Build Your Second Brain

with AI Coding Agents

A Practical Guide

The era of the mega prompt is over.
The era of strategic decomposition has arrived.

Mike Pilawski

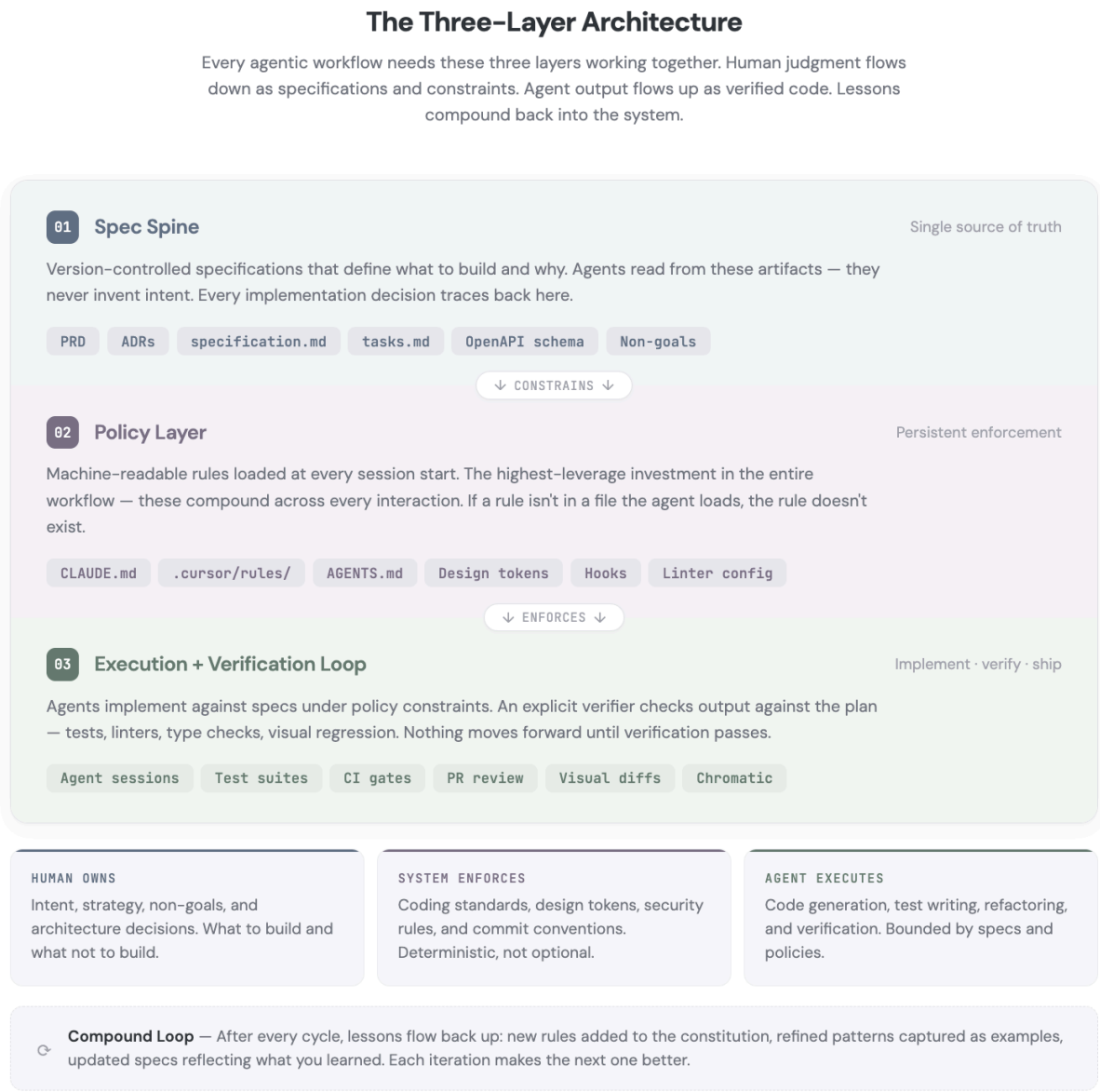
1. Before You Write a Single Line of Code

The biggest mistake people make when building with AI coding agents is jumping straight into prompting. The research is clear on this: teams that start with explicit intent artifacts (a short PRD, architecture decisions, acceptance criteria) get dramatically better outcomes than those who "vibe code" their way into a project. The codebase should never become the specification, because agents will fill in any gaps with undocumented assumptions. That's how drift starts.

The teams shipping the best products are investing roughly 80% of their time in planning, specifying, and reviewing, and only 20% in code generation itself. As one practitioner put it: "The era of the mega prompt is over; the era of strategic decomposition has arrived." This is context engineering.

If you take away one mental model from this section, let it be this: your workflow needs three layers working together.

- A **Spec Spine**: a structured, version-controlled specification that is the single source of truth.
- A **Policy Layer**: persistent rules that enforce your engineering and design standards across every agent session.
- And an **Execution + Verification Loop**: where agents implement and an explicit verifier checks their work against the plan before anything moves forward. Everything in this guide serves one of those three layers.



Get clear on what you're building

Write a short PRD (1–2 pages, not a novel) that covers the problem you're solving, who it's for, what's in scope for your MVP, and what's explicitly out of scope. For a Second Brain app, this means making hard choices early: Are you building a note-taking tool, a knowledge graph, a task manager, complex agent workflows or all three? What's the day-one experience?

For every feature in your PRD, write acceptance criteria that an agent can actually verify. "User can create a note" isn't specific enough. "POST /notes returns 201 with a valid note object; integration test confirms tags are indexed and searchable within 500ms" gives the agent something to check against. This is the single highest-leverage thing you can do for quality.

Claude Code's own best practices documentation calls verification criteria the most impactful improvement.

