ChatGPT

Recursive Language Models (RLM): Concept, Applications, and Implementation

Conceptual Foundations

What is a Recursive Language Model? A *Recursive Language Model (RLM)* is not a new architecture but an **inference-time framework** that wraps around a standard language model (LM) to enable *recursive reasoning* on large inputs. In an RLM, the model can **call itself (or other models) as a subroutine** during inference, breaking a complex input or problem into smaller parts and solving them step by step [1]. From the user's perspective, using an RLM feels the same as a normal LM API call (e.g. calling `rlm.completion()` instead of `gpt5.completion()`), but under the hood the RLM employs a *recursive strategy* [1]. In effect, the RLM gives the *illusion of a near-infinite context window* by allowing the LM to iteratively digest pieces of the input rather than consuming the entire prompt at once [2]. This approach was proposed specifically to address *"context rot"* – the degradation in model performance when context is very long [3].

How does recursion work in prompting and reasoning? RLMs introduce an external *environment* (for example, a Python REPL notebook) where the entire input context is stored as a variable, and the model can interact with this environment through generated code [2] [4]. The *root* RLM (depth 0) receives only the user's query plus some instructions, *not* the full context text [5]. When answering, the model can output special code or commands (in the REPL) to inspect or manipulate the context variable. For instance, the model might **peek at a slice of the context or search it** using a regex, or even invoke a *recursive sub-model* on a portion of the context [6] [7]. Each such *sub-invocation* is essentially a smaller LM call (often a cheaper or smaller model) focused on a chunk of the data. The RLM runtime captures the results of these code executions or sub-calls and feeds them back into the model's prompt (often in truncated form) so the model can continue reasoning with that feedback [8]. This loop continues, with the model iteratively reading parts of the context and refining its understanding, until the model decides to output a final answer. The end of the process is marked by a special output syntax – e.g. the model prints `FINAL(answer)` or `FINAL_VAR(var_name)` to indicate the final answer [8] [9]. The RLM wrapper then returns that answer string as the result to the user.

Figure 1: Example of a recursive language model architecture. The RLM (root, depth=0) uses a base LM (e.g. GPT-5) to answer a query by interacting with an external environment that holds the full context. The context (which could be arbitrarily large text or data) is stored in a variable accessible in a Python REPL. The root LM can issue code (via the REPL) to inspect or search the context and can spawn recursive LM calls (depth=1 sub-calls) on subsections of the context. Each sub-call processes a chunk of the data (possibly using a smaller LM for efficiency) and returns an intermediate result. The root LM combines these results to produce the final answer, which it outputs with a special `FINAL(...)` *notation.* [2] [6]

Differences from traditional LMs (autoregressive or instruction-tuned): Traditional large language models operate in a purely *autoregressive* fashion – they take a fixed prompt (up to some maximum token length) and generate an output token by token. Even instruction-tuned LMs (like GPT-4, Claude, etc.) follow

this one-shot forward-pass paradigm where the entire context must be given upfront in the prompt window. This leads to two big limitations that RLMs address:

- **Fixed Context Window vs. Unbounded Context:** Standard LMs have a fixed context limit (e.g. 4K, 100K tokens, etc.), and even within that window their performance can degrade as the prompt grows longer (the *context rot* effect). RLMs, by contrast, treat the context as **external memory**. The full input (which could be hundreds of thousands or millions of tokens) is stored as a variable *outside* the model's prompt [10]. The model never "sees" the entire context at once; it only pulls in what it needs via recursive queries. This gives the RLM effectively *unbounded context length* handling – as long as the context can be stored in memory, the model can iteratively process it [2] [11]. In other words, instead of doing `llm.complete(prompt + huge_context)` (which would hit limits or cause context rot), one can do `rlm.completion(query, context=huge_document)` and let the model figure out how to navigate that huge document [12].

