



Bocconi Students Investment Club

Beyond PnL: Reward Design and Context Management for LLM-Driven Strategy Research

Innovative LLM architecture for strategy research

Authors

Bocconi Students Investment Club

Matteo Mendicini (Project Lead)

Preslav Georgiev

Luca Ioan Aron

Tudor-Andrei Dolineaschi

May 2026

bsic.it — Bocconi University, Milan

Abstract

Large language models are increasingly deployed as autonomous research agents in quantitative finance: they read briefs, write strategy code, evaluate simulator output, and propose refinements in a loop. This report argues that the dominant bottleneck in such pipelines is not model capability or simulation fidelity, but *reward design*, that is, the choice of what feedback signal an LLM consumes between iterations. PnL only rewards collapse rich behavioural information into a single noisy scalar, encouraging overfitting and obscuring whether a strategy succeeded for the right reasons. We outline a taxonomy of reward signals (PnL, risk adjusted, log derived, multi dimensional) and argue that the most informative signals, namely structured execution logs, are also the ones least compatible with the context windows of current LLMs. We then propose treating log aware reward computation as a long context problem and sketch how the Recursive Language Model (RLM) architecture of [Zhang et al. \(2025\)](#) can serve as a hierarchical evaluator that interrogates logs without flooding the strategy LLM's context. We compare this approach against grep style tool use and retrieval augmented generation. The framework is intended as a general design pattern for LLM driven research loops in finance.

Contents

1	Introduction	3
1.1	The LLM Research Loop in Finance	3
1.2	Two Coupled Problems	3
1.3	Thesis and Contribution	3
2	The Iterative LLM Research Loop	4
3	The Reward Design Bottleneck	4
3.1	Why Reward Choice Dominates	5
3.2	A Taxonomy of Reward Signals	5
3.3	The Case for Log Aware Rewards	5
4	Recursive Language Models as Long Context Evaluators	6
4.1	How RLMs Actually Work	6
4.1.1	The Three Components	6
4.1.2	A Worked Example	7
4.2	RLM as a Reward Function	8
4.3	Why RLMs Beat Grep and RAG for Log Aware Reward	8
4.3.1	Against Grep Style Tool Use	9
4.3.2	Against Retrieval Augmented Generation	9
4.3.3	Summary of the Comparison	9
4.4	Why This Helps Reward Design	10
5	Discussion	10
5.1	Reward Engineering as a First Class Activity	10
5.2	Long Context Tools Belong at the Evaluator	10
5.3	Generalization	10
5.4	Limitations	11
6	Conclusion	11

1 Introduction

1.1 The LLM Research Loop in Finance

A recurring pattern in applied AI for finance is the **closed loop research agent**: a system in which a language model is given a research brief, writes a candidate strategy, executes it inside a simulator, observes the result, and is then prompted to refine. The loop is attractive because it externalizes the slow part of quantitative research, namely iterating on hypotheses, to a model that does not tire and that can explore combinatorial design spaces faster than a human analyst. It also maps onto where most quant work actually lives in production: the bulk of a researcher's day to day is maintaining, retuning, and incrementally extending strategies that already exist, not inventing new ones from scratch. In that regime the strategy itself is not the bottleneck; the bottleneck is the rate at which sensible variations can be proposed, evaluated, and either kept or rejected, which is exactly what a closed loop is good at.

The pattern is general. It appears in equity factor research, in algorithmic execution, in derivative hedging policy search, and in any setting where a strategy is iterated against a simulator. In each case, the system has the same skeleton: a **generator** that produces strategy artifacts, an **evaluator** (typically a backtester or simulator) that produces signals about how those artifacts behaved, and a **feedback channel** that translates evaluator output back into something the generator can condition on.

1.2 Two Coupled Problems

Practitioners who have built such loops know that two problems show up immediately and refuse to leave.

The first problem is reward design. What does the LLM see between iterations? Most pipelines default to a small set of scalars, usually PnL, sometimes Sharpe or drawdown. This is the path of least resistance because backtesters report exactly these numbers. But a scalar reward erases the very information that distinguishes a strategy that worked for a good structural reason from one that won a coin flip. When the generator is conditioned only on scalars, refinement degenerates into hill climbing on a noisy surface.