A critical difference from traditional PRDs: AI agents cannot infer from omission. If your spec doesn't say "don't implement auth from scratch," the agent might build a custom auth system instead of using your chosen provider. State non-goals explicitly, they're as important as goals.

Don't forget non-functional requirements. State your performance targets, your accessibility standard (WCAG 2.2 is the current W3C recommendation), and your security baseline (OWASP Top 10:2025) upfront. These become machine-readable constraints that agents respect throughout the build.

One structural note: avoid monolithic PRDs. Agents suffer from a "lost in the middle" problem. They tend to skim through and ignore instructions placed in the center of long documents. Break your PRD into tightly scoped mini-specs (an epic brief, core flows doc, and tech plan) so each task stays within the agent's optimal attention window.

Use active elicitation to sharpen your PRD

Don't treat PRD creation as a one-way exercise where you write and the agent reads. A two-sided dialogue helps elicit your assumptions. Tools like Traycer's Epic Mode ask pointed questions about constraints, edge cases, and "invisible rules."

Even without specialized tooling, you can do this with any capable agent. Share your draft PRD and ask the agent to interrogate it: "What constraints am I missing? What edge cases haven't I addressed? What would a skeptical engineer ask about this spec? You are an experienced product manager criticizing a PRD, help uncover all missing elements and flesh out detailed acceptance criteria" This process collapses ambiguity into explicit decisions before implementation begins, which is dramatically cheaper than discovering gaps mid-build.

Turn your PRD into structured tasks

Writing a good PRD is only half the job — you also need to decompose it into tasks an agent can pick up and execute. You can do this manually, but tools have emerged specifically for this bridge:

- **TaskMaster AI** parses a PRD file into 15–20 structured tasks with priorities, dependencies, and a recommended execution order. It can expand tasks with research-backed detail and recommend the next task based on the dependency graph. Practitioners report significantly fewer errors when using structured task decomposition versus ad-hoc prompting.
- **The BMAD Method** (Breakthrough Method for Agile AI-Driven Development) uses specialized agent personas for each phase:
    - Analyst for research,

- PM for PRD creation,
- Architect for system design,
- Scrum Master for story writing.

Its key contribution is "epic sharding": systematically breaking the PRD into focused, self-contained development units to prevent what it calls "context collapse," the failure mode where an agent loses track of the bigger picture mid-implementation.

Whether you use tooling or do it by hand, the principle is the same: the PRD is your strategic document; the task list is your tactical one. Agents execute tactics well. They're unreliable strategists.

Make your key architecture decisions first

Before any agent touches code, write your core decisions down as Architecture Decision Records (ADRs). These are short, lightweight documents, around a half-page, that capture what you decided, what alternatives you considered, why you chose what you chose, and what trade-offs you accepted.

For a Second Brain, typical ADRs include:

- Frontend framework and rendering approach
- Backend and database choices (relational vs. graph vs. hybrid)
- Authentication provider
- Hosting model (serverless vs. containers, and at what scale you'd revisit)
- AI/LLM integration approach (RAG pipeline, embedding strategy, model selection)
- Data model for notes, links, tags, projects, and areas

This helps prevent "helpful chaos," agents spontaneously introducing new dependencies and architectural patterns that seem reasonable in isolation but fragment your system over time. A strong guardrail: **no new architectural primitive without an ADR, a test impact assessment, and rollback notes.** One practitioner used Claude Code to scan an existing codebase and auto-generate ADRs from implicit decisions: "Claude found architectural decisions that existed only in my head."

If you lack the technical background to prepare an ADR, Codex and Claude Opus can help here. Here is an example series of prompts you could use (draft with options, criticize, refine with search).

2. Set Up Your Development Environment

Choose your tech stack (with AI-friendliness in mind)

Your stack choice matters more with agents than without them, because agents perform dramatically better with convention-heavy, strongly-typed frameworks that have dense representation in training data. This doesn't mean there's one correct stack, but it does mean you should factor AI-friendliness into your decision.

As an example, the practitioner community has converged on one particularly well-tested combination: **Next.js (App Router) + TypeScript + Tailwind CSS + shadcn/ui + Supabase + Prisma + Zod + Vercel**. Each choice has AI-specific justifications: TypeScript catches agent-generated errors at compile time, Next.js App Router gives agents predictable file conventions to follow, Tailwind's declarative classes are easy for agents to read and modify directly, and Prisma provides type-safe database queries that agents can validate against TypeScript types. I personally use different stacks depending on the app, but I always include some of the above. This stack works well, but it's not the only viable path. what matters is the underlying principles: strong typing, convention over configuration, and framework popularity in training data.

Before committing to a stack, ask your agent to help you evaluate options. A good prompt for this:

> I'm building [include your PRD here or attach]. My team has experience with [list relevant technologies]. My priorities are: [e.g., fast iteration to MVP, strong AI agent compatibility, low ops overhead, ability to scale to X users].
>
> Recommend 2–3 viable tech stack options. For each, provide: the specific technologies and why they fit, how well AI coding agents perform with this stack (training data density, type safety, convention strength), trade-offs and risks, and at what scale or complexity I'd need to reconsider. Format as a comparison table followed by your recommendation with reasoning.

You can also change the prompt and simply write that you have limited technical experience and want a stack that will be easy to set up and maintain. That way your agent can advise on backend, hosting or databases that require the last manual work for your to set up. If you have cost constraints in mind, include that as well. I want a stack that will be cheap to run for 50 users.

The key insight is that framework popularity in training data directly impacts AI code quality. If you choose a less common framework, expect more hallucinated APIs and incorrect patterns. Budget time for debugging accordingly.