- **Single-pass Reasoning vs. Recursive Reasoning:** A normal instruction-following LM tries to produce an answer in one go, relying on whatever was encoded in its prompt (which might include the entire context or some retrieved snippets). RLMs differ by allowing **multiple reasoning steps** with *explicit control flow*. The model can *decide* to break a problem down: for example, it might first scan for relevant sections of a text, then summarize each section, then combine those summaries – all through separate LM calls orchestrated by the RLM framework. This is akin to how a human programmer might write a function that calls helper functions for subtasks. The recursion is dynamic and decided by the model's own outputs, not by a fixed prompt template alone. While techniques like Chain-of-Thought (CoT) prompting or ReAct agents also involve multi-step reasoning, those are usually implemented via a fixed prompting pattern or an external agent loop. RLMs push this further by letting the *model itself choose when and how to recurse* on data [13]. The *structure* of the solution (how the context is partitioned or which parts to focus on) is learned or decided by the LM in real-time, rather than predefined by human designers [13].

- **Autonomy and Format:** Instruction-tuned models are trained to follow natural language instructions and produce answers directly. An RLM still leverages such models as the base, but augments them with a *programmatic interface*. The RLM provides a **system prompt or toolset** that tells the model how to use the code environment and how to output final answers in a specific format (like the `FINAL()` tag). This means the model's generations are a mix of *code actions* and natural language reasoning. Prior methods like CoT or ReAct also impose a structured format (e.g. thinking step by step, or interleaving thought and action), and indeed RLMs build on that idea of using a fixed format to guide the model [14]. The key difference is that RLMs give the model a much more **powerful action space** (full Python code execution and the ability to spawn sub-model calls) to manage large contexts, rather than just a textual reasoning scratchpad. Importantly, *no special training or fine-tuning* of the base LM was required to enable this in the early RLM prototypes – it works by prompting alone. However, the RLM authors suggest that training future models explicitly for recursive reasoning (analogous to how models have been trained for CoT) could yield even better performance [15] [16].

RLMs vs. agent frameworks: It's useful to contrast RLMs with so-called "agent" approaches (like tool-using agents or planners). In agent frameworks (e.g. a ReAct agent), the decomposition of a task into steps is often hand-designed or follows a script: the agent might be forced to always recall, then act, then observe, etc., or a human engineer decides to plug in a search tool or summarizer at certain points. RLMs take a

more **model-driven approach**. As Zhang notes, *"Agents are designed based on human/expert intuition on how to break down a problem… RLMs are designed on the principle that the LM should decide how to break down a problem"* [17] [18]. In essence, RLMs remove the rigid "external controller" and let the model itself be the controller: the model's output can contain code that controls what happens next. This is a fundamentally different philosophy – one might call RLM an *end-to-end learned agent*. That said, current RLM implementations still require carefully crafted system prompts (to instruct the model about using the environment), so there is still some designer influence. But compared to traditional agents, RLMs give more freedom to the LM's own internal logic. Another difference is that many agents use *retrieval* or external knowledge tools to handle large information (e.g. search engines, vector databases), whereas RLMs handle large context by direct inspection (almost treating the context as its own "database" to be queried via code) [19].

In summary, a Recursive LM can be seen as a **text-to-text mapping that internally does far more than a standard LM call**. It partitions and processes the input text in parts, spawning recursive computations as needed [2]. Yet to the end user it behaves like a single model call that simply *can answer questions about very large inputs that normal LMs cannot*. This general strategy opens up a new axis of scaling: instead of scaling model size or context window, we scale the *inference process* via recursion [16].

