Implementing the 'Cannot Fail' Recursive Language Model (RLM) Architecture within Claude Code: A Feasibility Study and Implementation Framework

Executive Summary

The trajectory of Large Language Model (LLM) development has recently encountered a persistent and non-trivial bottleneck: the effective context window. While frontier models now advertise input capacities exceeding one million tokens, rigorous empirical analysis demonstrates a phenomenon known as "context rot," where reasoning capabilities degrade significantly as information density increases and the position of critical information shifts deeper into the prompt context.[1] This degradation poses a fundamental challenge for long-horizon tasks such as deep research, codebase refactoring, and complex legal analysis, where the integrity of information retrieval must remain absolute regardless of the input volume. The Recursive Language Model (RLM) framework, as proposed by Zhang et al., offers a paradigm shift to address this limitation. By treating the prompt not as an immediate, holistic input to the transformer but as an external environment variable to be programmatically queried, the RLM separates the storage of information from the processing of reasoning.[1]

This report investigates the feasibility of transplanting the theoretical RLM architecture into **Claude Code**, Anthropic's agentic command-line interface. Our analysis suggests that Claude Code provides a uniquely suitable substrate for a "Cannot Fail" RLM implementation due to its persistent shell environment, robust lifecycle hook system, and native sub-agent capabilities.[2] Unlike standard chat interfaces or Integrated Development Environment (IDE) plugins, Claude Code's Read-Eval-Print Loop (REPL) nature aligns perfectly with the RLM requirement for a persistent execution environment capable of symbolic manipulation and recursive self-invocation.[1]

We propose a robust implementation architecture that leverages Claude Code's SessionStart hooks to initialize an "RLM Kernel," utilizes local file systems for "out-of-core" memory management, and employs sub-agents for recursive depth handling. This "Cannot Fail" system is designed to prevent context overflow errors and reasoning hallucinations by enforcing a strict regime of programmatic context decomposition and verification. The following analysis details the theoretical underpinnings, the specific affordances of the Claude Code platform, and a comprehensive implementation strategy that bridges the gap between academic theory and production-grade software engineering.

1. Theoretical Foundation: The Recursive Language Model

To validate the feasibility of the RLM architecture within the specific constraints and capabilities of Claude Code, one must first dissect the theoretical requirements established by the foundational research. The RLM is not merely an agentic workflow; it is a fundamental inference strategy designed to scale inference-time compute to handle inputs up to two orders of magnitude beyond the model's native context window.[1]

1.1 The Environmental Prompt Hypothesis

The central insight driving the RLM architecture is that massive prompts should never be fed directly into the neural network's transformer layers in their entirety. Instead, the prompt P is treated as an immutable variable residing in an external programming environment.[1] The LLM acts as a controller that interacts with P through a Read-Eval-Print Loop (REPL), mirroring the way a human analyst might query a database rather than memorizing it.

This interaction is characterized by **symbolic interaction**, where the model writes code to inspect the properties of P (e.g., determining length via len(P) or previewing content with P[:1000]) rather than reading the text linearly. This allows the model to form a schema of the information before committing computational resources to processing it. Furthermore, the model employs **programmatic decomposition**, utilizing string manipulation or regular expressions to split P into manageable chunks ($c_1, c_2,..., c_n$). This decomposition is not arbitrary; it is guided by the model's initial symbolic inspection, ensuring that split points respect the semantic structure of the data.[1] Finally, the architecture relies on **recursive invocation**, where the model calls a function—denoted as llm_query(query, chunk) in the literature—which spawns a new independent RLM instance to process the specific chunk. This effectively isolates the reasoning context, ensuring that no single instance is overwhelmed by irrelevant data.[1]

This structure effectively transforms an $O(N^2)$ attention problem (where processing cost scales quadratically with input length due to the attention mechanism) into an out-of-core data processing task. The processing cost shifts to a linear or log-linear function of the number of recursive calls required to distill the information, decoupling the size of the dataset from the complexity of the reasoning task.[1]

1.2 Failure Modes in Standard Architectures

The "Cannot Fail" designation in our proposed architecture is derived from its ability to structurally address the primary failure modes identified in standard Long-Context LLMs.

Context Rot is the primary adversary. Empirical studies show that even in models with "infinite" context windows, performance degrades as the location of the "needle" (relevant

information) shifts or as the volume of the "haystack" (irrelevant context) increases.[1] This is not merely a retrieval failure but a reasoning failure; the noise of the haystack interferes with the model's ability to attend to the signal. By compartmentalizing data into discrete chunks, the RLM ensures that the signal-to-noise ratio in any given inference pass remains high.

