Architecture Decision Records (ADRs) - Puffin Project

This document contains architecture decision records extracted from the Puffin project history.

Core Architecture

ADR-001: SAM Pattern for State Management

Status: Accepted (Core)

Context: Puffin needed a predictable, debuggable state management system for an Electron-based GUI application managing complex, multi-turn conversations with Claude.

Decision: Adopt the SAM (State-Action-Model) pattern as the canonical state architecture: - **State** - Pure representation of current state - **Actions** - Pure functions proposing state changes - **Model** - Acceptors that validate/enforce business rules on proposals - **State Computers** - Compute derived state from model

Rationale: - Unidirectional data flow (User Intent → Action → Model → State → View) prevents circular dependencies - Functional decomposition enables testing - Time-travel debugging support built-in - Clear separation between representation (State), proposals (Actions), and validation (Model)

Consequences: - All state changes flow through SAM loop: View → Action → Acceptor → State update → Computed state - Async operations must use promises within action proposals - Plugin state extensions use same acceptor/computer pattern - Logging configured with `[SAM]` prefixes for debugging

---

ADR-002: Dual Claude Strategy (3CLI + Direct API)

Status: Accepted

Context: Puffin needs both agentic code generation (file reading, tool use, building) and lightweight ancillary assistance (questions, research) without distracting the main conversation.

Decision: Use 3CLI (Claude Code CLI) as primary builder, with optional direct Claude API for secondary tasks.

Rationale: - 3CLI is the source of truth for building; Puffin orchestrates it rather than replacing it - Direct API keeps ancillary work from cluttering the main conversation history - Separation maintains clear responsibilities and prevents context bloat - 3CLI features (tool use, bash, git) don't apply to lightweight queries

Consequences: - Must manage two separate Claude interfaces - Requires clear protocol for which tasks use which interface - Allows more context tokens for main conversation

---

ADR-003: Directory-Based Workflow (`.puffin/` Directory)

Status: Accepted

Context: Puffin needs to persist application state (history, configuration, sprints) somewhere accessible but separated from project code.

Decision: Create a `.puffin/` directory in each project root as the single source of truth for all Puffin state.

Rationale: - State is tied to a specific project, not global or user-level - Directory-in-project pattern allows team collaboration (state can be committed to version control if desired) - Clear boundary between project code and Puffin metadata - Automatic persistence without explicit save/load UI

Consequences: - `.puffin/` should typically be `.gitignore`d (personal state, not shared) - Project changes require project reload to pick up new structure - All file I/O must be proxied through IPC for Electron security

---

Data Persistence

ADR-004: Multi-Database Isolation (Per-Project SQLite)

Status: Accepted

Context: Puffin supports multiple projects simultaneously, each with its own user stories, sprints, and history. The original JSON file-based storage lacked schema versioning and query capability.

Decision: Migrate to SQLite with a per-project database model: each project gets its own `.puffin/puffin.db` file.

Rationale: - No shared global database avoids complex multi-tenant logic - Single-file SQLite database is portable and requires no server - Each project has complete data isolation - Schema versioning via migrations supports future extensions - JSON files retained as backup alongside SQLite

Consequences: - Increased complexity in database initialization on project load - Each project database must be independently migrated - Requires repository pattern abstraction layer for all data access - Better scalability and query performance for large sprint histories

---

ADR-005: Atomic Transactions for Data Consistency

Status: Accepted

Context: SQLite operations across related entities (sprints, stories, sprint_history) needed to maintain consistency without partial failures.

Decision: Use `immediateTransaction()` wrapper from BaseRepository: - All multi-step operations within single transaction - Read data inside transaction (not before) to prevent TOCTOU - Validate foreign keys before inserts

Rationale: SQLite transactions ensure all-or-nothing semantics; prevents race conditions between read and write; provides consistency guarantees for master-detail relationships.

Consequences: - Longer transactions increase lock duration (mitigated by WAL mode) - All repository methods must be transaction-aware - Simplifies error recovery (rollback automatic)

ADR-006: Project-Level Plugin Data Storage

Status: Accepted

Context: Plugins and core app need to persist data across sessions without cross-project interference.

