# Central Reasoning Engine (CRE) — Detailed Design

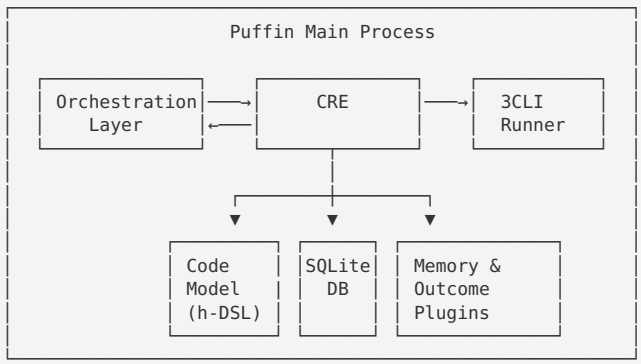
**Version:** 0.1 (First Iteration) **Status:** Draft **Specification:** See `CENTRAL_REASONING_ENGINE.md` **Depends on:** h-M3 v2, h-DSL Context Vault v3

---

## 1. Architectural Overview

### 1.1 Placement in Puffin

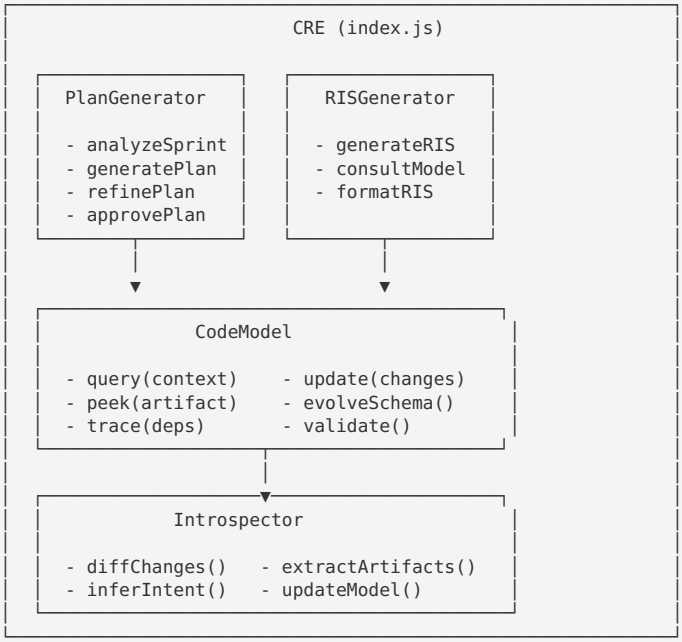
The CRE is a core module in Puffin's main process. It sits between the orchestration layer (which manages the user workflow) and the 3CLI execution layer.




### 1.2 Module Structure

```
src/main/cre/
├── index.js                # Public API, IPC registration
├── plan-generator.js       # Plan generation logic
├── ris-generator.js        # RIS generation logic
├── code-model.js           # h-DSL instance management
├── schema-manager.js       # h-DSL schema evolution
├── introspector.js         # Post-implementation code analysis
├── assertion-generator.js  # Inspection assertion generation
└── lib/
    ├── hdsl-types.js       # h-M3 primitive type definitions
    ├── hdsl-validator.js   # Schema/instance validation
    ├── context-builder.js  # Navigation context for code model queries
    └── ris-formatter.js    # RIS markdown formatting
```

### 1.3 Component Diagram




---

# 2. Component Design

## 2.1 CRE Entry Point (`index.js`)

The entry point initializes all sub-components and registers IPC handlers.

```
/**
 * @module CRE
 * Central Reasoning Engine - core Puffin module.
 * Provides plan generation, RIS generation, and code model maintenance.
 */
module.exports = {
  /**
   * Initialize the CRE with application context.
   * @param {Object} context - { ipcMain, app, db, config, projectRoot }
   */
  async initialize(context) { /* ... */ },

  /**
   * Graceful shutdown. Persists any pending state.
   */
  async shutdown() { /* ... */ }
};
```

