Summary

Recursive Language Models (RLMs) are a general inference strategy that allows LLMs to process arbitrarily long prompts—up to two orders of magnitude beyond model context windows.

Key Insight

The core innovation is treating **long prompts as part of an external environment** rather than feeding them directly into the neural network. The RLM:

1. Loads the input prompt as a variable in a Python REPL environment
2. Allows the LLM to programmatically examine, decompose, and peek into the prompt
3. Enables recursive self-calls over snippets of the prompt

How It Works

Given a prompt P, the RLM: - Initializes a REPL environment with P as a variable - Provides the LLM with context about P (e.g., length) - Permits the LLM to write code that peeks into and decomposes P - Crucially, allows the LLM to construct sub-tasks and invoke itself recursively

Key Results

- **Handles 10M+ tokens** effectively while base models fail beyond their context windows
- **Outperforms base LLMs by up to 2×** on long-context benchmarks
- **Comparable or cheaper costs** than alternatives like context compaction or retrieval agents
- On OOLONG-Pairs (quadratic complexity), RLMs achieved 58% F1 while base GPT-5 scored <0.1%

Benchmarks Used

1. **S-NIAH**: Single needle-in-haystack (constant complexity)

2. **BrowseComp-Plus**: Multi-hop QA over 1000 documents (6-11M tokens)
3. **OOLONG**: Long reasoning with linear complexity
4. **OOLONG-Pairs**: Pairwise reasoning with quadratic complexity
5. **LongBench-v2 CodeQA**: Code repository understanding

Emergent Patterns

RLMs exhibited several interesting behaviors without explicit training: - **Filtering via code**: Using regex and model priors to narrow search space - **Chunking and recursion**: Deferring reasoning chains to sub-LM calls - **Answer verification**: Using sub-calls to verify answers with smaller contexts - **Variable-based long outputs**: Building answers through REPL variables

Limitations

- Optimal implementation mechanisms remain underexplored
- Asynchronous sub-calls could reduce runtime
- Deeper recursion layers not fully investigated
- Models not explicitly trained as RLMs—current models make inefficient decisions

RLMs relevant to Puffin's architecture, as it demonstrates how an orchestration layer can dramatically extend LLM capabilities through programmatic context management—similar to how Puffin manages Claude Code CLI interactions.

I have added a skill that implements an RLM using a REPL python script `.claude/skills/rlm`, it works with an agent (`.claude/agents/rlm_subcall`) and manages its state in the `.claude/rlm_state` directory.

