
The Y-Combinator for LLMs: Solving Long-Context Rot with λ -Calculus

Amartya Roy

The School of Interdisciplinary Research
Indian Institute of Technology (IIT) Delhi
Robert Bosch GmbH, India
srz248670@iitd.ac.in

Rasul Tutunov

Huawei Noah’s Ark Lab
rasul.tutunov@huawei.com

Xiaotong Ji

Huawei Noah’s Ark Lab
xiaotong.ji1@h-partners.com

Matthieu Zimmer

Huawei Noah’s Ark Lab
matthieu.zimmer@huawei.com

Haitham Bou-Ammar

Huawei Noah’s Ark
UCL Centre for Artificial Intelligence
haitham.ammar@huawei.com

Abstract

LLMs are increasingly used as general-purpose reasoners, but long inputs remain bottlenecked by a fixed context window. Recursive Language Models (RLMs) address this by externalising the prompt and recursively solving subproblems. Yet existing RLMs depend on an open-ended read-eval-print loop (REPL) in which the model generates arbitrary control code, making execution difficult to verify, predict, and analyse.

We introduce λ -RLM, a framework for long-context reasoning that replaces free-form recursive code generation with a typed functional runtime grounded in λ -calculus. It executes a compact library of pre-verified combinators and uses neural inference only on bounded leaf subproblems, turning recursive reasoning into a structured functional program with explicit control flow. We show that λ -RLM admits formal guarantees absent from standard RLMs, including termination, closed-form cost bounds, controlled accuracy scaling with recursion depth, and an optimal partition rule under a simple cost model. Empirically, across four long-context reasoning tasks and nine base models, λ -RLM outperforms standard RLM in 29 of 36 model-task comparisons, improves average accuracy by up to **+21.9** points across model tiers, and reduces latency by up to **4.1 \times** . These results show that typed symbolic control yields a more reliable and efficient foundation for long-context reasoning than open-ended recursive code generation. The complete implementation of λ -RLMs, is open-sourced for the community at: github.com/lambda-calculus-LLM/lambda-RLM.

1 Introduction

Large language models (LLMs) are increasingly used as general-purpose problem solvers [Brown et al., 2020, Yao et al., 2023a, Mower et al., 2024, Zimmer et al., 2025b, Ji et al., 2026a], yet one of their most fundamental bottlenecks remains unchanged: *a Transformer consumes a fixed-length context window* [Dai et al., 2019]. When inputs exceed this limit, e.g., long documents, codebases,

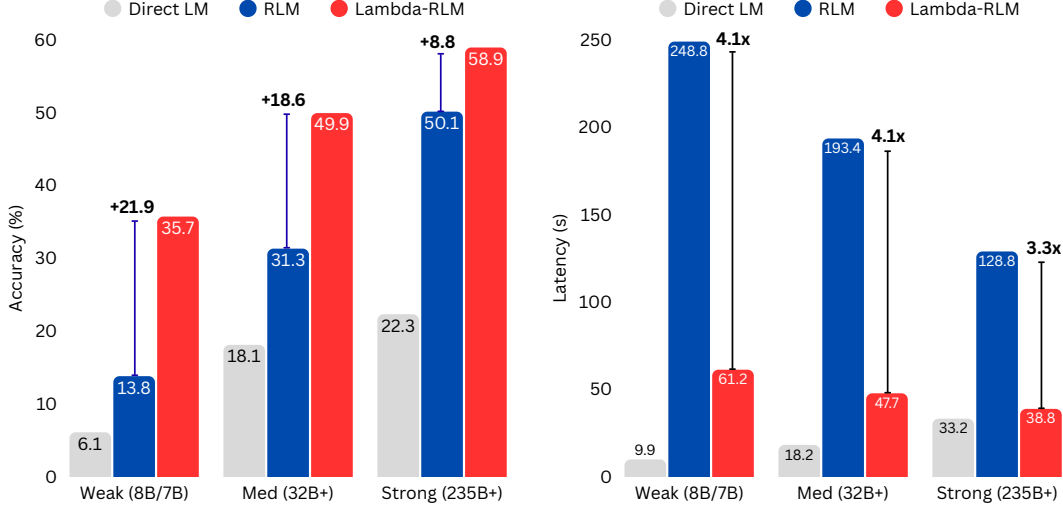


Figure 1: A summary of our results comparing λ -RLMs to base LLMs and recursive LLMs. Those results demonstrate improvements reaching **+21.9** in accuracy, with **4.1 \times** in latency reductions.

multi-file repositories, or large collections of evidence, naïvely truncating context or relying on sliding-window prompting forces the model to “forget” early information and often breaks tasks that require global consistency or systematic evidence gathering [Liu et al., 2023, Wang et al., 2024]. In response, a growing line of work reframes long-context reasoning as inference-time scaling: rather than increasing model parameters or training new architectures, we can scale computation at inference by decomposing problems into smaller subproblems and composing their solutions [Zhou et al., 2023, Yao et al., 2023b,a, Yang et al., 2025b].

A particularly compelling recent proposal is Recursive Language Models (RLMs), which argues that arbitrarily long user prompts should not be fed into the neural network directly [Zhang et al., 2026]. Instead, the prompt should be treated as part of an external environment that the model can interact with symbolically. Concretely, RLM initialises a programming environment (a REPL) in which the prompt is stored as a variable; the LLM then writes code to peek into the prompt, decompose it into slices, and recursively invoke itself on those slices as needed. This simple interface, prompt-as-environment plus symbolic recursion, enables models to handle inputs far beyond their native context length while retaining a standard “string-in, string-out” API.

However, RLM’s power comes with a practical cost: it relies on an LLM-driven control loop that emits and executes arbitrary code until the model decides it has finished. This open-ended REPL loop is difficult to bound and audit. In practice, it creates several failure modes that are orthogonal to the underlying reasoning task: code may not parse or may crash at runtime; recursion may be invoked excessively; intermediate outputs may be malformed; and computation may become unpredictable due to the model’s own control-flow decisions. More broadly, giving an LLM unrestricted freedom to program its own execution introduces an undesirable coupling between what the model knows and how it is allowed to search and compose evidence.

In this work, we propose λ -RLM, a framework that retains the key insight of RLM, prompt-as-environment with recursive decomposition but replaces open-ended code generation with a typed, functional runtime grounded in λ -Calculus. λ -RLM expresses all control flow through a small library of deterministic, compositional operators (e.g., SPLIT, MAP, FILTER, REDUCE) that are pre-verified and loaded into the REPL before execution. The base language model \mathcal{M} is invoked only at the leaves of the recursion, on sub-prompts that are guaranteed to fit within its context window K ; all higher-level decisions i) how to split, ii) how many chunks, iii) when to stop, iv) how to compose are made by a planner and executed symbolically, without any LLM-generated code. Recursion is encoded as a fixed-point over this operator library (Section 3), and the planner enforces predictable execution: maximum depth $d = \lceil \log_{k^*}(n/\tau^*) \rceil$, a pre-computed number of \mathcal{M} calls, and deterministic composition at every level. As a result, λ -RLM separates *semantic reasoning* from *structural control*: the model contributes understanding only where it is needed; at leaf sub-problems

small enough to process reliably while all orchestration is handled by an auditable, deterministic controller with formal guarantees on termination, cost, and accuracy.

We choose λ -Calculus as our foundation because it provides a minimalist yet universal interface for hierarchical reasoning that other formalisms lack. While Finite State Machines (FSMs) are insufficient for the arbitrary recursion depths required in complex document decomposition, and Planning Domain Definition Languages (PDDL) are optimised for state-space search rather than data transformation, λ -Calculus treats the prompt as a first-class functional object. Crucially, by utilising fixed-point combinators (e.g., the Y -combinator), λ -RLM "ties the knot" of recursion without requiring the LLM to manage function names or global state, effectively eliminating the reference errors and non-termination failures common in open-ended REPL loops.

Our design choices yield three benefits. First, λ -RLM provides termination by construction under mild conditions on the splitting operator, eliminating a common class of non-termination and runaway-execution failures in agentic scaffolds. Second, it yields predictable computation: we can bound the number of oracle calls and the total work as a function of input size and the chosen decomposition policy. Third, it improves reliability by reducing the number of "critical decisions" delegated to the language model. We evaluate λ -RLM on long-context task settings, using RLM as a primary baseline. In short, our contributions can be summarised as: *i)* We introduce λ -RLM, a typed functional runtime for prompt-as-environment long-context reasoning with recursion expressed as a fixed-point over deterministic combinators; *ii)* We formalise an operational semantics and prove termination and cost bounds under standard size-decreasing decomposition assumptions; and *iii)* We empirically compare against RLM, demonstrating improved reliability and more predictable compute while improving task performance.

We validate λ -RLM on four long-context task families spanning search, aggregation, pairwise reasoning, and code understanding, across nine base models and context lengths up to 128K. Compared with normal RLM, λ -RLM wins in 29/36 model-task comparisons (**81%** overall), improves average accuracy by up to **+21.9** points on weak models and **+18.6** points on medium models, and delivers consistent latency reductions of **3.3 \times** to **4.1 \times** . On the most structurally demanding benchmark, OOL-Pairs, the gain reaches **+28.6** points with a **6.2 \times** speedup. These results show that constraining control flow to a typed combinator runtime not only improves predictability but also leads to substantial empirical gains over open-ended recursive code generation.

2 A Short Primer on λ -Calculus

The lambda calculus is a minimal formal language for describing computation using only *functions* and *functional operations*. We include a brief primer here because λ -RLM uses a functional view of control flow: recursion and composition are expressed as combinations of small operators, rather than as an LLM-driven loop that generates arbitrary code.

We use Exp to denote the set of (untyped) lambda-calculus expressions, and Var to denote a countable set of variable names. The grammar is defined by:

$$\text{Exp} ::= x \mid (\lambda x. \text{Exp}) \mid (\text{Exp} \text{Exp}), \quad x \in \text{Var}.$$

Intuitively, the above is saying that every expression is one of three forms: *i)* A variable which is a placeholder: it gains meaning when it is bound by a function or substituted during evaluation; *ii)* An abstraction (function definition): If $e \in \text{Exp}$ and $x \in \text{Var}$, then $\lambda x. e \in \text{Exp}$. This is to be read as: "a function that takes an argument $x \in \text{Var}$ and returns $e \in \text{Exp}$. For instance, we could define an identity function as: $\text{id} = \lambda x. x$, or a constant function that returns its first argument as: $\text{const} = \lambda x. \lambda y. x$; and *iii)* An Application (functional call): If $e_1, e_2 \in \text{Exp}$, then $(e_1, e_2) \in \text{Exp}$, which is to be read as "apply e_1 to e_2 ". For example, the abstraction $(\lambda x. x) y$ applies the identity function to y . By convention, application associates to the left: $e_1 e_2 e_3 \equiv (e_1 e_2) e_3$. In other words, $f a b$ means "first apply f to a , then apply the result to b ". We may omit the outer parentheses when unambiguous.

Syntax tells us what expressions look like. Evaluation tells us how expressions compute. In the untyped lambda calculus, the central computational rule is β -reduction, which formalises what it means to apply a function to an argument. If we have a function $\lambda x. e$ and we apply it to an argument a , we reduce by substituting the argument a for the variable x inside the body e :

$$(\lambda x. e) a \xrightarrow{\beta} e[x := a], \quad \text{where } e[x := a] \text{ takes } e \text{ and replaces every free occurrence of } x \text{ by } a.$$

In other words, β -reduction is just a function application as substitution, exactly like evaluating a function call. Let us develop some examples to understand β -reduction better:

Examples of β -Reduction

Identity. Let $\text{id} = \lambda x. x$. Then $(\lambda x. x) y \xrightarrow{\beta} y$: applying the identity returns its input.

Constant function. Let $\text{const} = \lambda x. \lambda y. x$. By left associativity, $\text{const } a \ b = ((\lambda x. \lambda y. x) a) b$. Reducing step by step:

$$\underbrace{(\lambda x. \lambda y. x) a}_{\xrightarrow{\beta} \lambda y. a} b \xrightarrow{\beta} a.$$

The outer λ binds a , yielding a constant function $\lambda y. a$ that ignores its argument. Applying it to b returns a .

Recursion & Fixed-Point Combinators. λ -Calculus functions are anonymous, so recursion is not built-in. In Python one writes `def f(...): ... f(...) ...` the name `f` enables self-reference. Without names, the trick is *fixed points*: a value u satisfying $u = g(u)$ for a given function g .