### IPC Channels Registered:

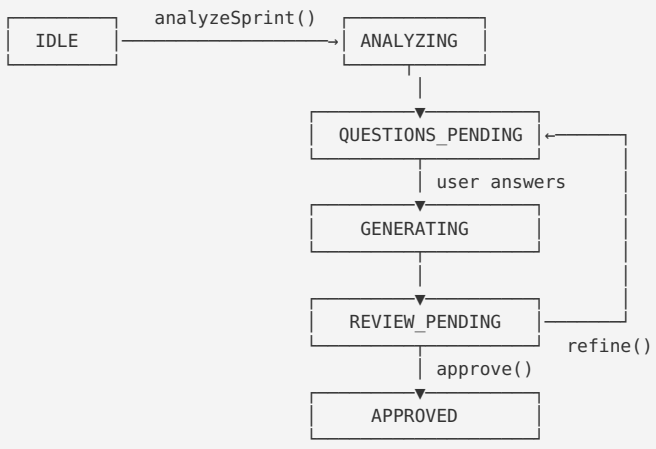
| Channel | Direction | Purpose |
| --- | --- | --- |
| | invoke | Start plan generation for a sprint |

| Channel | Direction | Purpose |
| --- | --- | --- |
| `cre:generate-plan` | | |
| `cre:refine-plan` | invoke | Submit user feedback for plan refinement |
| `cre:approve-plan` | invoke | Approve a plan, triggering assertion generation |
| `cre:generate-ris` | invoke | Generate RIS for a user story |
| `cre:update-model` | invoke | Trigger code model update after implementation |
| `cre:query-model` | invoke | Query the code model (for UI or debugging) |
| `cre:get-plan` | invoke | Retrieve a plan by sprint ID |
| `cre:get-ris` | invoke | Retrieve RIS by user story ID |

## 2.2 PlanGenerator (`plan-generator.js`)

Responsible for FR-01 through FR-06.

### State Machine



### Key Methods

```
class PlanGenerator {
  /**
   * Analyze sprint and user stories, identify ambiguities.
   * @param {string} sprintId
   * @returns {Object} { questions: string[], draft: Plan | null }
   */
  async analyzeSprint(sprintId) { /* ... */ }

  /**
   * Generate implementation plan using code model context.
   * @param {string} sprintId
   * @param {Object} answers - Answers to clarifying questions
   * @returns {Plan}
   */
  async generatePlan(sprintId, answers) { /* ... */ }

  /**
   * Refine plan based on user feedback.
   * @param {string} planId
   * @param {string} feedback - User's change request
   * @returns {Plan}
   */
  async refinePlan(planId, feedback) { /* ... */ }

  /**
   * Approve plan and generate inspection assertions.
   * @param {string} planId
   * @returns {Plan} - Plan with assertions populated
   */
  async approvePlan(planId) { /* ... */ }
}
```

### Plan Data Structure

```
/**
 * @typedef {Object} Plan
 * @property {string} id - UUID
 * @property {string} sprintId - FK to sprint
 * @property {string} status - idle|analyzing|questions_pending|generating|review_pending|approved
 * @property {PlanItem[]} items - Ordered implementation steps
 * @property {string[]} pendingQuestions - Unanswered clarifying questions
 * @property {Object} metadata - { createdAt, updatedAt, version }
 */

/**
 * @typedef {Object} PlanItem
 * @property {number} order - Execution sequence
 * @property {string} userStoryId - FK to user story
 * @property {string} branch - Git branch name
 * @property {string[]} dependsOn - IDs of prerequisite plan items
 * @property {string} summary - What this step accomplishes
 * @property {string} approach - How it will be implemented
 * @property {InspectionAssertion[]} assertions - Post-implementation checks
 */

/**
 * @typedef {Object} InspectionAssertion
 * @property {string} type - file_exists|function_exists|export_exists|pattern_match
 * @property {string} target - File path, function name, or pattern
 * @property {string} description - Human-readable description
 */
```

## 2.3 RISGenerator ( `ris-generator.js` )

Responsible for FR-07 through FR-11.

## Key Methods

```
class RISGenerator {
  /**
   * Generate RIS for a user story.
   * Consults code model and approved plan.
   * @param {string} userStoryId
   * @returns {RIS[]}
   */
  async generateRIS(userStoryId) { /* ... */ }

  /**
   * Query code model for artifacts relevant to this story.
   * Uses h-DSL navigation (PEEK, FOCUS, TRACE) to build context.
   * @param {string} userStoryId
   * @param {PlanItem} planItem
   * @returns {Object} relevantContext
   */
  async consultModel(userStoryId, planItem) { /* ... */ }
}
```

