Agentic Dream

📘 Agentic Dream Framework - Complete Guide

Status: 🟢 Production Ready | **Version:** 2.0 | **Format:** Consolidated Guide

📖 This is the complete Agentic Dream Framework guide consolidated into a single document.
All content is integrated in linear format for continuous reading.

📑 Table of Contents

👋 Introduction

Welcome to the **Agentic Dream Framework**, a revolutionary approach to AI-assisted software development that transforms linear "vibe coding" into a **compounding engineering system**.

This framework is built on a simple but powerful insight: *traditional AI coding relies on chat history (RAM), which decays over time. Instead, we move intelligence to the file system (Disk), creating a system that gets smarter with every bug fixed.*

> 💡 **Mission:** *We do not just ship code; we architect the system that ships the code.*

🧬 PART 1: THE MANIFESTO

From Linear to Exponential

The Problem: "Vibe Coding"

Traditional AI coding is **Linear**. You chat, you get code, you paste it.

- ❌ **Context Decay:** As the chat gets longer, the AI's IQ drops.
- ❌ **Zero Memory:** Fixing a bug in the chat fixes it *once*. It does not teach the system for the next time.
- ❌ **Fragility:** Without specs, the code drifts from the requirements instantly.

The Solution: Compounding Engineering

Under **Agentic Dream Framework**, every bug fixed must leave the system smarter than it was yesterday. We achieve this by moving intelligence from **RAM (Chat)** to **Disk (Files)**.

💡 **The Core Equation:**
`Bug Found` + `Update `agents.md`` = `Asset Appreciation`

🧠 PART 2: SYSTEM ARCHITECTURE

The Disk-Based Brain

We treat the file system as the brain of the AI. Do not rely on the chat history.

📂 Directory Topology

```
project-root/
├── .cursor/
│   └── rules/
│       └── global.mdc        # The "Constitution"
├── docs/
```

```
│   ├── specs/
│   │   ├──
```

🧩 Key Components

🏛️ 1. The Constitution (`.cursor/rules/global.mdc`)

- **Purpose:** The immutable laws. Tech stack, forbidden patterns, and safety rails.
- **Constraint:** Must be **< 200 lines**. If it's too long, the AI ignores it.
- **Content:** "Always use TypeScript", "No console.logs", "Follow PPRE cycle".

📚 2. Modular Context (`docs/reference/\\\*.md`)

- **Purpose: Context Sharding**. We do not load all documentation at once.
- **Mechanism:** If the agent is working on Auth, it loads `auth.md`. It does *not* load `billing.md`. This prevents **Context Poisoning**.

🦠 3. Fractal Memory (`src/\\\*\\\*/` `agents.md`)

- **Purpose: Tacit Knowledge**. This is where the system "learns".
- **Location:** Lives inside every folder (e.g., `src/components/` `agents.md`).
- **Content:** "In this folder, we use named exports only." "Button.tsx has a known issue with z-index."

📜 4. The Input Contract (`docs/specs/stories.json`)

- **Purpose:** The executable instruction.
- **Format:** JSON array of atomic tasks with **Binary Acceptance Criteria** (Pass/Fail).

Example:

```
{
  "id": "AUTH-03",
  "description": "Add JWT token validation middleware",
  "files_to_touch": ["src/middleware/auth.ts"],
  "acceptance_criteria": [
    "Middleware function named 'validateJWT' exists",
    "Returns 401 status code for invalid tokens",
    "Attaches decoded user object to req.user on success"
  ],
  "passes": false
}
```

⚙️ PART 3: THE PPRE ENGINE

The PPRE Cycle & The Kill Switch

We do not "chat" with code. We execute the **PPRE Cycle**.

🔄 The Cycle Phases

```
sequenceDiagram
    participant H as Human (Architect)
    participant A as Agent (Worker)
    participant S as System (Files)

    Note over H, A: Phase 1: PRIME
    H->>A: Load Context (Story + Reference Docs)

    Note over H, A: Phase 2: PLAN
    A->>H: Propose Implementation
    H->>A: Approve Plan
    Note over H, A: Phase 3: RESET (Kill Switch)
    H->>A: Clear Chat Context

    Note over H, A: Phase 4: EXECUTE
    A->>S: Write Code per Plan
    A->>A: Self-Verify vs Criteria

    Note over H, A: Phase 5: REVIEW
    A->>H: Mark passes: true if all pass
    H->>S: Commit + Update
```

🛑 The Kill Switch (Context Reset)

> ⚠️ *"Context Rot is the enemy of intelligence."*

You must **wipe the chat history** (reset) between the Plan and Execute phases.

1. **Plan Phase:** High creativity, lots of tokens.
2. **Reset:** Restore IQ to 100%.
3. **Execute Phase:** High precision, zero noise. Feed *only* the Plan and the active file.

Why it works:

- Eliminates conversational "noise"
- Prevents the agent from getting confused with obsolete information
- Maintains focus on the specific task
- Restores the model's "intelligence"

🤖 PART 4: THE RALPH PROTOCOL

Atomic Execution

How to write `stories.json` so the agent cannot fail.

📏 Rule of Atomicity

A story is **Atomic** if it can be completed in **ONE** context window (approx. 1 file or 1 function). If it requires "figuring things out", it is not a story; it is research.

✅ Binary Acceptance Criteria

Criteria must be objectively **True** or **False**. No subjectivity allowed.

Examples:

❌ Bad Criteria (Vague)	✅ Good Criteria (Binary)
"Make the UI look modern"	"Button uses Tailwind class `rounded-lg` "
"Fix the bug"	"Function `calculateTotal` returns 100 for input [1,2,3]"
"Add error handling"	"API returns 400 status on invalid JSON"

🛡️ Self-Verification

The Agent must read `stories.json`, check its own code against the `acceptance_criteria`, and *only then* mark `passes: true`. Humans verify *after* the agent claims success.