A *fixed-point combinator* `fix` is a higher-order term satisfying $\text{fix}(g) = g(\text{fix}(g))$ for all g . Intuitively, g is a non-recursive *recipe* that says: “here is one step of the computation, assuming you already have a solver f for strictly smaller sub-problems.” The combinator `fix` *ties the knot*, converting this one-step recipe into a genuinely recursive function by ensuring $f = g(f)$.

In the untyped λ -Calculus, one concrete realisation is the Y-combinator \mathbf{Y} , satisfying $\mathbf{Y} g \xrightarrow{\beta} g(\mathbf{Y} g)$ —a fixed point of g without any external naming mechanism.

Definition 1 (Fixed-Point Combinator). *The Y-combinator enables recursion in the untyped lambda calculus: $\mathbf{Y} \equiv \lambda f. (\lambda x. f(x x)) (\lambda x. f(x x))$, satisfying $\mathbf{Y} g = g(\mathbf{Y} g)$ for all g .*

Worked Example: Factorial via the Y-Combinator

In Python, factorial calls itself by name: `def fact(n): return 1 if n==0 else n*fact(n-1)`. In λ -Calculus there are no names, so we separate the *one-step recipe* from the recursion mechanism.

Step 1: Write the recipe. Define a functional G that takes a candidate solver f and returns a one-step factorial procedure:

$$G \triangleq \lambda f. \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \cdot f(n - 1).$$

G is *not* recursive - it never calls itself. It says: “given a solver f for smaller inputs, here is one step.”

Step 2: Apply the Y-combinator. Recall $\mathbf{Y} = \lambda g. (\lambda x. g(x x)) (\lambda x. g(x x))$. Define $\text{fact} \triangleq \mathbf{Y} G$ and expand:

$$\begin{aligned} \text{fact} &= \mathbf{Y} G = (\lambda g. (\lambda x. g(x x)) (\lambda x. g(x x))) G \\ &\xrightarrow{\beta} (\lambda x. G(x x)) (\lambda x. G(x x)) && \text{(substitute } g := G) \\ &\xrightarrow{\beta} G(\underbrace{(\lambda x. G(x x)) (\lambda x. G(x x))}_{= \mathbf{Y} G = \text{fact}}) = G(\text{fact}). && \text{(the knot is tied)} \end{aligned}$$

The self-referential term $(x x)$ is the engine: each copy of $\lambda x. G(x x)$ feeds *itself* as the argument, producing $G(\text{fact})$ - exactly the identity $\mathbf{Y} G = G(\mathbf{Y} G)$.

Step 3: Verify. Expanding $G(\text{fact})$ recovers the familiar recursive definition:

$$\text{fact} = G(\text{fact}) \xrightarrow{\beta} \lambda n. \text{if } (n = 0) \text{ then } 1 \text{ else } n \cdot \text{fact}(n - 1).$$

Step 4: Trace $\text{fact}(3)$. Each recursive call re-triggers the same \mathbf{Y} machinery:

$$\text{fact}(3) \xrightarrow{\beta} 3 \cdot \text{fact}(2) \xrightarrow{\beta} 3 \cdot 2 \cdot \text{fact}(1) \xrightarrow{\beta} 3 \cdot 2 \cdot 1 \cdot \text{fact}(0) \xrightarrow{\beta} 3 \cdot 2 \cdot 1 \cdot 1 = 6.$$

2.1 Core Definitions for λ -RLM

In addition to what we presented above, this section also introduces additional definitions needed for the remainder of the paper. Namely, we introduce base language models, cost functions for invoking a base model, and accuracy decays for those models as a function of the prompt’s length.

Definition 2 (Base Language Model). *A base language model is a function $\mathcal{M} : \Sigma^* \rightarrow \Sigma^*$ with context window $K \in \mathbb{N}$, such that \mathcal{M} is only defined (or reliable) on inputs of length $|P| \leq K$.*

Definition 3 (Cost Function). *The cost of invoking \mathcal{M} on n tokens:*

$$\mathcal{C}(n) = c_{in} \cdot n + c_{out} \cdot \bar{n}_{out} \quad (1)$$

where c_{in}, c_{out} are per-token prices and \bar{n}_{out} is expected output length.

Definition 4 (Accuracy Decay). *The accuracy of \mathcal{M} on a prompt of length n :*

$$\mathcal{A}(n) = \mathcal{A}_0 \cdot \rho^{n/K}, \quad \rho \in (0, 1] \quad (2)$$

where \mathcal{A}_0 is peak accuracy and ρ is the context-rot decay factor.

Definition 5 (Composition Operator). *A composition operator $\oplus : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ is a deterministic function that combines partial results. We define a family $\{\oplus_\tau\}_{\tau \in \mathcal{T}}$ indexed by task type τ .*

3 The λ -RLM Framework

The key idea behind λ -RLM is simple: long-context reasoning should be recursive, but the recursion should be executed by a small trusted runtime rather than by arbitrary code written by the language model.

λ -RLM keeps the central insight of RLM—prompt-as-environment with recursive decomposition, but replaces open-ended code generation with a typed functional runtime. Instead of allowing the model to emit arbitrary programs, λ -RLM executes a fixed library of pre-verified combinators such as SPLIT, MAP, FILTER, and REDUCE. The base language model is used only as a bounded oracle on small leaf subproblems. In this way, λ -RLM separates reasoning content, which remains neural, from control flow, which becomes symbolic, deterministic, and auditable.

This design is appealing for three reasons. First, it makes execution more reliable by removing many failure modes associated with free-form code generation. Second, it makes computation predictable: once a decomposition strategy is chosen, the number of recursive calls is bounded in advance. Third, it makes the system easier to analyse formally, since recursion is expressed through a fixed functional structure rather than an open-ended loop.

3.1 From Open-Ended Control to a Restricted Runtime

Standard RLM operates through a REPL-style interaction. At each iteration, the model generates a code snippet from the conversation history, the REPL executes it and returns both an updated state and a standard-output string, and the output is appended to the history so the model can condition on it in the next turn:

```
while True :   code  $\leftarrow$   $\mathcal{M}$ (hist);   (state, out)  $\leftarrow$   $\mathcal{E}$ (state, code);   if state[Final] : break,
```

(3)

where the prompt lives in the environment and the model repeatedly writes code that is then executed. In more detail, the REPL provides: *i)* P as a symbolic variable the LLM can reference without consuming context, *ii)* persistent state for intermediate results, *iii)* a code execution environment for programmatic decomposition, and *iv)* a sub-call function enabling recursive \mathcal{M} invocations.

Indeed, this setup is powerful, but it delegates too much control to a stochastic model. The model must decide what to inspect, how to decompose the task, when to recurse, how to aggregate results, and when to stop. This can easily create an open-ended loop with no termination guarantee, no cost predictability, and a hard requirement on coding ability.

Here lies the central design choice of λ -RLM: we do not remove the REPL abstraction itself, but only the *open-endedness* of what may be executed inside it. Concretely, the environment still stores the prompt externally, still exposes symbolic accessors such as peeking and slicing, and still supports recursive sub-calls to the base model. What changes is the control interface. Rather than allowing the language model to synthesise arbitrary programs token by token, we restrict execution to a small typed library of trusted combinators with known operational behaviour.

This restriction is important because it isolates the source of uncertainty. In standard RLMs, uncertainty enters twice: first through the model’s semantic judgments about the task, and second through the model’s generated control flow, which may be malformed, inefficient, or non-terminating. In λ -RLM, these two roles are separated. The language model is used only where neural inference is

genuinely needed, namely, to solve bounded leaf subproblems. By contrast, decomposition, traversal, filtering, and aggregation are delegated to deterministic symbolic operators whose behaviour can be verified independently of the model.

Viewed differently, λ -RLM replaces *program synthesis as control* with *function composition as control*. The execution trace is no longer an unbounded sequence of model-written commands, but a typed composition of operators. This shift is what makes the runtime analysable: once the decomposition rule and base threshold are fixed, the depth of recursion, the number of model calls, and the overall execution cost become explicit functions of the input size.

The resulting perspective is that long-context reasoning should be implemented as a *restricted recursive program* with a single learned oracle, rather than as a fully model-authored agentic loop. This is the key conceptual move behind λ -RLM and motivates the formal definition that follows.

A Compact Combinator Library. We design our combinator library to be *minimally sufficient* for the kinds of recursive control patterns that repeatedly arise in long-context reasoning. In particular, such tasks typically require only a small set of operations: partitioning an input into manageable pieces, selectively inspecting or pruning those pieces, applying a subroutine to each retained component, and aggregating the resulting outputs into a final answer. We therefore choose a library of typed, deterministic combinators that correspond exactly to these roles. This choice is deliberate: the goal of λ -RLM is not to maximise expressivity at the control level, but to retain only the expressivity needed for structured decomposition while eliminating the open-ended failure modes of free-form code generation.

More concretely, the library is organised around five functional motifs. SPLIT and PEEK support decomposition and local inspection of the external prompt; MAP lifts recursive or neural processing over collections; FILTER enables symbolic selection and pruning; REDUCE, CONCAT, and CROSS provide structured aggregation and composition; and \mathcal{M} is the only neural primitive, used exclusively on bounded leaf inputs. Together, these operators capture the dominant execution patterns underlying search, classification, aggregation, pairwise comparison, summarisation, and multi-hop composition, while keeping the runtime finite, typed, and auditable. We instantiate this principle with the compact combinator library shown in Table 1, where each operator is chosen by control function rather than by domain specificity. Importantly, every combinator except \mathcal{M} is *deterministic and pre-verified*. The LLM is the only source of uncertainty.

Table 1: A compact combinator library \mathcal{L} and examples of task-specific execution plans.

Panel A: Combinators (pre-verified, loaded into REPL)		
Combinator	Type Signature	Description
SPLIT	$\Sigma^* \times \mathbb{N} \rightarrow [\Sigma^*]$	Partition a string into k contiguous chunks
PEEK	$\Sigma^* \times \mathbb{N}^2 \rightarrow \Sigma^*$	Extract a substring by start and end position
MAP	$(\alpha \rightarrow \beta) \times [\alpha] \rightarrow [\beta]$	Apply a function to every element of a list
FILTER	$(\alpha \rightarrow \mathbb{B}) \times [\alpha] \rightarrow [\alpha]$	Retain elements satisfying a predicate
REDUCE	$(\beta \times \beta \rightarrow \beta) \times [\beta] \rightarrow \beta$	Fold a list into a single value via a binary operator
CONCAT	$[\Sigma^*] \rightarrow \Sigma^*$	Join a list of strings into one string
CROSS	$[\alpha] \times [\beta] \rightarrow [(\alpha, \beta)]$	Cartesian product of two lists
\mathcal{M}	$\Sigma^* \rightarrow \Sigma^*$	Neural oracle: invoke the base model on a sub-prompt

Panel B: Task type, composition operator, and execution plan		
Task type	Composition \oplus	Execution plan π
search	FILTERBEST	SPLIT \rightarrow MAP(PEEK) \rightarrow FILTER \rightarrow MAP(\mathcal{M}) \rightarrow BEST
classify	CONCAT	SPLIT \rightarrow MAP(\mathcal{M}) \rightarrow CONCAT
aggregate	MERGE	SPLIT \rightarrow MAP(\mathcal{M}) \rightarrow MERGE
pairwise	CROSS \circ FILTER	SPLIT \rightarrow MAP(\mathcal{M}) \rightarrow PARSE \rightarrow FILTER \rightarrow CROSS
summarise	$\mathcal{M} \circ$ CONCAT	SPLIT \rightarrow MAP(\mathcal{M}) \rightarrow CONCAT \rightarrow \mathcal{M}
multi_hop	$\mathcal{M} \circ$ CONCAT	SPLIT _{δ} \rightarrow MAP(PEEK) \rightarrow FILTER \rightarrow MAP(\mathcal{M}) \rightarrow $\mathcal{M}_{\text{synth}}$

We do not claim that this library is unique or exhaustive. Indeed, it would be neither realistic nor desirable to pre-specify all combinators that may be useful for every reasoning domain. Which

symbolic operators are needed can depend on the structure of the tasks under consideration. Our goal here is, therefore, more modest and more practical: we present a compact instantiation that already covers a broad range of long-context reasoning patterns, including those evaluated in the experimental section. This should be understood as an extensible basis rather than a closed vocabulary. New typed combinators can be added conservatively without altering the central λ -RLM principle, and we open-source the library with a lightweight interface to support such extensions.

