This is a fascinating design challenge. Let me analyze how RLM concepts could enhance Puffin's orchestration capabilities.

RLM-Enhanced Puffin Architecture

Core Concept Mapping

RLM Concept	Puffin Application
Prompt as external variable	Specifications, user stories, and codebase as queryable context
REPL environment	Persistent workspace with project state, history, and artifacts
Recursive sub-calls	Specialized Claude Code threads (UI, Backend, Plugin) with scoped context
Programmatic decomposition	Intelligent chunking of specs/code for targeted implementation

---

Proposed Architecture

Layer 1: Context Management Layer

```
┌───────────────────────────────────────────────────────────────┐
│                         CONTEXT VAULT                         │
├───────────────────────────────────────────────────────────────┤
│  Specifications      │  Codebase Index     │  Conversation    │
│  ───────────────     │  ──────────────     │  History         │
│  • User Stories      │  • File manifest    │  ─────────────   │
│  • Acceptance        │  • Symbol index     │  • Thread logs   │
│  • Constraints       │  • Dependencies     │  • Decisions     │
│  • Domain rules      │  • Patterns         │  • Rationale     │
└───────────────────────────────────────────────────────────────┘
```

Key Innovation: Context is never fed directly to Claude Code. Instead, Puffin maintains a **queryable vault** that the orchestrator can slice, filter, and compose into targeted prompts.

Layer 2: Recursive Orchestrator

```
┌─────────────────────────────────────────────────────────────┐
│                    RECURSIVE ORCHESTRATOR│
├─────────────────────────────────────────────────────────────┤
│                                                             │
│   ┌─────────────┐                                           │
│   │   PLANNER   │  Analyzes task, determines decomposition  │
│   └──────┬──────┘                                           │
│          │                                                  │
│          ▼                                                  │
│   ┌─────────────┐      ┌─────────────┐      ┌─────────────┐ │
│   │  SPEC AGENT │      │ CODE AGENT  │      │ TEST AGENT  │ │
│   │  (depth=1)  │────▶│  (depth=1)  │────▶│  (depth=1)  │   │
│   └──────┬──────┘      └──────┬──────┘      └─────────────┘ │
│          │                    │                             │
│          ▼                    ▼                             │
│   ┌─────────────┐      ┌─────────────┐                      │
│   │ SUB-SPEC    │      │ SUB-CODE    │   Recursive depth    │
│   │ (depth=2)   │      │ (depth=2)   │   controlled         │
│   └─────────────┘      └─────────────┘                      │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Layer 3: Context Slicing Engine

This is where the RLM insight becomes powerful:

```
┌─────────────────────────────────────────────────────────────┐
│                     CONTEXT SLICER                          │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  Input: "Implement drag-and-drop for calendar notes"        │
│                                                             │
│  Slice Strategy:                                            │
│  ┌───────────────────────────────────────────────────────┐  │
│  │ 1. Extract relevant user stories (not all 50+)        │  │
│  │ 2. Index only calendar plugin files                   │  │
│  │ 3. Pull only drag-drop patterns from codebase         │  │
│  │ 4. Include only related completed stories             │  │
│  └───────────────────────────────────────────────────────┘  │
│                                                             │
│  Output: Focused context packet (~20K tokens vs ~500K)      │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

Detailed Component Specifications

1. Context Vault Schema

```
ContextVault/
├── specifications/
│   ├── user-stories.json       # Structured story data
│   ├── acceptance-criteria.json
│   ├── domain-rules.json       # Business logic constraints
│   └── constraints.json        # Technical constraints
├── codebase-index/
│   ├── file-manifest.json      # All files with metadata
│   ├── symbol-table.json       # Functions, classes, exports
│   ├── dependency-graph.json   # Import relationships
│   └── pattern-catalog.json    # Detected patterns
├── history/
│   ├── decisions.json          # Architectural decisions
│   ├── thread-summaries/       # Condensed conversation logs
│   └── implementation-log.json # What was built and why
└── active-context/
    ├── current-sprint.json
    ├── current-story.json
    └── working-set.json        # Files currently relevant
```

2. Recursive Task Decomposition

Task Analysis Protocol:

```
Given: User request + Context Vault access

Step 1: CLASSIFY task type
  - Specification (planning only)
  - Implementation (code changes)
  - Investigation (read-only analysis)
  - Hybrid (needs decomposition)

Step 2: DETERMINE information needs
  - Which user stories are relevant?
  - Which files need to be read?
  - What patterns exist in codebase?
  - What decisions were made before?

Step 3: SLICE context
  - Query vault for relevant slices
  - Compose minimal sufficient context
  - Estimate token budget

Step 4: DECOMPOSE if needed
  - Split into sub-tasks
  - Assign to specialized agents
  - Define information flow between agents

Step 5: EXECUTE with focused context
  - Each agent receives only its slice
  - Results aggregated by orchestrator
```

