Memory Plugin Design Specification

Version: 0.1.0 (Draft)
Status: Ready for Implementation
Plugin Name: `memory-plugin`

1. Overview

1.1 Purpose

The Memory Plugin is a Puffin plugin that automatically processes branch threads at regular intervals to extract, categorize, and memorize key decisions made during development sessions. It transforms ephemeral conversation history into structured, persistent knowledge that can be retrieved and used to provide context in future sessions.

1.2 Problem Statement

Development conversations in Puffin generate valuable knowledge: - Architectural decisions and their rationale - Bug fixes and root cause analyses - Feature implementation approaches - User preferences and project conventions

Without a memory system, this knowledge is lost when conversations end, leading to: - Repeated discussions of the same topics - Inconsistent decision-making - Loss of institutional knowledge about the project

1.3 Goals

1. **Automatic Extraction**: Process branch threads without manual intervention
2. **Decision Focus**: Identify and prioritize key decisions over routine exchanges

3. **Structured Storage**: Organize memories in a hierarchical, queryable format
4. **Intelligent Retrieval**: Surface relevant memories based on current context
5. **Memory Maintenance**: Implement decay and consolidation to prevent knowledge rot

---

2. Architecture

2.1 Memory Architecture Model

Following the principles from the Memory Architecture reference, this plugin implements a **File-Based Memory** system with three layers:

Layer 1: Resources (Raw Data - Source of Truth)

- Store raw branch thread conversations and context
- Immutable and timestamped for full traceability
- Enables replay, debugging, and audit trails
- Location: `~/.puffin/memory/<user-id>/resources/`
- Format: JSON with metadata (timestamp, branch, thread-id, session-id)

Layer 2: Items (Atomic Facts)

- Discrete, extracted facts from conversations
- Examples: "User prefers async/await over promises", "Architecture decision: Use event-driven pattern"
- Linked to source resources for full traceability
- Extracted automatically via LLM from raw conversations
- Location: `~/.puffin/memory/<user-id>/items/`
- Format: JSON with category, content, source-resource-id, extraction-timestamp

Layer 3: Categories (Evolving Summaries)

- High-level, human-readable knowledge organized by topic

- Examples: `coding_preferences.md`, `architectural_decisions.md`, `project_conventions.md`
- Actively synthesized from items to handle contradictions and updates
- Location: `~/.puffin/memory/<user-id>/categories/`
- Format: Markdown files, updated via LLM-driven synthesis

2.2 Write Path: Active Memorization

When processing a branch thread:

1. **Resource Ingestion**: Save raw conversation to resources layer (immutable)
2. **Extraction**: Use LLM to extract atomic facts from conversation
3. Prompt focuses on decisions, preferences, architectural choices
4. Returns structured JSON with extracted items
5. **Batching**: Group extracted items by category to minimize file I/O
6. Structure: `{ "coding_preferences": [...], "architectural_decisions": [...], ... }`
7. **Item Storage**: Save individual items with source resource links
8. **Category Evolution**: Load existing category summary, integrate new items via LLM
9. LLM rewrites summary to incorporate new information
10. Handles conflicts by overwriting outdated facts
11. Maintains narrative coherence and context
12. **Metadata**: Track extraction timestamp, confidence scores, and access counts

Example Extraction Prompt: Given a branch thread conversation, identify and extract: - **User Preferences**: Coding style, library choices, patterns favored - **Architectural Decisions**: Design patterns adopted, trade-offs accepted - **Project Conventions**: Naming conventions, file organization, tooling preferences - **Bug Patterns**: Recurring issues and their fixes - **Feature Implementation Notes**: Approaches tried, lessons learned

Return as JSON:

```
{
  "coding_preferences": [
    {
      "content": "User prefers async/await pattern",
      "confidence": 0.95,
      "context": "Mentioned in discussion about Promise handling"
    }
  ],
  "architectural_decisions": [
    {
      "content": "Adopt event-driven pattern for state management",
      "rationale": "Decouples components and improves testability",
      "trade_offs": "Requires careful event sequencing to prevent race conditions",
      "confidence": 0.90
    }
  ],
  "project_conventions": [
    {
      "content": "Use camelCase for variable names",
      "scope": "JavaScript files",
      "confidence": 0.95
    }
  ],
  "bug_patterns": [
    {
      "pattern": "Race conditions in async state updates",
      "fix": "Use explicit locking mechanisms",
      "occurrences": 2,
      "confidence": 0.85
    }
  ],
  "implementation_notes": [
    {
      "topic": "Component refactoring",
      "approach": "Use composition over inheritance",
      "lessons_learned": "Reduces coupling and improves reusability",
      "confidence": 0.88
    }
  ]
}
```