The second problem is context rot. Rich feedback (order by order execution traces, queue dynamics, fill timing, log lines from the strategy's own debug statements) is exactly the feedback that a human researcher would use to diagnose a backtest. But a single day of tick level logs from even a small instrument universe routinely exceeds the context window of frontier models, and stuffing logs into prompts well below those limits has been observed to degrade reasoning quality, a phenomenon often referred to colloquially as *context rot*. The result is a forced choice: either give the model a shallow scalar it can swallow whole, or drown it in detail it cannot effectively process.

1.3 Thesis and Contribution

The central claim of this report is that these two problems are the same problem, and that solving the second is a precondition for solving the first. Specifically:

1. Reward design, not model capability, is the dominant bottleneck in LLM driven strategy research loops. The community treats the reward function as a fixed constant and the model as the variable. The opposite framing is more productive.

2. The most useful reward signals available to a financial AI agent live in structured logs, not in scalar metrics. These logs are too long to fit in a single LLM call.
3. Recursive Language Models (RLMs) (Zhang et al., 2025) provide a natural mechanism for converting long horizon log streams into compact, decision relevant feedback for the generator, without forcing it to ingest the raw stream. The mechanism is mechanically distinct from, and strictly more expressive than, the grep based and retrieval based alternatives that are usually proposed.

The contribution is a design pattern and an argument for it. We attempted to validate it end to end, but the regime in which we expect an RLM evaluator to perform well, with capable models orchestrating tens of recursive sub calls per backtest over many refinement iterations, is expensive to run; the cost made it impractical for us to exercise the architecture at full capacity.

2 The Iterative LLM Research Loop

Figure 1 shows the canonical structure of an LLM research loop, abstracted away from any specific asset class. A research brief seeds a **combination generator**, which enumerates concrete strategy variants. Each variant is passed to a **code generator** that produces an executable artifact, which is then run inside a **simulator**. The simulator returns metrics and logs. A **reward function** compresses this output into the feedback the generator will condition on for the next iteration.

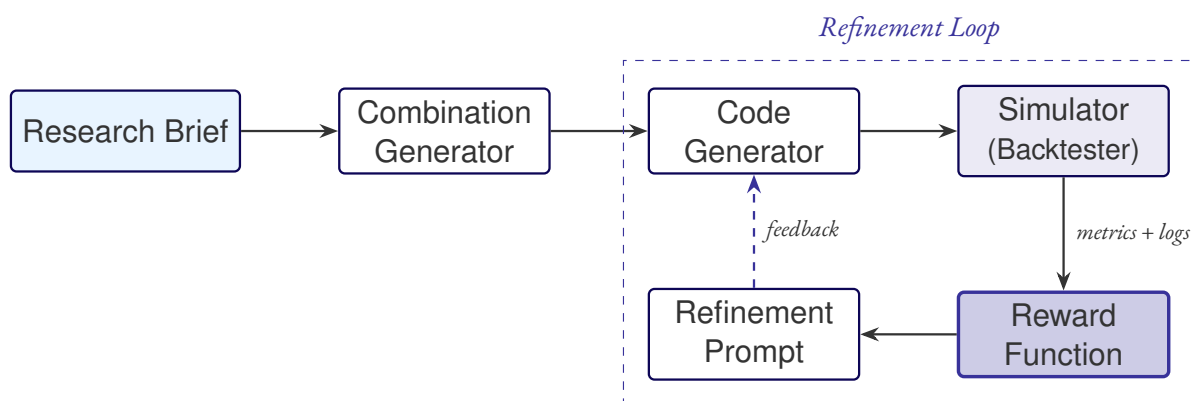


Figure 1: Canonical LLM research loop. The dashed box marks the iterative segment; everything inside it is repeated up to a fixed iteration budget. The reward function (highlighted) is the focus of this report.

The loop is asset agnostic. It applies to factor research over historical bars, to hedging policy search over Monte Carlo paths, to execution algorithm tuning over limit order book replays, and to algorithmic trading competitions in which a strategy is repeatedly evaluated against a fixed simulator. We have built and tested a concrete instantiation of the loop in this last setting, where the reward channel issues we describe below were straightforward to reproduce; the framework, however, is meant to apply wherever an LLM iterates against a simulator that produces more output than the model can read.

3 The Reward Design Bottleneck

3.1 Why Reward Choice Dominates

In a fixed budget refinement loop, the LLM samples from a posterior over strategy space conditioned on the reward signals it has observed. The shape of that posterior, and therefore the trajectory of the search, is determined almost entirely by what the reward function reveals and what it hides. Two pipelines with identical models, prompts, and simulators but different reward functions will explore meaningfully different regions of strategy space.