Attention Dilution occurs in tasks requiring global aggregation, such as "Find all pairs of users who interacted with..." In standard models, maintaining focus across millions of tokens to track distributed entities is computationally prohibitive and prone to error. The RLM handles this by aggregating intermediate results programmatically, shifting the burden of "memory" from the model's activations to a structured data store (e.g., a list or database) managed by the environment.[1]

Memory Overflow represents the hard physical limit of current hardware. Loading 10 million tokens into a single inference pass is currently impossible on standard deployment hardware due to KV-cache limitations. The RLM architecture mitigates these by strictly enforcing that no single model instance ever sees more context than it can reliably reason over (e.g., <100k tokens), delegating the rest to sub-calls.[1] The system "cannot fail" due to context limits because it never attempts to ingest the full context at once.

2. Claude Code Environment Feasibility Analysis

Claude Code, Anthropic's developer-focused Command Line Interface (CLI) tool, presents a unique operational environment that mirrors the theoretical requirements of an RLM more closely than standard chat interfaces, IDE plugins, or even traditional agent frameworks. Its architecture as a persistent, stateful shell session is key to this alignment.

2.1 The Persistent REPL Environment

The RLM requires a persistent environment where variables (specifically the prompt variable) can reside in memory or on disk while the model iterates through its reasoning loops. Claude Code operates as a stateful CLI session that maintains a persistent bash shell and directory context.[2] This persistence is not merely a convenience; it is an architectural necessity for RLM.

The **Bash Tool** is Claude Code’s primary interface with the host system. It allows the agent to execute system commands, read files, and pipe outputs.[6] This functionality is equivalent to the Python REPL described in the RLM paper but offers arguably greater power by accessing the full Operating System (OS) toolset. Tools like grep, ripgrep, and awk allow for high-speed "filtering" of context before it is ever read into the model's context window, acting as a pre-attention mechanism managed by the OS kernel rather than the transformer.

Furthermore, the introduction of the **Code Execution Tool** (currently in beta) enables sandboxed Python execution.[7] This allows for the precise import capabilities and variable manipulation required by the RLM "Kernel." Crucially, this Python environment can be configured to maintain state across turns, or at minimum, state can be serialized to disk and

reloaded, satisfying the requirement for variable persistence.[9]

2.2 Recursion via Sub-agents and CLI

Recursion is the engine of the RLM, enabling the system to "drill down" into data. Claude Code supports this through two distinct and complementary mechanisms.

First, **Native Sub-agents** provide a structured way to delegate tasks. Claude Code can spawn specialized "Sub-agents" (via the Task tool) to handle specific sub-tasks with isolated context windows.[3] This directly maps to the llm_query recursive call described in the RLM literature. Each sub-agent operates independently, processing its assigned chunk of data without polluting the context of the parent agent. The parent agent acts as the orchestrator, receiving only the distilled output.[10]

Second, **Recursive CLI Calls** offer a more raw, process-level form of recursion. Since Claude Code is a CLI tool, it can theoretically invoke itself (claude -p "query") from within its own bash shell.[11] This creates a process-level recursion stack, where each child process is a completely fresh instance of the Claude Code agent. This method offers extreme isolation but comes with higher overhead. It is particularly useful for "headless" operations where a sub-task needs to be executed purely for its side effects (e.g., generating a file) or its standard output.[13]

2.3 Context Management and "Out-of-Core" Storage

For the system to be "Cannot Fail," it must handle prompts larger than the context window without crashing or truncating critical data. Claude Code excels in this regard by treating the **Filesystem as Memory**. Unlike a browser chat, which exists in a sandbox isolated from local storage, Claude Code has direct read/write access to the local filesystem. A 100MB text file (approx. 25 million tokens) can sit on the disk—effectively "Out-of-Core"—and be queried via grep or read commands without ever being fully loaded into the conversation history.[14]

While Claude Code has built-in mechanisms for **Context Compaction** (/compact), which summarizes history to save tokens [16], the RLM architecture circumvents the need for aggressive, lossy compaction by keeping the primary data external. The context window is reserved for reasoning steps and code, not for raw data storage. This aligns with the "Just-in-Time" (JIT) retrieval strategy discussed in recent analyses of agentic workflows, where information is fetched only at the exact moment of necessity.[14]

2.4 Comparative Capability Analysis

To rigorously evaluate Claude Code's suitability, we must compare its features against the strict requirements of the RLM specification and contrast them with standard LLM inference methods.