2.3 Read Path: Intelligent Retrieval

The memory system uses tiered retrieval to minimize token consumption while providing relevant context:

1. **Query Synthesis**: Transform user query into a search-optimized form
2. Filter out noise and convert to semantic keywords
3. Identify query intent (asking for a decision, preference, or pattern)
4. **Category Selection**: Use LLM to identify relevant category files
5. Load only summaries from categories likely to contain answers
6. Skip categories that don't match query intent
7. **Sufficiency Check**: Determine if category summaries are enough
8. If summaries answer the query comprehensively → Return them

9. If insufficient or ambiguous → Proceed to atomic items search

10. Perform explicit content-based assessment rather than assuming coverage

11. **Hierarchical Search**: Fall back to more granular searches

12. Search atomic items (extracted facts) if summaries are vague or incomplete
13. Search resources (raw conversations) only as last resort for full context

14. Maintain audit trail of retrieval depth for transparency

15. **Conflict Detection and Resolution**: Address contradictions across memory sources

16. Identify similar memories from different sources that contradict each other
17. Highlight conflicting entries with source information and extraction timestamps
18. Present conflicts to user for manual resolution when confidence is split

19. Flag outdated memories that conflict with newer, higher-confidence facts

20. **Relevance Filtering**: Apply multi-factor scoring with flexible thresholds

21. Rank results by recency, relevance, and access frequency
22. Apply adaptive relevance thresholds based on task context (default: 0.7, adjustable 0.5–0.9)
23. Prioritize frequently-accessed memories that remain valid

24. Support domain-specific threshold customization for different memory categories

25. **Temporal Constraints**: Validate facts against temporal validity windows

26. Identify and flag memories with explicit expiration dates

27. Invalidate outdated facts superseded by newer information
28. Support validity windows (e.g., "valid until Dec 2025") for time-sensitive facts

29. Exclude memories outside their valid temporal range from results

30. **Context Assembly**: Select memories that fit within token budget

31. Sort by final score (relevance × time decay × confidence)
32. Pack top-scoring memories until token limit reached
33. Include timestamp, confidence, and validity window metadata
34. Ensure conflict information is surfaced when relevant

Temporal Decay Formula:

```
final_score = relevance_score × time_decay_factor
time_decay_factor = 1.0 / (1.0 + (age_in_days / half_life_days))
```

Where `half_life_days` is configurable (default: 30). This ensures recent memories are prioritized while still allowing access to older insights.

2.4 Memory Maintenance

To prevent knowledge rot and maintain system health, the plugin implements scheduled maintenance tasks:

Nightly Consolidation (Daily at 3 AM)

- Review conversations from the past 24 hours
- Identify and merge redundant or contradictory memories
- Promote frequently-accessed items to higher priority
- Remove low-confidence duplicates

Weekly Summarization (Every Monday at 2 AM)

- Compress memories older than 30 days into higher-level insights
- Archive infrequently-accessed items
- Prune memories not accessed in 90 days
- Update category summaries based on consolidated items

Monthly Re-indexing (First day of month at 1 AM)

- Regenerate embeddings with latest LLM model
- Reweight graph relationships based on actual usage patterns
- Archive unused nodes not touched in 180 days
- Optimize storage layout for faster retrieval

Future Enhancement: Advanced Memory Evolution (Out of Scope for v1)

The following sophisticated memory evolution mechanisms are planned for future releases:

Consolidation Enhancements: - **Cluster-level fusion**: Group semantically related memories into coherent clusters - **Global integration**: Synthesize contradictory facts into resolved summaries - **Hierarchical abstraction**: Consolidate repeated patterns into higher-level insights

Updating Enhancements: - **Conflict resolution strategy**: Explicit handling when new information contradicts existing memories - **Temporal validity windows**: Support time-bounded facts with explicit expiration - **Soft updating**: Decay weights rather than hard replacements

Forgetting Enhancements: - **Frequency-based forgetting**: LRU-style pruning for rarely accessed memories - **Importance-driven forgetting**: Consider semantic value, not just recency - **Soft forgetting**: Gradual decay weights rather than hard deletion

---

3. Plugin Implementation

3.1 Plugin Structure

```
plugins/memory-plugin/
├── main.js                 # Main process initialization and IPC handlers
├── renderer.js             # Renderer process UI integration
├── config/
│   └── default-config.json # Default configuration
├── lib/
│   ├── memory-manager.js   # Core memory operations
│   ├── llm-extractor.js    # LLM-based extraction logic
│   ├── retriever.js        # Tiered retrieval system
│   └── maintenance.js      # Scheduled maintenance tasks
└── README.md               # Plugin documentation
```