RLM Integration Design Specification: Branch/Thread History Search

Overview

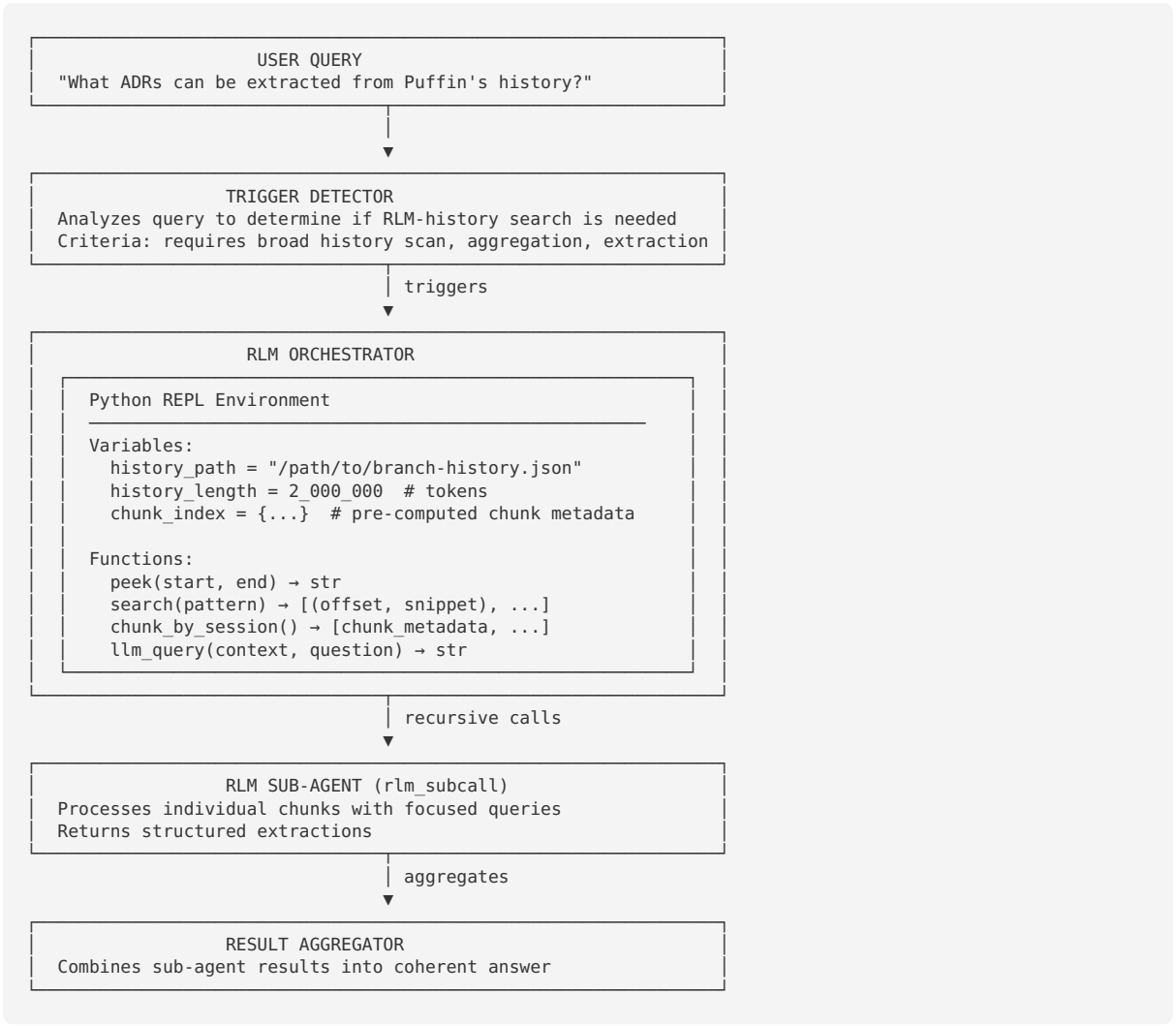
This specification defines how to integrate the RLM skill with Puffin's branch/thread history files to enable intelligent searches across arbitrarily large conversation histories (2M+ tokens).

Problem Statement

Puffin maintains comprehensive branch/thread history files that can grow to millions of tokens. Users need to query this history for insights like: - "What Architecture Decision Records can be extracted from this history?" - "What bugs were discussed and how were they resolved?" - "What design patterns were considered for feature X?" - "Show me all discussions about performance optimization"

Direct context injection is impossible due to model context limits. The RLM approach treats history as an external environment to be programmatically explored.

Architecture



Component Specifications

1. Trigger Detector

Purpose: Determine when a user query requires RLM-based history search rather than direct context or simple file reads.

Trigger Criteria:

Criterion	Description	Examples
Scope Keywords	Query contains terms indicating broad search	"all", "every", "throughout", "history", "extract from"

Criterion	Description	Examples
Aggregation Intent	Query requires collecting/summarizing across sessions	"list of", "how many", "what patterns", "summarize"
Temporal Span	Query references time ranges or evolution	"over time", "how did X evolve", "when did we decide"
Extraction Tasks	Query asks for structured extraction	"ADRs", "decisions", "bugs fixed", "features added"

Implementation:

```
// Proposed location: src/renderer/sam/actions.js or new file
const HISTORY_SEARCH_TRIGGERS = {
  scopePatterns: [
    /\b(all|every|entire|whole|throughout)\b.*\bhistory\b/i,
    /\bextract\s+from\b/i,
    /\bsearch\s+(across|through)\b/i
  ],
  aggregationPatterns: [
    /\b(list|enumerate|catalog|inventory)\s+of\b/i,
    /\bhow\s+many\b/i,
    /\bwhat\s+(patterns|decisions|bugs|features)\b/i
  ],
  extractionKeywords: [
    'ADR', 'architecture decision', 'decision record',
    'bugs', 'issues', 'fixes', 'resolutions',
    'design patterns', 'conventions'
  ]
};

function shouldTriggerRLMHistorySearch(query) {
  // Returns { triggered: boolean, confidence: number, reason: string }
}
```

2. History Index Structure

Purpose: Pre-compute metadata about history file to enable efficient chunking and navigation.