## RIS Data Structure

```
/**
 * @typedef {Object} RIS
 * @property {string} id - UUID
 * @property {string} userStoryId - FK to user story
 * @property {string} planId - FK to plan
 * @property {string} sprintId - FK to sprint
 * @property {string} branch - Target git branch
 * @property {string} content - Markdown content (the actual specification)
 * @property {string} status - generated|sent|completed
 * @property {Object} metadata - { createdAt, codeModelVersion }
 */
```

## RIS Content Format

Each RIS is a markdown file with the following structure:

```
# RIS: [User Story Title]

## Context
- Branch: `feature/xyz`
- Depends on: [list of prerequisite branches/RIS]
- Code Model Version: [hash]

## Objective
[One-sentence statement of what this implementation achieves]

## Instructions

### 1. [First task]
- File: `src/path/to/file.js`
- Action: create | modify | delete
- Details:
  [Specific implementation instructions]

### 2. [Second task]
...

## Conventions
- [Relevant patterns from the code model]
- [Naming conventions, error handling patterns, etc.]

## Assertions
- [ ] [Inspection assertion 1]
- [ ] [Inspection assertion 2]
```

## 2.4 CodeModel (`code-model.js`)

Responsible for FR-12 through FR-18. This is the runtime interface to the h-DSL instance.

### Storage Layout

```
.puffin/
└── cre/
    ├── schema.json      # h-DSL schema definition
    ├── instance.json     # h-DSL instance (the code model)
    └── memo.json         # Cached navigation results
```

### Key Methods

```
class CodeModel {
  /**
   * Load schema and instance from disk.
   * @param {string} projectRoot
   */
  async load(projectRoot) { /* ... */ }

  /**
   * Persist current state to disk.
   */
  async save() { /* ... */ }

  /**
   * PEEK: Get summary of an artifact without full load.
   * @param {string} path - Artifact path or identifier
   * @returns {ArtifactSummary}
   */
  peek(path) { /* ... */ }

  /**
   * FOCUS: Load full artifact details.
   * @param {string} path
   * @returns {Artifact}
   */
  focus(path) { /* ... */ }

  /**
   * TRACE: Follow dependency chain.
   * @param {string} from - Starting artifact
   * @param {Object} opts - { direction, kind, depth }
   * @returns {Artifact[]}
   */
  trace(from, opts) { /* ... */ }

  /**
   * FILTER: Find artifacts matching criteria.
   * @param {Object} criteria - { tags, kind, pattern }
   * @returns {Artifact[]}
   */
  filter(criteria) { /* ... */ }

  /**
   * Apply incremental updates to the instance.
   * @param {ModelDelta[]} deltas
   */
  async update(deltas) { /* ... */ }

  /**
   * Query artifacts relevant to a task description.
   * Implements ORIENT → FILTER → EXPLORE from the h-DSL protocol.
   * @param {string} task - Natural language task description
   * @returns {Artifact[]}
   */
  async queryForTask(task) { /* ... */ }
}
```

### Model Delta Format

```
/**
 * @typedef {Object} ModelDelta
 * @property {string} op - add|update|remove
 * @property {string} type - artifact|dependency|flow
 * @property {string} path - Target path/identifier
 * @property {Object} data - New or updated data
 */
```

## 2.5 SchemaManager (`schema-manager.js`)

Responsible for schema evolution (FR-14, FR-15, FR-16).

```
class SchemaManager {
  /**
   * Load the current h-DSL schema.
   * @returns {Schema}
   */
  async load() { /* ... */ }

  /**
   * Extend the schema with a new element type.
   * Validates h-M3 annotation is present.
   * @param {SchemaExtension} extension
   */
  async extend(extension) { /* ... */ }

  /**
   * Validate that an instance conforms to the schema.
   * @param {Object} instance
   * @returns {ValidationResult}
   */
  validate(instance) { /* ... */ }
}
```

## 2.6 Introspector (`introspector.js`)

Responsible for FR-13 (post-implementation code model updates).

```
class Introspector {
  /**
   * Analyze code changes from a completed implementation.
   * @param {string} branch - Branch that was implemented
   * @param {string} baseBranch - Branch it was based on
   * @returns {ModelDelta[]}
   */
  async analyzeChanges(branch, baseBranch) { /* ... */ }

  /**
   * Extract artifacts from source files.
   * Parses files to identify modules, classes, functions, exports.
   * @param {string[]} filePaths - Changed files
   * @returns {Artifact[]}
   */
  async extractArtifacts(filePaths) { /* ... */ }

  /**
   * Infer intent and relationships from code structure.
   * Uses AI to generate PROSE descriptions.
   * @param {Artifact[]} artifacts
   * @returns {ModelDelta[]}
   */
  async inferIntent(artifacts) { /* ... */ }
}
```