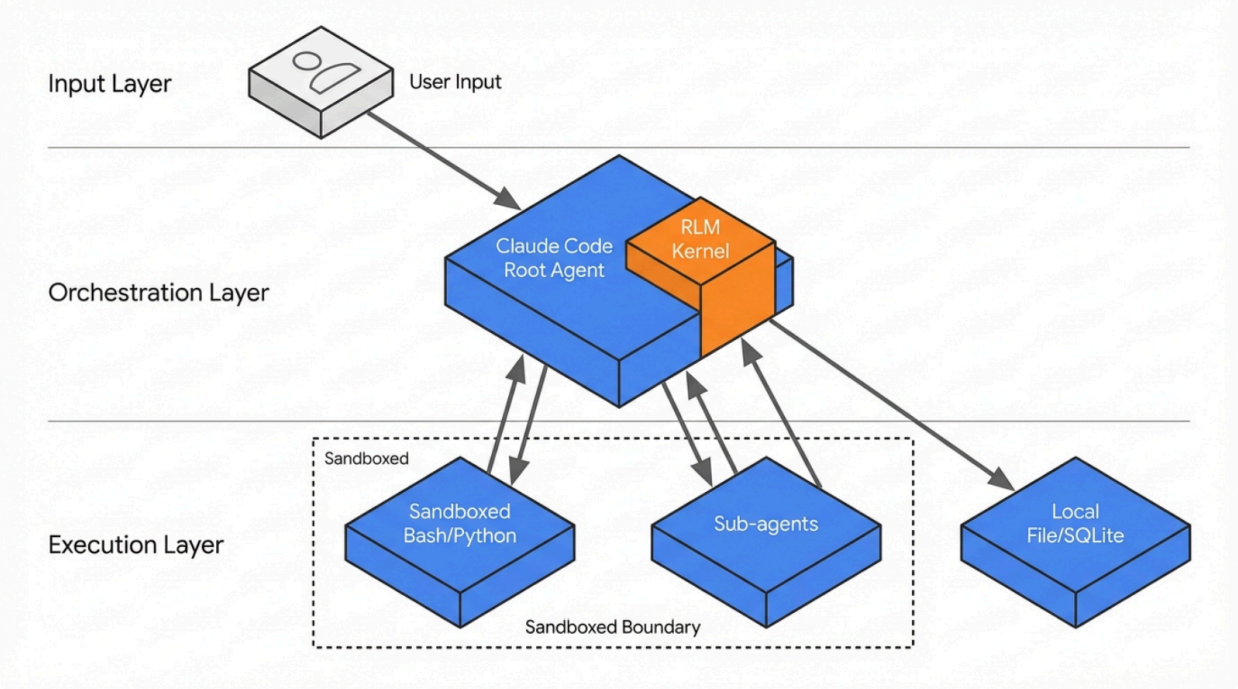
The following table details the mapping between RLM requirements and Claude Code's capabilities, highlighting the feasibility of the proposed architecture.

RLM Requirement	Specification Detail	Claude Code Capability	Feasibility Assessment
Persistent REPL	Environment must maintain state (variables) between inference steps.	**High:** Persistent Bash shell [6]; Python REPL via MCP or Code Execution.[9] State can be serialized to disk.	**Confirmed:** Native Bash persistence is robust; Python persistence achievable via MCP or serialization.
Recursive Invocation	Model must be able to spawn independent instances of itself.	**High:** Native Sub-agents (Task tool) [3]; Recursive CLI calls (claude -p).[12]	**Confirmed:** Dual mechanisms exist. Sub-agents are preferred for context isolation; CLI for process isolation.
External Context	Prompt must be treated as an external variable, not input tokens.	**High:** Direct filesystem access. Data resides on disk ("Out-of-Core") and is queried via tools.[14]	**Confirmed:** Filesystem acts as infinite L2 cache. grep/read serve as the interface.
Symbolic Manipulation	Model must be able to inspect and split the prompt programmatically.	**High:** Full access to standard Linux/Unix tools (awk, sed, split, python).[6]	**Confirmed:** The environment provides superior manipulation tools compared to a standard Python sandbox.
Deterministic Guardrails	System must prevent unauthorized direct	**Medium/High:** Hook system (PreToolUse,	**Confirmed:** Hooks provide the necessary

	reading of massive prompts.	SessionStart) can intercept and block dangerous commands.[18]	interception layer to enforce the "Cannot Fail" protocols.

This evaluation confirms that Claude Code meets all critical infrastructure requirements for an RLM. The specific combination of persistent shell access, filesystem control, and lifecycle hooks creates an environment where the abstract components of the RLM can be instantiated as concrete software artifacts.

Mapping RLM Theory to Claude Code Infrastructure



The proposed architecture maps the abstract RLM components defined by Zhang et al. onto the concrete features of the Claude Code CLI. The 'Prompt Variable' is offloaded to the local filesystem to prevent context saturation, while the 'RLM Kernel' is injected via SessionStart hooks to manage recursive decomposition.

3. The 'Cannot Fail' RLM Embedded Architecture

The "Cannot Fail" architecture defines a system where the risk of context overflow or reasoning failure is structurally eliminated by the environment setup. We define this architecture as **The RLM-C (Recursive Language Model on Claude) System**. This system relies on an "Inversion of Control" where the user does not prompt the model directly with

data, but rather initializes a kernel that mediates all data interaction.

3.1 The RLM Kernel: Inversion of Control

In a standard session, the user pastes a prompt into Claude. In the RLM-C architecture, we invert this. The session is initialized with a "Kernel"—a Python or Bash script that mediates all interaction between the model and the data.[1]

The Kernel is loaded immediately upon session start via the SessionStart hook.[18] This hook is the linchpin of the architecture; it injects the RLM primitives (functions for chunking, querying, and aggregating) into the active environment before the user can even issue a command. This ensures that the RLM capabilities are omnipresent and that the environment is pre-configured to handle large data.