📈 PART 5: SYSTEM EVOLUTION

The Golden Rule

This is the most important section for the "Guardian" role.

🧬 The Evolution Loop

When a bug occurs or the Agent gets stuck:

1. Fix the Code (Linear work)

- Solves the immediate problem
- Does NOT prevent future occurrences

2. MANDATORY - Codify the Lesson (Exponential work)

- Ensures the system learns

Decision Tree:

- **Logic Error?** → Update `src/\\\*\\\*/`agents.md` ("Warning: This API returns strings, not numbers")
- **Style Error?** → Update `.cursor/rules/global.mdc`
- **Process Error?** → Update `docs/reference/`workflow.md`

⚠️ **Mandate:** If you fix a bug manually and do not update a `.md` file, you have **failed the protocol**. The system has not learned.

Objective

The system must be **smarter at the end of the day** than it was at the start.

📚 PART 6: DOCUMENTATION STANDARDS

Documentation is not an afterthought. It is the **permanent memory** of the system.

🎯 The Documentation Hierarchy

Level 1: Constitutional (`.cursor/rules/global.mdc`)

When to update: When you establish a new universal rule or architectural constraint.

Examples:

- "Database migrations must be reversible"
- "All API calls must use the centralized `apiClient` wrapper"
- "All components must be tested with React Testing Library"

Lifespan: Permanent until a major architecture change.

Level 2: Reference (`docs/reference/\\*.md`)

When to update: When you add new features, APIs, or integration patterns.

Required files:

- `api_` `endpoints.md` - All backend endpoints with request/response examples
- `auth.md` - Authentication flow, token management, permissions
- `db_` `schema.md` - Database tables, relationships, indexes
- `deployment.md` - Build, test, and deployment procedures
- `env_` `variables.md` - All environment variables and their purpose

Level 3: Tactical (`src/\\*/` `agents.md`)

When to update: Every time you fix a bug, discover a gotcha, or establish a local convention.

Structure:

```
# 🧠 Agent Memory: [Folder Name]

## 📌 Critical Lessons
- **YYYY-MM-DD:** Lesson learned with context

## 🛑 Known Issues
- Issue description and workaround

## 🏗️ Local Conventions
- "This folder uses X pattern because Y"
```

```
## 🔗 Dependencies
- "Module A depends on Module B for Z functionality"
```

Level 4: Execution (`logs/progress.txt`)

When to update: After completing each story in `stories.json`.

Format:

```
[2026-01-16 15:30] ✅ AUTH-01: Login Form Component
- Created LoginForm.tsx with email/password fields
- Added form validation with react-hook-form
- All acceptance criteria passed
- Updated src/components/
```

📐 Documentation Rules

⚡ **Rule 1: No Orphan Fixes**
Every bug fix MUST update at least one `.md` file. If you fix it in code but not in docs, you will hit the same bug again.

⚡ **Rule 2: Date Every Entry**
Always prefix lessons with `YYYY-MM-DD`. This creates a timeline of system evolution.

⚡ **Rule 3: Be Specific**
"The API is weird" ❌
"The `/users` endpoint returns strings for IDs, not numbers" ✅

⚡ **Rule 4: One Source of Truth**
Do not duplicate information. Use references:
"For authentication, see `docs/reference/`[`auth.md`]"

⚡ **Rule 5: Archive Completed Specs**
When a feature is complete, move `docs/specs/feature.json` to `docs/done_specs/feature.json`. Keep the active folder clean.

🚀 PART 7: QUICK START GUIDE

From Zero to Hello World in 5 Minutes

Time: 5 Minutes | **Goal:** Verify the Agentic Loop

1. 🏗️ Scaffold the Brain (The Setup)

Don't create folders manually. Copy and paste this command in your terminal (project root) to generate the **Agentic Dream Topology** instantly.

```
mkdir -p .cursor/rules docs/specs docs/reference docs/done_specs logs src/scripts
touch .cursor/rules/global.mdc logs/progress.txt src/scripts/hello_agentic.ts
```

2. 📜 The Input Contract (Hello World)

Create the file `docs/specs/stories.json`. This is your first **Input Contract**.

Copy this exact block:

```json
{
  "epic": "System Initialization",
  "status": "active",
  "stories": [
    {
      "id": "INIT-001",
      "description": "Create a Hello World script to verify the Agentic Dream Engine execution.",
      "files_to_touch": ["src/scripts/hello_agentic.ts", "package.json"],
      "acceptance_criteria": [
        "File 'src/scripts/hello_agentic.ts' exists",
        "The script prints exactly: 'Agentic Dream Engine: ONLINE'",
        "A 'package.json' script 'start:agentic' runs this file",
        "Running 'npm run start:agentic' succeeds without errors"
      ],
      "passes": false
    }
  ]
}
```

3. 🔄 Execute the PPRE Cycle (The "How-To")

Follow this cycle **exactly** to complete the story above.

Phase 1: PRIME 🧠

- Open Chat (Command+L)
- Drag/Attach: `docs/specs/stories.json` and `.cursor/rules/global.mdc`
- **Prompt:** "Read the active story in stories.json. Do not code yet."

Phase 2: PLAN 📝

- **Prompt:** "Create a step-by-step plan to implement story INIT-001. Output as Markdown."
- **Human Review:** Verify the plan includes creating the `ts` file and editing `package.json`

Phase 3: RESET (The Kill Switch) 🛑

- **Action:** Press `Command+K` (or use `/clear` / New Chat)
- **Why:** We eliminate conversation noise to restore the Agent's IQ to 100%

> ⚠️ **CRITICAL:** Do not skip this step. RESET is mandatory between Plan and Execution. If you skip it, code quality drops by 40%.