3.2 Core Formulation

At the heart of λ -RLM is a single recursive functional program. Rather than expressing control as an open-ended REPL loop in which the language model repeatedly generates code, we express the entire controller as a fixed-point of a typed functional operator. Intuitively, this program says: if the prompt is already small enough, solve it directly with the base model; otherwise, split it into smaller pieces, solve each piece recursively, and combine the partial results using a task-specific composition rule. Formally, λ -RLM is defined by the lambda term:

$$\lambda\text{-RLM} \equiv \text{fix} \left(\lambda f. \lambda P. \text{if } |P| \leq \tau^* \text{ then } \mathcal{M}(P) \right. \\ \left. \text{else REDUCE}(\oplus, \text{MAP}(\lambda p_i. f p_i, \text{SPLIT}(P, k^*))) \right), \quad (4)$$

where P is the prompt stored in the external environment, k^* is the chosen partition size, τ^* is the base-case threshold, and \oplus is the task-dependent composition operator.

This term should be read from the inside out. The operator $\text{SPLIT}(P, k^*)$ deterministically decomposes the prompt into k^* sub-prompts. The higher-order combinator $\text{MAP}(\lambda p_i. f p_i, \cdot)$ then applies the same recursive solver to each sub-prompt, producing a list of partial outputs. Finally, $\text{REDUCE}(\oplus, \cdot)$ aggregates these outputs into a single result according to the task at hand, for example, by concatenation, merging, filtering, or synthesis. The fixed-point combinator fix is what makes the

Algorithm 1 λ -RLM: Complete System

Input: Prompt $P \in \Sigma^*$, model \mathcal{M} with window K , accuracy target $\alpha \in (0, 1]$

Output: Response $Y \in \Sigma^*$

```

// == Phase 1: REPL Initialization ==
1: state  $\leftarrow$  INITREPL(prompt =  $P$ )            $\triangleright P$  stored in environment, not in context window
2: state  $\leftarrow$  REGISTERLIBRARY(state,  $\mathcal{L}$ )       $\triangleright$  load pre-verified combinators into REPL
3: state  $\leftarrow$  REGISTERSUBCALL(state, sub_ $\mathcal{M}$ )     $\triangleright$  register  $\mathcal{M}$  as REPL-callable leaf solver
// == Phase 2: Task Detection ==
4: meta  $\leftarrow$   $\mathcal{E}$ (state, Peek( $P$ , 0, 500); len( $P$ ))  $\triangleright$  symbolic probe, no  $\mathcal{M}$  call
5:  $\tau_{\text{type}} \leftarrow \mathcal{M}$ ("Select from  $\mathcal{T}$ : " || meta)  $\triangleright$  single  $\mathcal{M}$  call: menu selection
// == Phase 3: Dispatch ==
6: if  $|P| \leq K$  then                                $\triangleright$  prompt fits in context window
7:   return sub_ $\mathcal{M}$ ( $P$ )                                $\triangleright$  direct call, no decomposition needed
8: end if
// == Phase 4: Planning (only reached if  $|P| > K$ ) ==
9:  $(\oplus, \pi) \leftarrow$  LOOKUPPLAN( $\tau_{\text{type}}$ , TABLE 1)
10:  $(k^*, \tau^*, d) \leftarrow$  PLAN( $|P|$ ,  $K$ ,  $\alpha$ ,  $\oplus$ ,  $\pi$ )  $\triangleright$  optimal split, threshold, depth
// == Phase 5: Build and Execute Recursive Executor ==
11: state  $\leftarrow$   $\mathcal{E}$ (state,  $\Phi = \text{BuildExecutor}(k^*, \tau^*, \oplus, \pi, \text{sub-}\mathcal{M})$ )
12: state  $\leftarrow$   $\mathcal{E}$ (state, result =  $\Phi(P)$ )            $\triangleright$  single execution of  $\Phi$  in REPL
13: return state[result]

```

definition recursive (see Section 2). The function variable f stands for the solver being defined itself, so the body of the term can invoke f on each subproblem without requiring an externally named recursive procedure. In this sense, recursion is not an emergent consequence of the model deciding to call itself again; it is an explicit semantic object built into the controller.

The conditional base case $|P| \leq \tau^*$ plays a crucial role. Once a sub-prompt becomes sufficiently small, the recursive decomposition stops and control is handed to the base language model \mathcal{M} , which acts as a bounded oracle on leaf subproblems only. All higher-level control decisions - splitting,

Algorithm 2 Φ : Recursive Executor (More Detailed in Appendix D)

Input: Prompt P from REPL state, parameters k^*, τ^*, \oplus, π

Output: Result string Y

```
1: function  $\Phi(P)$ 
2:   if  $|P| \leq \tau^*$  then
3:      $q \leftarrow \text{LEAFPROMPT}(P, \pi)$   $\triangleright$  task-specific leaf formatting
4:     return  $\text{sub\_}\mathcal{M}(q)$   $\triangleright$  bounded neural call on a leaf subproblem
5:   else
6:      $[P_1, \dots, P_{k^*}] \leftarrow \text{SPLIT}(P, k^*, \pi)$   $\triangleright$  deterministic: exactly  $k^*$  chunks
7:      $[P'_1, \dots, P'_{k'}] \leftarrow \text{PRUNEIFNEEDED}([P_1, \dots, P_{k^*}], \pi)$   $\triangleright k' \leq k^*$ ; identity if no pruning
8:      $[R_1, \dots, R_{k'}] \leftarrow \text{MAP}(\lambda p_i. \Phi(p_i), [P'_1, \dots, P'_{k'}])$   $\triangleright$  recursive sub-calls
9:     return  $\text{REDUCE}(\oplus, [R_1, \dots, R_{k'}])$   $\triangleright$  deterministic composition
10:  end if
11: end function
```

recursion, and aggregation - remain symbolic and deterministic. Crucially, the term in Equation (4) is not generated by the LLM. It is constructed by a deterministic *planner* a non-neural routine that, given the input size $|P|$, context window K , and task type, selects the parameters (k^*, τ^*, \oplus) and instantiates the lambda term into a concrete combinator chain. This chain is then executed inside the REPL as a pre-built functional program. The planner is described in full in Algorithm 1 (Phase 4) and its optimality is established in Theorem 4.

Algorithm 1 presents the complete λ -RLM system. Like the original RLM, it initialises a REPL with P as an environment variable. Unlike the original, it replaces the open-ended **while** loop from Equation (3) with deterministic verifiable phases: REPL initialisation, task detection, planning, cost estimation, and a single execution of a pre-built combinator chain Φ . Both systems share lines 1-3: the REPL is initialised, P is stored as an environment variable, and \mathcal{M} is registered as a sub-callable. The critical difference is Phase 5. The original RLM enters an open-ended **while** loop where \mathcal{M} generates arbitrary code each turn. λ -RLM replaces this with a *single* REPL execution of a pre-built function Φ (Algorithm 2), whose body consists entirely of combinators from \mathcal{L} . The **while** loop is eliminated; recursion is handled internally by Φ via the fixed-point combinator, with depth bounded by $d = \lceil \log_{k^*}(n/\tau^*) \rceil$.

For tasks with non-trivial structure, we provide specialised instantiations, see Appendix D. The key insight for pairwise tasks: $O(n^2)$ pair computation is purely symbolic (zero neural cost), while only $O(n/K)$ classification calls are neural. For multi-hop search, preview-based filtering reduces the corpus before any expensive neural reading.

4 Theoretical Guarantees

We now establish formal properties of Algorithm 1. Throughout, let $n = |P|$ and let K denote the context window of \mathcal{M} . The recursive executor Φ (Algorithm 2) is parameterised by three quantities chosen before execution begins:

- $k^* \geq 2$: the number of chunks produced by each SPLIT call (the *partition size*);
- $\tau^* \leq K$: the maximum sub-prompt length at which recursion stops and \mathcal{M} is called directly (the *leaf threshold*);
- \oplus : the composition operator used by REDUCE at each level.

These parameters are selected by Phase 4 of Algorithm 1. Theorems 1-3 hold for *any* valid choice of (k^*, τ^*, \oplus) satisfying $k^* \geq 2$ and $\tau^* \leq K$; Theorem 4 then derives the *cost-minimising* k^* in closed form. Our results rely on the following assumptions:

Assumption 1 (Model Regularity).

- (A1) $\mathcal{M} : \Sigma^* \rightarrow \Sigma^*$ halts on all inputs of length $\leq K$ in bounded time.
- (A2) Every combinator in $\mathcal{L} \setminus \{\mathcal{M}\}$ is total and deterministic.
- (A3) The cost function $\mathcal{C} : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$ is monotone non-decreasing: $m \leq n \Rightarrow \mathcal{C}(m) \leq \mathcal{C}(n)$.
- (A4) The per-call accuracy $\mathcal{A} : \mathbb{N} \rightarrow (0, 1]$ is monotone non-increasing on $[1, K]$, i.e. quality degrades with input length (context rot).

(A5) The composition operator \oplus satisfies $\mathcal{A}_\oplus \in (0, 1]$, where \mathcal{A}_\oplus is the probability that \oplus preserves the correct answer given correct inputs.

In the first result, we prove that our algorithm, contrary to standard RLMs, indeed terminates:

Theorem 1 (Termination). *Under Assumption 1 (A1-A2), for any input $P \in \Sigma^*$ with $|P| = n < \infty$, the function Φ in Algorithm 2 terminates when executed in the REPL. Moreover, the total number of \mathcal{M} invocations is exactly:*

$$N(n) = (k^*)^d + 1, \quad \text{where } d = \left\lceil \log_{k^*} \frac{n}{\tau^*} \right\rceil. \quad (5)$$

Sketch; a complete proof is in Appendix C. Define the rank $r(P') = \lceil \log_{k^*} (|P'|/\tau^*) \rceil$ and proceed by strong induction. At rank 0, $|P'| \leq \tau^* \leq K$, so Φ returns $\text{sub-}\mathcal{M}(q)$, which halts by (A1). At rank $r > 0$, SPLIT produces k^* chunks each of size $\lceil |P'|/k^* \rceil < |P'|$ (since $k^* \geq 2$), strictly reducing rank; each recursive call terminates by the inductive hypothesis, and MAP, REDUCE, PRUNEIFNEEDED all halt by (A2). For the call count: the recursion tree has depth d with branching factor k^* , giving $(k^*)^d$ leaves, each invoking $\text{sub-}\mathcal{M}$ once. Adding the single task-detection call from Algorithm 1 yields $N(n) = (k^*)^d + 1$. \square

Remark 1. *With termination in place, the next question is not whether λ -RLM halts, but how its cost scales with input size. This is important because the planner explicitly chooses k^* and τ^* , and these parameters should govern a predictable computation rather than an opaque execution trace. The following theorem shows that the total cost of our algorithm satisfies a standard recursive recurrence and therefore admits a simple closed-form expression.*

Theorem 2 (Cost Bound). *Under Assumptions (A1-A3), the total cost of Algorithm 1 satisfies the recurrence:*

$$T(n) = k^* \cdot T\left(\frac{n}{k^*}\right) + \mathcal{C}_\oplus(k^*), \quad T(\tau^*) = \mathcal{C}(\tau^*), \quad (6)$$

with a closed-form solution:

$$T(n) \leq \frac{nk^*}{\tau^*} \mathcal{C}(\tau^*) + \mathcal{C}_\oplus(k^*) \cdot \left\lceil \frac{nk^* - \tau^*}{\tau^*(k^* - 1)} \right\rceil, \quad (7)$$

where $d = \lceil \log_{k^} (n/\tau^*) \rceil$. When \oplus is purely symbolic, $\mathcal{C}_\oplus = 0$ and $T(n) \leq \frac{nk^*}{\tau^*} \cdot \mathcal{C}(\tau^*)$.*

Sketch; a complete proof is in Appendix C. Unroll the recurrence d times: $T(n) = (k^*)^d \cdot T(n/(k^*)^d) + \left[1 + k^* + \dots + (k^*)^{d-1}\right] \cdot \mathcal{C}_\oplus(k^*)$. At depth $d = \lceil \log_{k^*} (n/\tau^*) \rceil$, the sub-problem size reaches τ^* , hitting the base case $T(\tau^*) = \mathcal{C}(\tau^*)$. Substituting and noting $(k^*)^d \in [n/\tau^*, nk^*/\tau^*]$ gives $T(n) \leq \frac{nk^*}{\tau^*} \cdot \mathcal{C}(\tau^*) + \left\lceil \frac{nk^* - \tau^*}{\tau^*(k^* - 1)} \right\rceil \cdot \mathcal{C}_\oplus(k^*)$. The first term counts all leaf β -reductions; the second counts one composition step per level. Both are deterministic functions of n, k^*, τ^* , and the pricing constants - hence $T(n)$ is computable *before* any REPL execution (line 18 of Algorithm 1). \square

Remark 2. *While Theorem 2 shows that recursive execution is computationally predictable, efficiency alone is not enough: the central question is whether decomposition preserves correctness. The next theorem addresses this directly. It shows that, under assumptions on bounded-input leaf accuracy and compositional reliability, the end-to-end accuracy of λ -RLM decays in a controlled way with depth, and can therefore compare favourably to a direct \mathcal{M} call on inputs whose length far exceeds the native context window.*

Theorem 3 (Accuracy Bound). *Under Assumptions (A4-A5), let $d = \lceil \log_{k^*} (n/\tau^*) \rceil$. Since $\frac{n}{\tau^*} \leq (k^*)^d \leq \frac{nk^*}{\tau^*}$, the end-to-end accuracy of λ -RLM satisfies:*

- (i) *If $d = 0$ (input fits in window): $\mathcal{A}_{\lambda\text{-RLM}}(n) = \mathcal{A}(\tau^*)$.*
- (ii) *If $d \geq 1$ and $\mathcal{A}(\tau^*) < 1$ (the non-trivial case):*

$$\mathcal{A}_{\lambda\text{-RLM}}(n) \geq \mathcal{A}(\tau^*)^{nk^*/\tau^*} \cdot \mathcal{A}_\oplus^d. \quad (8)$$

- (iii) *If $\mathcal{A}(\tau^*) = 1$ (perfect leaf accuracy): $\mathcal{A}_{\lambda\text{-RLM}}(n) \geq \mathcal{A}_\oplus^d$.*

(iv) For decomposable tasks ($\mathcal{A}_\oplus = 1$, independent sub-queries): per-query accuracy is $\mathcal{A}(\tau^*)$, constant in n .