3. Agent Specialization

Agent Type	Context Slice	Output
Spec Agent		

Agent Type	Context Slice	Output
	User stories, domain rules, constraints	Refined specs, acceptance criteria
Architecture Agent	Codebase patterns, dependency graph, decisions	Implementation plan, file targets
Implementation Agent	Target files, related patterns, specific story	Code changes
Review Agent	Changed files, acceptance criteria, patterns	Validation results
Test Agent	Implementation, acceptance criteria	Test cases

4. Context Slicing Strategies

Strategy A: Story-Centric Slicing

```
Input: Single user story to implement
Slice:
  - The story + acceptance criteria
  - Files mentioned in previous similar stories
  - Patterns from related completed implementations
  - Relevant domain rules only
```

Strategy B: Investigation Slicing

```
Input: "How does X work in the codebase?"
Slice:
  - Symbol table entries matching X
  - Files containing X references
  - Related decision history
  - NO implementation context (read-only)
```

Strategy C: Sprint Slicing

```
Input: Sprint with multiple stories
Slice per story:
  - Individual story context
  - Shared sprint constraints
  - Cross-story dependencies only
Orchestrate:
  - Dependency-ordered execution
  - Shared context for related stories
```

Implementation Phases

Phase 1: Context Vault Foundation

- Structured storage for specifications
- Codebase indexing (files, symbols, patterns)
- History tracking for decisions

Phase 2: Basic Slicing Engine

- Query interface for vault
- Token budget estimation
- Simple relevance filtering

Phase 3: Recursive Orchestrator

- Task classification
- Agent spawning with sliced context
- Result aggregation

Phase 4: Intelligent Decomposition

- Automatic task breakdown
- Dependency detection
- Parallel execution where possible

Phase 5: Learning & Optimization

- Track which slices led to success
- Refine slicing heuristics
- Pattern recognition for task types

User Stories for RLM Architecture

Story 1: Context Vault Storage

Title: Structured Specification Storage

Description: As a developer, I want specifications stored in a queryable format so that the orchestrator can retrieve relevant context without loading everything.

Acceptance Criteria: 1. User stories stored as structured JSON with searchable fields 2. Acceptance criteria linked to parent stories 3. Domain rules stored with applicability conditions 4. Full-text search across all specification types 5. Retrieval by story ID, keyword, or related file

---

Story 2: Codebase Indexing

Title: Codebase Symbol and Pattern Index

Description: As the orchestrator, I want an index of the codebase so that I can identify relevant files without reading everything.

Acceptance Criteria: 1. File manifest with path, type, size, last modified 2. Symbol table with functions, classes, exports per file 3. Import/ dependency graph between files 4. Pattern catalog identifying common patterns (e.g., "plugin structure", "IPC handler") 5. Index updates on file changes

---

Story 3: Context Slicing

Title: Intelligent Context Slicing

Description: As a developer, I want the orchestrator to select only relevant context so that Claude Code receives focused, manageable prompts.

Acceptance Criteria: 1. Given a task, identify relevant user stories (not all) 2. Given a task, identify relevant files (not entire codebase) 3. Estimate token count before sending to Claude Code 4. Configurable token budget per agent type 5. Fallback to broader context if initial slice insufficient

---

Story 4: Recursive Task Decomposition

Title: Automatic Task Decomposition

Description: As a developer, I want complex tasks automatically broken into sub-tasks so that each can be handled with focused context.

Acceptance Criteria: 1. Tasks classified by type (spec, implementation, investigation, hybrid) 2. Hybrid tasks decomposed into typed sub-tasks 3. Dependencies between sub-tasks identified 4. Sub-tasks executed in dependency order 5. Results aggregated into coherent response

---

Story 5: Specialized Agent Routing

Title: Route Sub-tasks to Specialized Agents

Description: As the orchestrator, I want sub-tasks routed to specialized agents so that each agent has optimized context for its role.

Acceptance Criteria: 1. Spec agent receives only specification-related context 2. Implementation agent receives target files and patterns 3. Test agent receives implementation and acceptance criteria 4. Each agent's context fits within token budget 5. Agent outputs feed back to orchestrator for aggregation

---