Decision: Store plugin data in `.puffin/plugins/[pluginName]/` directory structure; core app data in `.puffin/` root; never store project-specific data in localStorage.

Rationale: - Project-level storage prevents interference when Puffin switches between projects - localStorage scoped to browser profile causes cross-project pollution - Explicit filesystem structure easy to backup/migrate

Consequences: - Requires file system permissions - Data format must be explicitly versioned - Cleaning up old data requires manual intervention

---

Plugin System

ADR-007: Plugin System Architecture (Microkernel Pattern)

Status: Accepted (Phase 1 Implemented)

Context: Puffin needed to become extensible beyond its core features without increasing coupling or making the monolith harder to maintain.

Decision: Adopt a plugin architecture using a Microkernel pattern with: - **PluginContext API** - provides actions, acceptors, components, IPC handlers, storage, and UI registration - **PluginLoader** - discovers and loads plugins from `.puffin/plugins/` directory - **Plugin lifecycle** - activate/deactivate hooks with manifest-based metadata - **Extension points** - SAM pattern integration, UI components, IPC handlers, sidebar items, modals

Rationale: Enables external plugins without modifying core code; follows existing patterns in VS Code and Eclipse IDE; supports gradual extraction of existing features into plugins.

Consequences: - Adds abstraction layer with PluginContext and PluginRegistry - Requires plugins to follow contract (manifest.json, activate/deactivate methods) - Phase-based rollout: Phase 1 (core), Phase 2 (feature extraction), Phase 3 (marketplace)

ADR-008: Plugin View Registration via IPC

Status: Accepted

Context: Plugins need to register UI views but plugin code runs in renderer process while view registry must be in main process.

Decision: Use IPC handlers (`plugin:register-view`, `plugin:unregister-view`) with ViewRegistry class in main process; push registration events back to renderer.

Rationale: Main process as single source of truth prevents race conditions; event-driven updates allow multiple renderers to stay in sync; IPC provides security boundary.

Consequences: - Requires serialization of view objects - Async communication adds latency - IPC channel names must be carefully versioned

---

ADR-009: Plugin Event Broadcasting

Status: Accepted

Context: Plugins need to emit events that external systems listen to (e.g., Claude Config Plugin updates branch focus).

Decision: Two-level event emission: 1. Plugin calls `context.emit('event-name', data)` within plugin context 2. PluginRegistry forwards to EventEmitter: `registry.emitPluginEvent('plugin-event', event)` 3. External listeners subscribe: `registry.on('plugin-event', handler)`

Rationale: Decouples plugin from listeners; allows multiple listeners; prevents tight coupling between plugin and service layers.

Consequences: - PluginRegistry extends EventEmitter - Events not namespaced by plugin (future improvement) - Requires careful listener cleanup

---

ADR-010: Plugin Stylesheet Injection Strategy

Status: Accepted

Context: Plugins provide CSS that must be loaded into the document but need isolation.

Decision: Create `StyleInjector` class that creates `<link>` elements with `data-plugin` attributes; register styles in plugin manifest under `renderer.styles`; cleanup on disable by removing matching elements.

Rationale: DOM-based injection allows graceful fallback; `data-plugin` attribute enables efficient cleanup; manifest-based registration keeps style list with metadata.

Consequences: - CSS naming collisions possible (mitigated by convention) - Style loading is asynchronous - Plugin developers responsible for CSS scoping

---

User Interface

ADR-011: Component-Based UI with Modal Manager

Status: Accepted

Context: Multiple UI components needed modal dialogs for forms, avoiding duplication.

Decision: Centralized **ModalManager** that: - Registers modal types with render functions - Handles open/close lifecycle - Manages stack for nested modals - Coordinates with state via SAM intents

Rationale: Single responsibility for modal lifecycle; reusable across components; decouples trigger logic from display logic.

Consequences: - 982+ lines in modal-manager.js for all modal definitions - Modal state separate from page state - Requires careful event binding to avoid memory leaks

---

ADR-012: Animation Transitions with Accessibility

Status: Accepted

Context: User story view switches between layouts; needed smooth transitions without motion-sensitive issues.