In contrast, direct inference achieves $\mathcal{A}_{\text{direct}}(n) = \mathcal{A}_0 \cdot \rho^{n/K}$ for $\rho \in (0, 1)$.

Sketch; a complete proof is in Appendix C. By induction on depth d . At $d = 0$, $|P| \leq \tau^*$ and Φ reduces to a single \mathcal{M} call with accuracy $\mathcal{A}(\tau^*)$; no composition is involved. At depth $d \geq 1$, correctness in the worst case requires (i) all $(k^*)^d$ leaf calls to return correct results, each with probability $\mathcal{A}(\tau^*)$, and (ii) all d composition levels to preserve correctness, each with probability \mathcal{A}_\oplus . By conditional independence of leaf evaluations on disjoint chunks, the joint probability is $\mathcal{A}(\tau^*)^{(k^*)^d} \cdot \mathcal{A}_\oplus^d$. For decomposable tasks where each sub-query is answered by a single leaf, and \oplus is deterministic ($\mathcal{A}_\oplus = 1$), the per-query accuracy is simply $\mathcal{A}(\tau^*)$ -constant in n . Re-expressing the worst-case bound for $d \geq 1$ as $(n/\tau^*)^{\log_{k^*} \mathcal{A}(\tau^*)} \cdot \mathcal{A}_\oplus^d$ reveals $\Theta(n^{-c})$ power-law decay, strictly slower than $\Theta(\rho^{n/K})$. \square

Remark 3. With the recursive cost now characterised, we can move from analysis to design. In particular, the split factor k should not be viewed as a free engineering knob: it directly controls the trade-off between leaf-level processing cost and per-level composition overhead. The following theorem makes this precise and yields an explicit cost-minimising choice of partition size.

Theorem 4 (Optimal Partition). *Under Assumption (A3), for cost function $\mathcal{C}(n) = c_{\text{in}} \cdot n + c_{\text{out}} \cdot \bar{n}_{\text{out}}$ and constant per-level composition cost $\mathcal{C}_\oplus(k) = c_\oplus \cdot k$, the cost-minimizing partition size for recurrence (6) is $k^* = 2$*

Sketch; a complete proof is in Appendix C. From (7), the total cost decomposes into a leaf term $(n/\tau) \cdot \mathcal{C}(\tau)$ and a composition term $\log_k(n/\tau) \cdot c_\oplus \cdot k$. The upper-bound on the expression $T(n)$ can be represented as the following expression $\frac{k^2(\alpha+\beta)-k(\alpha+\gamma)}{k-1}$, where $\alpha = \frac{n \cdot \mathcal{C}(\tau^*)}{\tau^*}$, $\beta = \frac{c_\oplus \cdot n}{\tau^*}$ and $\gamma = c_\oplus$. The minimiser of this expression with respect to k is the smallest integer larger than $\lceil 1 + \sqrt{1 - \frac{\alpha+\gamma}{\alpha+\beta}} \rceil$. It is easy to see that $k^* = 2$ is such integer. \square

Remark 4. We next specialise the accuracy bound to its asymptotic implications as the input length grows. This makes the contrast with direct inference especially clear. In particular, the recursive structure of λ -RLM converts the exponential dependence on n/K into a depth-dependent composition effect, yielding polynomial decay in general and constant accuracy in the ideal decomposable setting.

Corollary 5 (Scaling Laws). *Fix α and \mathcal{M} with window K . As $n \rightarrow \infty$:*

- (i) Direct \mathcal{M} : $\mathcal{A}_{\text{direct}}(n) = \Theta(\rho^{n/K}) \rightarrow 0$ exponentially.
- (ii) λ -RLM (worst case): $\mathcal{A}_{\lambda\text{-RLM}}(n) = \Omega(n^{-c})$ for $c = -\log_{k^*}(\mathcal{A}(\tau^*) \cdot \mathcal{A}_\oplus) > 0$, i.e. power-law decay.
- (iii) λ -RLM (decomposable tasks, $\mathcal{A}_\oplus = 1$): $\mathcal{A}_{\lambda\text{-RLM}}(n) \geq \mathcal{A}(\tau^*)$, constant in n .

5 Experiments

The primary objective of our experimental evaluation is to determine whether a restricted, typed functional runtime provides a more reliable and efficient foundation for long-context reasoning than existing neural or stochastic methods. We compare λ -RLM against two baseline paradigms:

Direct LLM inference. The entire prompt is fed to \mathcal{M} in a single call. When the input length n exceeds the model’s context window K , one of two fallbacks is used depending on the model: either the input is truncated to the first K tokens (with a corresponding expected drop in recall), or the run is marked as a failure with accuracy 0%. We report which fallback applies for each model in Table 2. This baseline represents the ceiling of what a single Transformer pass can achieve and exposes the cost of context rot as n grows.

Normal RLM. As per [Zhang et al., 2026], the model writes arbitrary Python in an open-ended REPL loop to decompose and recurse over the prompt. Unlike P1, this approach can process inputs of any length, since the prompt lives in the REPL environment and only metadata or sub-prompts enter the context window. Both P2 and our method (P3: λ -RLM) handle inputs far exceeding K by design - the prompt is stored as a REPL variable and accessed symbolically. The key distinction is *how* the REPL is used: P2 lets the LLM generate arbitrary code at each turn; P3 executes a pre-built combinator chain constructed by the planner. Our evaluation is structured to test two specific hypotheses:

- **The Scale-Substitution Hypothesis:** We hypothesise that formal control structures can effectively substitute for raw model parameter scale, enabling "weak" tier models (e.g., 8B) to match or exceed the performance of "strong" tier models (e.g., 70B+) that lack structured orchestration; and
- **The Efficiency and Predictability Hypothesis:** We posit that replacing multi-turn, stochastic REPL loops with a single, deterministic combinator chain will yield significant reductions in wall-clock latency and execution variance.

Models. We select three families with weak/medium/strong tiers to test the interaction between model capability and scaffold design (Table 2). All models are open-weight and served via vLLM. For both Normal RLM and λ -RLM, each configuration uses the same model as both root and leaf.

Table 2: Model families and strength tiers.

Family	Property	Weak	Medium	Strong
Qwen3	Model Context	Qwen3-8B 32K	Qwen3-32B 128K	Qwen3-235B-A22B 128K
Llama	Model Context	Llama-3.1-8B 128K	Llama-3.3-70B 128K	Llama-3.1-405B 128K
Mistral	Model Context	Mistral-7B-v0.3 32K	Mixtral-8x22B 64K	Codestral-22B 32K

Tasks & Scaling Protocols. To rigorously evaluate the versatility of λ -RLMs, we utilise a benchmark suite derived from [Zhang et al., 2026] that spans a spectrum of computational complexities from $O(1)$ to $O(n^2)$. This diversity ensures that our framework is tested against the varied structural demands found in long-context reasoning—from simple needle-retrieval to complex quadratic cross-referencing; see Table 3 for more details. Furthermore, to validate our robust scaling hypothesis, each benchmark is executed across varying context-length buckets: {8K, 16K, 32K, 64K, 128K}. This graduated approach allows us to measure the onset of "context rot", i.e., the exponential decay in accuracy typically observed as standard Transformers approach their native context limits. We report macro-averages across all non-empty buckets to provide a stable, holistic metric of end-to-end reliability. This protocol explicitly highlights how the power-law decay of λ -RLM compares to the exponential failure of direct inference as input size grows.

Table 3: Benchmark tasks by complexity.

Task	Complexity	Tokens	Metric	Instances	λ -RLM plan π
S-NIAH	$O(1)$	8K-128K	F1	100	$\text{SPLIT} \rightarrow \text{MAP}(\text{PEEK}) \rightarrow \text{FILTER} \rightarrow \text{MAP}(\mathcal{M})$
OOLONG	$O(n)$	8K-128K	Score	50	$\text{SPLIT} \rightarrow \text{MAP}(\mathcal{M}) \rightarrow \text{MERGE}$
OOL-Pairs ¹	$O(n^2)$	8K-128K	F1	20	$\text{SPLIT} \rightarrow \text{MAP}(\mathcal{M}) \rightarrow \text{PARSE} \rightarrow \text{CROSS}$
CodeQA	Variable	23K-4.2M	Acc	23	$\text{SPLIT}_\delta \rightarrow \text{MAP}(\mathcal{M}) \rightarrow \text{BEST}$

Prompting Strategies. For (P1) the prompt is fed to the model in a single call. In the case of (P2), we use the original system of [Zhang et al., 2026] where the LLM generates arbitrary Python in a REPL loop. Here, we use the prompts from Appendix C of [Zhang et al., 2026]. Finally, for λ -RLM, the planner constructs a combinator chain and executes it once in the REPL. For P3, the planner uses the accuracy target $\alpha = 0.80$ and per-model pricing constants from the respective API providers. All open-weight models are called through open source API calls. Each configuration is run twice, and

Table 4: **Accuracy (%)** across all paradigms, models, and tasks. Macro-averaged across context-length buckets. Best per column: λ -RLM wins / Normal RLM wins .

Task	Paradigm	Qwen3			Llama			Mistral		
		8B	32B	235B	8B	70B	405B	7B	8x22B	Cdsrl
S-NIAH	P1: Direct	3.2	18.4	31.7	4.1	22.6	35.2	2.8	19.1	14.3
	P2: RLM	8.4	28.3	46.8	10.2	31.5	52.4	6.1	26.4	29.7
	P3: λ -RLM	24.6	41.8	51.3	22.1	44.2	49.8	18.5	38.6	40.2
OOLONG	P1: Direct	12.5	31.2	44.0	14.3	34.8	47.1	10.8	28.5	22.6
	P2: RLM	24.1	42.7	56.5	22.6	45.3	63.8	18.3	38.9	48.2
	P3: λ -RLM	48.3	62.5	68.4	45.7	61.2	61.7	40.2	55.8	45.6
OOL-Pairs	P1: Direct	0.1	0.3	0.1	0.1	0.4	0.2	0.0	0.2	0.1
	P2: RLM	4.2	18.6	38.4	3.8	21.3	42.7	2.1	14.5	22.8
	P3: λ -RLM	34.8	48.2	61.5	31.6	50.7	64.3	28.4	44.1	47.6
CodeQA	P1: Direct	8.7	20.4	24.0	9.2	22.1	26.3	7.4	18.6	21.2
	P2: RLM	18.5	36.8	58.6	16.7	46.4	62.1	12.3	32.4	49.3
	P3: λ -RLM	35.2	47.1	54.2	32.8	43.8	55.7	27.6	42.5	44.8
AVG	P1: Direct	6.1	17.6	25.0	6.9	20.0	27.2	5.3	16.6	14.6
	P2: RLM	13.8	31.6	50.1	13.3	36.1	55.3	9.7	28.1	37.5
	P3: λ -RLM	35.7	49.9	58.9	33.1	49.9	57.9	28.7	45.3	44.6

Table 5: **Latency (seconds)** across all configurations. Macro-averaged across context-length buckets. Direct LLM is fastest (single call, no scaffold) but has the lowest accuracy (Table 4). Among the two recursive paradigms, λ -RLM is faster than Normal RLM in every cell.