Schema:

```
{
  "historyFile": "branches/main/history.json",
  "totalTokens": 2100000,
  "lastIndexed": "2026-01-19T10:30:00Z",
  "sessions": [
    {
      "id": "session-001",
      "startOffset": 0,
      "endOffset": 45000,
      "tokenCount": 12000,
      "timestamp": "2025-11-01T...",
      "summary": "Initial architecture discussion",
      "topics": ["architecture", "electron", "SAM pattern"]
    }
  ],
  "topicIndex": {
    "architecture": ["session-001", "session-015", "session-042"],
    "bugs": ["session-008", "session-023"],
    "plugins": ["session-012", "session-018", "session-030"]
  }
}
```

Index Generation: - Run periodically or on history file changes - Use lightweight LLM calls to generate session summaries and topic tags - Store alongside history file: `history.index.json`

3. REPL Environment Extensions

New Functions for History Search:

```
# Extensions to .claude/skills/rlm/repl_env.py

def load_history_index(branch: str) -> dict:
    """Load pre-computed index for a branch's history."""
    pass

def get_sessions_by_topic(index: dict, topic: str) -> list[dict]:
    """Filter sessions relevant to a topic."""
    pass

def read_session(history_path: str, session: dict) -> str:
    """Read a specific session's content by offset."""
    pass

def extract_structured(session_content: str, schema: str) -> dict:
    """Use sub-LLM to extract structured data from session."""
    pass
```

4. Search Strategies

Strategy A: Topic-Filtered Search

```
1. Load history index
2. Identify relevant topics from query
3. Filter sessions by topic
4. For each relevant session:
   a. Read session content
   b. Call llm_query() with extraction prompt
   c. Collect structured results
5. Aggregate and deduplicate results
```

Strategy B: Chunked Full Scan

```
1. Divide history into N chunks (e.g., 50K tokens each)
2. For each chunk in parallel (where possible):
   a. Call llm_query() with chunk + extraction prompt
   b. Collect partial results
3. Merge and reconcile partial results
4. Final llm_query() to synthesize answer
```

Strategy C: Hierarchical Summarization

```
1. If session summaries exist, search summaries first
2. Identify candidate sessions from summary matches
3. Deep-dive only into candidate sessions
4. Extract detailed information from candidates
```

5. Result Aggregation

Purpose: Combine results from multiple sub-agent calls into a coherent response.

Aggregation Rules:

Result Type	Aggregation Method
Lists (ADRs, bugs)	Union + deduplication by key fields
Counts	Sum with overlap detection
Timelines	Merge-sort by timestamp
Summaries	Hierarchical summarization

Deduplication Heuristics: - Same decision mentioned in multiple sessions → keep richest description - Similar items with different wording → LLM-based similarity check - Conflicting information → flag for user review with sources

---

Data Flow Example

Query: "What Architecture Decision Records can be extracted from Puffin's history?"

```
1. TRIGGER DETECTION
   - "extract from" + "history" → triggered
   - "Architecture Decision Records" → extraction task
   - Confidence: 0.95

2. INDEX LOOKUP
   - Load history.index.json
   - Filter by topics: ["architecture", "decisions", "ADR"]
   - Found: 12 relevant sessions out of 150 total

3. RECURSIVE EXTRACTION
   For each relevant session:
   ┌──────────────────────────────────────────────┐
   │ llm_query(                                   │
   │   context: session_content,                  │
   │   query: "Extract any architecture           │
   │           decisions in ADR format:           │
   │           - Title                            │
   │           - Context                          │
   │           - Decision                         │
   │           - Consequences                     │
   │           Return JSON array."                │
   │ )                                            │
   └──────────────────────────────────────────────┘

4. AGGREGATION
   - Collected 23 potential ADRs from 12 sessions
   - Deduplicated to 15 unique ADRs
   - Sorted by implied chronology

5. RESULT SYNTHESIS
   Final llm_query() to format results:
   - Group by category (architecture, state, IPC, etc.)
   - Add session references for traceability
   - Generate summary statistics
```