Decision: CSS transitions with accessibility: - 250ms duration - `requestAnimationFrame` coordination - `@media (prefers-reduced-motion: reduce)` for accessibility - Staggered card animations for visual flow

Rationale: Smooth animations improve UX perception; respects accessibility needs; RAF ensures no jank.

Consequences: - ~80 lines CSS + ~30 lines JS - Total transition ~375ms - No performance impact (animations purely CSS)

ADR-013: Contextual Branch Buttons (Per-Story)

Status: Accepted

Context: After sprint plan approval, developers need quick branch access without UI clutter.

Decision: Show contextually relevant branch buttons (UI, Backend, Full Stack) below each story, hidden until sprint status is `planned`.

Rationale: - Reduces UI clutter by showing only relevant branches - Dynamically showing based on sprint status prevents confusion - Improves workflow: plan sprint → see branches → click to start

Consequences: - Requires dynamic button rendering based on sprint status - Full Stack button reuses active branch - Main branch tabs still needed for other types

---

Document Editor

ADR-014: JSON Change Format for Document Editor

Status: Accepted

Context: The Document Editor Plugin used markdown `<<<CHANGE>>>` blocks for edits. This format was error-prone and difficult to parse.

Decision: Switch to structured JSON format with legacy markdown fallback:

```
{
  "changes": [
    {"op": "replace", "find": "...", "content": "..."}
  ]
}
```

Rationale: - JSON is machine-parseable without ambiguity - Structured format captures metadata (line numbers, change type) - Legacy fallback ensures graceful degradation

Consequences: - DocumentMerger parses both JSON and markdown formats - Response parsing more complex but more reliable - Claude needs new format examples in prompts

---

ADR-015: Inline Prompt Marker Syntax

Status: Accepted

Context: Users needed to embed Claude instructions directly in documents without a separate prompt interface.

Decision: Use symmetric marker syntax `/@puffin: instruction @/`: - Opening: `/@puffin:` - Closing: `@/`

Rationale: - Symmetric pattern (`/@...@/`) is easy to spot - `@/` rarely used in code, reducing false matches - Universal format works in any text file

Consequences: - Visual highlighting with yellow background and penguin icon - Multiline support for complex instructions - Clean markers button removes all markers when done

---

Memory System (Proposed)

ADR-016: File-Based Memory Architecture (3-Layer System)

Status: Proposed (Ready for Implementation)

Context: Puffin conversations generate valuable knowledge that's lost when conversations end, causing repeated discussions.

Decision: Implement a three-layer memory system: - **Layer 1 (Resources)**: Raw branch conversations, immutable and timestamped - **Layer 2 (Items)**: Extracted atomic facts with confidence scores - **Layer 3 (Categories)**: Human-readable markdown summaries

Rationale: - Immutable resources enable audit trails - Atomic items with confidence scores support conflict detection - Category summaries provide accessible knowledge representation - Tiered retrieval minimizes token usage

Consequences: - Requires LLM extraction pipeline for each conversation - Memory management overhead (cleanup, conflict resolution) - User must review contradictory memories - Enables knowledge reuse across projects

---

Security

ADR-017: XSS Prevention Strategy

Status: Accepted

Context: User-provided content (branch names, story descriptions) could contain malicious scripts.

Decision: Create centralized `utils/escape.js` module with `escapeHtml()` and `escapeAttr()` functions; apply escaping at multiple layers.

Rationale: Centralization prevents duplication; enables code reuse; defensive escaping at multiple layers.

Consequences: - Adds utility module - Slight performance overhead (negligible) - Requires thorough testing of edge cases

---

ADR-018: Memory Leak Prevention in Event Listeners

Status: Accepted

Context: Long-running sessions accumulated event listeners causing memory leaks.

Decision: Add listener tracking with `boundListeners[]` and `documentListeners[]` arrays; implement cleanup methods called from `destroy()` and on re-render.

Rationale: Explicit tracking enables controlled cleanup; separate tracking for element vs document listeners allows targeted removal.