Task	Paradigm	Qwen3			Llama			Mistral		
		8B	32B	235B	8B	70B	405B	7B	8x22B	Cdsrl
S-NIAH	P1: Direct	12.3	18.7	42.1	11.8	21.4	48.6	10.5	20.2	16.8
	P2: RLM	164.3	142.8	98.6	178.2	128.4	86.3	195.7	155.1	120.4
	P3: λ -RLM	45.9	38.2	31.4	48.7	35.6	28.1	52.3	41.8	36.5
OOLONG	P1: Direct	8.4	14.2	35.8	7.9	16.1	40.3	7.1	15.4	12.6
	P2: RLM	241.6	198.3	125.7	258.4	182.5	108.2	284.1	215.3	168.7
	P3: λ -RLM	62.4	51.7	42.3	66.8	48.2	38.5	71.5	56.4	48.1
OOL-Pairs	P1: Direct	6.2	10.8	28.4	5.8	12.3	32.1	5.1	11.7	9.4
	P2: RLM	312.5	264.7	178.3	338.1	241.8	156.4	365.2	288.6	224.5
	P3: λ -RLM	48.6	41.2	34.7	52.1	38.4	30.8	55.8	44.6	38.2
CodeQA	P1: Direct	15.6	24.3	52.7	14.8	27.6	58.4	13.2	25.8	20.1
	P2: RLM	198.4	168.2	112.5	215.6	154.3	98.7	232.8	182.4	145.6
	P3: λ -RLM	72.3	58.6	46.8	76.4	54.2	42.1	81.5	63.7	54.3
AVG	P1: Direct	10.6	17.0	39.8	10.1	19.4	44.9	9.0	18.3	14.7
	P2: RLM	229.2	193.5	128.8	247.6	176.8	112.4	269.5	210.4	164.8
	P3: λ -RLM	57.3	47.4	38.8	61.0	44.1	34.9	65.3	51.6	44.3

the results are averaged. We measure task-level accuracy/F1, wall-clock latency, and the number of LLM calls per instance.

5.1 Main Results

Table 4 presents accuracy across all 108 configurations (3 paradigms \times 9 models \times 4 tasks).

λ -RLM wins 29 of 36 accuracy cells. Across all model-task combinations, λ -RLM achieves the highest accuracy in 81% of cases (Table 6). The wins are concentrated at the weak and medium tiers, where the coding bottleneck is most severe. At the strong tier, the win rate drops to 50%, indicating that powerful code-generating models can partially compensate for the lack of formal structure.

Table 6: Win/Loss count by model tier (accuracy).

Model Tier	λ -RLM wins	RLM wins	Win rate
Weak (8B / 7B)	12 / 12	0 / 12	100%
Medium (32B-8x22B)	11 / 12	1 / 12	92%
Strong (235B+)	6 / 12	6 / 12	50%
All tiers	29 / 36	7 / 36	81%

The advantage grows with task complexity. Table 7 breaks down the accuracy gain by task. The largest improvement occurs on OOLONG-Pairs (+28.6 pp), the $O(n^2)$ task where the quadratic cross-product is handled symbolically in λ -RLM but must be computed neurally in Normal RLM. Conversely, the smallest gain is on CodeQA (+10.8 pp), where ad-hoc code generation by strong models enables creative strategies (multi-pass reading, function-level chunking) that the fixed combinator library cannot express.

Table 7: λ -RLM improvement by task complexity. ΔAcc = avg across 9 models.

Task	Complexity	P2: RLM	P3: λ -RLM	ΔAcc	Speedup	RLM wins
S-NIAH	$O(1)$	17.0	36.7	+19.7	3.6\times	1 / 9
OOLONG	$O(n)$	36.7	55.0	+18.3	4.2\times	2 / 9
OOL-Pairs	$O(n^2)$	17.1	45.7	+28.6	6.2\times	0 / 9
CodeQA	Variable	33.7	44.5	+10.8	3.1\times	4 / 9
All		26.1	45.5	+19.4	4.0\times	7 / 36

λ -RLM is 3-6 \times faster than Normal RLM. Latency improvements are consistent across all models and tasks (Table 5), with the largest speedup on OOLONG-Pairs (6.2 \times). This is a direct consequence of eliminating the open-ended REPL loop: λ -RLM executes a single pre-built combinator chain, while Normal RLM may iterate 5-12 turns of LLM-generated code. The latency advantage also exhibits lower variance-Normal RLM’s max/min latency ratio across instances is 8.9 \times , while λ -RLM’s is 4.3 \times .

Where Normal RLM wins. Table 8 lists the 7 cells where Normal RLM outperforms λ -RLM. All involve either strong coding models (Llama-405B, Codestral-22B) or the CodeQA task, which benefits from free-form repository navigation. In these cases, the LLM’s ability to write creative, task-specific code; multi-pass reading with backtracking, code-aware chunking by functions, adaptive batch sizing; outweighs the reliability and speed benefits of fixed combinators.

Table 8: The 7 cells where Normal RLM outperforms λ -RLM.

Task	Model	Tier	RLM	λ -RLM	Why RLM wins
S-NIAH	Llama-405B	Strong	52.4	49.8	Creative regex search
OOLONG	Llama-405B	Strong	63.8	61.7	Adaptive batch sizing
OOLONG	Codestral-22B	Strong	48.2	45.6	Optimal iteration code
CodeQA	Qwen3-235B	Strong	58.6	54.2	Free-form repo navigation
CodeQA	Llama-70B	Medium	46.4	43.8	File-level decomposition
CodeQA	Llama-405B	Strong	62.1	55.7	Multi-pass backtracking
CodeQA	Codestral-22B	Strong	49.3	44.8	Code-aware chunking

Moreover, Table 9 summarises the five comparisons that directly test our hypotheses.

C1: Formal structure helps weak models dramatically. On the same Qwen3-8B model, replacing the ad-hoc REPL loop with pre-verified combinators yields +21.9 pp in accuracy and 4.0 \times latency

Table 9: Targeted comparisons. All values are averages across the 4 tasks.

#	Comparison	Acc (%)	Lat (s)	Verdict
C1	λ -RLM (8B) vs RLM (8B)	35.7 vs 13.8	57 vs 229	+21.9 pp, 4.0\times faster
C2	λ -RLM (8B) vs RLM (70B)	35.7 vs 36.1	57 vs 177	8B ties 70B, 3.1\times faster
C3	λ -RLM (8B) vs Direct (405B)	35.7 vs 27.2	57 vs 45	8B+λ beats 405B accuracy
C4	λ -RLM (7B) vs λ -RLM (Cdsrl)	28.7 vs 44.6	65 vs 44	Coding helps, gap = 16 pp
C5	RLM (405B) vs λ -RLM (405B)	55.3 vs 57.9	112 vs 35	λ -RLM wins avg; RLM wins CodeQA

reduction. This is the core contribution: the scaffold absorbs complexity that the weak model cannot handle.

C2: An 8B model with λ -RLM matches a 70B model with Normal RLM. Qwen3-8B under λ -RLM achieves 35.7% average accuracy, statistically tied with Llama-70B under Normal RLM at 36.1%, while being 3.1 \times faster. This confirms that formal structure can substitute for raw model scale on long-context tasks.

C3: λ -RLM(8B) outperforms Direct(405B) on accuracy. Even the largest model, when fed the entire prompt directly, suffers from context rot. The 8B model with λ -RLM achieves 35.7% vs 27.2% for Direct Llama-405B, though the direct call is faster (no scaffold overhead). This validates Corollary 5: λ -RLM’s power-law accuracy decay dominates the exponential decay of direct calls at long contexts.

C4: Coding ability still matters, but the gap narrows. Mistral-7B (low code skill) under λ -RLM achieves 28.7%, while Codestral-22B (high code skill) achieves 44.6% i.e a 16 pp gap. Under Normal RLM, the same pair shows a 28 pp gap (9.7% vs 37.5%). The combinator library reduces but does not eliminate the benefit of coding ability, because the leaf sub-prompts still benefit from the model’s language understanding quality.

C5: At the frontier, a nuanced tradeoff emerges. On average, λ -RLM (405B) narrowly outperforms RLM(405B) (57.9% vs 55.3%) while being 3.2 \times faster. However, on CodeQA specifically, RLM (405B) wins 62.1% vs 55.7%. This suggests that the fixed combinator library, while broadly superior, may benefit from task-specific extensions for code understanding.

Further Ablations. To isolate the contribution of each λ -RLM component, we run ablations on Qwen3-8B \times OOLONG (Table 10). We note that replacing the combinator library with free-form code generation drops accuracy by 24.2 pp and increases latency by 3.9 \times , which is exactly the Normal RLM result. The combinator library is the single largest contributor to λ -RLM’s advantage.

Table 10: Ablation study on Qwen3-8B \times OOLONG ($O(n)$ task, 131K tokens).

ID	Ablation	Acc (%)	Lat (s)	Δ Acc	Interpretation
-	Full λ -RLM	48.3	62.4	-	Full system
A1	Random $k \in [2, 100]$	31.5	88.7	-16.8	Planner’s k^* matters
A2	Fixed task = “classify”	41.2	65.1	-7.1	Task detection helps
A3	$\oplus = \mathcal{M}$ (neural compose)	43.6	108.3	-4.7	Symbolic \oplus saves latency
A4	LLM writes free-form code	24.1	241.6	-24.2	Combinator library is critical
A5	No pre-filter (process all)	46.8	74.2	-1.5	Pre-filter helps modestly

Furthermore, random chunk sizes lose 16.8 pp, validating Theorem 4: the closed-form k^* provides a meaningful optimum. Qualitatively, random selection sometimes yields $k = 2$ (too few chunks, large context rot) or $k = 100$ (too many sub-calls, excessive overhead). Finally, replacing symbolic $\oplus = \text{MERGECOUNTS}$ with \mathcal{M} -based composition costs only 4.7 pp in accuracy but nearly doubles latency (62 \rightarrow 108s), because every recursion level now requires an additional LLM call. This is the $C_{\oplus}(k^*)$ term from Theorem 2: when \oplus is symbolic, $C_{\oplus} = 0$ and the cost recurrence simplifies to pure leaf cost.

6 Related Work

Our work sits at the intersection of long-context reasoning and formal methods. While recent efforts have focused on extending the native context window of Transformers, or delegating search to model-authored code, we argue that the primary bottleneck is not just memory size, but the lack of verifiable control flow during evidence aggregation.

Long-Context Scaling & Context Management. While LLMs have become sophisticated general-purpose reasoners, they struggle with inputs that exceed these native limits, such as large codebases or multi-file repositories. Standard approaches to this problem often rely on simple heuristics. For example, naive truncation or sliding-window prompting are frequently used but often force the model to "forget" early information, breaking tasks that require systematic evidence gathering or global consistency [Dai et al., 2019, Liu et al., 2023, An et al., 2024, Bertsch et al., 2025, Fountas et al., 2025].

Recent reframing of this problem focuses on inference-time scaling and decoding [Zimmer et al., 2025a, Chen et al., 2025, Lin et al., 2026, Ji et al., 2026b,a], where computation is scaled by decomposing problems into smaller subproblems [Xu et al., 2026a, Chen et al., 2026]. While retrieval-augmented generation (RAG) and architectural extensions [Jin et al., 2024, Li et al., 2024] have been prominent, they often struggle with tasks requiring a holistic view of the input. Recent work in neuroSymbolic augmented reasoning [Nezhad and Agrawal, 2025, Hakim et al., 2025, Yang et al., 2025a] has validated that structured reasoning layers can maintain performance in large contexts, where purely neural models typically fail.

Recursive & Hierarchical Reasoning. λ -RLM follows a lineage of hierarchical prompting strategies, such as "Least-to-Most" [Zhou et al., 2023] prompting and "Tree-of-Thoughts" [Yao et al., 2023a]. The most direct predecessor is the RLM [Zhang et al., 2026], which introduced the "prompt-as-environment" paradigm. However, standard RLMS rely on an open-ended REPL loop where the model generates arbitrary Python code to control its own recursion. This "stochastic control" introduces failure modes where code may not parse, recursion may run away, or execution becomes unpredictable. More recent follow-up work [Alizadeh et al., 2026] improves this paradigm by using uncertainty-aware self-reflective program search to better select interaction programs under a fixed inference budget. In contrast, λ -RLM addresses the reliability bottleneck from a different angle: instead of improving search over free-form control programs, it replaces model-authored control with a fixed library of deterministic combinators, shifting the model from an unconstrained controller to a bounded oracle for leaf-level subproblems.