### 2.7 AssertionGenerator (`assertion-generator.js`)

Responsible for FR-04.

```
class AssertionGenerator {
  /**
   * Generate inspection assertions for a plan item.
   * @param {PlanItem} planItem
   * @param {CodeModel} codeModel
   * @returns {InspectionAssertion[]}
   */
  async generate(planItem, codeModel) { /* ... */ }

  /**
   * Run assertions against the current codebase.
   * @param {InspectionAssertion[]} assertions
   * @param {string} projectRoot
   * @returns {AssertionResult[]}
   */
  async verify(assertions, projectRoot) { /* ... */ }
}
```

---

## 3. h-DSL Schema Design

### 3.1 Base Schema (`schema.json`)

The schema defines what element types exist in the code model and how they map to h-M3.

```
{
  "version": "1.0.0",
  "m3Version": "2.0",
  "elementTypes": {
    "module": {
      "m3Type": "SLOT",
      "fields": {
        "path": { "m3Type": "TERM", "required": true },
        "kind": { "m3Type": "TERM", "required": true, "enum": ["module", "file", "config"] },
        "summary": { "m3Type": "PROSE", "required": true },
        "intent": { "m3Type": "PROSE", "required": false },
        "exports": { "m3Type": "TERM", "array": true },
        "tags": { "m3Type": "TERM", "array": true },
        "size": { "m3Type": "TERM", "required": false }
      }
    },
    "function": {
      "m3Type": "SLOT",
      "fields": {
        "path": { "m3Type": "TERM", "required": true },
        "signature": { "m3Type": "TERM", "required": false },
        "summary": { "m3Type": "PROSE", "required": true },
        "intent": { "m3Type": "PROSE", "required": false },
        "behavior": {
          "pre": { "m3Type": "PROSE" },
          "post": { "m3Type": "PROSE" },
          "err": { "m3Type": "PROSE" }
        },
        "tags": { "m3Type": "TERM", "array": true }
      }
    },
    "dependency": {
      "m3Type": "RELATION",
      "fields": {
        "from": { "m3Type": "TERM", "required": true },
        "to": { "m3Type": "TERM", "required": true },
        "kind": { "m3Type": "TERM", "required": true, "enum": ["imports", "calls", "extends", "implements", "configures", "test
        "weight": { "m3Type": "TERM", "enum": ["critical", "normal", "weak"] },
        "intent": { "m3Type": "PROSE", "required": false }
      }
    },
    "flow": {
      "m3Type": "SLOT",
      "fields": {
        "name": { "m3Type": "TERM", "required": true },
        "summary": { "m3Type": "PROSE", "required": true },
        "steps": {
          "array": true,
          "fields": {
            "order": { "m3Type": "TERM" },
            "artifact": { "m3Type": "TERM" },
            "intent": { "m3Type": "PROSE" }
          }
        },
        "tags": { "m3Type": "TERM", "array": true }
      }
    }
  },
  "extensionLog": []
}
```

## 3.2 Instance Format (`instance.json`)

```
{
  "schemaVersion": "1.0.0",
  "lastUpdated": "2026-01-31T00:00:00Z",
  "artifacts": {
    "src/main/plugin-loader.js": {
      "type": "module",
      "path": "src/main/plugin-loader.js",
      "kind": "file",
      "summary": "Discovers and loads plugins from the plugins directory",
      "exports": ["loadPlugins", "validatePlugin"],
      "tags": ["core", "plugins", "lifecycle"]
    }
  },
  "dependencies": [
    {
      "from": "src/main/plugin-loader.js",
      "to": "src/main/plugin-loader.js:PluginManager",
      "kind": "calls",
      "weight": "critical"
    }
  ],
  "flows": {
    "plugin-activation": {
      "name": "plugin-activation",
      "summary": "Plugin discovery through activation",
      "steps": [
        { "order": 1, "artifact": "PluginLoader.loadPlugins", "intent": "Scan plugins directory" },
        { "order": 2, "artifact": "Plugin.activate", "intent": "Run plugin activate hook" }
      ],
      "tags": ["lifecycle", "startup"]
    }
  }
}
```

## 3.3 Schema Evolution

When the Introspector encounters a concept not expressible in the current schema:

1. The Introspector identifies the gap (e.g., a "middleware" concept with ordering semantics).
2. It proposes a `SchemaExtension` with the new element type and its h-M3 annotation.
3. The SchemaManager validates the extension and appends it to `schema.json`.
4. Existing instance data remains valid (additive-only changes).

```
/**
 * @typedef {Object} SchemaExtension
 * @property {string} name - New element type name
 * @property {string} m3Type - h-M3 primitive it maps to
 * @property {Object} fields - Field definitions
 * @property {string} rationale - Why this extension is needed
 */
```

Extensions are logged in `schema.json.extensionLog` for auditability.