Use Cases and Applications

Handling extremely long documents and context-heavy queries: The original motivation for RLMs was to cope with tasks involving *very large contexts*. In practical scenarios (like lengthy conversations, large documents, or codebases), users often hit context length limits or notice the model's quality dropping as the session grows. RLMs excel at tasks like **question answering or summarization of giant texts** – things that would break a normal model. Early experiments demonstrated dramatic gains. For example, on the OOLONG benchmark (a challenging long-context QA dataset with documents on the order of 132K tokens), an RLM using GPT-5-mini *more than doubled* the number of correct answers compared to a vanilla GPT-5, **while also reducing the average cost per query** [20] [21]. (GPT-5 alone managed around 33% accuracy on those long inputs, whereas the RLM achieved roughly 67% accuracy [22].) This highlights that even a smaller model, when used recursively, can outperform a larger model that tries to ingest the whole context at once. In another test, RLMs showed **no performance degradation even with 10+ million tokens of context** – the model maintained its accuracy when tasked with a *Deep Research* query that involved reading 1,000 documents (10M tokens) sequentially [20] [23]. In fact, on a benchmark called BrowseComp-Plus (which requires answering multi-hop questions by combining information from many documents), the RLM not only handled the huge context smoothly, but also **outperformed a ReAct-style retrieval agent** built to tackle the same task [24] [25]. This suggests that for open-ended research questions or any task where one might ordinarily use a search engine or retriever, a sufficiently powerful RLM can ingest *all* the data and reason over it holistically – potentially giving better results since it doesn't have to rely on imperfect retrieval heuristics.

Needle-in-a-haystack search and data extraction: RLMs are particularly suited for scenarios where a small piece of information must be found or extracted from a *massive* text. Because the model can execute code, it can perform efficient searches or filtering on the context variable. For instance, an RLM can use Python string operations or regex to scan through a long text for keywords or patterns [19] [11]. The blog authors give the example: if you need to find a specific fact in a huge log ("needle in haystack"), the RLM's root model might first grep through the text via the REPL to narrow down candidate segments, and then launch recursive LM calls on those segments to interpret them and extract the answer [19]. This is far more

efficient than forcing the LM to read every word of the log sequentially. In a "Hello World" demo of RLM, a random number was hidden inside a million lines of random text and the RLM was asked to find it – the RLM succeeded by programmatically searching the context, something a normal LM would be very unlikely to do correctly [26] [27]. Similarly, RLMs can be used for **structured data extraction** from large text dumps. With the ability to write code, the model can parse a huge text into a table or pull out all occurrences of a pattern. In tests on structured data queries (like counting occurrences or applying filters in a 60K-token dataset), a GPT-5-mini RLM answered 80% of queries correctly, whereas the same queries completely stumped the direct GPT API (0% correct when trying to cram the data into the prompt) [28]. These cases show RLMs turning unbounded text into actionable information by combining brute-force computation (via code) with language understanding.

Multi-step reasoning and multi-hop questions: Because RLMs allow complex chains of thought, they naturally apply to tasks requiring reasoning through multiple steps or combining information from multiple sources. In the *BrowseComp-Plus* example, the queries were *compositional multi-hop questions* – essentially, the model had to retrieve and synthesize facts from many documents [29]. A traditional pipeline might use a retrieval model to fetch relevant snippets, then an LM to compose an answer. The RLM approach instead shoved all documents into the context and let the model figure out which parts to read. Impressively, the RLM maintained strong performance as the number of documents increased (10, 100, up to 1000 documents) without needing any external search component [23]. Only methods that allowed iterative reading (like RLM or an agent that does retrieval loops) were able to even remain viable beyond 100 documents, and among those RLM had an edge [30]. This demonstrates RLMs' advantage in **agent-like scenarios**: the RLM essentially *became its own retrieval agent*, but with the flexibility to define arbitrary strategies (not just TF-IDF or embedding searches). That said, extremely large multi-hop tasks will be slow for an RLM to handle entirely in-context – there is a trade-off between doing it all with the model vs. using external tools. But as base models grow more powerful, letting them internally reason may yield higher quality answers.