3.2 Main Process (`main.js`)

The main process plugin: - Initializes the memory storage system - Registers IPC handlers for memory operations - Schedules maintenance tasks (cron jobs) - Manages file system access to memory directories

Key Responsibilities: - Create memory directories on first run - Load configuration from `config/default-config.json` - Set up scheduled tasks (nightly, weekly, monthly) - Handle all file I/O operations - Validate IPC inputs before processing

3.3 Renderer Process (`renderer.js`)

The renderer process plugin: - Provides UI indicators for memory system status - Displays memory statistics (total items, categories, last update) - Allows manual memory review and editing - Shows retrieval confidence scores in tooltips - Implements conflict visualization highlighting contradictory memories for user resolution - Displays side-by-side comparison of conflicting facts with source, timestamp, and confidence metadata - Provides user interface for manual conflict resolution (accept newer, keep both, mark as resolved) - Color-codes conflicting entries to improve visual discoverability - Tracks resolution history and reasoning for audit purposes - Supports batch resolution for multiple related conflicts

3.4 Configuration

File: `plugins/memory-plugin/config/default-config.json`

```json
{
  "enabled": true,
  "storage": {
    "basePath": "~/.puffin/memory",
    "layers": {
      "resources": "resources",
      "items": "items",
      "categories": "categories"
    }
  },
  "extraction": {
    "enabled": true,
    "minConfidence": 0.75,
    "categories": [
      "coding_preferences",
      "architectural_decisions",
      "project_conventions",
      "bug_patterns",
      "implementation_notes"
    ]
  },
  "retrieval": {
    "maxTokens": 2000,
    "relevanceThreshold": 0.7,
    "timeDecayHalfLifeDays": 30,
    "timeDecayFunction": "exponential"
  },
  "maintenance": {
    "nightly": {
      "enabled": true,
      "schedule": "0 3 * * *",
      "description": "Daily consolidation"
    },
    "weekly": {
      "enabled": true,
      "schedule": "0 2 * * 1",
      "description": "Weekly summarization"
    },
    "monthly": {
      "enabled": true,
      "schedule": "0 1 1 * *",
      "description": "Monthly re-indexing"
    }
  }
}
```

---

4. API Specification

4.1 IPC Channels

All IPC channels are prefixed with `memory:` to maintain plugin namespace isolation.

Memory Operations

`memory:memorize` - Direction: Renderer → Main - Request: `{ conversationText: string, branchId: string, sessionId: string }` - Response: `{ resourceId: string, itemsExtracted: number, categoriesUpdated: string[] }` - Purpose: Trigger memory extraction from a conversation

`memory:retrieve` - Direction: Renderer → Main - Request: `{ query: string, maxResults?: number }` - Response: `{ memories: Memory[], totalTokens: number, retrievalTime: number }` - Purpose: Retrieve relevant memories for a query

`memory:get-category` - Direction: Renderer → Main - Request: `{ category: string }` - Response: `{ category: string, content: string, lastUpdated: string, itemCount: number }` - Purpose: Load a specific category summary

Statistics and Monitoring

`memory:get-stats` - Direction: Renderer → Main - Request: `{}` - Response: `{ totalItems: number, totalCategories: number, totalResources: number, lastMemorized: string, nextMaintenance: string }` - Purpose: Get system-wide memory statistics

`memory:get-status` - Direction: Renderer → Main - Request: `{}` - Response: `{ status: 'ready'|'processing'|'error', lastError?: string, maintenanceInProgress: boolean }` - Purpose: Check plugin health and operational status

Administrative

`memory:clear-category` - Direction: Renderer → Main - Request: `{ category: string }` - Response: `{ success: boolean, itemsDeleted: number }` - Purpose: Clear all memories in a category (careful operation)

`memory:run-maintenance` - Direction: Renderer → Main - Request: `{ type: 'nightly'|'weekly'|'monthly'|'full' }` - Response: `{ success: boolean, duration: number, results: object }` - Purpose: Manually trigger maintenance tasks

4.2 Memory Object Structure

```
interface Memory {
  id: string;                    // Unique memory identifier
  category: string;              // Category classification
  content: string;               // Memory text
  sourceResourceId: string;      // Link to original resource
  extractedAt: string;           // ISO timestamp of extraction
  lastAccessed: string;          // ISO timestamp of last retrieval
  accessCount: number;           // Number of times retrieved
  confidence: number;            // Confidence score 0.0-1.0
  metadata: {
    branchId?: string;
    sessionId?: string;
    tags?: string[];
  };
}
```