---

# 4. Database Schema

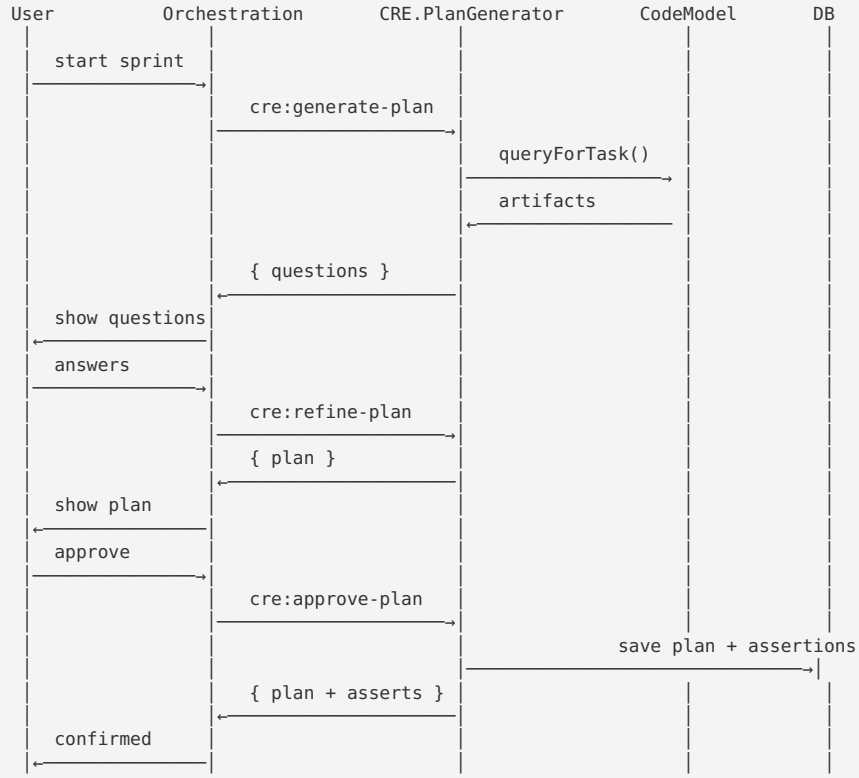
New tables added to the existing SQLite database.

```
-- Plans: one per sprint
CREATE TABLE plans (
  id TEXT PRIMARY KEY,
  sprint_id TEXT NOT NULL UNIQUE,
  status TEXT NOT NULL DEFAULT 'idle',
  version INTEGER NOT NULL DEFAULT 1,
  content TEXT NOT NULL,  -- JSON serialized plan items
  questions TEXT,         -- JSON array of pending questions
  created_at TEXT NOT NULL,
  updated_at TEXT NOT NULL,
  FOREIGN KEY (sprint_id) REFERENCES sprints(id)
);

-- Ready-to-Implement Specifications: many per user story
CREATE TABLE ris (
  id TEXT PRIMARY KEY,
  user_story_id TEXT NOT NULL,
  plan_id TEXT NOT NULL,
  sprint_id TEXT NOT NULL,
  branch TEXT NOT NULL,
  content TEXT NOT NULL,    -- Markdown RIS content
  status TEXT NOT NULL DEFAULT 'generated',
  code_model_version TEXT,  -- Hash of code model at generation time
  created_at TEXT NOT NULL,
  FOREIGN KEY (user_story_id) REFERENCES user_stories(id),
  FOREIGN KEY (plan_id) REFERENCES plans(id),
  FOREIGN KEY (sprint_id) REFERENCES sprints(id)
);

-- Inspection assertions: many per plan item
CREATE TABLE inspection_assertions (
  id TEXT PRIMARY KEY,
  plan_id TEXT NOT NULL,
  user_story_id TEXT NOT NULL,
  type TEXT NOT NULL,       -- file_exists, function_exists, export_exists, pattern_match
  target TEXT NOT NULL,     -- What to check
  description TEXT NOT NULL,
  result TEXT,              -- pass|fail|pending
  verified_at TEXT,
  FOREIGN KEY (plan_id) REFERENCES plans(id),
  FOREIGN KEY (user_story_id) REFERENCES user_stories(id)
);
```

