Recursive Language Models

We propose Recursive Language Models (RLMs), an inference strategy where language models can decompose and recursively interact with input context of unbounded length through REPL environments.

AUTHORS
Alex Zhang
Omar Khattab

AFFILIATIONS
MIT CSAIL
MIT CSAIL

PUBLISHED
Oct. 15, 2025

The full paper is now available here: https://arxiv.org/abs/2512.24601v1.

You can find the official codebase for Recursive Language Models (RLMs) here: https://github.com/alexzhang13/rlm

tl;dr

We explore language models that **recursively call themselves or other LLMs** before providing a final answer. Our goal is to enable the processing of essentially unbounded input context length and output length and to mitigate degradation "context rot".

We propose **Recursive Language Models**, or **RLM**s, a general inference strategy where language models can decompose and recursively interact with their input context as a variable. We design a specific instantiation of this where GPT-5 or GPT-5-mini is queried in a Python REPL environment that stores the user's prompt in a variable.

We demonstrate that an **RLM using GPT-5-mini outperforms GPT-5** on a split of the most difficult long-context benchmark we got our hands on (OOLONG [1]) by more than **double** the number of correct answers, and is **cheaper** per query on average! We also construct a new long-context Deep Research task from BrowseComp-Plus [2]. On it, we observe that RLMs outperform other methods like ReAct + test-time indexing and retrieval over the prompt. Surprisingly, we find that RLMs also do not degrade in performance when given 10M+ tokens at inference time.

We are excited to share these very early results, as well as argue that RLMs will be a powerful paradigm very soon. We think that RLMs trained explicitly to recursively reason are likely to represent the next milestone in **general-purpose inference-time scaling** after CoT-style reasoning models and ReAct-style agent models.

We have a compressed summary in the original tweet: https://x.com/a1zhang/status/1978469116542337259

We also now have a minimal implementation for people to build on top of: https://github.com/alexzhang13/rlm-minimal

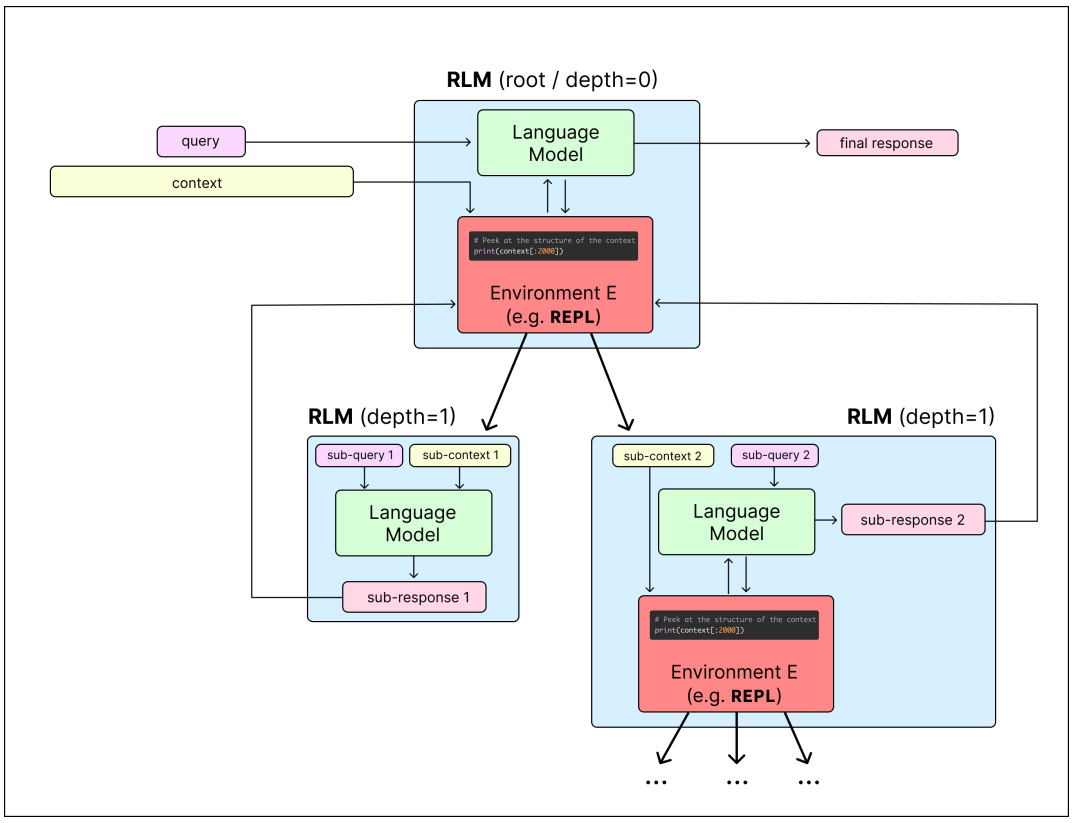


Figure 1. An example of a recursive language model (RLM) call, which acts as a mapping from text → text, but is more flexible than a standard language model call and can scale to near-infinite context lengths. An RLM allows a language model to interact with an environment (in this instance, a REPL environment) that stores the (potentially huge) context, where it can recursively sub-query "itself", other LM calls, or other RLM calls, to efficiently parse this context and provide a final response.

Prelude: Why is “long-context” research so unsatisfactory?

There is this well-known but difficult to characterize phenomenon in language models (LMs) known as “context rot”. Anthropic defines context rot as “[when] the number of tokens in the context window increases, the model’s ability to accurately recall information from that context decreases”, but many researchers in the community know this definition doesn’t *fully* hit the mark. For example, if we look at popular needle-in-the-haystack benchmarks like RULER, most frontier models actually do extremely well (90%+ on 1-year old models).

I asked my LM to finish carving the pumpkin joke it started yesterday. It said, “Pumpkin? What pumpkin?” — the context completely rotted.

But people have noticed that context rot is this weird thing that happens when your Claude Code history gets bloated, or you chat with ChatGPT for a long time — it’s almost like, as the conversation goes on, the model gets…dumber? It’s sort of this well-known but hard to describe failure mode that we don’t talk about in our papers because we can’t benchmark it. The natural solution is something along the lines of, “well maybe if I split the context into two model calls, then combine them in a third model call, I’d avoid this degradation issue”. We take this intuition as the basis for a recursive language model.

Recursive Language Models (RLMs).