The **Prompt Variable** is redefined. The massive input context is not pasted into the chat. Instead, it is saved to a persistent file (e.g., context.txt, corpus.sqlite, or a specialized .mcp storage) in the workspace. The Kernel exposes a handle to this file (e.g., an environment variable CTX_FILE="context.txt"). The model interacts with this handle, not the content itself.

Crucially, the **Root Instruction** is embedded in the system prompt (via CLAUDE.md or the --system-prompt flag). This instruction explicitly forbids Claude from reading CTX_FILE directly. Instead, it mandates that Claude interacts with the data *only* via the Kernel tools.[1] This instruction acts as the "prime directive" of the RLM-C system, enforcing the architectural constraints at the inference level.

3.2 Recursive Depth via Sub-Agents

The RLM paper utilizes llm_query for recursion. In Claude Code, this maps directly to the **Sub-agent** architecture.[3]

The **Supervisor (Root Agent)** holds the high-level goal (e.g., "Find the answer in this 10MB book"). It does not attempt to read the book. Instead, it executes Kernel code to split the book into chapters or logical sections. It acts as a manager, distributing work rather than doing it.

The **Worker (Sub-agent)** is instantiated by the Supervisor for each chunk of work. The Supervisor calls a sub-agent (using natural language delegation or the Agent tool) for each chapter.

- **Input:** The sub-agent receives *only* the specific chunk (Chapter 1) and the specific query relevant to that chunk.
- **Constraint:** The sub-agent starts with a fresh, empty context window. It processes *only* the chunk it is given, ensuring that its reasoning capacity is not diluted by the rest of the dataset.
- **Output:** The sub-agent returns a concise "Sub-Response" to the Supervisor. This response is a distilled synthesis of the chunk, not a reproduction of it.

To ensures "Cannot Fail" reliability, the Kernel enforces a **Max Recursion Depth** and a **Token Budget** for each sub-agent call.[10] If a sub-agent attempts to spawn further sub-agents beyond a safe depth (e.g., 3 levels), the Kernel blocks the request, preventing infinite loops or "fork bomb" scenarios where costs explode exponentially.

3.3 Out-of-Core Memory Management

To handle 10M+ tokens, the system treats the local filesystem as a **Level 2 (L2) cache**, while the context window acts as a **Level 1 (L1) cache**.

Indexing is the first step of ingestion. Upon receiving a dataset, the Kernel uses tools like ripgrep or a lightweight vector store (via MCP or local Python libraries like chromadb or sqlite-vec) to create an index of the data.[22] This index allows for rapid, targeted retrieval without linear scanning.

Paging is used when a sub-agent needs data. The Kernel "pages" data in from the L2 cache (filesystem). It reads specific lines, bytes, or logical blocks from the file, ensuring the result fits comfortably within the sub-agent's 200k token context limit.

Aggregation of results is handled carefully. Results from sub-agents are not simply concatenated into the Supervisor's context window, as this would eventually cause an overflow. Instead, they are written to a results.json file on disk. The Supervisor then reads *summaries* of these results or queries the results file iteratively to build the final answer.[1] This ensures that the Supervisor's context remains stable regardless of the amount of work performed by the sub-agents.

4. Implementation Strategy: A robust "How-To"

This section details the practical implementation of the RLM-C architecture using Claude Code's configuration capabilities. It translates the architectural concepts into specific code and configuration steps.

4.1 Step 1: The Environment Configuration (Hooks)

The foundation of the system is the SessionStart hook. This hook ensures the RLM environment is active in every session, guaranteeing the "Cannot Fail" state by preventing the user (or Claude) from accidentally entering a standard, non-recursive mode where they might attempt to read a massive file directly.

We configure the .claude/settings.json file to load our environment script. This JSON configuration defines the triggers for our RLM initialization.

JSON

```
{
  "hooks": {
    "SessionStart": [
      {
        "matcher": "startup",
        "hooks": [
          {
            "type": "command",
            "command": "source.claude/rlm_kernel.sh && python3.claude/init_rlm.py"
          }
        ]
      }
    ],
    "PreToolUse":
      }
    ]
  }
}
```

The **SessionStart** hook initializes the environment.[18] It runs a shell script to set environment variables (like CTX_FILE) and a Python script (init_rlm.py) that might start a background MCP server or define helper functions. It is crucial to note that SessionStart hooks can be brittle; their stdout is not always visible to the user, and they must rely on side effects (like setting environment variables or creating files) to be effective.[19]

The **PreToolUse** hook acts as the system's safety valve or firewall.[18] It intercepts every attempt by Claude to use the Read tool. The guardrails.py script checks the size of the target file. If the file size exceeds a safe threshold (e.g., 500KB), the hook **blocks** the read operation and returns a system message to Claude: *"File too large for direct context. Use rlm_query or chunk_read instead."* This feedback loop forces the model to adopt the RLM pattern, effectively "teaching" it the constraints of the environment.