This is not a novel observation in reinforcement learning. It is, however, routinely under appreciated in LLM in the loop systems, because the reward function in those systems is rarely engineered: it is whatever the simulator happened to print. The reward is inherited rather than designed.

3.2 A Taxonomy of Reward Signals

Table 1 sketches the main reward classes encountered in financial AI loops. Each row trades off informational content against the cost of consuming it.

Table 1: Reward signal classes in LLM driven strategy iteration.

Class	Content	Context cost	Failure mode
PnL only	Total profit per run, possibly per day	Negligible	Overfitting on noise; rewards lucky paths
Risk adjusted scalar	Sharpe, Sortino, max draw-down	Negligible	Punishes legitimate convex bets; opaque cause
Aggregated behavioural	Win rate, fill ratio, inventory turnover, queue position	Low	Hides interaction effects between products
Log derived narrative	Natural language summary of execution traces	Medium	Summary quality bounds reward quality
Raw logs	Full per tick order/trade/state stream	Prohibitive	Context rot; degraded reasoning

The qualitative pattern we observed across our own runs, conducted on a budget that did not let us push the loop to the model and iteration scale at which we believe it would perform well, is that PnL only and risk adjusted scalar rewards both produce the same pathology. The generator latches onto whichever variant happened to score well in the most recent iteration and proposes minor perturbations of it, regardless of whether the underlying economic story is sound. The refinement loop becomes a noisy gradient ascent on a metric the model cannot trust, because the metric never tells the model *why* a run succeeded.

3.3 The Case for Log Aware Rewards

The information that distinguishes a structurally sound strategy from a lucky one almost always lives in execution detail: which trades fired, against which counterparties, at what queue positions, in response to which inventory state. A human researcher reviewing a backtest reads logs first and PnL second, because logs are causal and PnL is consequential. Any reward function that does not surface log level evidence is asking the LLM to do less than the human does.

The obstacle is mechanical rather than conceptual. A single backtest can produce on the order of 10^4 to 10^5 structured events; a tick level FX backtest several orders of magnitude more. None of these fit in a frontier model context window, and even when they nominally do, model performance on long inputs degrades well before the documented limit.

This is why the reward design problem and the context management problem are the same problem. The reward signals worth giving the model are precisely the ones the model cannot directly read.

4 Recursive Language Models as Long Context Evaluators

4.1 How RLMs Actually Work

Zhang et al. (2025) introduce Recursive Language Models (RLMs), an architecture in which a language model treats a long input as an external environment rather than as in context tokens. The framing converts long context handling from an architectural problem (longer attention, better positional encoding) into an algorithmic one (decomposition strategy, recursion budget).

4.1.1 The Three Components

An RLM has exactly three moving parts:

1. A **root LLM**, which is the only component the outside world talks to. The root never reads the long input directly. Its context contains the user's question and a transcript of its own interactions with a Python environment.
2. A **persistent Python REPL**, initialised with the long input bound to a variable (typically called `context`). This REPL is the only place where the long input physically lives. It persists for the entire RLM session, so any variables the root creates remain available across reasoning steps.
3. A **sub LLM call primitive**, exposed in the REPL as a function `llm_query(prompt)`. Calling it spawns a fresh, independent LLM invocation with whatever prompt the root constructed programmatically, and returns the response as a Python string.

The root interacts with the REPL by emitting Python code; the REPL executes it and returns whatever the code printed. The root also has `FINAL(answer)` to end the session. That is the entire interface. Figure 2 shows the data flow.