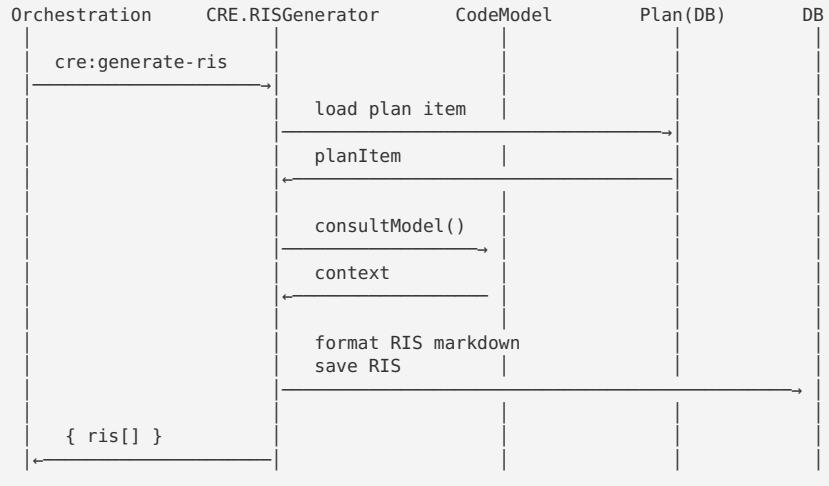
---

# 5. Sequence Diagrams

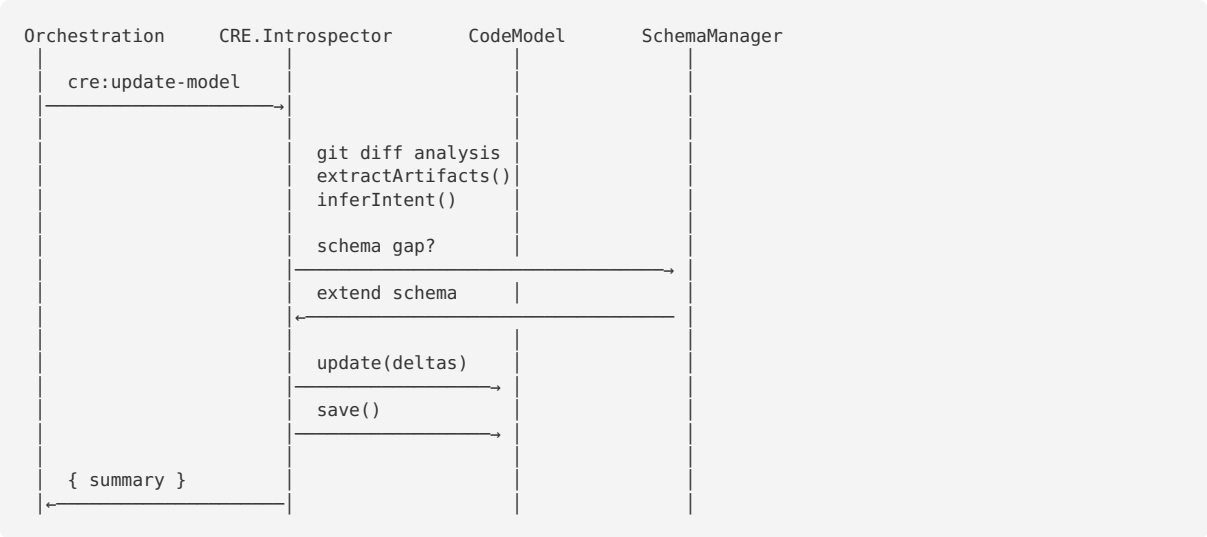
## 5.1 Plan Generation Flow




## 5.2 RIS Generation Flow



### 5.3 Post-Implementation Code Model Update




---

## 6. AI Integration Points

The CRE delegates reasoning to 3CLI (Claude) at specific points. These are not arbitrary AI calls — each maps to an h-M3 interpreter operation.