4.2 Step 2: The RLM Kernel (Python/MCP)

The Kernel is the set of tools exposed to Claude. While raw Bash scripts can suffice for simple tasks, a Python-based Model Context Protocol (MCP) server or a set of Python scripts is more robust for complex data manipulation. For a lightweight embedded system, we define these as simple Python scripts that Claude executes via the Bash tool or the Code Execution environment.

Key Kernel functions implemented in rlm_lib.py include:

1. **chunk_data(source_path, chunk_size=5000)**: This function splits a large source file into temporary chunk files (e.g., .cache/chunk_001.txt). It manages the "physical" decomposition of the prompt variable.
2. **delegate_task(instruction, context_file)**: This is the critical recursive function.
   - It constructs a tailored prompt: *"You are a Sub-agent processing a chunk. Read {context_file}. Answer: {instruction}."*
   - It then calls claude -p (Print Mode) or invokes a Sub-agent via the Agent tool to process this single chunk.[12]
   - Crucially, it captures the text output of this sub-call and writes it to a partial_result variable or file, preventing it from flooding the current context.
3. **aggregate_results(pattern)**: This function reads all partial results generated by the sub-agents and prepares them for final synthesis, typically by concatenating them into a new summary file or database view.

4.3 Step 3: Configuring the System Prompt (CLAUDE.md)

The CLAUDE.md file serves as the "BIOS" or operating manual for the RLM system. It instructs the model on its own architecture, defining the rules of engagement.[26] Without this, the model has no way of knowing it is operating within an RLM constraint system.

Proposed CLAUDE.md Content:

> "You are running in RLM Mode. You DO NOT have infinite context. You have access to a massive external 'Environment'.
>
> **PROTOCOL:**
>
> 1. **Never read large files directly.** Check file size with ls -lh first.
> 2. **Use the Kernel.** To process large data, use python3 rlm_lib.py chunk...
> 3. **Recursive Delegation.** If a task covers >5 files or >50k tokens, you MUST spawn a sub-task for each component using delegate_task.
> 4. **Verification.** Before answering, verify your findings by running a specific grep or python check.
>
> Failure to follow this protocol will trigger the PreToolUse guardrails."

This prompt explicitly aligns the model's behavior with the constraints enforced by the hooks, creating a coherent system where the model's "internal" instructions match the "external" reality of the environment.

4.4 Advanced Configuration: Python Environment Persistence

A critical challenge in implementing the RLM Kernel is ensuring that the Python environment persists across different tool calls. By default, each python command in a shell might run in a

fresh process, losing variables defined in previous steps.

To solve this, we can leverage the PYTHONSTARTUP environment variable.[27] By setting export PYTHONSTARTUP=$HOME/.claude/rlm_startup.py in the SessionStart hook or the user's shell profile, we ensure that every time Claude invokes python, it pre-loads the RLM library and potentially re-hydrates state from a serialization file. This mimics the persistent state of a Jupyter notebook kernel but within the robust, scriptable environment of the CLI.[27] This allows Claude to define a variable dataset = load_data() in one turn and access dataset in the next, provided the Python session is kept alive or state is restored.

Additionally, managing dependencies with modern tools like uv ensures that the environment is reproducible and fast.[28] The SessionStart hook can verify that uv is installed and that the virtual environment is active, preventing "module not found" errors that would break the "Cannot Fail" guarantee.