Program synthesis and code understanding: While the current RLM research has focused on long text inputs, the underlying idea can benefit coding tasks as well. In fact, the RLM environment *is* a live Python interpreter, which aligns perfectly with code-related tasks. We can imagine using an RLM to navigate a large codebase or to solve programming problems by breaking them into functions. For example, an RLM could be tasked with debugging a code repository: it could open one file at a time (since the files can be loaded as strings in the REPL), search for a bug pattern, and recursively call a code-specialized LM on specific functions to explain or fix issues. This isn't far-fetched – the RLM authors were inspired by **CodeAct**, a framework where LLM agents use code execution as actions [31]. RLM extends this concept by adding the *recursive model call* as an action. So for complex programming challenges, an RLM could iteratively write and execute test code, or generate sub-functions piecewise and verify them, before synthesizing a final program. Early adopters have provided examples like `multi_file.py`, where an RLM processes multiple files/documents in one session [32]. Another example script, `data_extraction.py`, shows using RLM to extract structured data – something very relevant to log parsing or large JSON/XML processing [32]. Although we don't yet have published benchmark results for RLMs on code generation tasks, the ability to handle long contexts (e.g. an entire codebase) and do stepwise execution suggests that domains like **large-scale program synthesis or analysis** could greatly benefit from RLMs. In general, any domain requiring *adaptive planning* over large input data – including scholarly research (reading many papers), legal analysis (parsing lengthy contracts), or multi-step mathematical reasoning – is a candidate for RLM application.

Performance and cost considerations: RLMs introduce overhead in that they may call the base LM multiple times (recursively) instead of once. However, they often *save* cost by avoiding extremely long prompts. By treating the prompt as data, RLMs reduce the token footprint of each LM call. Empirical results showed huge token savings: one report noted that a direct approach to answer a question with a 150K-token context failed due to API limits, whereas the RLM handled it by using only a few thousand tokens internally [33]. In another case, RLM used ~2–3K tokens to process what would normally require a 95K-token prompt [34]. This is because the RLM never sends the whole 95K as input to the LM; it only sends queries and small chunks as needed. As a result, **RLM can be cheaper and faster for long contexts**, up to a point. Moreover, one can optimize RLM calls by using *different model sizes* for different depths. For example, Zhang et al. used GPT-5-mini (a smaller, cheaper model) for the recursive sub-calls, while using a stronger model at the root for the high-level reasoning [35] [36]. This two-tier approach means most of the heavy lifting on text chunks is done by a less expensive model, and the top-level only synthesizes final answers. Such optimizations make RLMs economically attractive for handling huge inputs. The overall success of RLMs in early benchmarks suggests that this approach is a viable way to extend the capabilities of LMs without needing to drastically retrain models for longer context or wait for new architectures – it's a clever reuse of existing models in a recursive fashion.

Implementation Details

Available RLM implementations: The concept of RLM was introduced in late 2025 by Alex L. Zhang and Omar Khattab, along with a blog post and a minimal code release. The official **RLM minimal implementation** (open-sourced by Zhang) is a Python package that demonstrates the core idea [37]. It provides an `RLM_REPL` class that wraps around an OpenAI (or other provider) model API [38]. Soon after, community contributors released improved versions such as the `recursive-llm` project by ysz [39], which is built on top of a library called LiteLLM for broad model support. These implementations share the same design: a *wrapper object* that manages a REPL environment, crafting prompts and parsing the model's outputs. The structure typically includes:

- **System Prompt & Tooling:** When the user calls `rlm.completion(query, context)`, the RLM sets up a system message that explains the special interface to the model. For example, the model is told that the full context text is stored in a variable (often named `context`) in a Python environment, and the model can output Python code to interact with it [7]. The system instructions also define how to indicate the final answer (e.g. "When you have the answer, output `FINAL(answer)`"). This guided prompting is crucial – it's effectively an API contract that the model must follow.