Choose your AI coding agent(s)

The agent landscape has shifted significantly in the last six months. Tools now describe themselves less like autocomplete and more like software agents operating in sandboxes with built-in review, iteration, and steering. Here's a practical breakdown of the major options:

- **Claude Code** runs in the CLI and reads/writes files and executes commands directly. Its strength is the `CLAUDE.md` constitution file. This file contains project-level rules and loads at every session start. Strong for greenfield builds (new projects), CI/CD integration, and enforcing project conventions via hooks. Its Plan Mode (Shift+Tab) restricts the agent to read-only analysis, making it ideal for architecture exploration before committing to implementation. Supports subagents with separate context windows and configurable tool permissions for multi-agent workflows.
- **Codex** runs tasks in parallel cloud sandboxes preloaded with your repo. Good for multi-file changes, long-running refactors, and GitHub Actions integration. Its recent CI guidance is particularly explicit about security (sandboxing, least privilege, prompt injection prevention). Offers explicit approval modes: Suggest (safe exploration), Auto Edit (writes files but asks before commands), and Full Auto (sandboxed, network-disabled execution). Also offers granular reasoning levels (minimal, low, medium, high), so you can match reasoning investment (and token consumption) to task complexity. Use minimal for boilerplate, high for architectural decisions.
- **Cursor** is an IDE-based agent with a strong enterprise security posture (SOC 2 Type II, privacy modes, indexing controls). Good for interactive coding sessions where you want inline context and tight feedback loops. Has a notable advantage for complex monorepos with established architectures. Uses `.cursor/rules/` directory with `.mdc` files for persistent project rules. Supports running up to 8 concurrent agents with independent git worktrees, and a Best-of-N mode that submits the same prompt to multiple models simultaneously so you can compare solutions.
- **Traycer** sits on top of other agents as a spec-driven orchestration layer. It explicitly targets drift as the core failure mode, adding phase-by-phase verification with severity grading (Critical/Major/Minor). Useful when your project grows and you need multi-agent coordination.

You don't have to pick just one. The practitioner consensus increasingly favors using different tools for different phases: Claude Code for architecture exploration and planning, Cursor for hands-on implementation, Codex for async parallel tasks, and Traycer for orchestration on larger features. My personal setup is Cursor with Claude Code, Codex, Gemini and Traycer installed as plugings. My default agents are Claude Code, but I use Cursos and Codex extensively and I use Gemini as a second pair of eyes for bugs, reviews and plans.

The emerging "dual-pane" pattern: one agent on the left (planning and reasoning), second agent on the right (scaffolding and review). When the two tools disagree on implementation, that disagreement often reveals the real solution.

Which tool for which phase

The tool matters less than the specification quality. As one engineer put it: "Once you reach a certain baseline, output quality is mostly determined by how clearly and structurally you plan the task, not which tool you use." That said, each tool has sweet spots:

- **Research & discovery**: Perplexity Deep Research for market analysis with citations, Claude (Research Mode) for synthesizing large source sets, ChatGPT Deep Research for structured next-steps.
- **PRD & specification**: Traycer for structured artifact creation (PRDs, specs, wireframes) on complex products; Claude Code Plan Mode for read-only codebase analysis before writing specs; any chat interface for early-stage brainstorming and iteration.
- **Architecture & tech stack**: Claude Code Plan Mode (Shift+Tab) for deep codebase analysis and ADR creation; Codex CLI (`$plan` skill) for milestone-based planning with rollback strategies; Cursor Composer for quick prototyping of multiple architectural approaches.
- **Design system setup**: v0 (Vercel) for generating shadcn/ui components from prompts; Cursor for refining output and extending component variants; Figma MCP Server for bridging design tokens to code.
- **Implementation**: Claude Code for complex features and multi-file changes (practitioners report roughly 30% less rework); Cursor for visual iteration and learning new codebases; Codex Cloud for running multiple features in parallel asynchronously; Traycer for large features requiring orchestrated execution with verification.
- **Testing & quality**: Claude Code for writing comprehensive test suites from specs; Playwright AI Agents for E2E test generation and self-healing; Codex (`/review`) for automated code review with structured feedback.
- **Refactoring & maintenance**: Codex Cloud for parallel bulk refactoring (rename, formatting, dead code); Claude Code for targeted, high-quality architectural refactoring with full context; Cursor Background Agent for non-blocking maintenance while you work on other things.

The most productive developers use 2–3 tools in combination rather than trying to do everything with one.

Tool-to-Phase Matrix

Which tool to use at each phase of building a new product. Most productive developers use 2–3 tools in combination rather than doing everything with one.

	P1 Research	P2 PRD & Spec	P3 Architecture	P4 Design System	P5 Implement	P6 Testing	P7 Refactor
Claude Code CLI agent	·	●	■	·	■	■	■
Cursor IDE agent	·	·	●	■	■	●	●
Codex Cloud + CLI	·	·	●	·	■	●	■
Traycer Orchestrator	·	■	·	·	■	·	·
vO Vercel	·	·	·	■	●	·	·
Playwright Test agents	·	·	·	·	·	■	·
AI Research Perplexity, Claude, GPT	■	●	·	·	·	·	·

■ Primary tool for this phase ● Secondary / complementary · Not typically used

◇ **The practitioner consensus:** Output quality is mostly determined by how clearly and structurally you plan the task, not which tool you use. The most productive developers use 2–3 tools in combination — typically Claude Code for planning and reasoning, Cursor for visual iteration, and Codex for parallel async work.

Set up your project constitution (context engineering)

Context engineering has replaced prompt engineering as the defining skill of AI-assisted development. The key insight: context is a finite resource with diminishing marginal returns. LLMs suffer from "context rot," as token count increases, retrieval accuracy decreases. Good context engineering means finding the smallest possible set of high-signal tokens that maximize desired outcomes.