Phase 4: EXECUTE 🚀

- Paste the **Approved Plan** (copied from Phase 2) into the new chat
- **Prompt:** "Execute this plan. Verify against the acceptance criteria in stories.json. If successful, update 'passes' to true."

4. ✅ Certification Quiz

Before committing to production, verify you understand the rules:

☐ **Where does intelligence live?** (Answer: In `agents.md` files, NOT in chat)

☐ **When do I Reset?** (Answer: ALWAYS between Plan and Execution)

☐ **When is my work done?** (Answer: When `stories.json` says `true` AND I've updated `agents.md` with learnings)

> 🎓 **Mandatory Certification**
> To complete your onboarding, you must pass the official framework quiz:
> 🔗 **Certification Quiz:** https://forms.gle/ABvRg4qtwXCZZ

⚙️ PART 8: COMPLETE TECHNICAL SPECIFICATION

Technical Reference for Implementation

Version: 1.0 | **Type:** System Architecture & Workflow Protocols

1. Philosophy: Compounding Agentic Engineering

The Agentic Dream Framework shifts development from a linear **"Feature Factory"** model to an **Exponential System Evolution** model.

Core Principles

Spec-Driven Development

- No code is written without a rigorous, approved specification (PRD)
- The Chat History is ephemeral; the PRD is the absolute truth
- Location: `docs/specs/`\`prd.md\`

Atomic Execution

- Work is broken down into binary units (Atoms) verifiable by scripts
- Each atom must be completable in ONE agent context window
- Binary acceptance criteria only (Pass/Fail)

Compounding Context

- Every bug fixed or lesson learned must be "codified" into the system's memory
- `.mdc` rules for style/architecture decisions
- \`agents.md\` for tactical knowledge and gotchas

- Ensures the same error never occurs twice

2. The Agentic Dream Directory Structure

All projects **MUST** strictly adhere to this standardized topology to ensure agent autonomy.

Complete Directory Tree

```
project-root/
├── .cursor/
│   └── rules/
│       └── global.mdc        # The "Constitution": Tech stac
k, critical constraints
├── logs/
│   └── progress.txt          # Short-Term Memory: Active PPRE
cycle logs
├── scripts/
│   └──
```

Component Roles

`.cursor/rules/global.mdc` - The Constitution

- Always loaded by the AI
- Contains universal, high-level rules
- Must be < 200 lines to prevent context overload
- Examples: "Use TypeScript", "No console.logs in production"

`logs/progress.txt` - Short-Term Memory

- Session-level continuity between PPRE cycles
- Format: `\\\[YYYY-MM-DD HH:MM\\\] ✅/❌ STORY-ID: Description`
- Cleared or archived after epic completion

`scripts/`**`autopilot.sh`** - The Execution Engine

- Automates the PPRE cycle
- Enforces the Kill Switch (context reset)
- Handles git commits and story verification

`docs/specs/`**`prd.md`** - The Blueprint

- Single source of truth for business logic
- Must include: Mission, Technical Architecture, Data Dictionary, Implementation Plan
- Overrides chat history in all conflicts

`docs/specs/stories.json` - The Execution Contract

- Array of atomic tasks
- Each story has binary `acceptance_criteria`
- Agent self-verifies and marks `passes: true/false`

`docs/reference/\\*.md` - Context Sharding Layer

- Loaded **on-demand only**
- Prevents "Context Poisoning"
- Example: Load `auth_``flow.md` only when working on auth

`**src//**agents.md` - Long-Term Memory

- Fractal: Lives in every relevant directory
- Contains: Critical Lessons, Known Issues, Local Conventions, Dependencies
- Date-stamped entries for timeline tracking

3. The Workflows

Phase 1: The Blueprint (PRD)

Rule: The Chat History is ephemeral; the PRD is the absolute truth.

Location: `docs/specs/``prd.md`

Required Sections:

1. **Mission:** What problem are we solving?
2. **Technical Architecture:** High-level system design
3. **Data Dictionary:** All entities, fields, relationships
4. **Implementation Plan:** Step-by-step breakdown

Validation: PRD must be approved by stakeholders before any coding begins.

Phase 2: Atomic Decomposition

Action: Convert `prd.md` into `docs/specs/stories.json`

The Atomicity Rule:

- A "Story" must be small enough to be completed in **one single agent context window**
- Approximately 1 file or 1 function
- If it requires "figuring things out", it's research, not a story

Validation:

- Every story MUST have an `acceptance_criteria` array
- Criteria must be **Binary (Pass/Fail)**
- Vague criteria (e.g., "Make it look good") are **prohibited**

Phase 3: The PPRE Execution Loop

Agents must not "vibe code." They must follow the **PPRE Cycle** enforced by `scripts/` `autopilot.sh`:

1. PRIME

- Load **only** the current story from `stories.json`
- Load relevant `docs/reference/` files
- Do NOT load the entire chat history

2. PLAN

- Generate a specific implementation plan in the chat

- Human reviews and approves the plan
- Plan is saved to context

3. RESET (The Kill Switch) ⚠️

- **Critical:** Clear the context window/chat history
- This prevents "Context Rot"
- Only the approved Plan and relevant files remain in context

4. EXECUTE

- Write code based strictly on the Plan and Story constraints
- No improvisation or "creative interpretation"
- Follow binary acceptance criteria exactly

5. VERIFY

- Agent checks its own code against `acceptance_criteria`
- Each criterion must evaluate to Pass or Fail
- No partial passes allowed

6. COMMIT

- If all criteria pass: Set `"passes": true` in JSON
- Log to `logs/progress.txt` with timestamp
- Git commit with story ID in message
- If any criterion fails: Update `agents.md` with lesson learned

4. Context Management Strategy

Context Sharding

To prevent "Context Poisoning" (confusing the AI with too much info), we strictly separate context:

Global Context (`.cursor/rules/global.mdc`)

- Always loaded

- Contains strictly high-level rules
- Examples: "Use TypeScript", "No console.logs in production", "Follow PPRE cycle"
- Constraint: < 200 lines

On-Demand Context (`docs/reference/`)

- Loaded **only** when requested
- Mechanism: If working on Auth, load `docs/reference/auth_`[`flow.md`]
- Do NOT load `ui_`[`components.md`] when working on Auth
- Prevents information overload

Memory Systems

Short-Term Memory (`logs/progress.txt`)

- **Question it answers:** "What did I just do?"
- **Purpose:** Maintain continuity between PPRE cycles without bloating context
- **Lifespan:** Current epic or sprint
- **Format:** Timestamped log entries

Long-Term Memory (`src/\\\*\\\*/`[`agents.md`])

- **Question it answers:** "What have I learned?"
- **Purpose:** Store project-specific nuances and prevent repeated errors
- **Lifespan:** Permanent (until architecture change)
- **Examples:**
    - "This API endpoint requires a specific header format"
    - "Button.tsx has a known z-index issue with Modal.tsx"
    - "Use react-hook-form with zod validation for all forms"

🤖 PART 9: AUTOPILOT SCRIPT

Automating the Ralph Loop

Status: 🟢 Production Ready | **Dependency:** `jq` (JSON Processor)

Mission: Enforce the PPRE Cycle and automate the administrative overhead of the Ralph Protocol.

The Complete Script

File location: `scripts/` `autopilot.sh`

```
#!/bin/bash

# --- CONFIGURATION ---
SPECS_FILE="docs/specs/stories.json"
PROGRESS_FILE="logs/progress.txt"
MEMORY_FILE="src/scripts/
```

Line-by-Line Explanation

Line	Concept	Technical Explanation
`18`	**The Infinite Loop**	Script runs until no tasks remain. Simulates "Autonomous Agent" behavior.
`20`	**Atomic Selection**	Uses `jq` to find first `false` item. Forces **Sequential Execution**.
`24`	**Exit Condition**	Empty `STORY_ID` means `stories.json` is 100% complete.
`37`	**Human-in-the-Loop**	Acts as "Gatekeeper". Waits for human validation before proceeding.
`43`	**State Mutation**	Rewrites JSON "in-place", changing `false` to `true`. Updates the Contract.
`51`	**Auto-Commit**	Automatic commit per story. Ensures granular, clean Git history.

Line	Concept	Technical Explanation
`58`	**Forced Evolution**	Blocks progress until human confirms `agents.md` update. Prevents "brute forcing".

Customization Examples

For Automated Tests (CI/CD)

Replace line 37 with:

```
echo "🧪 Running Tests..."
if npm test; then
    RESULT="y"
else
    RESULT="n"
fi
```

For Python Projects

Change configuration:

```
SPECS_FILE="requirements/stories.json"
MEMORY_FILE="app/api/
```

For Silent Mode (Advanced)

```
claude --prompt "Execute story $STORY_ID from $SPECS_FILE. Output only code."
```

Installation

1. Install dependencies:

```
brew install jq  # Mac
sudo apt-get install jq  # Linux/WSL
```

2. Make executable:

```
chmod +x scripts/
```

3. Run:

```
./scripts/
```

🔧 PART 10: TROUBLESHOOTING GUIDE

Common Debugging Scenarios

Purpose: Common debugging scenarios for the Agentic Dream Framework

🚨 **Before reporting a bug:** 90% of framework problems are due to not strictly following the process. Review this guide first.

🤖 Problem 1: "Agent doesn't follow PPRE cycle"

Symptoms

- Agent starts coding without making a plan
- Agent "guesses" requirements not in `stories.json`
- Generated code doesn't meet acceptance criteria
- Agent refactors unrelated code

Diagnosis

The agent is in **"Vibe Coding Mode"** because it didn't receive explicit PPRE cycle instructions. Without structured context, the LLM falls back to default behavior: write code immediately.

Solution

Step 1: Verify `global.mdc` exists

```
ls -la .cursor/rules/global.mdc
```

If it doesn't exist, create it with the template from the Quick Start Guide.

Step 2: Force PPRE cycle explicitly

Use these **exact** prompts in each chat:

PRIME:

```
Read the active story in docs/specs/stories.json.
Also read .cursor/rules/global.mdc.
Do NOT code yet. Just confirm you understand.
```

PLAN:

```
Create a step-by-step implementation plan.
Output as Markdown. Include:
1. Files to create/modify
2. Dependencies to check
3. Acceptance criteria to verify
```

RESET: Press `Cmd+K` (Mac) or `Ctrl+K` (Windows). **Mandatory.**

EXECUTE:

```
[Paste plan here]
```

```
Execute this plan. After completion, verify against
acceptance criteria and update stories.json.
```

Step 3: If problem persists

- Close and reopen Cursor completely
- Verify no multiple chat tabs open
- Check for old `.cursorrules` in project root

🧠 Problem 2: "Context Decay still happening"

Symptoms

- Agent "forgets" decisions made 5 minutes ago
- Agent asks same question multiple times
- Agent introduces bugs already fixed
- Code quality degrades in long chats

Diagnosis

You're experiencing **Context Window Pollution**. Each message consumes tokens. At context limit (100k-200k tokens), the model starts "forgetting" information at the beginning.

Solution

Cause #1: Not doing RESET

- **Golden rule:** ALWAYS RESET between PLAN and EXECUTE
- If chat has 20+ messages, you're doing something wrong

Cause #2: Not using **`agents.md`**

The file system is "permanent brain". Chat is "volatile RAM".

✅ **Correct:**

```
# src/components/
```

❌ **Incorrect:** Letting agent fix same bug 3 times without documenting.