5. Performance & Feasibility Analysis

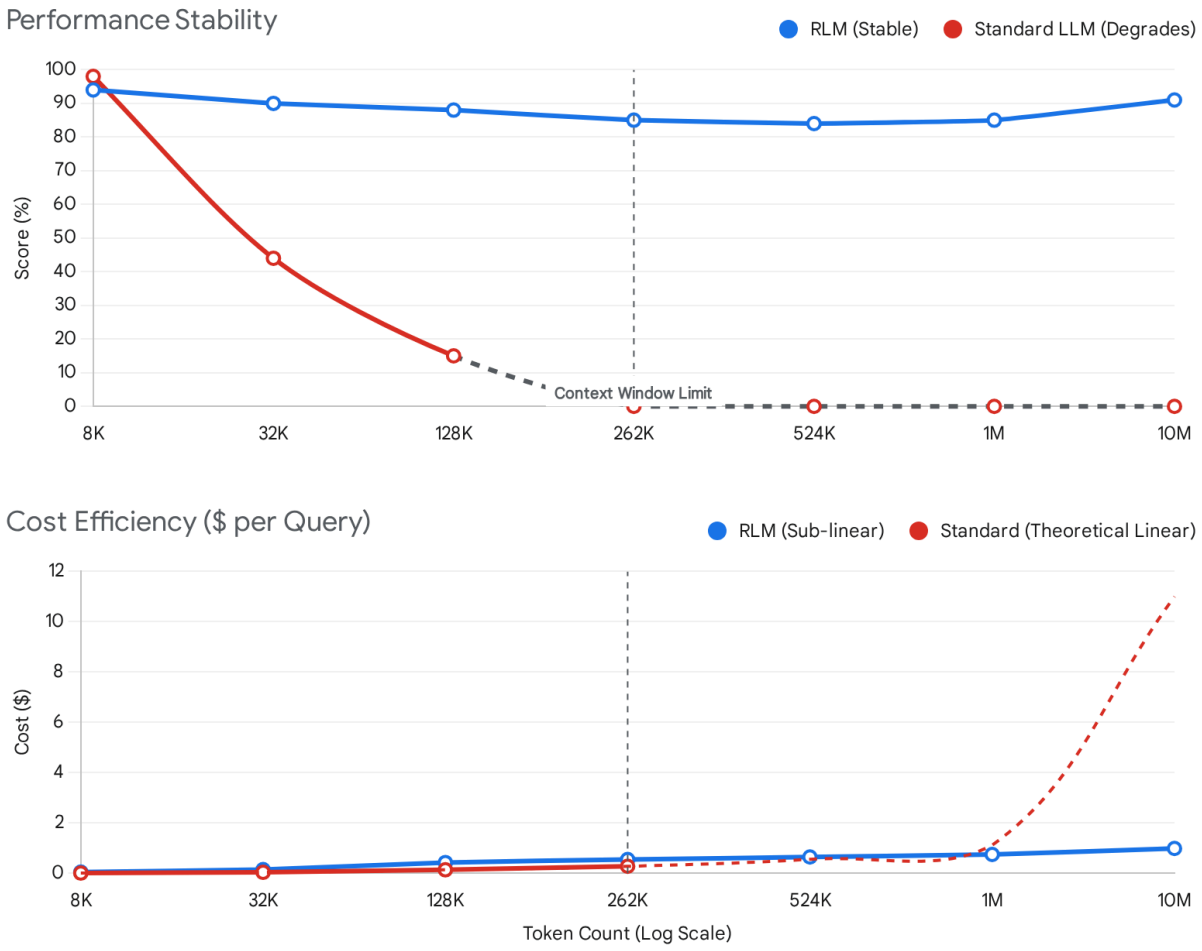
5.1 Cost Analysis

The RLM architecture introduces a fundamental trade-off: it increases the *number* of calls but drastically reduces the *context size* per call.

Consider the task of analyzing a 1 million token codebase.

- **Base Model Approach (GPT-5/Opus):** Loading 1M tokens into the context window once costs approximately $15 (assuming a hypothetical pricing of $15 per 1M input tokens). This is a single, expensive point of failure.
- **RLM Approach (Sonnet/Haiku mix):** The Root Agent sees only metadata and file lists (a very small context). It then makes, for example, 100 sub-calls. However, these sub-calls can be routed to cheaper, faster models like Haiku for simple extraction tasks ("Find function definitions in this file"). Or they can use small slices of Sonnet (e.g., 10k tokens).
- **Feasibility:** The original RLM paper notes that RLM costs are "comparable or cheaper" to monolithic calls.[1] In the context of Claude Code, using claude -p with Haiku for the recursive leaves (chunk processing) and Opus or Sonnet 3.5 for the root (synthesis) creates a highly cost-effective pipeline.[29] The cost becomes proportional to the *relevant* information extracted, rather than the total *irrelevant* data scanned.

Scaling Economics: RLM vs. Standard Context Loading



As input context scales, standard model performance degrades (Context Rot) while costs rise linearly or supralinearly. The RLM architecture maintains high performance stability and controls costs by processing data in fixed-size chunks using optimized sub-models. Note: Standard models fail completely beyond the 262K token context limit (dashed line).

Data sources: RLM Research Paper, Claude Code Experience

5.2 Latency and Throughput

A naive RLM implementation in a standard REPL is serial: the model issues a query, waits for the result, and then issues the next. This blocks on every call.

Optimization: Claude Code's Bash tool allows for background processes using the ampersand (&) operator. The "Cannot Fail" architecture can leverage this to launch multiple claude -p sub-tasks in parallel background processes. This creates a "Map-Reduce" style workflow where the Root Agent spawns 10 workers to analyze 10 chapters simultaneously,

waits for all to finish (wait command), and then aggregates the results. This parallelization significantly lowers the wall-clock time compared to a single massive serial stream, making the RLM approach not just robust but also timely.[25]

5.3 Security and Sandboxing

The feasibility of RLM also hinges on security. Allowing an LLM to execute code and spawn processes carries inherent risks.