4.3 Error Handling

All IPC handlers return structured error responses:

```
interface ErrorResponse {
  error: true;
  code: string;                  // Error code: EXTRACTION_FAILED, RETRIEVAL_FAILED, etc.
  message: string;               // Human-readable error description
  context?: any;                 // Additional debugging context
}
```

---

5. Integration with Puffin

5.1 Branch Thread Processing

When a branch completes or at regular intervals: 1. Memory plugin receives branch thread conversation 2. Extracts and stores memories automatically 3. Updates category summaries 4. Makes memories available for future retrieval

5.2 Context Injection for Claude Code

When Claude Code initializes for a new task: 1. Current context (user message, branch) triggers memory retrieval 2. Relevant memories are injected into system prompt 3. Claude has access to historical decisions and conventions 4. Improves consistency and reduces repeated discussions

5.3 UI Integration

The memory plugin renders: - Status indicator showing memory system health - Quick access to category summaries - Search interface for manual memory lookup - Statistics dashboard (memory growth, access patterns)

Document Status: Ready for Detailed Design

Appendix A: Implementation Readiness Checklist

- [x] Three-layer architecture defined (Resources, Items, Categories)
- [x] Write path with LLM extraction specified
- [x] Read path with tiered retrieval defined
- [x] Conflict detection and resolution mechanisms included
- [x] Temporal decay formula provided
- [x] Plugin structure outlined
- [x] IPC API channels specified
- [x] Configuration schema documented
- [x] Error handling patterns defined
- [x] Maintenance schedules specified
- [x] Detailed design steps documented

Appendix B: Detailed Design Steps

The following steps outline the recommended approach for moving from this specification to detailed design and implementation.

Step 1: Define Data Models and Schemas

1. **Resource Schema**: Define the JSON structure for raw conversation resources
2. Required fields: `id`, `timestamp`, `branchId`, `sessionId`, `conversationText`

3. Optional metadata fields for future extensibility

4. **Item Schema**: Define the structure for extracted memory items

5. Required fields: `id`, `category`, `content`, `sourceResourceId`, `extractedAt`, `confidence`

6. Index fields: `lastAccessed`, `accessCount`

7. **Category Summary Format**: Define the markdown template for category files

8. Header structure, metadata section, content organization

Step 2: Design the LLM Extraction Prompts

1. Create the extraction prompt template based on Section 2.2
2. Define the expected JSON response schema with validation rules
3. Design fallback prompts for handling edge cases (empty conversations, unclear content)
4. Create the category evolution prompt for integrating new items into existing summaries

Step 3: Implement Core Memory Manager

1. **File System Layer**
2. Directory creation and management
3. Atomic file write operations

4. File locking for concurrent access

5. **Write Path Implementation**

6. Resource ingestion with immutable storage
7. Batched item storage by category

8. Category summary evolution logic

9. **Read Path Implementation**

10. Query synthesis function
11. Category selection via LLM

12. Sufficiency checking logic
13. Hierarchical search fallback

Step 4: Implement Retrieval System

1. **Temporal Decay Calculator**: Implement the decay formula from Section 2.3
2. **Relevance Scoring**: Design the multi-factor scoring algorithm
3. **Conflict Detection**: Implement logic to identify contradictory memories
4. **Token Budget Manager**: Context assembly with size limits

Step 5: Design IPC Interface

1. Map each IPC channel to its handler function
2. Define input validation schemas for each channel
3. Implement error handling and response formatting
4. Create integration tests for each IPC endpoint

Step 6: Implement Maintenance Tasks

1. **Scheduler Setup**: Configure cron-style scheduling for maintenance
2. **Nightly Consolidation**: Implement redundancy detection and merging
3. **Weekly Summarization**: Implement compression and archival logic
4. **Monthly Re-indexing**: Placeholder for future embedding support

Step 7: Build Renderer UI Components

1. **Status Indicator**: Memory system health display
2. **Statistics Dashboard**: Memory metrics visualization
3. **Conflict Resolution UI**: Side-by-side comparison interface
4. **Manual Review Interface**: Memory browsing and editing

Step 8: Integration Testing

1. End-to-end tests for memorization workflow
2. Retrieval accuracy tests with sample queries

3. Conflict detection and resolution tests
4. Maintenance task verification
5. Performance benchmarks for large memory stores

Step 9: Documentation

1. Plugin README with installation instructions
2. API documentation for IPC channels
3. Configuration reference guide
4. Troubleshooting guide