Consequences: - Adds state tracking overhead - Requires discipline to call cleanup at lifecycle points - Testing needed for edge cases (rapid interactions)

Sprint Execution

ADR-019: Sprint Execution Control Flow

Status: Accepted

Context: Sprints need iteration limits, auto-continue timers, stuck detection, and story limits to prevent runaway execution.

Decision: Create unified `sprintExecution` state with four coordinated components: 1. **Iteration Counter**: Tracks current/max iterations 2. **Auto-Continue Timer**: Implements delay countdown 3. **Stuck Detection**: Compares response similarity (0.85 threshold) 4. **Story Limit Validator**: Prevents sprints with >4 stories

Rationale: - All features share common state infrastructure - Prevent rather than reject approach (disable UI before invalid state) - Iteration history enables stuck detection

Consequences: - Single sprintExecution state object manages all concerns - Story limit enforced before sprint creation - UI reflects live iteration counter and countdown

---

ADR-020: Sprint/User Story Persistence Refactoring

Status: In Progress

Context: Sprint and user story persistence had critical bugs (non-atomic operations, TOCTOU bugs, orphaned references).

Decision: Three-phase refactoring: 1. **Phase 1** - Make sprint close atomic; validate story IDs; fix TOCTOU bugs 2. **Phase 2** - Add title field; implement bidirectional status sync 3. **Phase 3** - Create SprintService layer; consolidate IPC handlers

Rationale: Prevents data loss by ensuring atomic transactions; validates foreign keys; fixes race conditions.

Consequences: - Database migration for existing null titles - New SprintService abstraction - All operations use `immediateTransaction()`

RLM Integration

ADR-021: Context Management Strategy for Long-Context Reasoning

Status: Accepted (RLM Enhancement)

Context: Processing long-context reasoning (2M+ tokens) for multi-file codebases and project histories requires a strategy that balances token efficiency, semantic accuracy, and computational cost.

Options Discussed:

1. **Direct Prompt Injection** (Traditional)
2. Concatenate all context into a single prompt

3. Cons: Deteriorates with length, loses semantic structure, attention degrades over 100K+ tokens

4. **RAG (Retrieval-Augmented Generation)** (Semantic Search)

5. Use embedding-based similarity to fetch relevant context

6. Cons: Misses structural dependencies (contracts, code cross-references)

7. **REPL-Based External Environment** (Chosen)

8. Store context as queryable variables in Python REPL

9. Agent writes code to index, search, follow references, verify
10. Recursion chases dependency graphs, building evidence trees

Rationale: - MIT CSAIL RLM paper validates 100x context window scaling beyond base model limits - Separates length (token count) from complexity (information density) - Structural dependencies can be programmatically resolved - Recursive sub-calls prevent attention fragmentation - Cost efficient: often lower than simpler long-context scaffolds

Consequences: - Requires REPL environment setup in orchestration layer - Agent must write queries (Python code) to explore context - Implementation complexity increases but becomes manageable with proper abstractions - Enables handling of truly large artifacts (multi-thousand-line codebases)

ADR-022: Plugin-Based Designer Storage Architecture

Status: Accepted (RLM Enhancement)

Context: GUI definitions must be stored in a way that: 1. Prevents namespace collisions (unique names per design) 2. Remains portable across projects 3. Integrates with plugin architecture for extensibility 4. Separates designer ownership from core state

Options Discussed:

1. **Core `.puffin/gui-definitions/` Directory**
2. Simple, readily available in puffin-state.js

3. Cons: Couples GUI storage to core, no namespace isolation

4. **Designer Plugin Namespace** (Chosen)

5. Stores definitions under plugin ownership: `.puffin/plugins/designer/designs/`
6. Implements namespace enforcement (unique names within plugin)
7. Designer Plugin manages schema evolution independently

Rationale: - Designer Plugin becomes first-class owner of GUI definition lifecycle - Namespace enforcement via Plugin schema prevents collisions - Enables future plugins to store complementary design artifacts - Demonstrates plugin pattern as viable architecture

Consequences: - Designer Plugin must be created before other plugins reference GUI definitions - PluginStateStore requires initialization before GUI operations - Migration path needed if core currently stores definitions

---

ADR-023: Multi-Select GUI Definition Inclusion

Status: Accepted (RLM Enhancement)

Context: When composing prompts, users need to include multiple GUI reference materials for context. Strategy must support reusing previously saved definitions, current work, and combining them.