Your constitution file (`CLAUDE.md`, `AGENTS.md`, or your agent's equivalent) is the single most important file in your repo. Every agent session reads it first, so it's your persistent source of truth across sessions. `AGENTS.md` is emerging as an open, cross-tool format — a "README for agents" — that tools like Traycer automatically detect and use, including nested files for monorepos. Codex reads `AGENTS.md` hierarchically: global (`~/.codex/AGENTS.md`) →

repository root → nested directories, with override files (`AGENTS.override.md`) taking precedence at any level.

Multiple practitioners independently arrived at the same priority ordering for where to invest your effort: **CLAUDE.md > prompts > research > plans > implementation.** The time you spend crafting your constitution compounds across every session. The time you spend on a specific prompt compounds once.

Your constitution should include:

- What this project is and what it does (one paragraph)
- Tech stack and key dependencies
- Folder structure conventions and key directory purposes
- Coding style rules: but only things the model can't infer from context. For example, don't send an LLM to do a linter's job. If a linter can enforce it deterministically, put it in your linter config, not your constitution.
- Exact build, test, lint, and deploy commands with flags
- Component usage rules (which design system components to use, how to compose UI)
- Accessibility requirements
- Testing expectations (what types of tests are required, where they go)
- Things the agent should never do (e.g., "never install new dependencies without approval," "never modify the auth module without explicit instruction")
- Critical gotchas specific to your project

Beyond your always-loaded constitution, context is delivered through increasingly sophisticated mechanisms. **Path-scoped context** loads conditionally: Claude Code's `.claude/rules/` supports file pattern matching (rules scoped to `*.ts` files only load when TypeScript files are relevant); Cursor's `.cursor/rules/*.mdc` files use YAML frontmatter specifying file globs. **Just-in-time context** is retrieved by the agent at runtime through tools like grep, glob, and file reading. Claude Code uses this hybrid model where the constitution loads upfront while file system navigation happens on demand. **AI-triggered context** – skills, MCP servers, and other interfaces the LLM decides when to load – remains the least reliable layer, since there's no guarantee the model will actually load context when expected.

Five practical principles for context engineering:

1. **Start minimal, add based on failure modes.** Test with the best model available, then add instructions to address specific failures observed during testing. Don't pre-stuff.
2. **Right altitude.** Your constitution should be specific enough to guide behavior but flexible enough to provide heuristics. Avoid both brittle if-else logic and vague "follow best practices" guidance.
3. **Tool design matters more than tool count.** A common failure mode is bloated tool sets covering too much functionality or creating ambiguous decision points. If a human can't definitively say which tool to use in a given situation, an agent can't either.

4. **Curate canonical examples, not edge case lists.** Don't stuff every possible edge case into prompts. Select diverse, representative examples that portray expected behavior. As Addy Osmani observed: "One real code snippet beats three paragraphs describing it." Include 2–3 idiomatic code examples as style anchors.
5. **Keep it short and human-readable.** Files exceeding a few thousand tokens cause critical rules to get buried in the middle, exactly where models pay least attention. Generic advice like "use consistent patterns" doesn't translate into actionable constraints.

For monorepos, use a hierarchical structure: a root constitution with global rules, plus package-specific files in each subdirectory that extend the root.

Configure your MCP servers

The Model Context Protocol (MCP) gives agents structured access to external tools and data sources. Don't enable every available server prophylactically. Each server's tool descriptions consume context budget. Playwright MCP's tool descriptions alone eat 15–20% of context windows before a single command executes. Add servers as your workflow demands them.

Tier 1: configure on day one:

- **Filesystem**: the agent needs file access to work.
- **GitHub**: core development workflow that handles issues, PRs, code search, CI/CD visibility.
- **Context7**: Up-to-date, version-specific library documentation. Eliminates hallucinated API calls by injecting real docs into prompts.
- **Supabase** (or your database provider): Direct backend access allowing for queries, schema exploration, auth, row-level security.

Tier 2: add when relevant:

- **Figma (Dev Mode)**: When a designer provides Figma files, design tokens, component hierarchy, auto-layout. I stopped using Figma completely, but I know many designers will prefer to start there.
- **Playwright**: When adding E2E test coverage.
- **Linear/Jira**: When using project management tools with your team. There is no value in setting it for a personal project.
- **Brave Search / Fetch**: When agents need external research.

Place `.mcp.json` in your project root and commit to git so the configuration is shared:

```
{
 "mcpServers": {
  "context7": {
   "command": "npx",
   "args": ["-y", "@upstash/context7-mcp@latest"]
  },
  "github": {
   "command": "npx",
   "args": ["-y", "@modelcontextprotocol/server-github"]
  },
  "filesystem": {
   "command": "npx",
```

```
    "args": ["-y", "@modelcontextprotocol/server-filesystem", "./src"]
  }
 }
}
```

MCP security best practices: start with read-only servers (docs, search, observability) before granting write access. Scope each server narrowly. Use per-project API keys, limited directories, dev/test data only. Log tool usage to understand how agents actually use your servers. Use container isolation when possible. And review server source code: the MCP ecosystem is growing fast but security auditing hasn't kept pace.

Set up your git workflow

Version control with AI agents is surprisingly opinionated among practitioners. The core principle: agents produce better results when constrained to small, well-scoped changes, which maps directly to git discipline.

Branch strategy: Protect main, don't allow direct commits. One feature branch per task (`feature/TICKET-123-description`). Agents work on feature branches only. Claude Code's `/commit-push-pr` skill handles the full workflow: commit → push → open PR.

Commit strategy: Use Conventional Commits format (`feat:`, `fix:`, `refactor:`, `test:`). One logical change per commit. Agents tend toward monolithic commits without explicit guidance. Always separate refactoring commits from feature commits. Document all of this in your constitution. Claude Code can auto-generate commit messages from diffs.