- **REPL Environment:** The RLM creates a sandboxed Python interpreter session (often using libraries like RestrictedPython to ensure safety) [40]. Into this environment, it *pre-loads* the context data (e.g., `context = "<huge text string>"`). The environment also defines a helper function or hook for making recursive calls – for instance, a function `recursive_llm(query, text_chunk)` might be made available in the REPL, which when called will invoke the LM (perhaps a smaller model) on that chunk [7]. This function essentially performs a nested RLM call of depth 1 and returns the result to the environment. By design, each recursive call is isolated in its own sub-environment to avoid side-effects: the sub-call can read the chunk (as its context) and produce an output, which is then passed back up to the parent's environment [41]. This is analogous to a function call stack – each level only has access to its own context variable unless explicitly returned upwards.

- **Main Loop (Controller):** The RLM controller enters a loop where it feeds the model a prompt, gets the model's output, and then interprets that output. If the model's output contains a code block or an `execute` command, the controller will execute that code in the REPL and capture the result [8]. The result (or a truncated version of it, to avoid overflow) is then appended to the model's context for the next prompt, so the model can see what happened when its code ran [8]. If the model's output instead contains a `FINAL(...)` marker, the controller recognizes that as the end of reasoning and will extract the final answer string. This loop continues for a certain number of iterations (configurable, e.g. a `max_iterations` parameter to prevent infinite loops) [42] [43]. The controller thus orchestrates a back-and-forth: model proposes an action (code or sub-query), the action is executed, result is given back to model, and repeat.

- **Prompt formatting:** The prompt sent to the model at each step typically includes: the system message with instructions (persisting across turns), the conversation history of prior steps (e.g. the code it wrote and the output it got), and the user's original query (often kept visible to maintain focus). By using a conversational format, the RLM can leverage the model's conversational/instruction-following training. Some implementations use special tokens or markdown to delimit code. For example, the model might output Python code inside markdown triple backticks ```` ```py ... ``` ```` – the controller will detect those and execute them. In Zhang's prototype, they mention the model can output **code blocks** and receive the outputs inline [8]. The *parser* component of the RLM is responsible for detecting these patterns (code vs. final answer vs. errors) [40]. If the model outputs something not understood, the controller can even nudge it or remind it of the format (this is part of prompt engineering to keep the model on track).

- **Termination and finalization:** When the model has gathered enough information and formulated an answer, it uses the `FINAL` syntax. The controller then stops the loop, possibly cleans up any ephemeral state, and returns the answer. There are two forms used: `FINAL(text)` means the model is directly giving the answer text, whereas `FINAL_VAR(varName)` means the model has constructed the answer in a variable in the REPL and wants to return that (the controller will fetch the content of that variable) [44]. This gives the model flexibility to build a complex answer piece by piece in code if needed (for example, concatenating several retrieved bits) and then output it.

Key implementation patterns (caching, tree structure, memory management): Since RLMs are a new paradigm, there are emerging best practices to make them efficient:

- *Tree-structured recursion:* Under the hood, the sequence of recursive LM calls forms a tree (or at least a call graph). In the current implementations, the depth was limited to 1 (a root calling level-1 submodels) [45]. This was a conscious choice – the authors note that depth=1 was sufficient for benchmarks they tried, but the system could easily allow deeper recursion (a sub-RLM could call another sub-RLM, and so on) [45]. Allowing arbitrary depth turns the process into a general tree search or divide-and-conquer algorithm. One can imagine the model splitting a document into halves, then quarters, etc., with a binary recursion pattern; or it might create a hierarchy of summaries. Each recursive call is like a node in a tree whose children are further calls. Managing this tree is currently straightforward (with depth=1), but as depth increases, one must consider how to aggregate results from multiple branches. The RLM framework would need to decide how to pass intermediate results back up. In a trivial case, the model itself explicitly handles aggregation (e.g., the root asks two sub-models and then combines their answers in code). This is already within the model's capabilities if prompted properly.