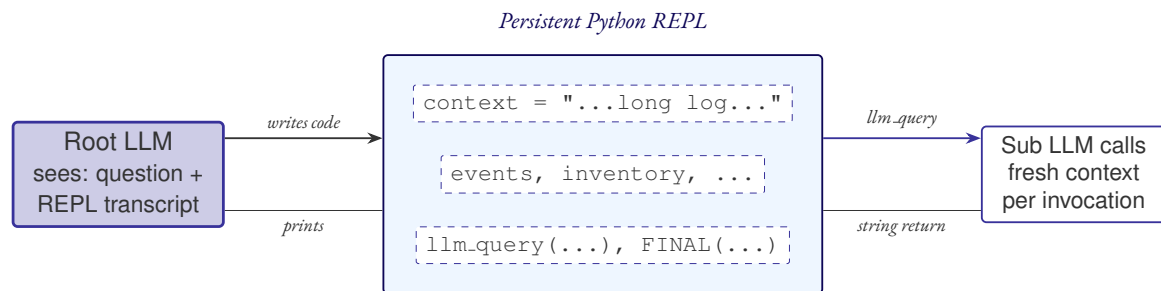


Figure 2: Internal data flow of an RLM. The root LLM never touches the long input directly. The REPL holds it as a variable and exposes only `print` output and `llm_query` return values back to the root. Sub LLM calls operate on whatever short slice the root chooses to construct.

4.1.2 A Worked Example

To make the mechanics concrete, consider the diagnostic question *Did inventory drift correlate with adverse selection?* applied to a backtest log of roughly 80,000 events. The log will not fit in any frontier model's context window, and even truncated to a token budget that does fit, a single shot model would have to perform parsing, two derived time series computations, and statistical aggregation in token space. An RLM session for the same question proceeds as in Listing 1.

Listing 1: Abbreviated RLM trace. `>>>` lines are code the root LLM emits; `#out` lines are what the REPL prints back into the root's context.

```

# Initial: context = "<80,000 lines of backtest events>" (set by runtime)
>>> print(context[:200]) # 1. peek at schema
#out: t=0 mid=10000 best_bid=9999 ... t=1 fill side=BUY price=10001 ...
>>> events = [parse_line(l) for l in context.splitlines() if l] # 2. parse
>>> fills = [e for e in events if e['type']=='fill']
>>> adverse = [(f['mid_after_50t']-f['price'])*(-1 if f['side']=='BUY' else 1)
...             for f in fills] # 3. derive series in Python
>>> rho = pearson([events[f['t']]['pos'] for f in fills], adverse)
>>> print(f"corr(inv, adverse) = {rho:.3f}") # zero LLM tokens so far
#out: corr(inv, adverse) = 0.41
>>> windows = find_high_inv_high_adverse_windows(events, adverse) # 4.
localise
>>> print([(w.start, w.end) for w in windows])
#out: [(4210,4612), (8104,8377), (15880,16021)]
>>> verdicts = [llm_query("Diagnose mechanism for this window:\n" # 5. sub-LLM
...                     + fmt(events[w.start:w.end])) for w in windows] # on slices
>>> FINAL(llm_query("Synthesize: " + str(verdicts))) # 6. return

```

Three properties of this trace are worth highlighting. First, the long input is read *programmatically*, not by an LLM: steps 2 to 4 do parsing, derivation, and filtering in Python and consume zero LLM tokens. Second, every LLM call in the session has a short context: the root sees only what is printed (a few hundred tokens total across the whole session), and each sub call sees only the slice the root chose for it (a few hundred to a few thousand tokens). Third, intermediate state (`events`, `adverse`, `windows`, `verdicts`) lives in REPL variables across steps; nothing has to be re-derived or re-summarised on each turn, which is the primitive that distinguishes an RLM from a tool using agent that issues isolated `grep` or `read` calls.

The root's own context, viewed as a transcript, never contains a single full event line of the original log. It

contains the user’s question, the small things it chose to print, and the short answers its sub calls returned. That is the precise mechanism by which RLMs handle inputs “up to two orders of magnitude beyond model context windows” (Zhang et al., 2025): by ensuring the long input is never in any single model’s window in the first place.

4.2 RLM as a Reward Function

We propose a straightforward reuse of the RLM idea outside its original setting. Rather than placing an RLM at the strategy generator, where its long context capability would be spent reading market history, we place it at the *reward function*, where its job is to read the simulator’s output and emit a compact, decision relevant feedback message back to the generator. Figure 3 sketches the resulting architecture.