Cause #3: Stories too large

If story touches 10+ files, it's not atomic. Split it:

❌ **Bad:**

```
{"id": "AUTH-01",
 "description": "Implement complete auth system"}
```

✅ **Good:**

```
[
  {"id": "AUTH-01a", "description": "Create login UI"},
  {"id": "AUTH-01b", "description": "Add validation"},
  {"id": "AUTH-01c", "description": "Connect to API"}
]
```

Cause #4: Dragging large PDFs into chat

- Extract relevant text to `.md` in `docs/reference/`
- Drag small `.md` instead of full PDF

📋 Problem 3: "Stories.json not validating"

Symptoms

- Error: `Unexpected token` when running autopilot
- Agent doesn't find stories marked `false`
- Script says "All complete" but tasks remain
- Acceptance criteria not being evaluated

Diagnosis

JSON is strict. A syntax error breaks the entire file.

Solution

Step 1: Validate syntax

Online: jsonlint.com

Terminal:

```
jq . docs/specs/stories.json
```

Step 2: Common errors

❌ **Extra comma:**

```
{"stories": [
  {"id": "TASK-01", "passes": false},
]}
```

❌ **Wrong quotes on booleans:**

```
{"passes": "false"}  // Wrong (string)
{"passes": false}    // Correct (boolean)
```

Step 3: Valid template

```
{
  "epic": "Feature Name",
  "status": "active",
  "stories": [
    {
      "id": "FEAT-001",
      "description": "Short description",
      "files_to_touch": ["src/file.ts"],
      "acceptance_criteria": [
        "Criterion 1",
        "Criterion 2"
      ],
      "passes": false
    }
  ]
}
```

📝 Problem 4: "Agents.md not being updated"

Symptoms

- Agent makes same mistake multiple times
- `agents.md` files empty or outdated
- Team doesn't know what lessons learned
- No "compounding" (system doesn't get smarter)

Diagnosis

"System Evolution" protocol not being followed. Developers fix bugs without codifying learning.

Solution

Step 1: Understand evolution protocol

When you find a bug:

1. **DO NOT fix it silently**
2. Open `agents.md` in relevant folder
3. Add entry with date + lesson
4. **Then** fix the bug
5. Commit both changes together

Step 2: Correct structure

```
# 🧠 Agent Memory: [Module Name]

## ⚠ Critical Lessons Learned

- **2026-01-16:** The `formatDate()` utility expects
  ISO strings, not Date objects. Always convert first.
- **2026-01-14:** Never mutate props directly.
  Clone first with `structuredClone()`.

## 🚫 Forbidden Patterns

- Do NOT use `any` type in this module.
- Do NOT import from `@/utils/deprecated`.

## 📚 Context

- This module interfaces with legacy Python API
- Date formats must match backend: YYYY-MM-DD
- All IDs are UUIDs (strings), never integers
```

❓ FAQ: Common Errors

Q1: "Agent changed my tech stack"

- **Cause:** `global.mdc` doesn't define tech stack
- **Solution:** Add "Tech Stack Standards" section with exact stack

Q2: "Autopilot stuck in loop"

- **Cause:** Story has `passes: true` but script detects `false`
- **Solution:** Look for invisible characters: `cat -A docs/specs/stories.json \\\| grep "passes"`

Q3: "Agent hallucinates functions"

- **Cause:** Agent assumes libraries not installed

- **Solution:** Force verification in plan: "Before planning, check package.json for dependencies."

Q4: "Auto-commits breaking workflow"

- **Cause:** Script commits without review
- **Solution:** Modify script to stage instead of auto-commit

Q5: "Team not following framework"

- **Cause:** Lack of accountability
- **Solution:** Implement gates: pre-commit hooks validating `stories.json` and `agents.md`

Q6: "Framework too slow for hotfixes"

- **Cause:** Scope misunderstanding. Framework is for features, not emergencies
- **Solution:** Create bypass protocol for critical production bugs

---

📋 PART 11: PRD TEMPLATE & CREATION GUIDE

How to Create Professional PRDs in 30 Minutes

Version: 1.0 | **Type:** Spec-Driven Development Template

This guide shows you how to create **professional Product Requirements Documents (PRDs)** in 30 minutes using AI, aligned with the Agentic Dream Framework's spec-driven development philosophy.

What you'll get: A complete PRD ready to be converted into `stories.json` for atomic execution.

📦 What's Inside

2 AI Prompts + This Guide = Complete PRD System

1. **ChatGPT Scaffold Prompt** - Creates PRD structure + Claude prompt
2. **Claude Generation Prompt** - Generates detailed 15-20 page PRD
3. **Review & Polish Prompt** - Reviews and improves your PRD
4. **Agentic Dream Conversion Guide** - How to convert PRD → `stories.json`

That's it! No complex setup, no multiple documents to manage.

🚀 How It Works (4 Simple Steps)

Step 1: Generate Your PRD Creation Prompt (5 minutes)

Using ChatGPT (GPT-4 recommended):

Scaffold Prompt Template:

```
You are a senior product manager creating a PRD for a new AI
project.

Project Description: [YOUR PROJECT DESCRIPTION HERE]

Create a comprehensive prompt for Claude that will generate a
complete PRD with these sections:

1. Executive Summary
2. Problem Statement
3. Solution Overview
4. User Personas
5. Technical Architecture
6. Functional Requirements (as user stories)
7. API Specifications
8. Implementation Plan
9. Success Metrics
10. Risk Assessment

The PRD must follow the Agentic Dream Framework principles:
- Spec-driven (PRD is source of truth)
- Atomic decomposition (requirements broken into verifiable u
nits)
- Binary acceptance criteria (Pass/Fail only)
- Implementation-ready (developers can start immediately)

Format the prompt so Claude will generate a 15-20 page profes
sional document.
```

Action:

1. Replace `\\\[YOUR PROJECT DESCRIPTION HERE\\\]` with your project
2. Paste into ChatGPT
3. **Result:** Complete prompt ready for Claude

Step 2: Generate Your PRD (5 minutes)

Using Claude:

1. Copy ChatGPT's entire response
2. Paste directly into Claude chat
3. **Result:** 15-20 page professional PRD

💡 **Pro Tip:** Claude Pro generates faster, but any tier works. If the response is cut off, simply type "continue" to get the rest.