Options Discussed:

1. **Single Selection Only**
2. Choose one GUI definition per prompt
3. Cons: Can't reference multiple design variants
4. **Current Design Only**
5. Auto-include active GUI designer state
6. Cons: Can't reference past saved states
7. **Multi-Select with Current Design Toggle** (Chosen)
8. Checkbox dropdowns for saved GUI definitions
9. Separate toggle for current in-progress design
10. Combined descriptions concatenated in prompt context

Rationale: - Real-world use case: comparing current design against approved versions - Current design toggle addresses in-progress work without saving - Prompt-editor already implements this pattern - Enables comprehensive UI design context (responsive variants, states, etc.)

Consequences: - Increases UI complexity in prompt-editor dropdowns - State: `selectedGuiDefinitions` array in prompt model - Requires careful ordering in combined descriptions

---

ADR-024: Hierarchical Context Vault Organization

Status: Accepted (RLM Enhancement)

Context: Context Vault stores specifications, codebase indexes, traceability, history, and active context. Organization must enable efficient slicing and queries without coupling domains.

Decision:

```
ContextVault/
├── specifications/         (user stories, domain rules, assumptions)
├── codebase-index/         (files, symbols, dependencies, patterns)
├── traceability/           (story-to-code, code-to-tests, impact analysis)
├── quality-gates/          (inspection assertions, deliberation triggers)
├── history/                (decisions, thread summaries, deployment)
└── active-context/         (current sprint, working set, risk hotspots)
```

Rationale: - Four domains separate concerns (specs ≠ code ≠ traceability ≠ history) - Hierarchical structure maps to RLM query patterns - Each domain independently queryable and sliceable - Scale-friendly: add new files without restructuring

Consequences: - Requires vault initialization on first RLM setup - Slicing engine must understand domain relationships - Migration tooling needed if moving from flat schemas

---

ADR-025: History Index Generation Strategy

Status: Accepted (RLM Enhancement)

Context: RLM queries need to efficiently navigate large histories (2M+ tokens). Pre-computed indexes prevent O(n) scans and enable targeted recursion.

Options Discussed:

1. **On-Demand Generation**

2. Generate index when requested

3. Cons: High latency, duplicated work

4. **Continuous Real-Time Generation**

5. Update index on every prompt/response

6. Cons: Overhead on every interaction

7. **Sprint-Based + Periodic Checkpoints** (Chosen)

8. Generate comprehensive index at sprint end
9. Lightweight periodic checkpoints (daily/weekly) for recent history
10. Index structure: topic tags, session boundaries, byte offsets

Rationale: - Sprint boundaries are natural semantic breakpoints - Reduces indexing overhead (not on every interaction) - Two-tier approach: heavy analysis at sprint end, light updates in-between - Enables RLM queries like "find all stories about authentication in previous sprints"

Consequences: - Requires HistoryIndex file alongside history.json - Sprint closing workflow must include index generation - Recent history requires linear scan (acceptable)

ADR-026: RLM Parallel Execution Strategy

Status: Accepted (RLM Enhancement)

Context: RLM's recursive sub-calls can explore independent branches of a dependency graph. Orchestrator must determine when parallelization is safe and beneficial.

Options Discussed:

1. **Fully Sequential Execution**
2. Process sub-calls one at a time

3. Cons: Underutilizes API rate limits

4. **Aggressive Parallelization**

5. Spawn all possible sub-calls immediately
6. Cons: Rapid rate limit exhaustion, uncontrolled token spend
7. **Smart Parallelization with Dependency Detection** (Chosen)
8. Analyze task dependency graph
9. Parallelize independent branches
10. Serialize dependent branches

Rationale: - Balances latency vs cost (parallel on independent paths, serial on dependencies) - Respects API rate limits through controlled concurrency - Matches real-world development patterns

Consequences: - Orchestrator must implement dependency graph analysis - Aggregation logic handles varying completion times - Requires careful testing of dependency detection

---

ADR-027: RLM Result Aggregation Strategy

Status: Accepted (RLM Enhancement)

Context: When RLM executes parallel sub-calls or multiple recursive levels, results must be combined coherently into a single response.