Implementation Phases

Phase 1: History Indexing (Foundation)

- [ ] Define index schema
- [ ] Implement index generator (can use existing RLM for summarization)
- [ ] Add index storage alongside history files
- [ ] Create index update triggers (on history append)

Phase 2: Trigger Detection

- [ ] Implement trigger pattern matching
- [ ] Add confidence scoring
- [ ] Integrate with prompt submission flow
- [ ] Add user override (force RLM / skip RLM)

Phase 3: REPL Extensions

- [ ] Add history-specific functions to REPL environment

- [ ] Implement session reading by offset
- [ ] Add topic filtering utilities
- [ ] Test with sample queries

Phase 4: Search Strategies

- [ ] Implement topic-filtered search (Strategy A)
- [ ] Add chunked full scan fallback (Strategy B)
- [ ] Implement strategy selection heuristics
- [ ] Add progress reporting for long searches

Phase 5: Result Handling

- [ ] Implement result aggregation logic
- [ ] Add deduplication heuristics
- [ ] Create result formatting templates
- [ ] Add source traceability (link results to sessions)

Phase 6: Integration & UX

- [ ] Add UI indicator when RLM search is active
- [ ] Show progress (sessions processed, results found)
- [ ] Allow query refinement mid-search
- [ ] Cache results for repeated queries

Configuration Options

```
// Proposed: src/shared/constants.js or project config
const RLM_HISTORY_CONFIG = {
  // Trigger sensitivity
  triggerConfidenceThreshold: 0.7,

  // Chunking
  maxChunkTokens: 50000,
  maxParallelChunks: 4,

  // Search limits
  maxSessionsToSearch: 50,
  maxResultsPerSession: 10,

  // Caching
  indexCacheTTL: 3600000, // 1 hour
  resultCacheTTL: 300000, // 5 minutes

  // User control
  allowUserOverride: true,
  showProgressUI: true
};
```

History File Schema

The branch/thread history is stored in `.puffin/history.json` with this structure:

```
{
  "branches": {
    "<branch-id>": {
      "id": "specifications",
      "name": "Specifications",
      "prompts": [
        {
          "id": "uuid",
          "parentId": "uuid | null",      // null = conversation root
          "content": "User's prompt text",
          "timestamp": 1765061544113,      // Unix ms
          "response": {
            "content": "Claude's full response text",
            "sessionId": "uuid",           // For --resume
            "cost": 0.288,                  // API cost in USD
            "turns": 5,                     // Agentic turns
            "duration": 138798,             // Response time in ms
            "timestamp": 1765061685679      // Completion time
          },
          "children": ["uuid1", "uuid2"]   // Child prompt IDs
        }
      ]
    }
  }
}
```

Key Observations: - Multiple branches exist (specifications, architecture, ui, backend, etc.) - Prompts form a tree structure via `parentId` and `children` - `parentId: null` indicates a conversation root (new thread) - Response content can be very large (full Claude responses with reasoning) - Typical file size: 8-20 MB for active projects

Chunking Strategy Implications: - Natural chunking boundary: individual prompts (prompt + response pairs) - Session boundaries: prompts with `parentId: null` start new conversation trees - Topic inference: can be derived from prompt content and branch name

---

Resolved Design Decisions

Question	Resolution	Notes
Index freshness	Sprint-boundary reindex	Full reindex occurs at sprint close (after code review and/or bug fixing phases, if performed). Outside of sprints, reindex every 4 hours if any interactions occurred. No reindex

Question	Resolution	Notes
		when Puffin is idle. Maximum staleness: 4 hours (or less during active sprints).
Parallel execution	Yes, parallel sub-calls	Sub-agent calls will run in parallel where possible. Each sub-call processes an independent chunk, so no REPL state conflicts.
Cost management	Deferred	No token budget limits initially. Scaffolding can be added later for: `maxChunksPerSearch`, `maxTokensPerSearch`, `costWarningThreshold`.
Incremental results	Final only	Results are aggregated and shown only after all chunks are processed. Scaffolding for streaming can be added later via event emitters.
History format	Documented above	JSON with branch/prompt tree structure in `.puffin/history.json`.