A recursive language model is a thin wrapper around a LM that can spawn (recursive) LM calls for intermediate computation — from the perspective of the user or programmer, it is the same as a model call. In other words, you query a RLM as an “API” like you would a LM, i.e. `rlm.completion(messages)` is a direct replacement for `gpt5.completion(messages)`. We take a **context-centric view** rather than a **problem-centric view** of input decomposition. This framing retains the functional view that we want a system that can answer a particular **query** over some associated **context**:

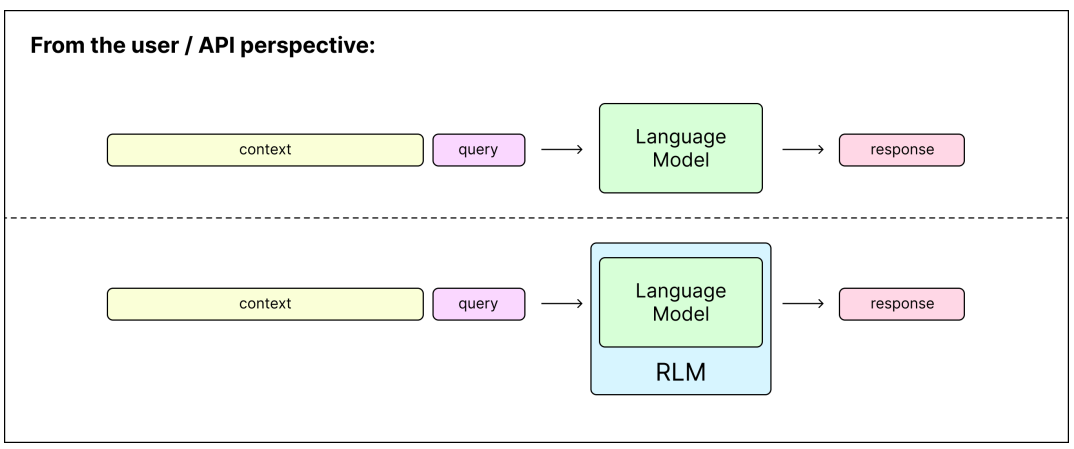


Figure 2. A recursive language model call replaces a language model call. It provides the user the illusion of near infinite context, while under the hood a language model manages, partitions, and recursively calls itself or another LM over the context accordingly to avoid context rot.

Under the hood, a RLM provides only the **query** to the LM (which we call the **root LM**, or LM with depth=0), and allows this LM to interact with an **environment**, which stores the (potentially huge) **context**.

We choose the **environment** to be a loop where the LM can write to and read the output of cells of a Python REPL Notebook (similar to a Jupyter Notebook environment) that is pre-loaded with the **context** as a variable in memory. The **root LM** has the ability to call a recursive LM (or LM with depth=1) inside the REPL **environment** as if it were a function in code, allowing it to naturally peek at, partition, grep through, and launch recursive sub-queries over the **context**. **Figure 3** shows an example of how the RLM with a REPL **environment** produces a final answer.

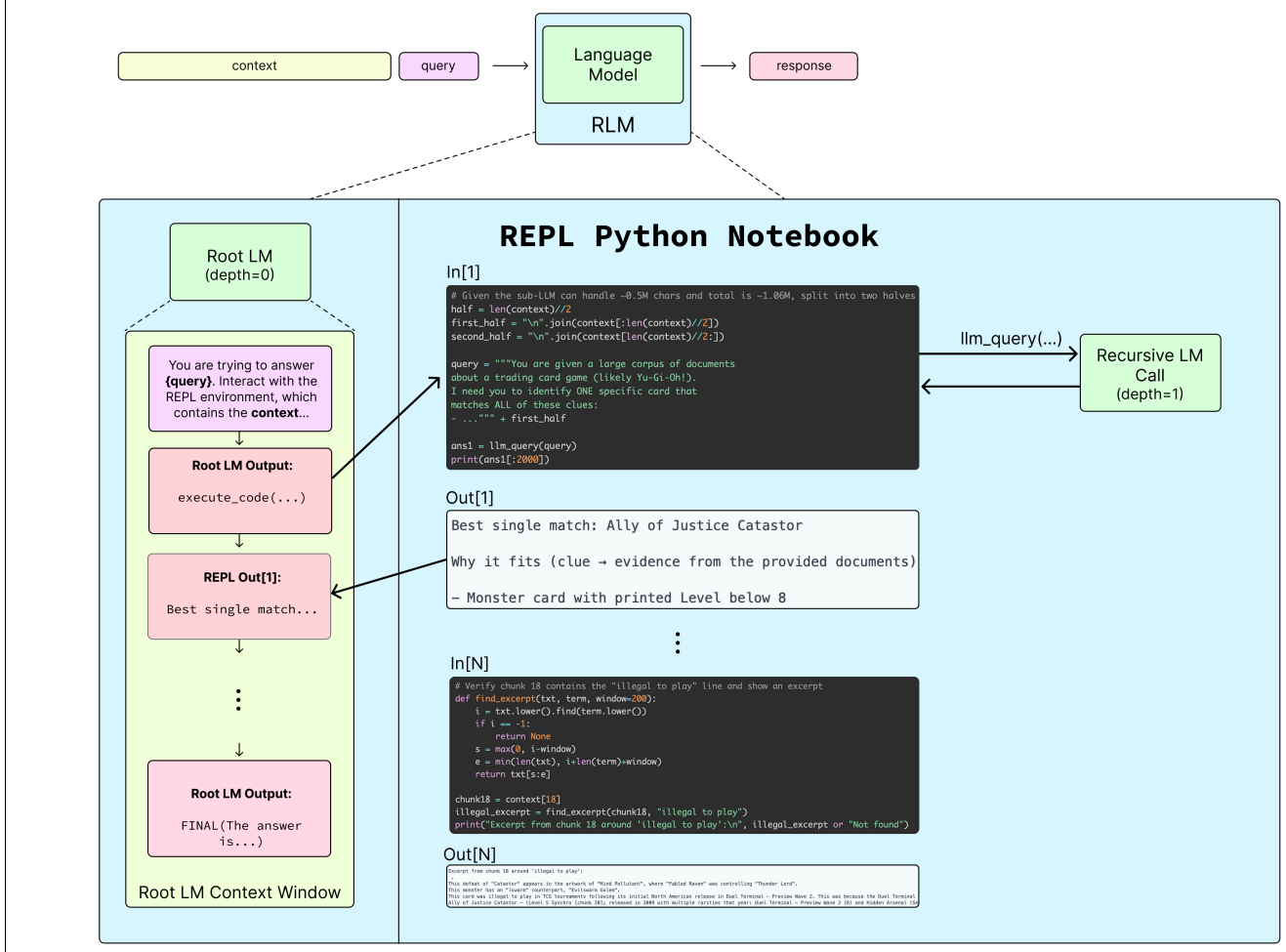


Figure 3. Our instantiation of the RLM framework provides the root LM the ability to analyze the context in a Python notebook environment, and launch recursive LM calls (depth=1) over any string stored in a variable. The LM interacts by outputting code blocks, and it receives a (truncated) version of the output in its context. When it is done, it outputs a final answer with `FINAL(...)` tags or it can choose to use a string in the code execution environment with `FINAL_VAR(...)`.

When the **root LM** is confident it has an answer, it can either directly output the answer as `FINAL(answer)`, or it can build up an answer using the variables in its REPL environment, and return the string inside that answer as `FINAL_VAR(final_ans_var)`.

This setup yields several benefits that are visible in practice:

1. The context window of the root LM is rarely clogged — because it never directly sees the entire context, its input context grows slowly.
2. The root LM has the flexibility to view subsets of the context, or naively recurse over chunks of it. For example, if the query is to find a needle-in-the-haystack fact or multi-hop fact, the root LM can use `regex` queries to roughly narrow the context, then launch recursive LM calls over this context. This is particularly useful for arbitrary long context inputs, where indexing a retriever is expensive on the fly!
3. The context can, in theory, be any modality that can be loaded into memory. The root LM has full control to view and transform this data, as well as ask sub-queries to a recursive LM.

Relationship to test-time inference scaling. We are particularly excited about this view of language models because it offers another axis of scaling test-time compute. The trajectory in which a language model chooses to interact with and recurse over its context is entirely learnable, and can be RL-ified in the same way that reasoning is currently trained for frontier models. Interestingly, it does not directly require training models that can handle huge context lengths because **no single language model call should require handling a huge context**.