Step 3: Review & Polish (5 minutes)

Using ChatGPT again:

Review Prompt Template:

```
Review this PRD for completeness and quality. Check for:

1. Vague requirements (flag anything that isn't binary/verifi
able)
```

```
2. Missing technical details
3. Unrealistic timelines
4. Gaps in user stories or acceptance criteria
5. Missing API specifications
6. Unclear success metrics

Provide specific suggestions for improvement.

PRD:
[PASTE YOUR PRD HERE]
```

Action:

1. Paste your PRD from Claude
2. Apply ChatGPT's suggestions
3. **Result:** Production-ready PRD

Step 4: Convert to Stories.json (15 minutes)

Now convert your PRD into atomic, executable stories.

Conversion Prompt for ChatGPT:

```
Convert this PRD into a stories.json file following the Agent
ic Dream Framework format.

For each functional requirement, create atomic stories that:
- Are completable in ONE context window (1 file or 1 functio
n)
- Have binary acceptance criteria (Pass/Fail only)
- Specify exact files to touch
- Are ordered by dependency

Format:
{
  "epic": "Epic Name",
  "stories": [
    {
      "id": "EPIC-01",
      "description": "Atomic task description",
      "files_to_touch": ["path/to/file.ts"],
      "acceptance_criteria": [
        "Specific, binary criterion 1",
        "Specific, binary criterion 2"
      ],
      "passes": false
    }
  ]
}

PRD:
[PASTE YOUR PRD HERE]
```

Result: Ready-to-execute `stories.json` file

Total Time: 30 minutes

Total Effort: 4 copy-paste operations

✅ What You Get

After 30 minutes, you'll have a comprehensive PRD with:

📄 Documentation

- **Executive Summary** - Clear vision and objectives
- **Problem Statement** - Market-backed pain points
- **Solution Overview** - Technical approach and differentiators
- **User Personas** - Detailed target user profiles
- **Technical Architecture** - Complete system design

🔧 Implementation Ready

- **Functional Requirements** - User stories with acceptance criteria
- **API Specifications** - Endpoints and data models
- **Implementation Plan** - Sprint timeline and team needs
- **Success Metrics** - Measurable KPIs and targets
- **Risk Assessment** - Identified risks with mitigation strategies

⚙️ Agentic Dream Framework Integration

- **stories.json** - Atomic, executable tasks
- **Binary Acceptance Criteria** - No ambiguity
- **File-level Granularity** - Agent knows exactly what to touch
- **PPRE Cycle Ready** - Can start Prime → Plan → Reset → Execute immediately

💡 Pro Tips for Better Results

For Your Project Description

Be specific about:

- ✅ Target users and main goal
- ✅ Key integrations or platforms
- ✅ Constraints (budget, timeline, team size)
- ✅ **Focus on WHAT you're building**, not HOW

Good Examples ✅

"A commercial furniture design assistant that generates 3D renders using real, in-stock SKUs from our catalog. Target users are furniture dealers creating proposals for corporate clients. Must integrate with existing PIM system and validate pricing/availability in real-time."

Bad Examples ❌

"An AI thing that helps people"

"A chatbot using machine learning"

🎯 Success Indicators

Your PRD is ready when:

- ✅ **15+ pages** of detailed content
- ✅ **All 10 sections** are comprehensive
- ✅ **Developers can start building** from your specs
- ✅ **ChatGPT review** shows only minor improvements needed
- ✅ **stories.json** has 10-30 atomic stories
- ✅ **All acceptance criteria** are binary (Pass/Fail)

Red Flags 🚩

- 🚩 **Under 10 pages** - Too generic or missing details
- 🚩 **Vague requirements** - "User-friendly interface" instead of specific user stories
- 🚩 **No technical specs** - Developers can't estimate effort
- 🚩 **Impossible timeline** - 6 months of work planned for 6 weeks
- 🚩 **Subjective criteria** - "Should look modern" instead of "Uses Tailwind class rounded-lg"

🎯 After Your PRD is Complete

Immediate Next Steps (Week 1)

1. **Share with stakeholders** - Get feedback and approval
2. **Technical review** - Have tech lead assess feasibility
3. **User validation** - Show to 5-10 potential customers
4. **MVP scoping** - Identify minimum viable features
5. **Save to docs/specs/** - Store `prd.md` and `stories.json` in your repo

Development Ready (Month 1)

- **Development team** can start building using PPRE cycle
- **User stories** ready for sprint planning
- **Success metrics** defined for tracking progress
- **Risk mitigation** plans in place
- **agents.md** **files** can start accumulating lessons learned

---

📦 PART 12: EXAMPLE REPOSITORY

Reference Implementation

Status: 🟢 Live Sample | **Stack:** React + TypeScript + Node

Objective: A "clean slate" repository demonstrating the Agentic Dream file topology.