- *Caching and re-use:* One challenge with recursion is the potential for repeated work – for example, multiple sub-queries might overlap on parts of the context. Currently, the RLM implementations do **not** have sophisticated caching of intermediate results [46]. Each call is made fresh, which could be redundant. A future improvement would be to cache answers to sub-queries so that if the model asks the same thing twice, it can be retrieved without an actual API call. Another type of caching is *prefix caching*: if the model tends to execute the same initial code (like always doing a quick regex search through the whole context at the start), that could be optimized. The authors acknowledge these optimizations as future work [46]. As RLM workflows become more complex, caching will likely become important to keep costs down and avoid unnecessary loops.

- *Memory and state management:* By design, the entire context is kept in memory (e.g. in a Python variable). This means the system's RAM usage grows with input size, but this is usually not a bottleneck until we get to truly enormous texts (for instance, 10 million tokens might be on the order of ~5GB of text, which is not trivial but also not impossible on a server). The crucial point is that the *model's context window* is not filled with all that text. The model only sees small chunks at a time. This not only avoids token limits but also prevents the prompt from being "cluttered." As the blog notes, *"the context window of the root LM is rarely clogged – because it never directly sees the entire context"* [47]. Instead of one huge prompt, the model's input at any moment is just the query plus maybe a snippet of text or an intermediate result. This keeps the token usage efficient. To manage the iterative process, the RLM loop maintains a running "transcript" of what's happened (the code run and outputs). One has to be careful to truncate or limit what portion of the context output is fed back, especially if printing a large chunk – typically only a summary or the first part of a result is fed to the model, to avoid overflow [8]. Developers can also enforce iteration limits or timeouts so that if a model gets stuck in a recursive loop, the process halts gracefully [43].

- *Use of smaller models for sub-calls:* A practical pattern in RLM usage is to allocate model resources in a hierarchy. The *root* call can use a powerful model (for better reasoning and decision-making), while the *leaf* calls use cheaper, possibly faster models. For example, one might configure `RLM(model="gpt-5", recursive_model="gpt-5-mini")` so that whenever the root spawns a sub-query, it actually calls the smaller "mini" model [36]. This significantly reduces cost, since the bulk of the token consumption might happen in those leaf calls when reading chunks of text. The small model can handle that adequately, and the large model only deals with the high-level combination of results. This idea mirrors the human strategy of delegating grunt work to assistants and only handling the most complex part oneself. The open-source RLM library supports this out of the box as an *optional configuration* [48].

- *Ensuring safe and correct code execution:* Since the RLM runs arbitrary model-generated code, there are obvious safety and reliability concerns. The implementations mitigate this by sandboxing. Using a restricted execution environment means the model cannot perform disallowed operations (like reading files, making network calls, or modifying the system) – it can only manipulate the given `context` data and standard Python libraries like `re` for regex [49]. This prevents abuse or accidents (for example, if the model somehow decided to delete variables or run an infinite loop, the RLM can catch that or limit it). Additionally, error handling is important: if the model's code throws an exception, the controller can catch it and feed the error message back to the model, allowing it to adjust its strategy. This kind of feedback loop helps the model learn what works (much like a programmer debugging code). The RLM logger can even visualize the sequence of actions – Zhang mentioned building a simple visualization of the RLM's reasoning path to understand what it's

"thinking" via code [50] [51]. This traceability is a nice side-effect: we gain some interpretability because the model's intermediate steps are externalized as code and results, which a developer or user can inspect.

Prompting techniques enabling recursion: The success of RLMs relies on clever prompting to steer the base LM. Typically, the system prompt will include explicit instructions and maybe examples. For instance, it might say: *"You are a recursive language model. You have access to a Python environment with a variable* `context` *that contains the user's data. You can output Python code to analyze the context. Use* `print()` *or* `re.findall()`*, etc., to inspect the data. You can also call* `recursive_llm(query, text_chunk)` *to ask a sub-model a question about a part of the text. Do not reveal the context directly. When you find the answer, output it as* `FINAL(answer)`*."* This kind of instruction primes the model to use the tools at hand. Moreover, few-shot examples can be provided (though in Zhang's blog case, it seems they mostly relied on zero-shot with a well-crafted system message). The model being used is often one already trained on code (e.g. GPT-4/5 are trained on tons of code and have the ability to produce correct Python). This is crucial – RLM leverages the model's latent programming skills. Essentially, the model is *writing a program to solve the task*. If the model did not understand Python or the concept of making a function call, it would struggle. So an implicit requirement is a code-capable LM.