RLMs with REPL environments are powerful. We highlight that the choice of the **environment** is flexible and not fixed to a REPL or code environment, but we argue that it is a good choice. The two key design choices of recursive language models are 1) treating the prompt as a Python variable, which can be processed programmatically in arbitrary REPL flows. This allows the LLM to figure out what to peek at from the long context, at test time, and to scale any decisions it wants to take (e.g., come up with its own scheme for chunking and recursion adaptively) and 2) allowing that REPL environment to make calls back to the LLM (or a smaller LLM), facilitated by the decomposition and versatility from choice (1).

We were excited by the design of CodeAct [3], and reasoned that adding recursive model calls to this system could result in significantly stronger capabilities — after all, LM function calls are incredibly powerful. However, we argue that RLMs fundamentally view LM usage and code execution differently than prior works: the **context** here is an object to be understood by the model, and code execution and recursive LM calls are a means of understanding this context efficiently. Lastly, in our experiments we only consider a recursive depth of 1 — i.e. the root LM can only call LMs, not other RLMs. It is a relatively easy change to allow the REPL environment to call RLMs instead of LMs, but we felt that for most modern "long context" benchmarks, a recursive depth of 1 was sufficient to handle most problems. However, for future work and investigation into RLMs, enabling larger recursive depth will naturally lead to stronger and more interesting systems.

▼ **The formal definition (click to expand)**

Consider a general setup of a language model M receiving a query q with some associated, potentially long context $C = [c_1, c_2, \ldots, c_m]$. The standard approach is to treat $M(q, C)$ like a black box function call, which takes a query and context and returns some `str` output. We retain this frame of view, but define a thin

scaffold on top of the model to provide a more **expressive** and **interpretable** function call $RLM_M(q, C)$ with the same input and output spaces. Formally, a recursive language model $RLM_M(q, C)$ over an environment $\mathcal{E}$ similarly receives a query q and some associated, potentially long context $C = [c_1, c_2, \ldots, c_m]$ and returns some `str` output. The primary difference is that we provide the model a tool call $RLM_M(\hat{q}, \hat{C})$, which spawns an isolated sub-RLM instance using a new query $\hat{q}$ and a transformed version of the context $\hat{C}$ with its own isolated environment $\hat{\mathcal{E}}$; eventually, the final output of this recursive callee is fed back into the environment of the original caller. The environment $\mathcal{E}$ abstractly determines the control flow of how the language model M is prompted, queried, and handled to provide a final output. In this paper, we specifically explore the use of a Python REPL environment that stores the input context C as a variable in memory. This specific choice of environment enables the language model to **peek at**, **partition**, **transform**, and **map** over the input context and use recursive LMs to answer sub-queries about this context. Unlike prior agentic methods that rigidly define these workflow patterns, RLMs defer these decisions entirely to the language model. Finally, we note that particular choices of environments $\mathcal{E}$ are flexible and are a generalization of a base model call: the simplest possible environment $\mathcal{E}_0$ queries the model M with input query and context q, C and returns the model output as the final answer.

Some early (and very exciting) results!

We've been looking around for benchmarks that reflect natural long-context tasks, e.g. long multi-turn Claude Code sessions. We namely were looking to highlight two properties that limit modern frontier models: 1) the context rot phenomenon, where model performance degrades as a function of context length, and 2) the system-level limitations of handling an enormous context.

We found in practice that many long-context benchmarks offer contexts that are not really that long and which were already solvable by the latest generation (or two) of models. In fact, we found some where **models could often answer queries without the context**! We luckily quickly found two benchmarks where modern frontier LLMs struggle to perform well, but we are actively seeking any other good benchmark recommendations to try.

Exciting Result #1 — Dealing with Context Rot.

The **OOLONG** benchmark [1] is a challenging new benchmark that evaluates long-context reasoning tasks over fine-grained information in context. We were fortunate to have the (anonymous *but not affiliated with us*) authors share the dataset upon request to run our experiments on a split of this benchmark.

Setup. The `trec_coarse` split consists of 6 different types of queries to answer distributional queries about a giant list of "question" entries. For example, one question looks like:

```
For the following question, only consider the subset of instances that are associated with user IDs 67144, 53321, 38876, 59219, 18145, 64957, 32617, 55177, 91019, 53985, 84171, 82372, 12053, 33813, 82982, 25063, 41219, 90374, 83707, 59594. Among instances associated with these users, how many data points should be classified as label 'entity'? Give your final answer in the form 'Answer: number'.
```

The query is followed by ~3000 - 6000 rows of entries with associated user IDs (not necessarily unique) and instances that **are not explicitly labeled** (i.e. the model has to infer the labeling to answer). They look something like this:

```
Date: Dec 12, 2022 || User: 63685 || Instance: How many years old is Benny Carter ?
Date: Dec 30, 2024 || User: 35875 || Instance: What war saw battles at Parrot 's Beak and Black Virgin ?
Date: Apr 13, 2024 || User: 80726 || Instance: What Metropolis landmark was first introduced in the Superman cartoons of the 1940 's ?
Date: Feb 29, 2024 || User: 59320 || Instance: When was Calypso music invented?
...
```

The score is computed as the number of queries answered correctly by the model, with the caveat that for numerical / counting problems, they use a continuous scoring metric. This benchmark is extremely hard for both frontier models and agents because they have to **semantically** map and associate thousands of pieces of information in a single query, and cannot compute things a-priori! We evaluate the following models / agents:

- **GPT-5.** Given the whole context and query, tell GPT-5 to provide an answer.
- **GPT-5-mini.** Given the whole context and query, tell GPT-5-mini to provide an answer.
- **RLM(GPT-5-mini).** Given the whole context and query, tell RLM(GPT-5-mini) to provide an answer. GPT-5-mini (root LM) can recursively call GPT-5-mini inside its REPL environment.
- **RLM(GPT-5) without sub-calls.** Given the whole context and query, tell RLM(GPT) to provide an answer. GPT-5 (root LM) cannot recursively call GPT-5 inside its REPL environment. This is an ablation for the use of a REPL environment without recursion.
- **ReAct w/ GPT-5 + BM25.** We chunk every lines into its own "document", and gives a ReAct loop access to a BM25 retriever to return 10 lines per search request.

Results. We focus explicitly on questions with contexts over 128k tokens (~100 queries), and we track both the performance on the benchmark, as well as the overall API cost of each query. In all of the following results (Figure **4a,b**), **the entire input fits in the context window of GPT-5 / GPT-5-mini** — i.e., incorrect predictions are never due to truncation or context window size limitations:

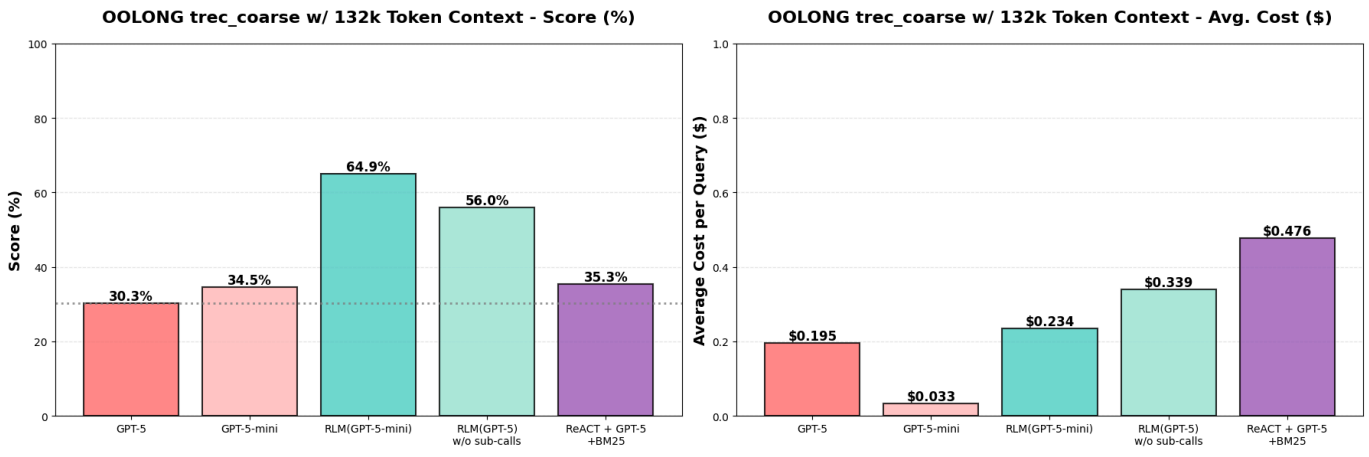


Figure 4a. We report the overall score for each method on the `trec_coarse` dataset of the OOLONG benchmark for queries that have a context length of 132k tokens. We compare performance to GPT-5. RLM(GPT-5-mini) outperforms GPT-5 by over **34 points (~114% increase)**, and is nearly as cheap per query (we found that the median query is cheaper due to some outlier, expensive queries).

It turns out actually that **RLM(GPT-5-mini)** outperforms **GPT-5** and **GPT-5-mini** by **>33%**↑ raw score (over double the performance) while maintaining roughly the same total model API cost as **GPT-5** per query! When ablating recursion, we find that RLM performance degrades by ~10%, likely due to many questions requiring the model to answer semantic questions about the data (e.g. label each question). We see in **Figure 4b** that these gains roughly transfer when we double the size of the context to ~263k tokens as well, although with some performance degradation!

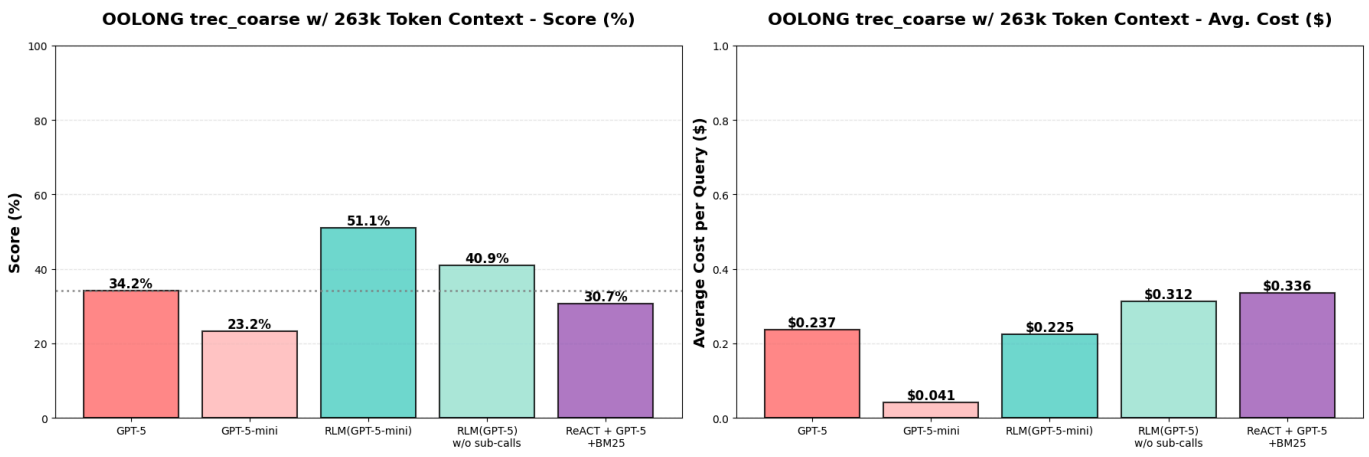


Figure 4b. We report the overall score for each method on the trec_coarse dataset of the OOLONG benchmark for queries that have a context length of 263k tokens, nearly the limit for GPT-5/GPT-5-mini. We compare performance to GPT-5. RLM(GPT-5-mini) outperforms GPT-5 by over **15 points (~49% increase)**, and is cheaper per query on average.

Notably, the performance of **GPT-5-mini** drops while **GPT-5** does not, which indicates that context rot is more severe for GPT-5-mini. We additionally noticed that the performance drop for the RLM approaches occurs for ***counting*** problems, where it makes more errors when the context length increases — for **GPT-5**, it already got most of these questions incorrect in the 132k context case, which explains why its performance is roughly preserved. Finally, while the **ReAct + GPT-5 + BM25** baseline doesn't make much sense in this setting, we provide it to show retrieval is difficult here while **RLM** is the more appropriate method.

Great! So we're making huge progress in solving goal (1), where GPT-5 has *just* enough context window to fit the 263k case. But what about goal (2), where we may have 1M, 10M, or even 100M tokens in context? *Can we still treat this like a single model call?*

Exciting Result #2 — Ridiculously Large Contexts

My advisor Omar is a superstar in the world of information retrieval (IR), so naturally we also wanted to explore whether RLMs scale properly when given thousands (or more!) of documents. OOLONG [1] provides a giant block of text that is difficult to index and therefore difficult to compare to retrieval methods, so we looked into DeepResearch-like benchmarks that evaluate answering queries over documents.

Retrieval over huge offline corpuses. We initially were interested in BrowseComp [4], which evaluates agents on multi-hop, web-search queries, where agents have to find the relevant documents online. We later found the BrowseComp-Plus [2] benchmark, which pre-downloads all possible relevant documents for all queries in the original benchmark, and just provides a list of ~100K documents (~5k words on average) where the answer to a query is scattered across this list. For benchmarking RLMs, this benchmark is perfect to see if we can just throw ridiculously large amount of context into a single `chat.completion(...)` RLM call instead of building an agent!

Setup. We explore how scaling the # documents in context affects the performance of various common approaches to dealing with text corpuses, as well as RLMs. Queries on the BrowseComp-Plus benchmark are multi-hop in the sense that they require associating information across several different documents to answer the query. What this implies is that even if you retrieve the document with the correct answer, you won't know it's correct until you figure out the other associations. For example, query `984` on the benchmark is the following:

```
I am looking for a specific card in a trading card game. This card was released between the years 2005 and 2015 with more than one rarity present during the year it was released. This card has been used in a deck list that used by a Japanese player when they won the world championship for this trading card game. Lore wise, this card was used as an armor for a different card that was released later between the years 2013 and 2018. This card has also once been illegal to use at different events and is below the level 8. What is this card?
```

For our experiments, we explore the performance of each model / agent / RLM given access to a corpus of sampled documents of varying sizes — the only guarantee is that the answer can be found in this corpus. In practice, we found that GPT-5 can fit ~40 documents in context before it exceeds the input context window (272k tokens), which we factor into our choice of constants for our baselines. We explore the following models / agents, similar to the previous experiment:

- **GPT-5.** Given all documents in context and the query, tell GPT-5 to provide an answer. If it goes over the context limit, return nothing.
- **GPT-5 (Truncated).** Given all documents in context and the query, tell GPT-5 to provide an answer. If it goes over the context limit, truncate by most recent tokens (i.e. random docs).
- **GPT-5 + Pre-query BM25.** First retrieve the top 40 documents using BM25 with the original query. Given these top-40 documents and the query, tell GPT-5 to provide an answer.
- **RLM(GPT-5).** Given all documents in context and the query, tell RLM(GPT-5) to provide an answer. GPT-5 (root LM) can "recursively" call GPT-5-mini inside its REPL environment.
- **RLM(GPT-5) without sub-calls.** Given the whole context and query, tell RLM(GPT-5) to provide an answer. GPT-5 (root LM) cannot recursively call GPT-5 inside its REPL environment. This is an ablation for the use of a REPL environment without recursion.
- **ReAct w/ GPT-5 + BM25.** Given all documents, query for an answer from a ReAct loop using GPT-5 with access to a BM25 retriever that can return 5 documents per request.