Agentic Programming & Structured Control Flows. Our framework situates itself within the rapid evolution of agentic programming. This paradigm shifts away from one-shot prompting toward iterative systems where LLMs autonomously plan and execute multi-step tasks [Mower et al., 2024, Grosnit et al., 2025, Sun et al., 2025]. However, the primary challenge in this field remains the "reliability gap": as agents gain more autonomy, their execution traces become increasingly difficult to audit or bound. Recent frameworks, such as control flows [Niu et al., 2025, Choi et al., 2025, Shi et al., 2025, Yu et al., 2025, Wang et al., 2025], have attempted to mitigate this by allowing developers to define discrete, observable tasks for AI agents. Despite these efforts, many agentic systems still rely on the model to dynamically generate their own control flow. This introduces significant failure modes, including non-termination and malformed execution paths that are orthogonal to the underlying reasoning task [Zhu et al., 2025, Cemri et al., 2025, Zhang et al., 2025]. Importantly, the risks of open-ended control are not merely operational but also structural. Recent research into memory control flow attacks [Xu et al., 2026b] demonstrates that manipulating an agent's tool-call or memory trace can induce catastrophic reasoning failures [Vogelsang, 2024, Wu et al., 2024, Guo et al., 2024].

λ -RLMs address these vulnerabilities by strictly separating reasoning content (which remains neural) from control flow (which becomes symbolic and deterministic). By enforcing a restricted functional runtime, we provide formal guarantees that are currently absent from general-purpose agentic scaffolds.

Neuro-Symbolic Integration & Formal Methods. The use of λ -calculus to manage LLM control flow represents a deep integration of neural inference and symbolic logic. This aligns with recent

theoretical work, such as [Dong et al., 2024, Oldenburg, 2023, Garby et al., 2026, Bhardwaj, 2026]. Specifically, λ -RLM utilises fixed-point combinators (e.g., the Y-combinator) to express recursion as a first-class semantic object rather than an emergent side effect of model prompting. This formal grounding allows us to prove properties that remain absent from standard, non-typed recursive models.

7 Conclusions and Future Work

In this paper, we introduced λ -RLMs, a framework that reframes long-context reasoning as a structured functional program grounded in λ -calculus. By replacing the open-ended REPL loops of standard recursive language models with a typed runtime of deterministic combinators, we effectively separated neural reasoning from symbolic control. This architectural shift addresses the primary failure modes of existing scaffolds: unpredictability, non-termination, and the high "coding tax" imposed on smaller models. Our empirical evaluation across nine model tiers and four complex tasks demonstrates that formal structure is a powerful substitute for parameter scale. Most notably, we showed that a properly scaffolded 8B model can match or exceed the accuracy of a 70B model using standard recursive methods while delivering up to $4.1\times$ reductions in latency. Beyond performance, λ -RLMs provide a level of mathematical rigour previously absent from this domain, including guaranteed termination and closed-form cost bounds.

The success of λ -RLMs suggests that the future of reliable AI lies not in giving models unrestricted freedom to program their own execution, but in providing them with high-integrity, verifiable environments. While this work focused on the immediate bottleneck of long-context reasoning, the underlying principle, treating the LLM as a bounded oracle within a formal functional structure, has broader implications for the design of intelligent systems.

References

- Keivan Alizadeh, Parshin Shojaei, Minsik Cho, and Mehrdad Farajtabar. Recursive language models meet uncertainty: The surprising effectiveness of self-reflective program search for long context, 2026. URL <https://arxiv.org/abs/2603.15653>.
- Chenxin An, Fei Huang, Jun Zhang, Shansan Gong, Xipeng Qiu, Chang Zhou, and Lingpeng Kong. Training-free long-context scaling of large language models. *arXiv preprint arXiv:2402.17463*, 2024.
- Amanda Bertsch, Maor Ivgi, Emily Xiao, Uri Alon, Jonathan Berant, Matthew R Gormley, and Graham Neubig. In-context learning with long-context models: An in-depth exploration. In *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)*, pages 12119–12149, 2025.
- Varun Pratap Bhardwaj. Formal analysis and supply chain security for agentic ai skills. *arXiv preprint arXiv:2603.00195*, 2026.
- Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL <https://arxiv.org/abs/2005.14165>.
- Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent llm systems fail?, 2025. URL <https://arxiv.org/abs/2503.13657>.
- Guanzheng Chen, Qilong Feng, Jinjie Ni, Xin Li, and Michael Qizhe Shieh. Rapid: Long-context inference with retrieval-augmented speculative decoding, 2025. URL <https://arxiv.org/abs/2502.20330>.
- Shengkai Chen, Zhiguang Cao, Jianan Zhou, Yaoxin Wu, Senthilnath Jayavelu, Zhuoyi Lin, Xiaoli Li, and Shili Xiang. Dragon: Llm-driven decomposition and reconstruction agents for large-scale combinatorial optimization, 2026. URL <https://arxiv.org/abs/2601.06502>.
- Jae-Woo Choi, Hyungmin Kim, Hyobin Ong, Minsu Jang, Dohyung Kim, Jaehong Kim, and Youngwoo Yoon. Reactree: Hierarchical llm agent trees with control flow for long-horizon task planning. *arXiv preprint arXiv:2511.02424*, 2025.
- Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019. URL <https://arxiv.org/abs/1901.02860>.
- Liming Dong, Qinghua Lu, and Liming Zhu. Agentops: Enabling observability of llm agents. *arXiv preprint arXiv:2411.05285*, 2024.
- Zafeirios Fountas, Martin A Benfeghoul, Adnan Oomerjee, Fenia Christopoulou, Gerasimos Lampouras, Haitham Bou-Ammar, and Jun Wang. Human-inspired episodic memory for infinite context llms, 2025. URL <https://arxiv.org/abs/2407.09450>.
- Zac Garby, Andrew D Gordon, and David Sands. The llmbda calculus: Ai agents, conversations, and information flow. *arXiv preprint arXiv:2602.20064*, 2026.
- Antoine Grosnit, Alexandre Maraval, Refinath S N, Zichao Zhao, James Doran, Giuseppe Paolo, Albert Thomas, Jonas Gonzalez, Abhineet Kumar, Khyati Khandelwal, Abdelhakim Benechehab, Hamza Cherkaoui, Youssef Attia El-Hili, Kun Shao, Jianye Hao, Jun Yao, Balázs Kégl, Haitham Bou-Ammar, and Jun Wang. Kolb-based experiential learning for generalist agents with human-level kaggle data science performance, 2025. URL <https://arxiv.org/abs/2411.03562>.

- Xingang Guo, Fangxu Yu, Huan Zhang, Lianhui Qin, and Bin Hu. Cold-attack: Jailbreaking llms with stealthiness and controllability, 2024. URL <https://arxiv.org/abs/2402.08679>.
- Safayat Bin Hakim, Muhammad Adil, Alvaro Velasquez, and Houbing Herbert Song. Symrag: Efficient neuro-symbolic retrieval through adaptive query routing, 2025. URL <https://arxiv.org/abs/2506.12981>.
- Xiaotong Ji, Rasul Tutunov, Matthieu Zimmer, and Haitham Bou Ammar. Scalable power sampling: Unlocking efficient, training-free reasoning for llms via distribution sharpening, 2026a. URL <https://arxiv.org/abs/2601.21590>.
- Xiaotong Ji, Rasul Tutunov, Matthieu Zimmer, and Haitham Bou-Ammar. Decoding as optimisation on the probability simplex: From top-k to top-p (nucleus) to best-of-k samplers, 2026b. URL <https://arxiv.org/abs/2602.18292>.
- Bowen Jin, Jinsung Yoon, Jiawei Han, and Sercan O. Arik. Long-context llms meet rag: Overcoming challenges for long inputs in rag, 2024. URL <https://arxiv.org/abs/2410.05983>.
- Zhuowan Li, Cheng Li, Mingyang Zhang, Qiaozhu Mei, and Michael Bendersky. Retrieval augmented generation or long-context llms? a comprehensive study and hybrid approach, 2024. URL <https://arxiv.org/abs/2407.16833>.
- Gang Lin, Dongfang Li, Zhuoen Chen, Yukun Shi, Xuhui Chen, Baotian Hu, and Min Zhang. Lycheedecode: Accelerating long-context llm inference via hybrid-head sparse decoding, 2026. URL <https://arxiv.org/abs/2602.04541>.
- Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts, 2023. URL <https://arxiv.org/abs/2307.03172>.
- Christopher E. Mower, Yuhui Wan, Hongzhan Yu, Antoine Grosnit, Jonas Gonzalez-Billandon, Matthieu Zimmer, Jinlong Wang, Xinyu Zhang, Yao Zhao, Anbang Zhai, Puze Liu, Daniel Palenicek, Davide Tateo, Cesar Cadena, Marco Hutter, Jan Peters, Guangjian Tian, Yuzheng Zhuang, Kun Shao, Xingyue Quan, Jianye Hao, Jun Wang, and Haitham Bou-Ammar. Ros-llm: A ros framework for embodied ai with task feedback and structured reasoning, 2024. URL <https://arxiv.org/abs/2406.19741>.
- Sina Bagheri Nezhad and Ameeta Agrawal. Enhancing large language models with neurosymbolic reasoning for multilingual tasks, 2025. URL <https://arxiv.org/abs/2506.02483>.
- Boye Niu, Yiliao Song, Kai Lian, Yifan Shen, Yu Yao, Kun Zhang, and Tongliang Liu. Flow: Modularized agentic workflow automation, 2025. URL <https://arxiv.org/abs/2501.07834>.
- Reinhard Oldenburg. Limitations of and lessons from the learning of large language models. 2023.
- Yuchen Shi, Siqi Cai, Zihan Xu, Yulei Qin, Gang Li, Hang Shao, Jiawei Chen, Deqing Yang, Ke Li, and Xing Sun. Flowagent: a new paradigm for workflow agent. 2025.
- Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.
- Terry Vogelsang. Llm controls execution flow hijacking. In *Large Language Models in Cybersecurity: Threats, Exposure and Mitigation*, pages 99–104. Springer, 2024.
- Xindi Wang, Mahsa Salmani, Parsa Omid, Xiangyu Ren, Mehdi Rezagholizadeh, and Armaghan Eshaghi. Beyond the limits: A survey of techniques to extend the context length in large language models, 2024. URL <https://arxiv.org/abs/2402.02244>.
- Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, et al. Intent-driven network management with multi-agent llms: The confucius framework. In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 347–362, 2025.
- Fangzhou Wu, Ethan Cecchetti, and Chaowei Xiao. System-level defense against indirect prompt injection attacks: An information flow control perspective. *arXiv preprint arXiv:2409.19091*, 2024.

- Zhen Xu, Shang Zhu, Jue Wang, Junlin Wang, Ben Athiwaratkun, Chi Wang, James Zou, and Ce Zhang. When does divide and conquer work for long context llm? a noise decomposition framework, 2026a. URL <https://arxiv.org/abs/2506.16411>.
- Zhenlin Xu, Xiaogang Zhu, Yu Yao, Minhui Xue, and Yiliao Song. From storage to steering: Memory control flow attacks on llm agents, 2026b. URL <https://arxiv.org/abs/2603.15125>.
- Xiao-Wen Yang, Jie-Jing Shao, Lan-Zhe Guo, Bo-Wen Zhang, Zhi Zhou, Lin-Han Jia, Wang-Zhou Dai, and Yu-Feng Li. Neuro-symbolic artificial intelligence: Towards improving the reasoning abilities of large language models, 2025a. URL <https://arxiv.org/abs/2508.13678>.
- Zhuoyi Yang, Xu Guo, Tong Zhang, Huijuan Xu, and Boyang Li. Test-time scaling of llms: A survey from a subproblem structure perspective, 2025b. URL <https://arxiv.org/abs/2511.14772>.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023a. URL <https://arxiv.org/abs/2305.10601>.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023b. URL <https://arxiv.org/abs/2210.03629>.
- Chaojia Yu, Zihan Cheng, Hanwen Cui, Yishuo Gao, Zexu Luo, Yijin Wang, Hangbin Zheng, and Yong Zhao. A survey on agent workflow—status and future. In *2025 8th International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pages 770–781. IEEE, 2025.
- Alex L. Zhang, Tim Kraska, and Omar Khattab. Recursive language models, 2026. URL <https://arxiv.org/abs/2512.24601>.
- Shaokun Zhang, Ming Yin, Jieyu Zhang, Jiale Liu, Zhiguang Han, Jingyang Zhang, Beibin Li, Chi Wang, Huazheng Wang, Yiran Chen, and Qingyun Wu. Which agent causes task failures and when? on automated failure attribution of llm multi-agent systems, 2025. URL <https://arxiv.org/abs/2505.00212>.
- Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, and Ed Chi. Least-to-most prompting enables complex reasoning in large language models, 2023. URL <https://arxiv.org/abs/2205.10625>.
- Kunlun Zhu, Zijia Liu, Bingxuan Li, Muxin Tian, Yingxuan Yang, Jiaxun Zhang, Pengrui Han, Qipeng Xie, Fuyang Cui, Weijia Zhang, Xiaoteng Ma, Xiaodong Yu, Gowtham Ramesh, Jialian Wu, Zicheng Liu, Pan Lu, James Zou, and Jiaxuan You. Where llm agents fail and how they can learn from failures, 2025. URL <https://arxiv.org/abs/2509.25370>.
- Matthieu Zimmer, Milan Gritta, Gerasimos Lampouras, Haitham Bou Ammar, and Jun Wang. Mixture of attentions for speculative decoding, 2025a. URL <https://arxiv.org/abs/2410.03804>.
- Matthieu Zimmer, Xiaotong Ji, Rasul Tutunov, Anthony Bordg, Jun Wang, and Haitham Bou Ammar. Bourbaki: Self-generated and goal-conditioned mdps for theorem proving, 2025b. URL <https://arxiv.org/abs/2507.02726>.