1. 📂 Directory Structure

Create this exact structure in your repository:

```
.
├── .cursor/
│   └── rules/
│       └── global.mdc       # The Constitution
├── docs/
│   ├── specs/
│   │   └── stories.json     # Input Contract
│   └── reference/
│       ├── auth_
```

2. 📄 Critical Artifacts

🏛️ `.cursor/rules/global.mdc` (The Constitution)

```
---
description: GLOBAL AGENTIC DREAM LAWS - ALWAYS ACTIVE
globs: *
---
# 🏗️ Agentic Dream Constitution

1. **Zero Hallucination:** You DO NOT write code without a Pl
an.
2. **The Memory Rule:** Before editing ANY folder, you MUST r
ead the `
```

🧠 `src/components/`agents.md` (Memory Example: UI)

```
# 🧠 Agent Memory: Components Layer

## ⚠ Critical Lessons (DO NOT REPEAT MISTAKES)
- **2023-11-01:** The `Button` component crashes if `isLoadin
g` is undefined.
  Always provide default `false`.
- **2023-11-05:** We use `lucide-react` for icons. Do NOT imp
ort from `react-icons`.

## 📌 Standards
- All components must export as **Named Exports** (e.g., `exp
ort function Button`).
- Props must be defined via a TypeScript `interface`.
```

📜 `docs/specs/stories.json` (The Contract)

```
{
  "epic": "User Dashboard V1",
  "stories": [
    {
      "id": "DASH-01",
      "description": "Scaffold the Dashboard layout with side
bar.",
      "files_to_touch": [
        "src/layout/DashboardLayout.tsx",
        "src/App.tsx"
      ],
      "acceptance_criteria": [
        "Layout renders sidebar on left (w-64)",
        "Sidebar contains links to 'Home' and 'Settings'",
        "Main content area renders children props",
        "Mobile view hides sidebar behind hamburger menu"
      ],
      "passes": true
    },
    {
      "id": "DASH-02",
      "description": "Create UserProfile card fetching from A
PI.",
      "files_to_touch": [
        "src/components/UserProfile.tsx",
        "src/api/user.ts"
      ],
      "acceptance_criteria": [
        "Component accepts `userId` prop",
        "Fetches data from `/api/users/:id`",
        "Displays user avatar (rounded-full)",
        "Displays 'Loading...' state while fetching",
        "Handles 404 error with 'User not found' message"
      ],
      "passes": false
    }
  ]
}
```

3. 📖 README.md (Setup Guide)

```
# 🏗️ Agentic Dream Example Project

This repository demonstrates the **Agentic Dream Framework**
architecture.

## 🚀 How to Run "The Loop"

1. **Install Dependencies:**
```

npm install && brew install jq

```


2. **Check the Contract:**
   Open `docs/specs/stories.json`.
   Find first story where `"passes": false`.

3. **Start Autopilot:**
```

./scripts/autopilot.sh

```


4. **Manual Intervention:**
   If not using the script, follow the **PPRE Cycle**:
   - **PRIME:** Drag `stories.json` + `global.mdc` into Chat.
   - **PLAN:** Ask for a plan.
   - **RESET:** `Cmd+K` to wipe memory.
   - **EXECUTE:** Paste the plan and code.

## 🧠 Memory Locations

```

```
- **Global Rules:** `.cursor/rules/global.mdc`
- **Folder Specific:** Look for `
```

🚀 Implementation Steps

Step 1: Create the Repository

1. Go to GitHub and create new public repo: `agentic-dream-reference-impl`
2. Clone locally:

```
git clone
```

Step 2: Copy All Files

1. Copy all files from this section to the repo
2. Create the exact directory structure
3. Make `autopilot.sh` executable:

```
chmod +x scripts/
```

Step 3: Commit & Push

```
git add .
git commit -m "feat: initial Agentic Dream reference implementation"
git push origin main
```

✅ **Pro Tip:** This repo is your perfect "starting point". Clone it every time you start a new Agentic Dream project and you'll have the complete structure ready in seconds.

🔗 PART 13: LEARNING RESOURCES

External References and Tools

This section contains **external resources, tools, and references** that inspired and support the Agentic Dream Framework. These are curated links to repositories, video tutorials, documentation, and tools that will deepen your understanding of spec-driven development and agentic engineering.

💻 Repositories & Tools (Code & Rules)

Core Tooling

Ralph Asset Management (Official Repository)

- **URL:** https://github.com/allegro/ralph
- **Description:** Asset management system that inspired the "Ralph Protocol" for atomic execution
- **Use for:** Understanding task decomposition and asset tracking patterns

Cursor Best Practices (Rules & Guidelines)

- **URL:** https://github.com/madvaultllc/cursor-best-practices
- **Description:** Community best practices for using Cursor AI editor
- **Use for:** Learning effective prompting and context management strategies

OpenSpec (Spec-Driven Development)

- **URL:** https://github.com/Fission-AI/OpenSpec
- **Description:** Toolkit for structured, spec-driven AI coding projects
- **Use for:** Creating change proposals and living specifications
- 🌟 **Must-watch tutorial:** https://www.youtube.com/watch?v=gHkdrO6IExM

Awesome Cursor Rules (Directory of Rules)

- **URL:** https://github.com/PatrickJS/awesome-cursorrules
- **Description:** Curated collection of Cursor rules for different use cases
- **Use for:** Finding pre-built `.mdc` rules for your tech stack

Cursor Directory (Community Rules)

- **URL:** https://cursor.directory/
- **Description:** Community-contributed Cursor rules and configurations
- **Use for:** Browsing and sharing Cursor AI configurations

Spec-Kit (Toolkit for Spec-Driven Development)

- **URL:** https://github.com/github/spec-kit
- **Description:** GitHub's toolkit for specification-driven development
- **Use for:** Understanding spec authoring and validation patterns

🎬 Videos & Educational Channels

Ralph Wiggum Agent (Workflow)

Created by: Greg Isenberg & Ryan Carson

Resources:

- Workflow Overview: https://startup-ideas-pod.link/Ralph
- AMP Code Example: https://startup-ideas-pod.link/amp-code