Results. We want to emphasize that these preliminary results are not over the entire BrowseComp-Plus dataset, and only a small subset. We report the performance over 20 randomly sampled queries on BrowseComp-Plus when given 10, 50, 100, and 1000 documents in context in **Figure 5.** We always include the gold / evidence document documents in the corpus, as well as the hard-mined negatives if available.

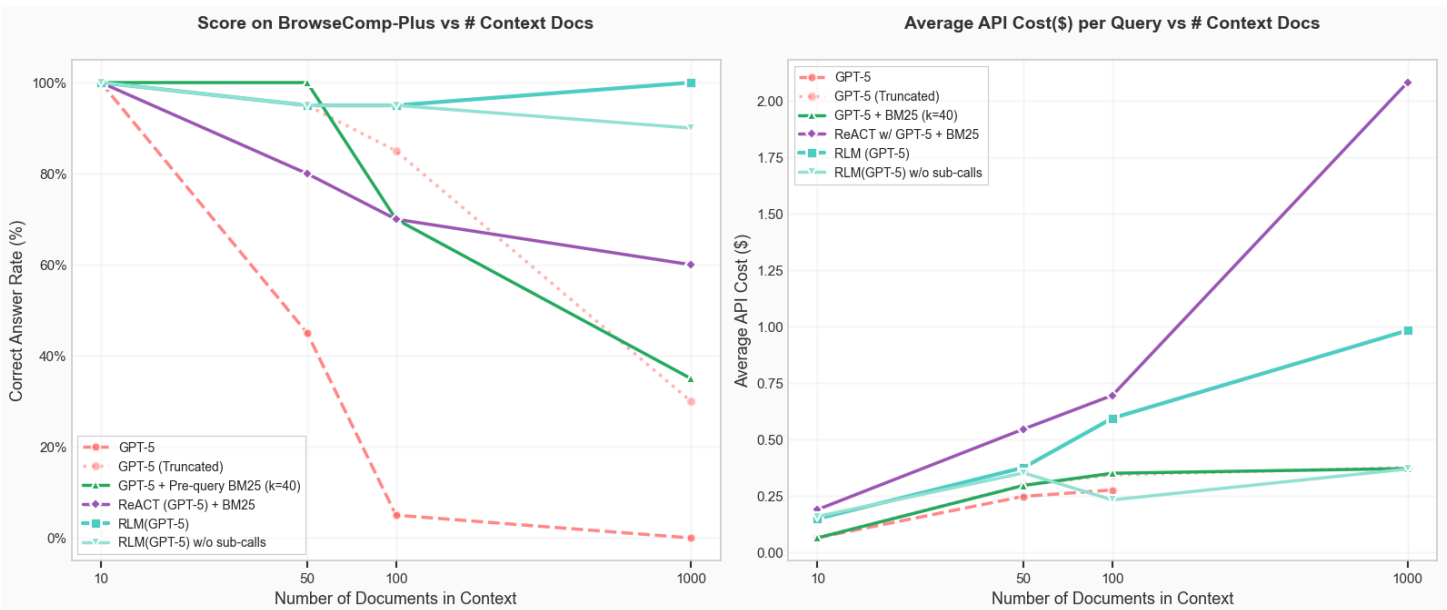


Figure 5. We plot the performance and API cost per answer of various methods on 20 random queries in BrowseComp-Plus given increasing numbers of documents in context. Only the iterative methods (RLM, ReAct) maintain reasonable performance at 100+ documents.

There are a few things to observe here — notably, `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%. The base `GPT-5` model approaches, regardless of how they are conditioned, show clear signs of performance dropoff as the number of documents increase. Unlike OOLONG [1], all approaches are able to solve the task when given a sufficiently small context window (10 documents), making this a problem of finding the right information rather than handling complicated queries. Furthermore, the cost per query of `RLM(GPT-5)` scales reasonably as a function of the context length!

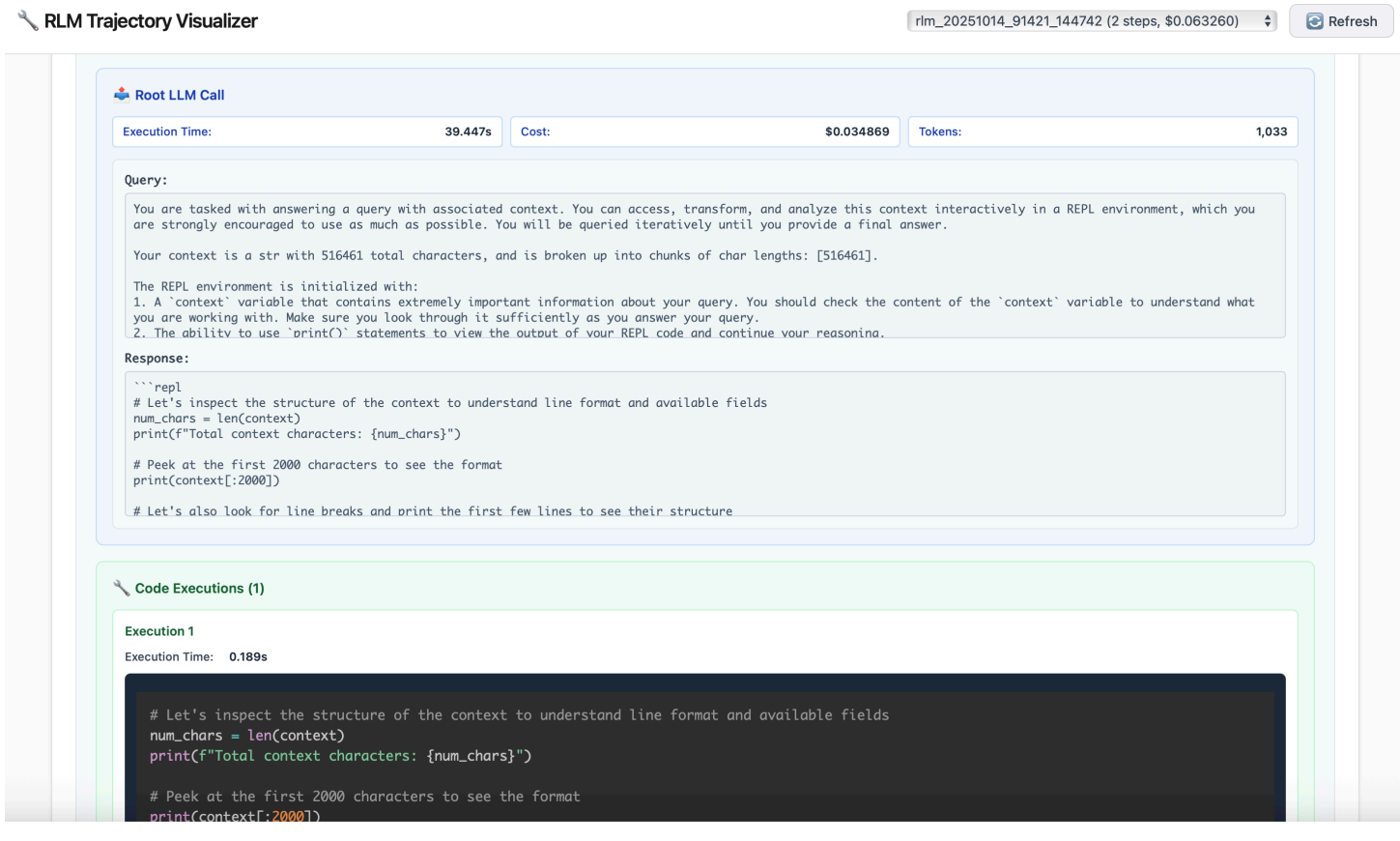
These experiments are particularly exciting because without any extra fine-tuning or model architecture changes, we can reasonably handle huge corpuses (10M+ tokens) of context on realistic benchmarks without the use of a retriever. It should be noted that the baselines here index BM-25 **per query**, which is a more powerful condition than indexing the full 100K document corpus and applying BM-25. Regardless, RLMs are able to outperform the iterative `ReAct + GPT-5 + BM25` loop on a retrieval style task with a reasonable cost!