A Complete Example Trace

We trace the λ -RLM on the OOLONG task: classifying 1000 questions in 131K tokens, with $K = 32K$.

Phase 1 - DETECTTASK: $\tau_{\text{type}} = \text{aggregate}$	[1 LLM call]
Phase 2 - PLAN: $k^* = 5, \tau^* = 26K, \oplus = \text{MERGECOUNTS}, d = 1$	[0 LLM calls]
Phase 3 - ESTIMATECOST: $\hat{C} = 5 \times \$0.03 + \$0.02 = \$0.17, \hat{N} = 6$ calls	[0 LLM calls]
Phase 4 - EXECUTE:	[5 LLM calls]

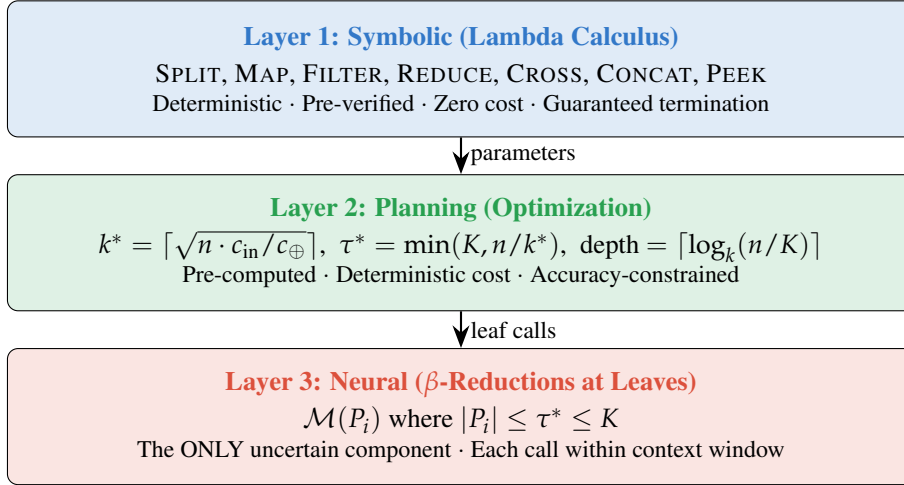
SPLIT($P, 5$) $\rightarrow [P_1, P_2, P_3, P_4, P_5]$	symbolic: free
MAP($\lambda p_i. \mathcal{M}(\text{"count categories: "} \parallel p_i), [P_{1:5}])$ $\rightarrow [\{\text{desc} : 45, \text{num} : 52, \dots\}, \dots, \{\text{desc} : 33, \text{num} : 42, \dots\}]$	neural: 5 β -reductions
REDUCE(MERGECOUNTS, results) $\rightarrow \{\text{desc} : 200, \text{num} : 240, \dots\}$	symbolic: free
Answer: "description is less common than numeric value"	✓

Total: 6 LLM calls, \$0.17, **correct**.

Normal RLM on same task: Huge no of LLM calls, \$1.12, **incorrect** (Example E.2 in [Zhang et al., 2026])

B The Hierarchy of Computation

The λ -RLM cleanly separates computation into three layers:



C Proofs

Theorem 1 (Termination). *Under Assumption 1 (A1-A2), for any input $P \in \Sigma^*$ with $|P| = n < \infty$, the function Φ in Algorithm 2 terminates when executed in the REPL. Moreover, the total number of \mathcal{M} invocations is exactly:*

$$N(n) = (k^*)^d + 1, \quad \text{where } d = \left\lceil \log_{k^*} \frac{n}{\tau^*} \right\rceil. \quad (5)$$

Proof. Define the *rank* of a call $\Phi(P')$ as $r(P') = \lceil \log_{k^*}(|P'|/\tau^*) \rceil$. We prove termination by strong induction on rank.

Base case ($r = 0$): $|P'| \leq \tau^* \leq K$, so Φ invokes $\text{sub_}\mathcal{M}(q)$ via the REPL, which halts by (A1).

Inductive step: Suppose Φ terminates for all inputs with rank $< r$.

Now, if $|P'|$ has rank $r > 0$, then $r - 1 \leq \log_{k^*} \frac{|P'|}{\tau^*} \leq r$. Hence the rank r is an integer and $r > 0$ implies $r \geq 1$, then $|P'| > \tau^*$ and Φ executes $\text{SPLIT}(P', k^*)$ in the REPL, which halts by (A2), producing chunks P_1, \dots, P_{k^*} with $|P_i| = \lceil |P'|/k^* \rceil$. Since

$$|P_i| = \left\lceil \frac{|P'|}{k^*} \right\rceil < |P'| \quad (\text{as } k^* \geq 2), \quad (9)$$

we have $r(P_i) < r(P')$, so each recursive call $\Phi(P_i)$ terminates by the inductive hypothesis. The REPL-executed operations MAP, REDUCE, and FILTER all halt by (A2). Thus $\Phi(P')$ terminates.

For the call count: at depth $\ell \in \{0, \dots, d-1\}$, there are $(k^*)^\ell$ nodes each spawning k^* children. The leaves reside at depth d , giving $(k^*)^d$ leaf calls to $\text{sub_}\mathcal{M}$. Adding the single task-detection call (line 6 of Algorithm 1) gives $N(n) = (k^*)^d + 1$.

Contrast with original RLM. The loop (3) has no rank function; the LLM may generate code that does not reduce input size, yielding unbounded iterations. λ -RLM eliminates this by construction: every recursive call strictly reduces rank. \square

Theorem 2 (Cost Bound). *Under Assumptions (A1-A3), the total cost of Algorithm 1 satisfies the recurrence:*

$$T(n) = k^* \cdot T\left(\frac{n}{k^*}\right) + \mathcal{C}_\oplus(k^*), \quad T(\tau^*) = \mathcal{C}(\tau^*), \quad (6)$$

with a closed-form solution:

$$T(n) \leq \frac{nk^*}{\tau^*} \mathcal{C}(\tau^*) + \mathcal{C}_\oplus(k^*) \cdot \left\lceil \frac{nk^* - \tau^*}{\tau^*(k^* - 1)} \right\rceil, \quad (7)$$

where $d = \lceil \log_{k^*}(n/\tau^*) \rceil$. When \oplus is purely symbolic, $\mathcal{C}_\oplus = 0$ and $T(n) \leq \frac{nk^*}{\tau^*} \cdot \mathcal{C}(\tau^*)$.

Proof. To start, we unroll the recurrence in Equation (6). At level ℓ , there are $(k^*)^\ell$ subproblems each of size $n/(k^*)^\ell$, and one composition step costing $\mathcal{C}_\oplus(k^*)$. Expanding:

$$\begin{aligned} T(n) &= k^* \cdot T(n/k^*) + \mathcal{C}_\oplus(k^*) = k^* [k^* \cdot T(n/(k^*)^2) + \mathcal{C}_\oplus(k^*)] + \mathcal{C}_\oplus(k^*) \\ &= (k^*)^2 \cdot T\left(\frac{n}{(k^*)^2}\right) + (k^* + 1) \cdot \mathcal{C}_\oplus(k^*) \\ &= (k^*)^d \cdot T\left(\frac{n}{(k^*)^d}\right) + ((k^*)^{d-1} + \dots + k^* + 1) \cdot \mathcal{C}_\oplus(k^*) \\ &= (k^*)^d \cdot T\left(\frac{n}{(k^*)^d}\right) + \left\lceil \frac{(k^*)^d - 1}{k^* - 1} \right\rceil \cdot \mathcal{C}_\oplus(k^*) \end{aligned}$$

At depth $d = \lceil \log_{k^*}(n/\tau^*) \rceil$, we have $d - 1 \leq \log_{k^*} \frac{n}{\tau^*} \leq d$, hence $(k^*)^d \leq \frac{nk^*}{\tau^*}$ and $\frac{n}{(k^*)^d} \leq \tau^*$. These two results allow us to bound the above expression:

$$T(n) \leq \frac{nk^*}{\tau^*} T(\tau^*) + \left\lceil \frac{nk^* - \tau^*}{\tau^*(k^* - 1)} \right\rceil \cdot \mathcal{C}_\oplus(k^*) = \frac{nk^*}{\tau^*} \mathcal{C}(\tau^*) + \left\lceil \frac{nk^* - \tau^*}{\tau^*(k^* - 1)} \right\rceil \cdot \mathcal{C}_\oplus(k^*)$$

hitting the base case $T(\tau^*) = \mathcal{C}(\tau^*)$. \square

Theorem 3 (Accuracy Bound). *Under Assumptions (A4-A5), let $d = \lceil \log_{k^*}(n/\tau^*) \rceil$. Since $\frac{n}{\tau^*} \leq (k^*)^d \leq \frac{nk^*}{\tau^*}$, the end-to-end accuracy of λ -RLM satisfies:*

- (i) *If $d = 0$ (input fits in window): $\mathcal{A}_{\lambda\text{-RLM}}(n) = \mathcal{A}(\tau^*)$.*
- (ii) *If $d \geq 1$ and $\mathcal{A}(\tau^*) < 1$ (the non-trivial case):*

$$\mathcal{A}_{\lambda\text{-RLM}}(n) \geq \mathcal{A}(\tau^*)^{nk^*/\tau^*} \cdot \mathcal{A}_{\oplus}^d. \quad (8)$$

- (iii) *If $\mathcal{A}(\tau^*) = 1$ (perfect leaf accuracy): $\mathcal{A}_{\lambda\text{-RLM}}(n) \geq \mathcal{A}_{\oplus}^d$.*
 - (iv) *For decomposable tasks ($\mathcal{A}_{\oplus} = 1$, independent sub-queries): per-query accuracy is $\mathcal{A}(\tau^*)$, constant in n .*
- In contrast, direct inference achieves $\mathcal{A}_{\text{direct}}(n) = \mathcal{A}_0 \cdot \rho^{n/K}$ for $\rho \in (0, 1)$.*

Proof. The recursion tree of Φ has depth $d = \lceil \log_{k^*}(n/\tau^*) \rceil$, branching factor k^* , and therefore $(k^*)^d$ leaf nodes. By definition of the ceiling function:

$$\frac{n}{\tau^*} \leq (k^*)^d \leq \frac{nk^*}{\tau^*}. \quad (10)$$

Base case ($d = 0$): $|P| \leq \tau^*$. A single call to $\text{sub_}\mathcal{M}$ yields accuracy $\mathcal{A}(\tau^*)$.

General case ($d \geq 1$): Correctness requires (i) all $(k^*)^d$ leaves correct, and (ii) all d compositions correct, giving $\mathcal{A}_{\lambda\text{-RLM}}(n) \geq \mathcal{A}(\tau^*)^{(k^*)^d} \cdot \mathcal{A}_{\oplus}^d$.