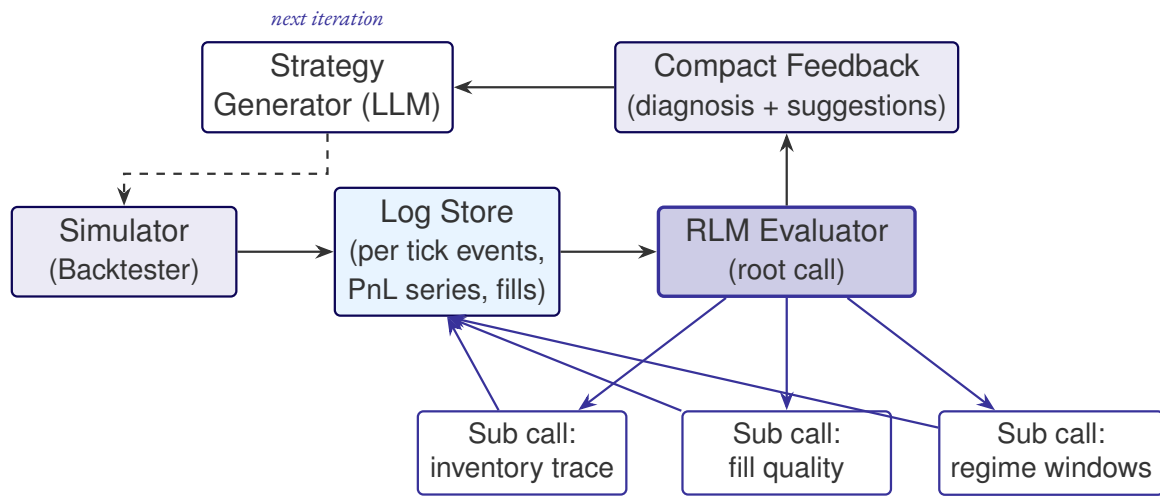


Figure 3: RLM based reward architecture. The strategy generator never reads raw logs; the RLM evaluator does, recursively, on demand. The feedback message returned to the generator is short by construction.

In this arrangement the generator’s context contains only (i) the brief, (ii) the current strategy, and (iii) a short feedback message produced by the RLM. The RLM is the only component that touches the long log stream, and it does so via recursive sub calls scoped to specific diagnostic questions: *Did inventory drift correlate with adverse selection?*, *Were our quotes consistently behind the queue?*, *Did the regime shift between morning and afternoon?* Each sub call returns a small structured answer; the root call composes those answers into a feedback message of bounded length. The REPL itself runs in a sandbox so that arbitrary LLM generated Python cannot reach the orchestrator host; we use [E2B](#) for this.

4.3 Why RLMs Beat Grep and RAG for Log Aware Reward

The natural objection to introducing a recursive evaluator is that simpler tools already exist. A tool using agent can be given `grep` and `read_file`; a retrieval pipeline can chunk the logs, embed them, and serve nearest neighbour passages on demand. Both alternatives are cheaper to build than an RLM. Both are also, mechanically, the wrong shape for the questions a log aware reward function actually has to answer.

4.3.1 Against Grep Style Tool Use

A `grep` based agent retrieves *lines* matching a textual pattern. The questions that distinguish a structurally sound strategy from a lucky one are not lookups. They are computations. *Did inventory drift correlate with adverse selection?* cannot be answered by matching a string, because neither “inventory drift” nor “adverse selection” appears in any log line. The answer requires (i) parsing the log into structured records, (ii) computing two derived time series, and (iii) checking their statistical relationship. A `grep` agent has nowhere to put the parsed records, no way to perform the arithmetic, and no way to retain intermediate results across tool calls. An RLM gets all three for free because its tool surface is a Python REPL, not a search command. Numerical work happens in Python and consumes zero LLM tokens; intermediate state lives in REPL variables that persist across the root’s reasoning steps.

4.3.2 Against Retrieval Augmented Generation

A RAG pipeline indexes the logs into an embedding store and retrieves passages by semantic similarity. This works when the question is a content lookup whose answer is textually present somewhere in the corpus. It fails for log aware reward computation for three independent reasons. First, the relevant evidence is rarely textually similar to the question; “fills clustered around regime breaks” has no semantically near match in raw event lines. Second, the relevant evidence is a *conjunction* of conditions (high inventory *and* fills moving against us *and* within a specific time window) that no embedding similarity reliably captures, because composite predicates have to be computed, not retrieved. Third, RAG retrieves passages without context about *why* they were selected, whereas an RLM’s diagnostic reasoning is preserved in the REPL state and can be revisited in subsequent sub calls.

4.3.3 Summary of the Comparison

Table 2 summarises the comparison along the dimensions that matter for reward computation.

Table 2: Mechanical comparison of long log access strategies.