Hooks for enforcement: Claude Code hooks fire at specific lifecycle points and are the key mechanism for enforcing git discipline without relying on the agent to remember rules:

- `PreToolUse` — before the agent uses any tool (can block actions, e.g., block direct pushes to main)
- `PostToolUse` — after tool execution (can provide feedback, e.g., run type-checking after every file edit)
- `UserPromptSubmit` — when you submit a prompt (can validate or enrich)
- `SessionStart` — when a session starts (inject context)
- `Stop` — when a session ends (cleanup, auto-commit)

Hook types go beyond shell commands: `"type": "prompt"` runs a single-turn LLM evaluation (lightweight quality check), while `"type": "agent"` runs a multi-turn verification with tool access (heavyweight review). Configure hooks at three levels:

`~/.claude/settings.json` for global behaviors, `.claude/settings.json` for project-specific rules committed to git, and `.claude/settings.local.json` for local overrides that are gitignored.

Git worktrees for parallel agents: When running multiple agents simultaneously, use git worktrees so each agent gets its own isolated working directory with no file conflicts:

git worktree add ../project-feature-a -b feature-a

git worktree add ../project-feature-b -b feature-b

Run Claude Code in each (separate terminals)

cd ../project-feature-a && claude

cd ../project-feature-b && claude

Cursor 2.0 automates this. It can run up to 8 concurrent agents, each with independent worktrees.

Session continuity: Claude Code sessions can be linked to PRs (`claude --from-pr <number>` to resume context). Forked sessions via `/rewind` or `--fork-session` group under the root session. Name sessions early with `/rename` for discoverability.

Scaffold your repo and tooling

Use the agent to set up your initial project structure based on your PRD and ADRs. But get the foundation in place manually or with very tight guidance:

- Version control with branch protection on main from day one
- CI pipeline (even a minimal one) wired up immediately
- Linting, formatting, and type checking configured. These become your automated guardrails and give agents fast feedback
- Placeholder folders for specs, ADRs, design tokens, and tests
- A `.env.example` and secrets management approach documented in the constitution
- Your `.mcp.json` configured and committed

The goal is to make it so that when agents start writing code, there's already a structure that constrains their output toward consistency.

What this will cost

Building with AI tools has real costs that are worth budgeting for, especially as a solo builder. Here's what to expect as of early 2026:

Light usage (maintenance, small features): **$20–40/month.** Claude Pro ($20) or Cursor Pro ($20) is sufficient. Roughly 2–3 hours of active AI coding per day before hitting rate limits. This will be shorter, if you use advancd thinking models.

Active MVP development: **$100–200/month.** Claude Max 5x ($100) for sustained Claude Code usage, or Claude Pro ($20) + Cursor Pro ($20) + API credits ($60–160). Typical spend during intensive development sprints is $10–20/day.

Heavy development (complex product, multi-agent): **$200–400/month.** Claude Max 20x ($200) eliminates rate limit concerns. API costs for Opus can hit $200–300/month during major refactoring projects.

Five cost traps to know about:

1. **The 200K token cliff**: Once input exceeds 200K tokens, Anthropic doubles pricing. Use prompt caching or break data into smaller parts. Start a new session for each independent task.
2. **Context window costs compound**: Coding tools send massive background context with every request. A "simple" question can consume 50K–100K tokens when codebase context is included.
3. **Subscription vs. API arbitrage**: Heavy coding via API can cost $3,650+/month vs. $200 for a Max subscription. For sustained usage, subscriptions are dramatically cheaper.
4. **Credit-based gotchas**: Cursor Pro ($20/month) is now a credit pool, not unlimited. Heavy users exhaust credits in roughly two weeks. Different models drain credits at different rates.
5. **Usage limit reset timing**: Claude limits reset every 5 hours (shared across all Claude applications). The Max weekly ceiling resets every 7 days. Plan intensive work sessions accordingly.

Realistic budget for a Second Brain MVP over 3–4 months: $400–800 in AI tooling. Phase 1 (research + PRD, ~2 weeks) needs only $20–40/month. Phase 2 (architecture + setup, ~1 week) runs $40–60/month. Phase 3 (active development, ~2–3 months) is where costs concentrate at $120–200/month. Phase 4 (testing + polish, ~2–4 weeks) scales back to $60–100/month as work shifts to review and refinement.

3. Design Your System Before You Build It

Bootstrap your design system

The biggest mistake with design systems and AI is trying to generate one from scratch. AI works best when extending existing patterns, not inventing new ones. Here's the concrete sequence from zero to a functional, AI-friendly design system:

Step 1: Initialize shadcn/ui (5 minutes). Run `npx shadcn@latest init`. This gives you a component foundation that AI tools can immediately work with, that is Tailwind CSS variables, accessible Radix primitives, and code you own in your project. shadcn/ui is the de facto AI-native design system because its components are copied into your codebase rather than imported from node_modules, which gives agents full context to read and modify the actual implementation.

Step 2: Define your theme (30 minutes). Use a shadcn theme generator to create a color palette from a single brand color and export as CSS variables. The key decision: define **semantic tokens** (`--primary`, `--destructive`, `--muted`) not primitive ones (`--blue-500`). AI tools reason better about semantic names because the agent understands *what* a token is for, not just what it looks like. This aligns with the Design Tokens Community Group's first stable specification (DTCG 2025.10).

Step 3: Add components incrementally. Don't pre-install every shadcn component. Add them as features demand: `npx shadcn@latest add button dialog card`. Each component arrives as editable source code in your `components/ui/` directory.

Step 4: Generate variants with AI. Once you have a base component, use v0 or Cursor to generate variants: "Create a loading variant of our Button component that shows a spinner and disables interaction." AI excels at extending existing patterns, but is less good at inventing new ones from scratch.