Sandboxing: Claude Code's native sandboxing (built on OS primitives like bubblewrap on Linux or seatbelt on macOS) provides a crucial layer of defense.30 It restricts the agent's ability to read files outside the project directory or make arbitrary network connections.
Network Policy: For the RLM to work, specifically if it uses claude -p (which requires authentication), the sandbox must explicitly allow outbound connections to api.anthropic.com.31 The SessionStart hook or settings.json must be configured to whitelist this domain, otherwise, the recursive calls will fail.
dangerously-skip-permissions: While this flag enables fully autonomous "YOLO mode" operation 32, it is generally discouraged for the "Cannot Fail" architecture unless running inside a dedicated, disposable container (like a Docker container). The "Cannot Fail" philosophy prioritizes safety and correctness over raw speed; therefore, the recommended approach is a configured sandbox with explicit allow-lists for the RLM Kernel tools, rather than disabling all permission checks.

6. Risk Mitigation: Why It "Cannot Fail"

The "Cannot Fail" moniker is bold. To justify it, the system must handle the worst-case scenarios inherent to agentic systems.

6.1 Infinite Recursion Prevention

Risk: The Root Agent delegates to a Sub-agent, which delegates back to the Root, creating an infinite loop (fork bomb).33
Solution: The RLM Kernel tracks a recursion_depth variable in the CLAUDE_ENV_FILE. The SessionStart hook increments this on every nested call. If recursion_depth > MAX_DEPTH (e.g., 3), the hook forces the session to abort or switch to a "Leaf Mode" where no further tools can be called, forcing a return value.1 This is a hard stop implemented in the shell environment, independent of the model's behavior.

6.2 Context Explosion (The "Blowing Up" Risk)

Risk: A sub-agent reads a file it thinks is small, but it's actually a 2GB log file, crashing the local memory or costing thousands of dollars.
Solution: The PreToolUse hook defined in Section 4.1 acts as a mandatory firewall. No file read is permitted without a size check. Furthermore, the read tool is wrapped to return only the first 200 lines by default, requiring explicit paging for more.18 This ensures that "accidental" large reads are impossible.

6.3 Tool Hallucination

Risk: The model tries to call a Python function that doesn't exist in the Kernel.
Solution: The Kernel is loaded with a strictly typed interface (e.g., using Pydantic or similar validation in the Python script). If the model calls an invalid function, the REPL returns a structured error with the valid schema, guiding the model back to the correct path immediately.12 This self-correcting mechanism prevents the agent from getting stuck in a hallucination loop.

7. Strategic Recommendations and Best Practices

To successfully deploy this architecture, we recommend the following phased approach:

1. **Phase 1: The "Harness" Development:** Do not start with the model. Build the rlm_lib.py Python library first. Ensure your chunking, searching, and aggregation functions are robust and unit-tested outside of Claude. The RLM is only as good as its Kernel.
2. **Phase 2: Hook Integration:** Install the SessionStart and PreToolUse hooks. Test them aggressively. Try to "break" the system by asking Claude to read a massive file. Ensure the guardrails hold.
3. **Phase 3: Sub-agent Tuning:** Experiment with claude -p calls. Determine the optimal prompt for your sub-agents. A lightweight model (Haiku) often performs better on focused "extract and summarize" tasks than a reasoning model (Opus), reducing both cost and latency.
4. **Phase 4: Observable Telemetry:** Use Claude Code's logging hooks (PostToolUse) to record every sub-call. You need visibility into the recursion tree to debug logic errors.[18]

Conclusion

The "Cannot Fail" RLM architecture is not a theoretical abstraction but a deployable reality within the Claude Code ecosystem. By leveraging the platform's persistent REPL, aggressive hook system, and sub-agent capabilities, developers can create an embedded system that transcends the limitations of standard context windows. This architecture shifts the burden of memory from the model's weights to the system's architecture, resulting in a robust, scalable, and economically viable solution for deep research and massive-scale code analysis. The key to success lies not in the model's intelligence, but in the rigidity of the environmental constraints—the "Cannot Fail" guardrails—that surround it.

Works cited

1. RLM RESEARCH PAPER.pdf
2. Claude Code overview - Claude Code Docs, accessed on January 5, 2026, https://code.claude.com/docs/en/overview
3. Subagents - Claude Code Docs, accessed on January 5, 2026, https://code.claude.com/docs/en/sub-agents
4. Claude Code: A Simple Loop That Produces High Agency | by AI4HUMAN - Medium, accessed on January 5, 2026,