| Operation | h-M3 Op | Where Used | Input | Output |
|---|---|---|---|---|
| Analyze ambiguities | GROUND | PlanGenerator.analyzeSprint | User stories | Clarifying questions |
| Generate plan | FOLLOW | PlanGenerator.generatePlan | Stories + answers + model | Ordered plan |
| Refine plan | FOLLOW | PlanGenerator.refinePlan | Plan + feedback | Updated plan |
| Generate assertions | DERIVE | AssertionGenerator.generate | Plan items | Testable assertions |
| Generate RIS | FOLLOW | RISGenerator.generateRIS | Story + plan + model | RIS markdown |

| Operation | h-M3 Op | Where Used | Input | Output |
|---|---|---|---|---|
| Infer intent | GROUND | Introspector.inferIntent | Source code | PROSE descriptions |
| Assess relevance | JUDGE | CodeModel.queryForTask | Task + artifacts | Ranked artifacts |
| Identify schema gaps | DERIVE | Introspector.analyzeChanges | New concepts | Schema extensions |

Each AI call is structured as a prompt with: 1. **System context**: The relevant h-DSL fragments and conventions. 2. **Task**: The specific operation (e.g., "generate an implementation plan"). 3. **Constraints**: Format requirements, length limits, required fields.

---

# 7. Error Handling Strategy

Per NFR-02, the CRE must not crash Puffin.

## 7.1 Error Categories

| Category | Example | Recovery |
|---|---|---|
| **AI Failure** | 3CLI returns malformed output | Retry once, then surface error to user |
| **Schema Violation** | Instance doesn't match schema | Log warning, skip invalid element, continue |
| **Storage Failure** | Can't write JSON/ SQLite | Surface error to user, keep in-memory state |
| **Model Inconsistency** | Code model references deleted file | Mark artifact as stale, continue |

| Category | Example | Recovery |
| --- | --- | --- |
| **Plan Conflict** | User story dependencies form cycle | Report cycle to user during plan generation |

### 7.2 Error Wrapping

All IPC handlers wrap their logic:

```
ipcMain.handle('cre:generate-plan', async (event, { sprintId }) => {
  try {
    return { success: true, data: await planGenerator.analyzeSprint(sprintId) };
  } catch (error) {
    console.error('[CRE] Plan generation failed:', error);
    return { success: false, error: error.message };
  }
});
```

---

## 8. Configuration

CRE configuration lives in Puffin's existing config system.

```
{
  "cre": {
    "codeModelPath": ".puffin/cre",
    "maxPlanIterations": 5,
    "risMaxLength": 5000,
    "introspection": {
      "autoAfterMerge": true,
      "excludePatterns": ["node_modules/**", "dist/**", ".git/**"]
    },
    "schema": {
      "allowAutoExtension": true,
      "extensionApprovalRequired": false
    }
  }
}
```

---

## 9. Dependencies and Integration

### 9.1 Internal Dependencies

| Dependency | Usage |
| --- | --- |
| SQLite database | Plan and RIS persistence |

| Dependency | Usage |
|---|---|
| Memory Plugin | Read branch memory, conversation history for context |
| Outcome Lifecycle Plugin | Read outcome states to inform planning |
| 3CLI Runner | Execute AI prompts for reasoning operations |
| Git integration | Diff analysis for introspection |

### 9.2 Integration with Orchestration Layer

The orchestration layer drives the CRE through IPC calls in this sequence:

1. User starts sprint → `cre:generate-plan`
2. User answers questions → `cre:refine-plan` (repeat as needed)
3. User approves → `cre:approve-plan`
4. For each story in order → `cre:generate-ris`
5. 3CLI implements RIS
6. After implementation + review + fixes → `cre:update-model`
7. Repeat from step 4 for next story

---

## 10. Open Questions

| # | Question | Impact |
|---|---|---|
| OQ-1 | Should the code model be versioned (git-tracked) or treated as derived state? | Affects recovery strategy and storage approach |
| OQ-2 | What is the maximum acceptable latency for plan generation? | May require streaming/ progressive output |
| OQ-3 | Should RIS generation be parallelizable across independent user stories? | Affects orchestration and model locking |

| # | Question | Impact |
|---|---|---|
| OQ-4 | How should the CRE handle codebase changes made outside of Puffin? | May need periodic full re-introspection |
| OQ-5 | Should schema extensions require user approval? | Trade-off between autonomy and control |