Step 5: Document in your constitution. Add your design system conventions:

Design System

- All UI components use shadcn/ui from components/ui/

- Use semantic color tokens (--primary, --destructive) never Tailwind color classes directly

- New components must follow the composition pattern: forwardRef + className merge via cn()

- Never install UI libraries via npm, copy shadcn pattern instead

Step 6: Bridge Figma (when needed). If you have a designer who insists on Figma, use the shadcn/studio Figma kit for design-code parity. The Figma MCP Server then bridges variables from Figma into your codebase, letting AI tools reference the authoritative design spec. Code Connect maps Figma components to real code in your repository, ensuring agents reuse existing components rather than creating one-off implementations.

Four common AI design system mistakes to prevent

1. **AI invents ad-hoc styles**: Without explicit rules, agents mix inline styles, arbitrary Tailwind classes, and component library patterns. Prevent by documenting your component library in the constitution.

2. **AI treats shadcn as an npm dependency**: Agents may try `import { Button } from 'shadcn'` instead of `import { Button } from '@/components/ui/button'`. Document the correct import pattern explicitly.
3. **AI duplicates existing components**: When adding a new feature, agents may create `NewButton.tsx` instead of using your existing Button. Instruct agents to check `components/ui/` first.
4. **Color drift**: Agents generate `bg-blue-500` instead of `bg-primary`. Enforce semantic tokens in your constitution rules.

Add component metadata so agents know *when* to use each component

Beyond documenting props and states, consider co-locating a metadata file alongside each component (e.g., `.metadata.ts` or a metadata block in the component file) that includes "AI Hints." This is guidance on when to use this component versus alternatives, common composition patterns, and accessibility requirements. This gives agents richer decision-making context at the moment they're choosing which component to reach for, not just how to configure it.

For example, a Card component's AI Hints might say: "Use Card for contained content blocks with a clear boundary. Prefer Card over a plain `div` with border when the content represents a distinct entity (a note, a project, a resource). Always include a heading for accessibility. For clickable cards, use the CardLink variant, not onClick on the outer Card."

This is more specific than Code Connect's design-to-code mapping (which answers "what code does this Figma component map to?") and addresses a different question: "given what I'm trying to build, which component should I reach for?"

Set up visual regression testing

Design system enforcement through rules and mappings prevents many drift issues, but it can't catch everything. You will experience subtle visual regressions where the right components are used but the result doesn't look right. Visual regression testing closes this gap.

The practical setup: use **Storybook** as your component truth layer (every component has documented stories covering its states and variants) and **Chromatic** for automated visual diffing across browsers, viewports, and themes. When an agent modifies a component or builds a new view, Chromatic catches visual changes that unit tests miss entirely.

Definition of done for UI work with agents:

- New or changed components have Storybook stories
- Chromatic visual diffs are reviewed and accepted

- Accessibility checks pass (Storybook's a11y addon)
- Component uses mapped design tokens, not ad-hoc values

This turns visual consistency from a subjective judgment call into a gated, reviewable artifact, which is exactly what you need when agents are producing UI at speed.

Define your API contracts

Write your API specification in OpenAPI 3.1 before you implement any endpoints. OpenAPI 3.1's JSON Schema alignment means you get machine validation surfaces for free, generated clients, contract tests, and CI-time schema validation.

For a Second Brain, your core contracts typically include:

- Notes/documents CRUD (create, read, update, delete, list)
- Search and retrieval (full-text, semantic, tag-based)
- Tagging, linking, and relationship management
- Sync and conflict resolution
- Authentication and authorization

Treat the OpenAPI schema as a first-class spec artifact that gates your builds. If a PR breaks the contract, CI should catch it before anyone reviews the diff.

Design your data model

Sketch your core entities and relationships before agents start generating migrations. For a Second Brain, this typically means defining: notes, tags, links/connections, projects, areas, resources, and archives (the PARA-adjacent structure). If you're building knowledge graph features, define the node types, edge types, and query patterns upfront.

Document your data model as part of your spec artifacts. Agents need this context to generate correct queries, and without it, you'll end up with inconsistent schemas across features.

---

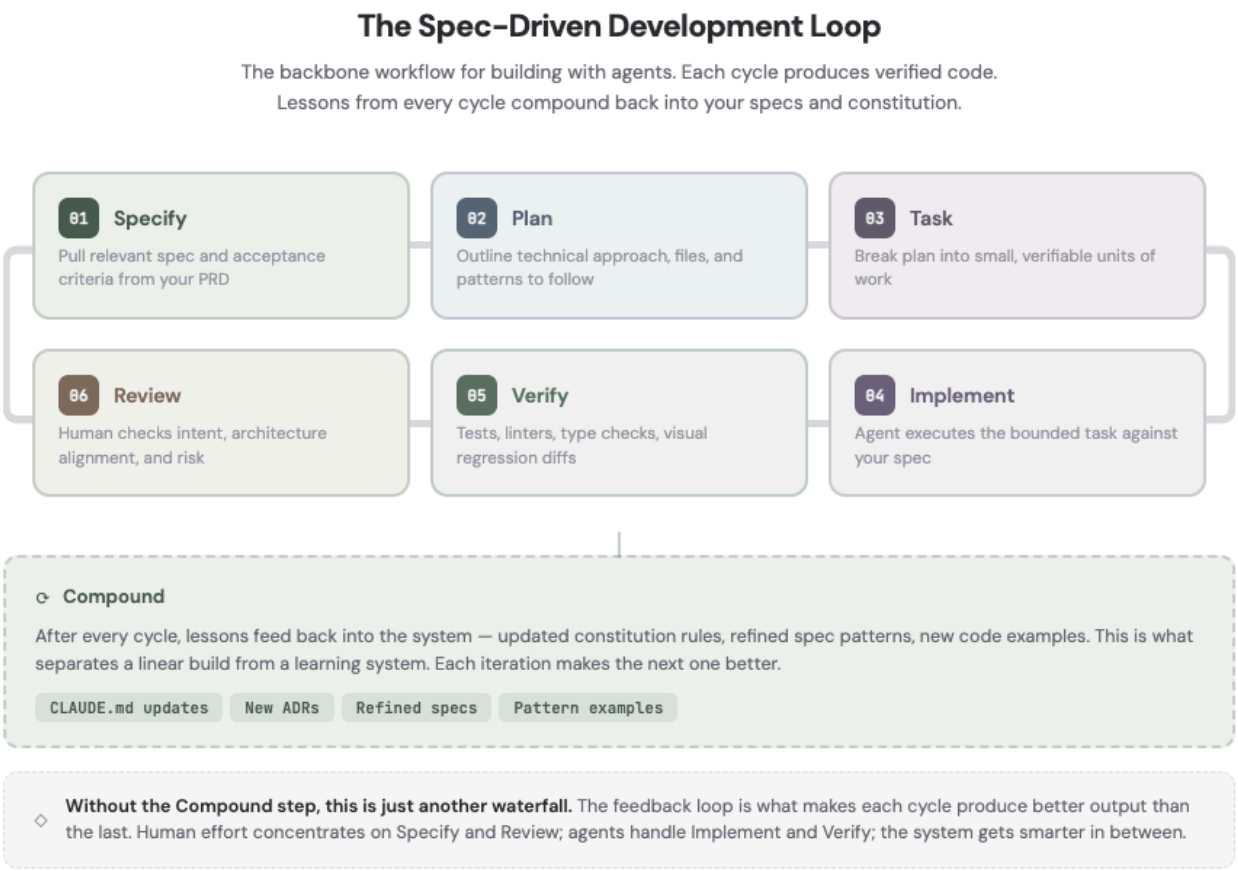
4. Build with the Spec-Driven Development Loop