Amazing! So RLMs are a neat solution to handle our two goals, and offer natural way to extend the effective context window of a LM call without incurring large costs. The rest of this blog will be dedicated to some cool and interesting behavior that RLMs exhibit!

What is the RLM doing? Some Interesting Cases…

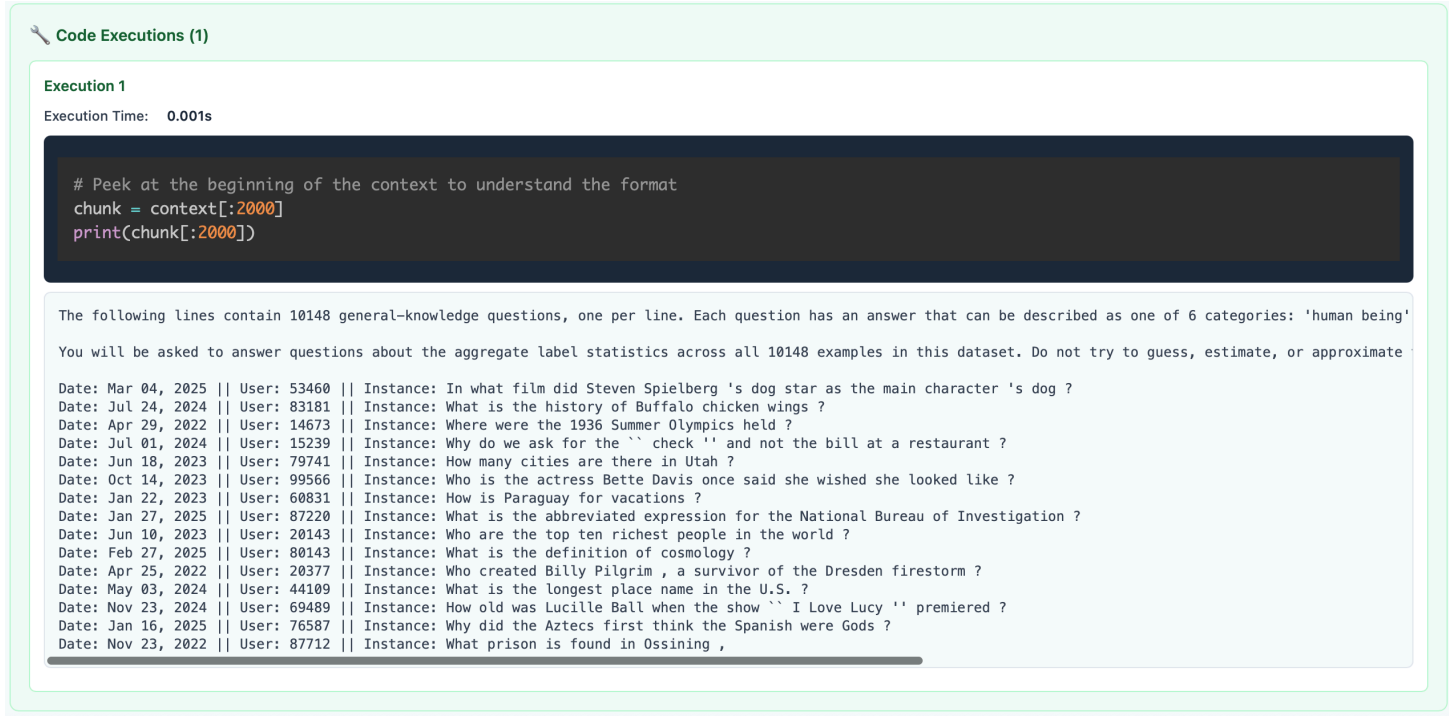
A strong benefit of the RLM framework is the ability to roughly interpret what it is doing and how it comes to its final answer.

We vibe-coded a simple visualizer to peer into the trajectory of an RLM, giving us several interesting examples to share about what the RLM is doing!



Strategies that have emerged that the RLM will attempt. At the level of the RLM layer, we can completely interpret how the LM chooses to interact with the context. Note that in every case, the root LM starts only with the query and an indication that the context exists in a variable in a REPL environment that it can interact with.

Peeking. At the start of the RLM loop, the root LM does not see the context at all — it only knows its size. Similar to how a programmer will peek at a few entries when analyzing a dataset, the LM can peek at its context to observe any structure. In the example below on OOLONG, the outer LM grabs the first 2000 characters of the context.



Grepping. To reduce the search space of its context, rather than using semantic retrieval tools, the RLM with REPL can look for keywords or regex patterns to narrow down lines of interest. In the example below, the RLM looks for lines with questions and IDs.

Execution 3

Execution Time: 0.001s

```
# From inspection we likely have user ID + question per line separated by a tab or delimiter.
# Let's try to robustly extract the question text by removing a leading user ID pattern.
import re

def extract_question(line):
    # Common patterns: "123\tQuestion", "123 ||| Question", "UserID Question", "123,Question", etc.
    # Heuristic: remove a leading token of digits/word chars plus common separators, then strip.
    # Try several separators
    for sep in ["\t", "|||", "|", ",", ";", " - ", " – ", ":", "    "]:
        if sep in line:
            parts = line.split(sep, 1)
            # Heuristic: if first part looks like an ID (mostly digits or alnum short), take the second as question
            if re.fullmatch(r"[A-Za-z0-9_.-]{1,40}", parts[0].strip()):
                return parts[1].strip()
    # If no clear separator, try to remove a leading ID followed by space
    m = re.match(r"^\s*([A-Za-z0-9_.-]{1,40})\s+(.*)$", line)
    if m:
        return m.group(2).strip()
    # Otherwise, assume the whole line is the question
    return line.strip()

# Test extraction on the first few lines
for i in range(10):
    q = extract_question(lines[i])
    print(f"Q{i}:", q)
```

```
Q9: Oct 14, 2023 || User: 99566 || Instance: Who is the actress Bette Davis once said she wished she looked like ?
```

Partition + Map. There are many cases where the model cannot directly grep or retrieve information due to some semantic equivalence of what it is looking for. A common pattern the RLM will perform is to chunk up the context into smaller sizes, and run several recursive LM calls to extract an answer or perform this semantic mapping. In the example below on OOLONG, the root LM asks the recursive LMs to label each question and use these labels to answer the original query.