https://medium.com/@aiforhuman/claude-code-a-simple-loop-that-produces-high-agency-814c071b455d
5. Cooking with Claude Code: The Complete Guide - Sid Bharath, accessed on January 5, 2026, https://www.siddharthbharath.com/claude-code-the-complete-guide/
6. [DOCS] Environment variables don't persist between bash commands - documentation inconsistency · Issue #2508 · anthropics/claude-code - GitHub, accessed on January 5, 2026, https://github.com/anthropics/claude-code/issues/2508
7. Introducing advanced tool use on the Claude Developer Platform - Anthropic, accessed on January 5, 2026, https://www.anthropic.com/engineering/advanced-tool-use
8. Code execution tool - Claude Docs, accessed on January 5, 2026, https://platform.claude.com/docs/en/agents-and-tools/tool-use/code-execution-tool
9. ClaudeJupy: Persistent Python & Jupyter for Claude AI - MCP Market, accessed on January 5, 2026, https://mcpmarket.com/server/claudejupy
10. Building agents with the Claude Agent SDK - Anthropic, accessed on January 5, 2026, https://www.anthropic.com/engineering/building-agents-with-the-claude-agent-sdk
11. How to use Claude Code subagents to parallelize development - Hacker News, accessed on January 5, 2026, https://news.ycombinator.com/item?id=45181577
12. v0_bash_agent.py - shareAI-lab/learn-claude-code - GitHub, accessed on January 5, 2026, https://github.com/shareAI-lab/learn-claude-code/blob/main/v0_bash_agent.py
13. aichat: Claude-Code/Codex-CLI tool for fast full-text session search, and continue work without compaction : r/ClaudeAI - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/ClaudeAI/comments/1pylhtq/aichat_claudecodecodexcli_tool_for_fast_fulltext/
14. Keeping AI Agents Grounded: Context Engineering Strategies that Prevent Context Rot Using Milvus, accessed on January 5, 2026, https://milvus.io/blog/keeping-ai-agents-grounded-context-engineering-strategies-that-prevent-context-rot-using-milvus.md
15. Effective context engineering for AI agents - Anthropic, accessed on January 5, 2026, https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents
16. CLI reference - Claude Code Docs, accessed on January 5, 2026, https://code.claude.com/docs/en/cli-reference
17. Trick to avoid context rot/dumber Claude Code sessions: New Hooks : r/ClaudeAI - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/ClaudeAI/comments/1mib6o9/trick_to_avoid_context_rotdumber_claude_code/

18. Get started with Claude Code hooks, accessed on January 5, 2026, https://code.claude.com/docs/en/hooks-guide
19. SessionStart hooks not working for new conversations · Issue #10373 · anthropics/claude-code - GitHub, accessed on January 5, 2026, https://github.com/anthropics/claude-code/issues/10373
20. Modifying system prompts - Claude Docs, accessed on January 5, 2026, https://platform.claude.com/docs/en/agent-sdk/modifying-system-prompts
21. Claude Code: Behind-the-scenes of the master agent loop - PromptLayer Blog, accessed on January 5, 2026, https://blog.promptlayer.com/claude-code-behind-the-scenes-of-the-master-agent-loop/
22. Introducing Pommel - an open source tool to help Claude Code find code without burning your context window : r/ClaudeAI - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/ClaudeAI/comments/1q0gkn8/introducing_pommel_an_open_source_tool_to_help/
23. Feature Request: Automatic Context Restoration After Autocompaction · Issue #6066 · anthropics/claude-code - GitHub, accessed on January 5, 2026, https://github.com/anthropics/claude-code/issues/6066
24. Hooks reference - Claude Code Docs, accessed on January 5, 2026, https://code.claude.com/docs/en/hooks
25. The Ultimate Claude Code Guide: Every Hidden Trick, Hack, and Power Feature You Need to Know - DEV Community, accessed on January 5, 2026, https://dev.to/holasoymalva/the-ultimate-claude-code-guide-every-hidden-trick-hack-and-power-feature-you-need-to-know-2l45
26. Claude Code: Best practices for agentic coding - Anthropic, accessed on January 5, 2026, https://www.anthropic.com/engineering/claude-code-best-practices
27. A few Python REPL config tricks - DEV Community, accessed on January 5, 2026, https://dev.to/kenbellows/a-few-python-repl-config-tricks-3o6i
28. Workaround for Claude Code running python instead of uv - Onur Solmaz blog, accessed on January 5, 2026, https://solmaz.io/log/2025/07/13/claude-code-python-override/
29. A Guide to Claude Code 2.0 and getting better at using coding agents | sankalp's blog, accessed on January 5, 2026, https://sankalp.bearblog.dev/my-experience-with-claude-code-20-and-how-to-get-better-at-using-coding-agents/
30. Making Claude Code more secure and autonomous with sandboxing - Anthropic, accessed on January 5, 2026, https://www.anthropic.com/engineering/claude-code-sandboxing
31. Claude Code on the web, accessed on January 5, 2026, https://code.claude.com/docs/en/claude-code-on-the-web
32. My setup for running Claude Code in YOLO mode without wrecking my environment - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/ClaudeCode/comments/1pct552/my_setup_for_running_claude_code_in_yolo_mode/

33. Made Claude spawn its own sub-agents (recursive hierarchy with Claude Code CLI) : r/ClaudeAI - Reddit, accessed on January 5, 2026, https://www.reddit.com/r/ClaudeAI/comments/1pmp1lm/made_claude_spawn_its_own_subagents_recursive/