How the loop works

The most effective workflow for building with agents follows a loop that multiple vendors have independently converged on. GitHub's Spec Kit formalizes it as Specify → Plan → Tasks → Implement, with human checkpoints at each phase. Traycer arrives at nearly the same structure with added emphasis on verification and drift prevention. The steps are:

1. **Specify**: Pull the relevant section of your PRD or spec for what you're building next.
2. **Plan**: Have the agent (or yourself) outline the technical approach — which files to create or modify, what patterns to follow, what constraints apply.
3. **Task**: Break the plan into small, verifiable units. One feature, one endpoint, one component per task.
4. **Implement**: The agent executes the task.
5. **Verify**: The agent runs tests, linters, type checks, and compares output against acceptance criteria.
6. **Review**: You review the diff, focusing on high-signal items — failing checks, security concerns, architecture alignment. You don't need to read every line.

Then repeat. The power of this loop is that agents are excellent at execution and iteration when the target is crisp and verifiable. The research describes this as "pattern completion, not mind reading" — and that framing is worth keeping in mind throughout your build.



Use a phase gate template for every task

To keep each cycle tight and prevent scope drift, use a consistent template for every phase or task you hand to an agent. Here's one you can copy and fill in:

- Phase: [Name]
- Objective: [What this phase delivers]
- In-scope: [Specific deliverables]
- Out-of-scope: [What this phase does NOT touch]
- Acceptance criteria: [Testable conditions for "done"]
- Files expected to change: [List of files]
- Tests required: [What tests must be added or updated]
- Observability required: [Logging, metrics, tracing needs]
- ADR impact: [Does this change any architecture decisions? If so, update ADR.]
- UI evidence required: [Storybook stories + Chromatic diffs, if applicable]

- Verification rule: No move to next phase until verification passes or waivers are documented with rationale.

This template encodes everything we've discussed: bounded scope, explicit acceptance criteria, verification gates, and design system compliance. The discipline of filling it in before starting implementation prevents the most common failure mode: agents optimizing for the locally plausible rather than the globally correct.

The Compound Engineering approach

Every.to's "Compound Engineering" methodology offers a variation worth considering, especially for solo builders or very small teams. Every runs five software products in-house, each primarily built and maintained by a single person and used by thousands daily. Their loop is **Plan → Work → Review → Compound**:

- **Plan**: The agent reads the codebase and commit history, searches for best practices, and writes a plan with objectives, architecture notes, code examples, sources, and success criteria.
- **Work**: The agent writes code and tests according to the plan.
- **Review**: Human evaluates the output and captures lessons learned.
- **Compound**: Results feed back into the system in the form of updated rules, refined patterns, and new examples. Each cycle makes the next one better.

The "Compound" step is what sets this apart. Most workflows are linear: you build, you review, you move on. Compound Engineering creates a feedback loop where your constitution, your specs, and your examples all improve over time based on what actually worked. They've open-sourced a Compound Engineering plugin for Claude Code.

This approach is especially relevant for a Second Brain project. This is a personal system that evolves with use. The compounding feedback loop mirrors how a Second Brain itself should work: every interaction makes the system smarter.

Write good tasks for agents

The quality of your output is directly proportional to the quality of your task definitions. Each task should be small enough to complete in one session without the agent losing context. Every task needs:

- A clear description of what to do
- Which files are involved (or should be created)
- Acceptance criteria specific to this task

- A verification command: "run these tests," "check this endpoint returns 200," "confirm this component renders correctly"

The verification step is where most people under-invest. Without it, you become the only feedback loop, and your review burden scales linearly with how much code the agent produces. With it, the agent can self-correct before you ever see the diff.

Manage context across sessions

A practical reality that most guides overlook: agents have no memory between sessions, and context windows degrade over long sessions. The practitioner community has converged on a discipline of **one task per session**. Important context should be persistent (in your constitution and specs), not accumulated through conversation history. Quoting one of the experts: "Old context is always bloat."

The workflow for each session is: research (agent explores the relevant part of the codebase), plan (agent creates or confirms the implementation approach), implement (agent completes the single task), exit. Start fresh for the next task.