Execution 2

Execution Time: 215.631s

```
# Define helper to call llm_query for classification
def classify_batch(batch):
    # batch is list of tuples (original_line_number, question)
    prompt_lines = []
    prompt_lines.append("You are given a numbered list of questions. For each question, choose exactly one label from the following six ca
    prompt_lines.append("['numeric value', 'entity', 'location', 'description and abstract concept', 'abbreviation', 'human being']")
    prompt_lines.append("Return ONLY lines in the format: original_index<TAB>label")
    prompt_lines.append("Do NOT include any extra text or explanation. Use these exact label strings.")
    prompt_lines.append("")
    prompt_lines.append("Here are the questions:")
    for idx, q in batch:
        # Escape newlines
        q_single = " ".join(q.splitlines())
        prompt_lines.append(f"{idx}\t{q_single}")
    prompt = "\n".join(prompt_lines)
    return llm_query(prompt)

# Batch size chosen to keep prompt manageable
batch_size = 50
batches = [jan_entries[i:i+batch_size] for i in range(0, len(jan_entries), batch_size)]

classified = {}  # map original_line_number -> label
for bi, batch in enumerate(batches):
    resp = classify_batch(batch)
    print(f"--- Batch {bi+1}/{len(batches)} response ---")
    print(resp[:1000])  # show first part of response for inspection
    # Parse response lines
    for line in resp.splitlines():
        line = line.strip()
        if not line:
            continue
```

Summarization. RLMs are a natural generalization of summarization-based strategies commonly used for managing the context window of LMs. RLMs commonly summarize information over subsets of the context for the outer LM to make decisions.

Execution 5

Execution Time: 26.519s

```
# As a final cross-check, summarize the opening chapter for narrator identity and reason for moving East.
query3 = f"""
Summarize the opening chapter(s) content related to:
- Who the first-person narrator is (by name)
- Why he moved East / to New York
Quote the exact lines where possible.

Text:
{context}
"""
verification3 = llm_query(query3)
print(verification3)
```

```
- Who the narrator is:
  - The narrator is Nick Carraway (usually called “Nick” or “Mr. Carraway”).
  - Exact lines: “I’d been writing letters once a week and signing them: ‘Love, Nick,’ …” and “’Good night, Mr. Carraway. See you anon.’” (also Gatsby introduces hi

- Why he moved East / to New York:
  - After the war he felt restless and “decided to go East” to enter the bond business; his father financed him for a year and he came East in spring 1922.
  - Exact lines: “I graduated from New Haven in 1915, … and a little later I participated in that delayed Teutonic migration known as the Great War. I enjoyed the c
```

Long-input, long-output. A particularly interesting and expensive case where LMs fail is in tasks that require long output generations. For example, you might give ChatGPT your list of papers and ask it to generate the BibTeX for all of them. Similar to huge multiplication problems, some people may argue that a model should not be expected to solve these programmatic tasks flawlessly — in these instances, RLMs with REPL environments should one-shot these tasks! An example is the **LoCoDiff** [5] benchmark, where language models are tasked with tracking a long `git diff` history from start to finish, and outputting the result of this history given the initial file. For histories longer than 75k tokens, GPT-5 can't even solve 10% of the histories! An example of what the model is given (as provided on the project website) is as follows:

```
> git log -p \
    --cc \
    --reverse \
    --topo-order \
    -- shopping_list.txt


commit 008db723cd371b87c8b1e3df08cec4b4672e581b
Author: Example User
Date:   Wed May 7 21:12:52 2025 +0000

    Initial shopping list

diff --git a/shopping_list.txt b/shopping_list.txt
new file mode 100644
index 0000000..868d98c
--- /dev/null
+++ b/shopping_list.txt
@@ -0,0 +1,6 @@
+# shopping_list.txt
+apples
+milk
+bread
+eggs
+coffee

commit b6d826ab1b332fe4ca1dc8f67a00f220a8469e48
Author: Example User
Date:   Wed May 7 21:12:52 2025 +0000

    Change apples to oranges and add cheese

diff --git a/shopping_list.txt b/shopping_list.txt
index 868d98c..7c335bb 100644
--- a/shopping_list.txt
+++ b/shopping_list.txt
@@ -1,6 +1,7 @@
 # shopping_list.txt
-apples
+oranges
 milk
 bread
 eggs
 coffee
+cheese
...
```

We tried **RLM(GPT-5)** to probe what would happen, and found in some instances that it chooses to one-shot the task by programmatically processing the sequence of diffs! There are many benchmark-able abilities of LMs to perform programmatic tasks (e.g. huge multiplication, diff tracking, etc.), but RLMs offer a framework for avoiding the need for such abilities altogether.

```
# Instead, capture with headers to retain data
commit_blocks = re.findall(r'(?:^|\n)commit [0-9a-f]{40}\n.*?(?=\ncommit [0-9a-f]{40}\n|\Z)', text, flags=re.S)

# Helper to extract diff blocks for our file from a commit block
def get_file_diffs(block, path='aider/onboarding.py'):
    diffs = []
    for m in re.finditer(r'diff --git a/{0} b/{0}.*?(?=(?:\ndiff --git |\Z))'.format(re.escape(path)), block, flags=re.S):
        diffs.append(m.group(0))
    return diffs

# Apply unified diff hunks to a list of lines
def apply_unified_diff(orig_lines, diff_text):
    """
    orig_lines: list of lines without trailing newline chars
    diff_text: the diff block text for this file
    Returns new_lines (list)
    """
    lines = diff_text.splitlines()
    # Find hunks starting with @@
    i = 0
    new_lines = orig_lines[:]
    # We'll build by applying hunks sequentially; to avoid index confusion, we'll reconstruct line-by-line
    # Strategy: For each hunk, we track output building: We maintain a cursor into new_lines and replace the range covered by the hunk.
    # But easier: We simulate applying hunks to a working list by walking through expected context lines to find positions.
    # Better approach: process hunks and construct result in one pass: take segments before hunk, then apply, then continue.

    # We'll parse hunks and apply on a working pointer.
    # To do this, we need to maintain a current position pointer in the output lines as per hunk new_start.
    # hunk header: @@ -old_start,old_len +new_start,new_len @@ optional section heading
    # Indices are 1-based.
    out = []
    cur_orig_index = 0  # 0-based index into orig_lines
    idx = 0
    while idx < len(lines):
        if lines[idx].startswith('@@'):
            # parse header
            m = re.match(r'@@ -(\d+)(?:,(\d+))? \+(\d+)(?:,(\d+))? @@', lines[idx])
            if not m:
                idx += 1
```

More patterns…? We anticipate that a lot more patterns will emerge over time when 1) models get better and 2) models are trained / fine-tuned to work this way. An underexplored area of this work is how *efficient* a language model can get with how it chooses to interact with the REPL environment, and we believe all of these objectives (e.g. speed, efficiency, performance, etc.) can be optimized as scalar rewards.