Decision: Final Aggregation Only - Collect all results, wait for all sub-calls to complete - Merge results in priority order (evidence quality, recency) - Synthesize into final response - Single transmission to user

Rationale: - Coherent result presentation (not fragmented updates) - Ability to reorganize results by quality - Easier to detect and handle contradictions - Matches SAM pattern (compute full state before rendering view)

Consequences: - User sees results only after all sub-calls complete (higher latency) - Orchestrator must buffer all sub-results (memory tradeoff) - Conflict resolution required if sub-calls produce contradictory evidence

---

ADR-028: Nested JSON Format for GUI Definitions

Status: Accepted (RLM Enhancement)

Context: GUI definitions stored in Designer Plugin must support serialization, semantic meaning, evolution, and traversal for RLM queries.

Decision: Nested JSON with Metadata

```
{
  "name": "...",
  "description": "...",
  "elements": [
    {"type": "...", "id": "...", "properties": {...}, "children": [...]}
  ],
  "metadata": {
    "timestamp": "...",
    "version": "...",
    "author": "...",
    "tags": [...]
  }
}
```

Rationale: - Matches gui-designer.js internal representation - Metadata enables audit trail - Tagging enables RLM queries like "find designs tagged with 'form'" - Version field decouples schema evolution

Consequences: - Requires schema version field for backward compatibility - Larger designs increase file size (manageable) - RLM indexing benefits from semantic structure

---

ADR-029: Fully Qualified IPC Channel Naming

Status: Accepted (RLM Enhancement)

Context: IPC channels connect renderer components to main process handlers. With multiple plugins, channel names must be unambiguous and prevent collisions.

Decision: Format: `plugin:<plugin-name>:<channel-name>` - Example: `plugin:designer:save`, `plugin:designer:load` - Central registry ensures plugin names are unique

Rationale: - Unambiguous in global namespace (no collisions) - Enables plugin discovery via naming pattern matching - Scales with unlimited plugins - Simple parsing: split on `:` to extract plugin name

Consequences: - Channel names are longer (minor verbosity) - IPC handler validates plugin existence - Logging and debugging benefit from explicit naming

---

ADR-030: Tiered Model Selection for RLM Agents

Status: Accepted (RLM Enhancement)

Context: RLM architecture spans multiple agents (root orchestrator, specialized agents, sub-calls). Each has different complexity and cost profiles.

Decision: - Root orchestrator: Claude Opus (complex task decomposition) - Specialized agents: Claude Opus (spec, implementation, test agents) - Sub-call recursive: Claude Haiku (targeted queries, evidence gathering)

Rationale: - Root orchestrator handles complex decomposition (needs Opus) - Recursive sub-calls are narrow in scope (Haiku sufficient) - Cost: Heavy hitters use Opus; work horses use Haiku - Total cost often lower than all-Opus

Consequences: - Must configure model selection per agent type - Sub-call responses may be less nuanced (acceptable for evidence) - Requires testing to validate Haiku sufficiency

---

Summary

The Puffin project architecture is built on these key principles:

1. **Predictability** - SAM pattern provides clear, traceable state management
2. **Isolation** - Per-project databases and storage prevent cross-project interference
3. **Modularity** - Plugin system enables extensibility without core changes
4. **Security** - XSS prevention, IPC boundaries, and careful event cleanup
5. **User Experience** - Animated transitions, contextual UI, accessibility support

6. **Long-Context Reasoning** - REPL-based external environment with hierarchical vault for efficient 2M+ token processing
7. **Extensibility** - Designer Plugin pattern demonstrates plugin ecosystem for domain-specific tools

These decisions have resulted in a maintainable, debuggable system suitable for complex multi-turn AI orchestration with enterprise-scale long-context reasoning capabilities.