Description: Demonstrates the "Ralph Protocol" for breaking down complex tasks into atomic, verifiable units

Key Concepts: Atomic execution, binary acceptance criteria, agent self-verification

SnapperAI (PRD Templates)

- **Channel:** https://youtube.com/@snapperAI

- **Templates:** https://snapperai.io/templates
- **Description:** Practical templates and examples for creating production-ready PRDs
- **Best for:** Learning how to write effective Product Requirements Documents

JeredBlu (MCP & PRD Creator)

- **Channel:** https://youtube.com/@JeredBlu
- **Description:** Tutorials on Model Context Protocol (MCP) and PRD creation workflows
- **Best for:** Understanding context management and specification authoring

OpenSpec Tutorial (Must Watch)

- **Video:** https://www.youtube.com/watch?v=gHkdrO6IExM
- **Duration:** ~12 minutes
- **Topics Covered:**
    - Initializing OpenSpec in your project
    - Creating change proposals and living specs
    - Integrating AI coding assistants (KiloCode, Claude Code, Cursor)
    - Moving from proposal to implementation
    - Keeping workflows clean, auditable, and scalable

👉 **Highly recommended:** Watch this before implementing the framework in your repository

📚 Documentation & Additional Resources

Official Documentation

Ralph Documentation (Quickstart Guide)

- **URL:** https://ralph-ng.readthedocs.io/en/latest/user/quickstart/

- **Description:** Official Ralph asset management quickstart
- **Use for:** Understanding the origin concepts behind atomic task execution

Thought Leadership

Every (Dan Shipper - Compounding AI)

- **URL:** https://every.to
- **Description:** Articles on AI-assisted productivity and compounding engineering
- **Key insight:** How to build systems that get smarter over time
- **Recommended reading:** Search for "Compounding AI" and "AI Memory Systems"

AI Development Tools

Amazon Q Developer

- **URL:** https://aws.amazon.com/q/developer
- **Description:** AWS's AI-powered development assistant
- **Use for:** Enterprise-scale code generation and review

Qodo Merge

- **URL:** https://qodo.ai/merge
- **Description:** AI-powered code review and merge assistance
- **Use for:** Automated PR reviews aligned with your framework rules

Maximus Labs (GEO/AI SEO)

- **URL:** https://maximuslabs.ai
- **Description:** AI-powered SEO and content optimization
- **Use for:** Documentation optimization and discoverability

📌 How to Use These Resources

👶 For Beginners

Recommended Learning Path:

1. 🎬 **Watch:** OpenSpec Tutorial (12 min)
2. 💻 **Explore:** Cursor Best Practices repo
3. 📖 **Read:** Articles on Every.to about Compounding AI
4. 🛠️ **Implement:** Clone OpenSpec and try it in a test project
5. 📄 **Reference:** Use SnapperAI templates for your first PRD

🚀 For Practitioners

Deep Dives:

- 🧑‍💻 **Study the code:** Fork Ralph and analyze its task decomposition patterns
- 🔧 **Build tooling:** Use Spec-Kit to create custom spec validators
- 🤝 **Collaborate:** Share your `.mdc` rules on Cursor Directory
- 📊 **Optimize:** Integrate Qodo Merge into your CI/CD for automated reviews

🏛️ For Enterprise Teams

Integration Resources:

- 🔐 **Security:** Evaluate Amazon Q Developer for enterprise compliance
- 📈 **Analytics:** Use Maximus Labs for documentation discoverability
- 📚 **Training:** Distribute Awesome Cursor Rules to your team
- 🛡️ **Guardrails:** Implement rules from Cursor Best Practices

---

🎓 PART 14: CERTIFICATION

Official Competency Assessment

⚠️ 🚨 **MANDATORY REQUIREMENT** 🚨
Before you are authorized to write code using this framework, you **MUST** complete and pass the official certification quiz.
Passing Score: 90% (Maximum 1 error allowed)
What it covers:

- The Mindset Protocols (understanding "Vibe Coding" vs. Compounding Engineering)
- System Architecture (Memory Architecture, Context Sharding)
- The PPRE Execution Loop (Prime → Plan → Reset → Execute)
- Guardian Scenarios (real-world decision-making)
- Protocol Compliance (Acceptance Criteria, Evolution Loop)

📋 Assessment Details

This is the **Official Competency Assessment** for the Agentic Dream Framework. It consists of 15 questions covering:

- **Section 1:** The Mindset Protocols (True/False)
- **Section 2:** System Architecture (Select All That Apply)
- **Section 3:** The PPRE Execution Loop
- **Section 4:** The Guardian Scenarios
- **Section 5:** Final Protocol Check

✅ Certification Requirements

1. **Complete the reading journey:**
    - Review all parts of this consolidated guide
    - Read the implementation guide completely
    - Understand all architecture components

- Review all learning resources

2. **Take the certification quiz:**
   - 📝 **Access the Official Certification Quiz:** https://forms.gle/ABvRg4qtwXCZZdqB7
   - You must score at least 90% (14/15 correct)
   - Maximum 1 error allowed
3. **Apply the framework:**
   - Only after passing can you begin coding with this framework
   - Your certification demonstrates you understand the Guardian role
   - You are now responsible for maintaining System Evolution

🏆 Ready to Certify?

Once you've completed your study of all framework materials, take the quiz:
👉 **START CERTIFICATION QUIZ**
Remember: This is not a test of memorization, but of understanding. The framework depends on Guardians who truly grasp the philosophy of compounding engineering.

🎓 **Remember:** Documentation is not overhead. It is **compound interest** on your engineering investment. Every minute spent documenting saves hours in the future.
The Agentic Dream Framework transforms linear coding into exponential system evolution. Every bug becomes a lesson. Every lesson becomes permanent intelligence. The system gets smarter every day.
Welcome to the future of AI-assisted development.