Since $0 < \mathcal{A}(\tau^*) < 1$, the map $x \mapsto \mathcal{A}(\tau^*)^x$ is strictly decreasing. Applying the upper bound from (10):

$$\mathcal{A}(\tau^*)^{(k^*)^d} \geq \mathcal{A}(\tau^*)^{nk^*/\tau^*}, \quad (11)$$

which yields the stated bound (8). \square

Theorem 4 (Optimal Partition). *Under Assumption (A3), for cost function $\mathcal{C}(n) = c_{\text{in}} \cdot n + c_{\text{out}} \cdot \bar{n}_{\text{out}}$ and constant per-level composition cost $\mathcal{C}_{\oplus}(k) = c_{\oplus} \cdot k$, the cost-minimizing partition size for recurrence (6) is $k^* = 2$*

Proof. From (7):

$$T(n, k) \leq \frac{nk}{\tau^*} \mathcal{C}(\tau^*) + \left\lceil \frac{nk - \tau^*}{\tau^*(k-1)} \right\rceil \cdot \mathcal{C}_{\oplus}(k)$$

or, after simplifying (treating τ^* and n as constants with respect to k):

$$\begin{aligned} T(n, k) &\leq k \underbrace{\frac{n \cdot \mathcal{C}(\tau^*)}{\tau^*}}_{\alpha} + \frac{k^2}{(k-1)} \underbrace{\frac{c_{\oplus} \cdot n}{\tau^*}}_{\beta} - \frac{k}{(k-1)} \underbrace{c_{\oplus}}_{\gamma} \\ &= \alpha k + \beta \frac{k^2}{k-1} - \gamma \frac{k}{k-1} \\ &= \frac{\alpha k(k-1) + \beta k^2 - \gamma k}{k-1} = \frac{k^2(\alpha + \beta) - k(\alpha + \gamma)}{k-1}, \end{aligned} \quad (12)$$

Algorithm 5 Pairwise Tasks

Input: P , predicate ϕ , \mathcal{M} , k^* , τ^*
Output: Pairs $\mathcal{S} \subseteq \mathbb{N} \times \mathbb{N}$
 // A: Linear neural - $O(n/K)$
 1: $[P_{1:k^*}] \leftarrow \text{SPLIT}(P, k^*)$
 2: $\text{labels} \leftarrow \text{MAP}(\text{sub_}\mathcal{M}_{\text{cls}}, P_{1:k^*})$
 3: $L \leftarrow \text{PARSE}(\text{CONCAT}(\text{labels}))$
 // B: Quadratic symbolic - FREE
 4: $Q \leftarrow \text{FILTER}(\phi, L.\text{ITEMS}())$
 5: $\mathcal{S} \leftarrow \{(i, j) \mid i, j \in Q, i < j\}$
 6: **return** \mathcal{S}

Algorithm 6 Multi-Hop Search

Input: Corpus $[D_1, \dots, D_m]$, query q , \mathcal{M}
Output: Answer Y
 // A: Filter - mostly symbolic
 1: $\text{prev} \leftarrow \text{MAP}(\lambda D. \text{PEEK}(D, 0, 500), D_{1:m})$
 2: $\text{rel} \leftarrow \text{FILTER}(\text{MATCH}(q), \text{ZIP}(D, \text{prev}))$
 // B: Read - $|\text{rel}| \ll m$
 3: $\text{evi} \leftarrow \text{MAP}(\text{sub_}\mathcal{M}_{\text{ext}}, \text{rel})$
 // C: Synthesize - 1 call
 4: $Y \leftarrow \text{sub_}\mathcal{M}(\text{"answer"} \parallel q \parallel \text{CONCAT}(\text{evi}))$
 5: **return** Y

where $\frac{n}{\tau^*} \leq (k)^d \leq \frac{nk}{\tau^*}$. Taking the derivative to the upper-bound with respect to k gives:

$$\begin{aligned}
 \frac{d}{dk} \left[\frac{k^2(\alpha + \beta) - k(\alpha + \gamma)}{k - 1} \right] &= \frac{[2(\alpha + \beta)k - (\alpha + \gamma)](k - 1) - [k^2(\alpha + \beta) - k(\alpha + \gamma)]}{(k - 1)^2} \\
 &= \frac{k^2(2\alpha + 2\beta - \alpha - \beta) - 2(\alpha + \beta)k + \alpha + \gamma}{(k - 1)^2} \\
 &= \frac{k^2(\alpha + \beta) - 2(\alpha + \beta)k + (\alpha + \gamma)}{(k - 1)^2} \\
 &= \frac{(\alpha + \beta) \left[\left(k - \left(1 - \sqrt{1 - \frac{\alpha + \gamma}{\alpha + \beta}} \right) \right) \left(k - \left(1 + \sqrt{1 - \frac{\alpha + \gamma}{\alpha + \beta}} \right) \right) \right]}{(k - 1)^2}
 \end{aligned}$$

The minimum of the function is k^* closest to the value $\lceil 1 + \sqrt{1 - \frac{\alpha + \gamma}{\alpha + \beta}} \rceil$. Because we have $k \geq 2$, this implies that the optimal value of $k^* = 2$. \square

D Algorithmic Details

This section provides the algorithmic details of λ -RLM that complement the framework description in Section 3. In particular, we make explicit the end-to-end pipeline and the construction and execution of the fixed combinator program Φ . We use a constant preview budget $b = 500$ for inexpensive symbolic inspection. Algorithm 1 specifies the full end-to-end pipeline, Algorithm 4 defines the recursive executor that carries out the planned decomposition, and Algorithms 5 and 6 provide representative task-specific realizations for pairwise tasks and multi-hop search. These algorithms show how the abstract fixed-point formulation is turned into a concrete, finite, and auditable execution procedure whose call structure, recursion depth, and composition behavior are fixed after task-type selection and planning, before the first recursive execution step is run.

Algorithm 1 presents the complete λ -RLM system as a finite sequence of phases. As in the original RLM formulation, execution begins by initializing a REPL state in which the prompt P is stored externally, the trusted combinator library \mathcal{L} is registered, and the base model is exposed as a callable leaf oracle through $\text{sub_}\mathcal{M}$. The key difference from standard RLM arises immediately after this shared setup: rather than entering an open-ended loop in which the model repeatedly emits arbitrary code, λ -RLM performs a single bounded task-type selection step, followed by deterministic planning and a one-shot execution of a pre-built recursive program. The model is asked to choose a task type from a fixed menu based on a lightweight symbolic probe of the prompt. This keeps neural uncertainty localized to semantic classification, while all subsequent control decisions remain symbolic. Once the task type is selected, the planner determines the execution rule by choosing the task-specific composition operator \oplus and execution plan π , together with the structural parameters (k^*, τ^*, d) . Here, k^* controls the branching factor of decomposition, τ^* determines the leaf threshold at which recursion terminates, and d bounds the resulting recursion depth. The planning phase therefore operationalizes the main design goal of λ -RLM: the shape of execution is fixed before recursive execution begins. Finally, the system builds the combinator chain in the REPL, executes it, and returns the resulting response Y .

Algorithm 3 λ -RLM: Complete System

Input: Prompt $P \in \Sigma^*$, model \mathcal{M} with window K , accuracy target $\alpha \in (0, 1]$

Output: Response $Y \in \Sigma^*$

```
// == Phase 1: REPL Initialization (same as original RLM) ==
1: state  $\leftarrow$  INITREPL(prompt =  $P$ )  $\triangleright P$  lives in environment, not context window
2: state  $\leftarrow$  REGISTERLIBRARY(state,  $\mathcal{L}$ )  $\triangleright$  load pre-verified combinators into REPL
3: state  $\leftarrow$  REGISTERSUBCALL(state, sub_ $\mathcal{M}$ )  $\triangleright$  register  $\mathcal{M}$  as callable, same as RLM
// == Phase 2: Task Detection (1 LLM call) ==
4: meta  $\leftarrow$   $\mathcal{E}$ (state, Peek( $P$ , 0, 500); len( $P$ ))  $\triangleright$  probe  $P$  via REPL, not neural
5:  $\tau_{\text{type}} \leftarrow \mathcal{M}$ ("Select from  $\mathcal{T}$ : " || meta)  $\triangleright$  menu selection, single call
// == Phase 3: Optimal Planning (0 LLM calls, pure math) ==
6:  $\oplus \leftarrow$  TABLE 1B[ $\tau_{\text{type}}$ ];  $\pi \leftarrow$  TABLE 1B[ $\tau_{\text{type}}$ ]
7: if  $|P| \leq K$  then
8:    $k^* \leftarrow 1$ ;  $\tau^* \leftarrow |P|$ 
9: else
10:   $k^* \leftarrow \lceil \sqrt{|P| \cdot c_{\text{in}} / c_{\oplus}} \rceil$   $\triangleright$  minimize  $T(n) = k \cdot T(n/k) + C_{\oplus}(k)$ , base  $T(\tau) = C(\tau)$ 
11:   $d \leftarrow \lceil \log_{k^*}(|P|/K) \rceil$ 
12:  while  $\mathcal{A}(K)^d \cdot \mathcal{A}_{\oplus}^d < \alpha$  and  $k^* < |P|/K$  do  $\triangleright$  accuracy constraint
13:     $k^* \leftarrow k^* + 1$ ;  $d \leftarrow \lceil \log_{k^*}(|P|/K) \rceil$ 
14:  end while
15:   $\tau^* \leftarrow \min(K, \lfloor |P|/k^* \rfloor)$ 
16: end if
// == Phase 4: Cost Estimation (deterministic, pre-execution) ==
17:  $\hat{C} \leftarrow (k^*)^d \cdot C(\tau^*) + d \cdot C_{\oplus}(k^*) + C(500)$   $\triangleright$  exact pre-execution bound
// == Phase 5: Build and Execute Combinator Chain in REPL ==
18: state  $\leftarrow$   $\mathcal{E}$ (state,  $\Phi = \text{BuildExecutor}(k^*, \tau^*, \oplus, \pi, \text{sub-}\mathcal{M})$ )  $\triangleright$  register  $\Phi$  in REPL
19: state  $\leftarrow$   $\mathcal{E}$ (state, result =  $\Phi(P)$ )  $\triangleright$  single execution of  $\Phi$  in REPL
20:  $Y \leftarrow \text{state}[\text{result}]$ 
21: return  $Y$ 
```

Algorithm 4 Φ : Combinator Executor (registered in REPL, not LLM-generated)

Input: Prompt P (from REPL state), parameters k^*, τ^*, \oplus, π (from planner)

Output: Result string

```
1: function  $\Phi(P)$ 
2:   if  $|P| \leq \tau^*$  then  $\triangleright$  base case: leaf  $\beta$ -reduction
3:      $q \leftarrow \text{TEMPLATE}[\tau_{\text{type}}].\text{FMT}(P)$   $\triangleright$  pre-defined prompt template
4:     return sub- $\mathcal{M}(q)$   $\triangleright$  invoke  $\mathcal{M}$  via REPL's registered sub-call
5:   else  $\triangleright$  recursive case: all REPL-executed combinators from  $\mathcal{L}$ 
6:      $[P_1, \dots, P_{k^*}] \leftarrow \text{SPLIT}(P, k^*)$   $\triangleright$  deterministic, pre-verified
7:     if  $\pi$  includes FILTER then  $\triangleright$  optional pre-filter for search/multi_hop
8:       previews  $\leftarrow \text{MAP}(\lambda p. \text{PEEK}(p, 0, \lfloor \tau^*/10 \rfloor), [P_1, \dots, P_{k^*}])$ 
9:        $[P_1, \dots, P_{k'}] \leftarrow \text{FILTER}(\text{RELEVANT}, \text{ZIP}([P_{1:k^*}], \text{previews}))$ 
10:    end if
11:     $[R_1, \dots, R_{k'}] \leftarrow \text{MAP}(\lambda p_i. \Phi(p_i), [P_1, \dots, P_{k'}])$   $\triangleright$  recursive sub-calls
12:    return REDUCE( $\oplus, [R_1, \dots, R_{k'}]$ )  $\triangleright$  deterministic composition
13:  end if
14: end function
```

Algorithm 4 then defines the executor Φ constructed from these planned components. This executor is the concrete realization of the fixed-point program in Eq. (4). Its behavior is intentionally simple. If the current sub-prompt is already below the threshold τ^* , Φ formats it with a task-specific leaf template and calls the base model exactly once. Otherwise, it applies a deterministic recursive pattern: split the input, optionally preview and filter chunks when the task plan requires pruning, recursively process the retained chunks, and combine the resulting partial outputs with $\text{REDUCE}(\oplus, \cdot)$. The recursive structure itself is not model-generated. The only neural operations occur at bounded leaves

and, for certain task types, in explicitly specified synthesis steps, while splitting, filtering, traversal, and aggregation are all handled by trusted combinators with fixed semantics.

Algorithms 5 and 6 illustrate how this general executor specializes to structured task families. The pairwise algorithm shows that the expensive neural portion can remain linear in the number of chunks: the model is used only to label or extract candidate items, after which the quadratic pairing step is computed symbolically at essentially zero additional neural cost. The multi-hop search algorithm follows the same principle in a different form. It first uses symbolic preview-based filtering to narrow a large corpus to a small relevant subset, then applies neural reading only to that subset, and finally performs a single synthesis step over the extracted evidence. These examples are not separate learning algorithms, but concrete instantiations of the same λ -RLM design principle: use the model only where semantic inference is needed, and realize the surrounding control flow through typed symbolic composition.