Limitations.

We did not optimize our implementation of RLMs for speed, meaning each recursive LM call is both blocking and does not take advantage of any kind of prefix caching! Depending on the partition strategy employed by the RLM's root LM, the **lack of asynchrony** can cause each query to range from a few seconds to several minutes. Furthermore, while we can control the length / "thinking time" of an RLM by increasing the maximum number of iterations, we do not currently have strong guarantees about controlling either the total API cost or the total runtime of each call. For those in the systems community (*cough cough*, especially the GPU MODE community), this is amazing news! There's so much low hanging fruit to optimize here, and getting RLMs to work at scale requires re-thinking our design of inference engines.

Related Works

Scaffolds for long input context management. RLMs defer the choice of context management to the LM / REPL environment, but most prior works do not. MemGPT [6] similarly defers the choice to the model, but builds on a single context that an LM will eventually call to return a response. MemWalker [7] imposes a tree-like structure to order how a LM summarizes context. LADDER [8] breaks down context from the perspective of problem decomposition, which does not generalize to huge contexts.

Other (pretty different) recursive proposals. There's plenty of work that invokes forking threads or doing recursion in the context of deep learning, but none have the structure required for general-purpose decomposition. THREAD [9] modifies the output generation process of a model call to spawn child threads that write to the output. Tiny Recursive Model (TRM) [10] is a cool idea for iteratively improving the answer of a (not necessarily language) model in its latents. Recursive LLM Prompts was an early experiment on treating the prompt as a state that evolves when you query a model. Recursive Self-Aggregation (RSA) is a recent work that combines test-time inference sampling methods over a set of candidate responses.

What We're Thinking Now & for the Future.

Long-context capabilities in language models used to be a model architecture problem (think ALiBi, YaRN, etc.). Then the community claimed it was a systems problem because "attention is quadratic", but it turned out actually that our MoE layers were the bottleneck. It now has become somewhat of a combination of the two, mixed with the fact that longer and longer contexts do not fall well within the training distributions of our LMs.

Do we have to solve context rot? There are several reasonable explanations for "context rot"; to me, the most plausible is that longer sequences are out of distribution for model training distributions due to lack of natural occurrence and higher entropy of long sequences. The goal of RLMs has been to propose a framework for issuing LM calls without ever needing to directly solve this problem — while the idea was initially just a framework, we were very surprised with the strong results on modern LMs, and are optimistic that they will continue to scale well.

RLMs are not agents, nor are they just summarization. The idea of multiple LM calls in a single system is not new — in a broad sense, this is what most agentic scaffolds do. The closest idea we've seen in the wild is the ROMA agent that decomposes a problem and runs multiple sub-agents to solve each problem. Another common example is code assistants like Cursor and Claude Code that either summarize or prune context histories as they get longer and longer. These approaches generally view multiple LM calls as decomposition **from the perspective of a task or problem**. We retain the view that LM calls can be decomposed by the context, and the choice of decomposition should purely be the choice of an LM.

The value of a fixed format for scaling laws. We've learned as a field from ideas like CoT, ReAct, instruction-tuning, reasoning models, etc. that presenting data to a model in predictable or fixed formats are important for improving performance. The basic idea is that we can reduce the structure of our training data to formats that model expects, we can greatly increase the performance of models with a reasonable amount of data. We are excited to see how we can apply these ideas to improve the performance of RLMs as another axis of scale.

RLMs improve as LMs improve. Finally, the performance, speed, and cost of RLM calls correlate directly with improvements to base model capabilities. If tomorrow, the best frontier LM can reasonably handle 10M tokens of context, then an RLM can reasonably handle 100M tokens of context (maybe at half the cost too).

As a lasting word, RLMs are a fundamentally different bet than modern agents. Agents are designed based on human / expert intuition on how to break down a problem to be digestible for an LM. RLMs are designed based on the principle that fundamentally, LMs should decide how to break down a problem to be digestible for an LM. I personally have no idea what will work in the end, but I'm excited to see where this idea goes!

--az

Acknowledgements

We thank our wonderful MIT OASYS labmates Noah Ziems, Jacob Li, and Diane Tchuindjo for all the long discussions about where steering this project and getting unstuck. We thank Prof. Tim Kraska, James Moore, Jason Mohoney, Amadou Ngom, and Ziniu Wu from the MIT DSG group for their discussion and help in framing this method for long context problems. This research was partly supported by Laude Institute.

We also thank the authors (who shall remain anonymous) of the OOLONG benchmark for allowing us to experiment on their long-context benchmark. They went from telling us about the benchmark on Monday 10:30am to sharing it with us by 1pm, and two days ago, we're able to tell you about these cool results thanks to them.

Finally, we thank Jack Cook and the other first year MIT EECS students for their support during the first year of my PhD!

Citation

You can cite this blog (before the full paper is released) here:

```
@article{zhang2025rlm,
  title   = "Recursive Language Models",
  author  = "Zhang, Alex and Khattab, Omar",
  year    = "2025",
  month   = "October",
  url     = "https://alexzhang13.github.io/blog/2025/rlm/"
}
```

References

1. **Oolong: Evaluating Long Context Reasoning and Aggregation Capabilities** [link]
   Anonymous,, 2025. Submitted to The Fourteenth International Conference on Learning Representations.
2. **BrowseComp-Plus: A More Fair and Transparent Evaluation Benchmark of Deep-Research Agent** [PDF]
   Chen, Z., Ma, X., Zhuang, S., Nie, P., Zou, K., Liu, A., Green, J., Patel, K., Meng, R., Su, M., Sharifymoghaddam, S., Li, Y., Hong, H., Shi, X., Liu, X., Thakur, N., Zhang, C., Gao, L., Chen, W. and Lin, J., 2025.
3. **Executable Code Actions Elicit Better LLM Agents** [link]
   Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H. and Ji, H., 2024. Forty-first International Conference on Machine Learning.
4. **BrowseComp: A Simple Yet Challenging Benchmark for Browsing Agents** [PDF]
   Wei, J., Sun, Z., Papay, S., McKinney, S., Han, J., Fulford, I., Chung, H.W., Passos, A.T., Fedus, W. and Glaese, A., 2025.
5. **LoCoDiff Benchmark**
   MentatAI, and AbanteAI,, 2025.
6. **MemGPT: Towards LLMs as Operating Systems** [PDF]
   Packer, C., Wooders, S., Lin, K., Fang, V., Patil, S.G., Stoica, I. and Gonzalez, J.E., 2024.
7. **Walking Down the Memory Maze: Beyond Context Limit through Interactive Reading** [PDF]
   Chen, H., Pasunuru, R., Weston, J. and Celikyilmaz, A., 2023.
8. **LADDER: Self-Improving LLMs Through Recursive Problem Decomposition** [PDF]
   Simonds, T. and Yoshiyama, A., 2025.
9. **THREAD: Thinking Deeper with Recursive Spawning** [link]
   Schroeder, P., Morgan, N.W., Luo, H. and Glass, J.R., 2025. Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), pp. 8418--8442. Association for Computational Linguistics. DOI: 10.18653/v1/2025.naacl-long.427
10. **Less is More: Recursive Reasoning with Tiny Networks** [PDF]
    Jolicoeur-Martineau, A., 2025.

© Copyright 2026 Alex L. Zhang.