Scalability and future improvements: One striking outcome of RLM research is the suggestion that we can scale to far larger inputs without changing the model itself. If a current model can handle *N* tokens context reliably, an RLM built on it might handle *10× or 100× N* tokens by recursive partitioning [52]. Zhang even speculates that if tomorrow's model can handle 10 million tokens, an RLM built on it could handle 100 million (perhaps at half the cost) [53]. This points to a potential roadmap: as base LMs improve (in both context window and reasoning ability), RLMs will proportionally improve their capacity. Another area of future work is training or fine-tuning models to be *better RLM participants*. Right now, the RLM relies on prompting alone. But one could imagine specially fine-tuning an LM to follow the RLM protocol (much like how models were fine-tuned for chain-of-thought or tool use). This might involve training data where the model is taught to decide when to call itself on sub-problems. By doing so, the "policy" of recursion can be learned, possibly with reinforcement learning on long-context tasks [16]. We might also see specialized models for the sub-tasks (for example, a model that is very efficient at summarizing 1000 tokens might be used as the recursive callee). All these are active research directions.

In conclusion, Recursive Language Models represent a promising method to push the boundaries of language model capabilities **without** needing new architectures or infinite context windows. By recursively leveraging existing LMs and treating the input as data to be explored, RLMs have shown superior performance on long-text understanding, the ability to integrate *programmatic actions* into reasoning, and improved cost efficiency for large inputs. This paradigm sits at the intersection of LLM reasoning and classical algorithmic decomposition, and it may very well become the next milestone in advanced AI systems [15]. The initial results are **exciting**: solving tasks that were previously infeasible for LLMs alone, and hinting that with further development (better prompts, fine-tuned recursive reasoning policies, caching mechanisms, etc.), RLMs could become a standard approach for any scenario that "doesn't fit" in a normal prompt [15] [52]. As one commentator put it, it's like turning the LM into an operating system for itself – managing its resources and calling subroutines as needed – opening up *virtually infinite* context and computation within the language model's grasp [2].

Sources:

- Alex L. Zhang's blog post introducing Recursive Language Models [35] [20] [1] [8]
- RLM Python implementation by Alex Zhang (GitHub README) [38] [26]
- *recursive-llm* open-source project (Grigori G.) – documentation and examples [7] [54] [55] [34]
- Discussion and summary of RLM results (Machine Heart report, Chinese) [56] [25] [18]
- Zhang & Khattab (2025), *"Recursive Language Models"* – pre-print/early results [2] [19] [17]

---

[1] [2] [3] [4] [5] [6] [8] [9] [14] [15] [16] [17] [19] [20] [24] [31] [35] [41] [44] [45] [47] Recursive Language Models | Alex L. Zhang
https://alexzhang13.github.io/blog/2025/rlm/

[7] [10] [12] [28] [32] [33] [34] [36] [39] [40] [42] [43] [46] [48] [49] [54] [55] GitHub - ysz/recursive-llm: Recursive Language Models for unbounded context processing. Process 100k+ tokens with any LLM by storing context as variables instead of prompts.
https://github.com/ysz/recursive-llm

[11] [13] [18] [21] [22] [23] [25] [29] [30] [50] [51] [52] [53] [56] 递归语言模型登场！MIT华人新作爆火，扩展模型上下文便宜又简单 - 腾讯云开发者社区-腾讯云
https://cloud.tencent.com/developer/news/3109557

[26] [27] [37] [38] GitHub - alexzhang13/rlm: Super basic implementation (gist-like) of RLMs with REPL environments.
https://github.com/alexzhang13/rlm