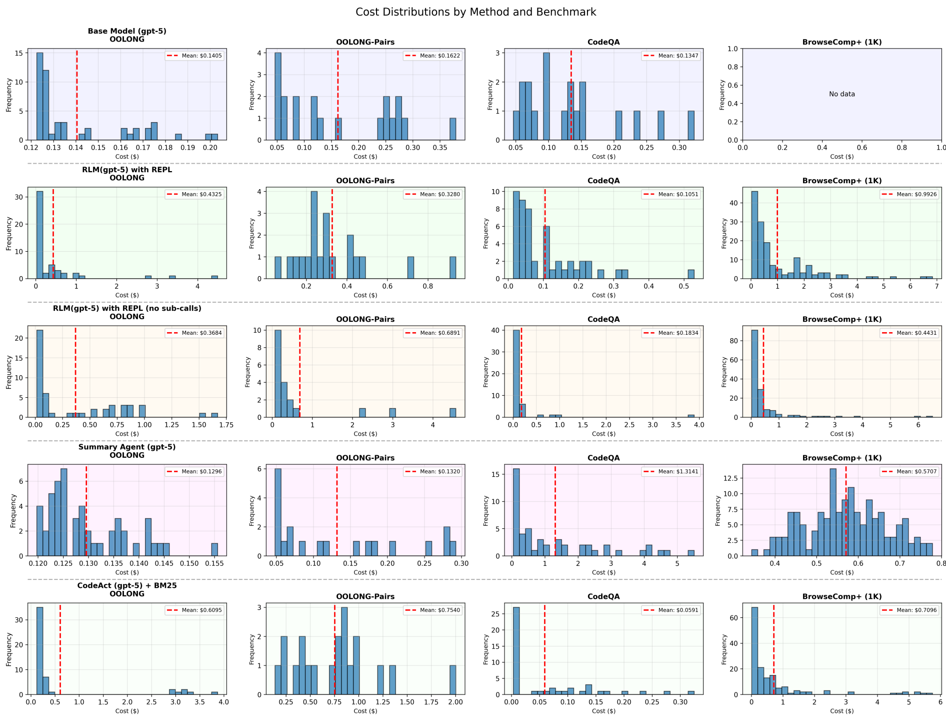


Figure 7: Histogram of the API costs for GPT-5 across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §2.2.

Figure 8: Histogram of the API costs for Qwen3-Coder-480B across OOLONG, OOLONG-Pairs, CodeQA, and BrowseComp+ (1K) for all methods described in §2.2.

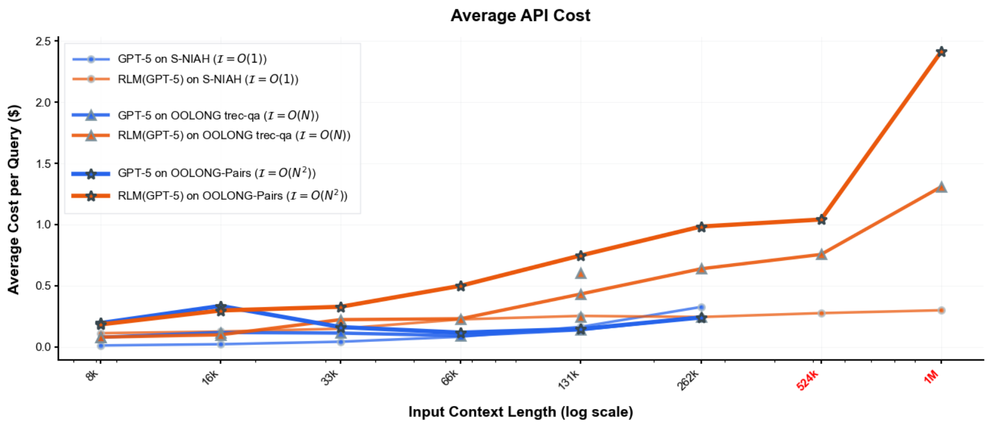


Figure 9: We plot the API cost in USD for the runs in Figure 1.

D Additional Methods and Baseline Details

D.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! In this section, we provide the system prompt used for all methods in §2.1 (other than the base model, which does not include a system prompt).

(1a) The system prompt for **RLM with REPL** 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.
+
```

(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.
````

D.2 Summary agent baseline

The summarization agent baseline follows the scaffold presented in Sun et al. (2025); Wu et al. (2025); Yu et al. (2025), which also mimics 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+, where the summary agent using Qwen3-Coder is nearly 20× 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.

E Additional Benchmark Details

We provide additional details about the benchmarks used to evaluate RLMs in §2.

E.1 OOLONG-Pairs Benchmark

To create OOLONG-Pairs, we synthetically generate 20 new 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 predicing the semantic mapping for each entry.

Ensuring $\mathcal{I} \approx O(N^2)$ on OOLONG-Pairs. We noticed that many tasks that aggregate over pairs of entries could 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), so we explicitly created questions that ask for all pairs satisfying some properties.

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.

E.2 Scaling Huge Document Corpuses in BrowseComp+

In addition to the BrowseComp+ (Chen et al., 2025) results for $k = 1000$ documents in §3, 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).

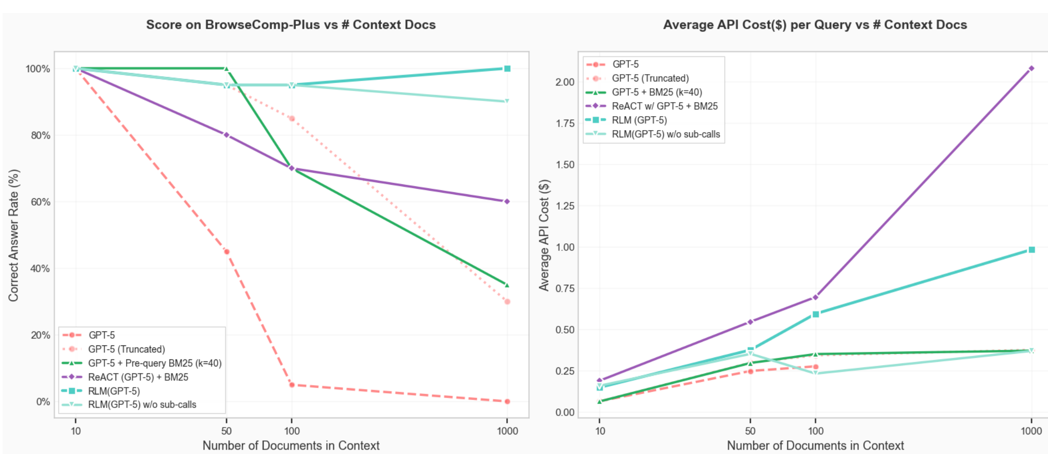


Figure 10: We plot the performance and API cost per answer of various methods using GPT-5 on 20 random queries in BrowseComp-Plus given increasing numbers of documents in context. Only the iterative methods (RLM, ReAct) maintain reasonable performance at 100+ documents.

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 increase.

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.