Capability	Grep agent	RAG	RLM
Textual pattern lookup	Yes	Yes	Yes
Numerical computation over logs	No	No	Yes (Python in REPL)
Composite/conjunctive predicates	Awkward	Unreliable	Yes
Persistent intermediate state	No	No	Yes (REPL variables)
Question driven decomposition	Manual	No	Yes (recursive sub calls)
Bounded root context	Yes	Yes	Yes

The RLM is the only option that can both *compute* on the long input and *reason recursively* about the results, while keeping the root LLM’s context bounded. For log derived reward signals, where the interesting

questions are composite computations rather than passage lookups, this combination is not a luxury; it is the minimum requirement.

4.4 Why This Helps Reward Design

Two properties of this design directly address the bottleneck identified in Section 3.

First, the **reward function becomes a research artifact**. The set of diagnostic sub calls the RLM is allowed to make is itself a design choice, separable from the generator and from the simulator. Improving the loop no longer requires touching the generator's prompt; it requires curating the evaluator's question set. This decouples reward iteration from strategy iteration, which is the discipline that scalar reward pipelines lack.

Second, the **generator is protected from context rot** without being deprived of log level information. It receives a synthesis of log derived evidence, not the logs themselves, and the synthesis is produced by a system whose entire purpose is to read long inputs well. The generator can be a smaller, cheaper, faster model than the evaluator, because the evaluator absorbs the cost of long horizon reasoning.

5 Discussion

5.1 Reward Engineering as a First Class Activity

The most direct recommendation arising from this work is that teams building LLM in the loop research systems should treat the reward function as a versioned, iterated artifact, on the same footing as prompts and model choice. In the canonical pipeline (Figure 1) the reward function is the smallest box and receives the least attention; in practice it does the most work to determine whether the loop converges to anything useful. Reward design deserves explicit experiments, ablations, and review.

5.2 Long Context Tools Belong at the Evaluator

The instinct in long context research has been to give the most context hungry component, usually the generator, the largest window. Our argument is that the evaluator is the better target. The evaluator's job is summarization; summarization rewards long inputs more cleanly than generation does, because the output is bounded by the question rather than by the input. RLMs and similar hierarchical reasoning architectures are a natural fit at this position in the pipeline.

5.3 Generalization

Although the loop is easiest to demonstrate in a simulator with cheap evaluation cycles, nothing in the framework depends on that setting. The same pattern applies wherever an LLM iterates over simulator output: factor research over historical bars, hedging policy search over Monte Carlo paths, execution algorithm tuning over LOB replays, and risk model calibration over scenario sets. In each case, the generator is bottlenecked on what the evaluator chooses to surface, and in each case the evaluator's input is too large for direct ingestion. The RLM as reward pattern is a general response to a general problem.

5.4 Limitations

Three limitations of the present report are worth stating plainly. First, while we attempted to validate the architecture, doing so at the scale we expect to be required for it to perform well, with frontier models orchestrating many recursive sub calls per evaluation across long refinement runs, is very expensive; we could not exercise the system at full capacity within the resources available to us, so the headline contribution remains a design pattern rather than a benchmarked result. Second, the RLM literature is recent and the engineering practices around it are not yet settled: decomposition strategies, recursion budget tuning, and evaluator alignment are open questions. Third, log derived rewards inherit whatever biases live in the simulator, and a more capable evaluator can amplify those biases as easily as it can correct for them; reward auditing remains a human responsibility.

6 Conclusion

LLM driven strategy research loops fail not because models cannot write strategy code and not because simulators cannot evaluate it, but because the channel between simulator and model is too narrow to carry the information that would make iteration productive. PnL first rewards are inherited from backtesters that were designed for human readers, who could supplement the scalar with their own log inspection. LLMs cannot, and a pipeline that gives them only the scalar is asking less of them than the system is capable of producing.

The two ways out are (a) richer reward signals, and (b) machinery for handling those signals without overflowing the generator's context. Recursive Language Models supply the second, and they do so in a way that grep style tool use and retrieval augmented generation cannot replicate, because the questions that matter for reward computation are computations over the logs rather than passage lookups inside them. The recommendation is small in scope and large in consequence: move the long context capability to where the long context actually is, and let the generator stay short, fast, and focused on the task it is good at.

References

Zhang, A. L., Kraska, T., & Khattab, O. (2025). Recursive Language Models. *arXiv preprint arXiv:2512.24601*. Available at <https://arxiv.org/abs/2512.24601>.