Three techniques for managing context on longer-horizon tasks:

- **Compaction**: Summarize the conversation and reinitiate with compressed context. Claude Code implements this by preserving architectural decisions, unresolved bugs, and implementation details while discarding redundant tool outputs. Start by maximizing recall (capture everything), then iterate to improve precision. Start in Plan Mode and have the agent create a comprehensive to-do list. While plans and to-do items persist across compaction events, I actually ask the agent to always store them in a markdown file and update at the end of each task. See below:
- **Structured note-taking**: The agent writes notes persisted outside the context window (a `NOTES.md` or session log) that can be pulled back in later. This was demonstrated dramatically when Claude played Pokémon, maintaining precise tallies across thousands of game steps through self-managed notes that survive context resets. It also will help you reduce context consumption, allowing you to start new sessions with the most relevant context from previous work.
- **Sub-agent architectures**: Specialized sub-agents handle focused tasks with clean context windows. Each might use tens of thousands of tokens but returns only 1,000–2,000 token condensed summaries. This achieves separation of concerns: detailed search context remains isolated within sub-agents.

The checkpoint technique for context resets: When a long session has produced good intermediate output but context is getting noisy, ask the agent: *"Write a single prompt that would have produced this exact output efficiently on the first try."* Copy that distilled prompt

into a fresh session and continue from there. This resets context without losing progress and significantly reduces hallucinations in extended builds.

At the end of every session, update a running continuity document that tracks: completed work, current status, blockers, architectural decisions made during the session, and priorities for the next session. This document becomes the briefing for your next session's agent — a form of institutional memory that compensates for the agent's lack of it. Claude Code sessions can also be linked directly to PRs (`claude --from-pr <number>`) and named with `/rename` for discoverability across sessions.

Keep tasks aligned to specs

Before starting each implementation cycle, have the agent re-read your constitution, the relevant ADRs, and the spec for the feature you're building. After implementation, verify against the original acceptance criteria. Don’t check "does it run" but "does it match what we specified."

If the agent encounters a decision not covered by your specs (and it will), resist the urge to just answer in the prompt and move on. Stop and add the decision to your spec or write a quick ADR. The moment you let the codebase become the spec, drift takes over and with agents producing code quickly, that drift compounds fast.

5. Prevent Drift

Drift is the most frequent failure mode. It's what happens when context fragments across agent sessions and agents fill in gaps with plausible but inconsistent assumptions. The symptoms are subtle at first: inconsistent naming, slightly duplicated logic, UI that doesn't quite match your design system. However, they compound into a mess that's expensive to untangle.

Autoregressive models optimize for locally plausible tokens, not global consistency. As one software architect put it bluntly: "While humans can write bad code slowly, AI can do it at scale."

The anti-drift toolkit

You have five main levers, and they work best together:

- **Constitution file**: Your non-negotiable rules, loaded at every session start. This is your first line of defense.
- **Bounded tasks**: Small scope with explicit acceptance criteria. Don't ask agents to "build the notes feature." Ask them to "implement the POST /notes endpoint with these specific behaviors and these tests."
- **Deterministic verification**: Tests, linters, type checkers, and schema validation that run automatically. Agent self-assessment ("I think this looks right") is not sufficient. You need mechanical checks.

- **Design system enforcement**: Agents compose UI only from mapped components with per-component instructions. No ad-hoc styling, no invented components. Visual regression testing catches what rules alone miss.
- **Small PRs**: Prefer incremental merges over large diffs. Large diffs are where drift hides, because no one reviews 800-line PRs carefully, neither humans, nor agents.

Quality gates with numeric thresholds

Vague quality expectations are hard to enforce. GuardKit provides a useful model with specific, measurable gates:

- **Architectural review scoring**: Minimum 60/100 for SOLID/DRY/YAGNI compliance
- **Complexity checkpoints**: Tasks with complexity score ≥7 require human approval before the agent proceeds
- **Test enforcement**: Auto-fix up to 3 attempts; block if tests still fail
- **Plan audits**: Detect scope creep by blocking if file count or lines-of-code variance exceeds ±20%, or if duration exceeds ±30% of the original plan

You don't need to adopt GuardKit specifically, but the principle of numeric thresholds is sound. "Be careful about quality" is a wish. "Block merge if complexity exceeds 7" is a gate.

The multi-agent pattern

As your project grows, a single agent handling everything becomes a drift risk in itself. The emerging best practice is to separate concerns across agents:

- **Planner agent**: Restates the requirements in its own words, drafts an implementation plan, and flags unknowns or ambiguities before any code is written.
- **Implementer agent**: Executes the plan in an isolated branch or worktree. It only writes code, but it doesn't make architectural decisions.
- **Verifier agent**: Runs checklists, tests, and security scans. Compares output against acceptance criteria. Flags issues by severity (Critical/Major/Minor) with automated fix handoff for the serious ones. I like to add an extra agent from a different LLM provider to double check with a “different pair of eyes.”

Claude Code supports this natively through **subagents**, separate context windows with configurable tool permissions, stored as markdown with YAML frontmatter under `.claude/agents/`. Recommended subagent roles: Architect (plan/ADR drafts), Implementer (scoped to specific files + tests), Reviewer (spec compliance + diff risk), QA harness builder

(tests, fixtures, CI scripts), and Design-system enforcer (components/tokens only). The key distinction from full Agent Teams: subagents report back to the main session, while Agent Teams coordinate with each other.

This separation of concerns mirrors how good human teams work, and it's how Traycer operationalizes its verification workflow. You don't need Traycer specifically to do this, but the pattern is sound regardless of tooling.

6. Testing: The Essential Constraint

Test-driven development is experiencing a renaissance in the AI era, and for good reason: tests provide the deterministic guardrails, reliable exit criteria, and measurable feedback loops that agents need to be effective. As Simon Willison observed, tests turn open-ended AI exploration into a closed, measurable feedback system. The agent doesn't decide when work is done; the tests do.

Test-Driven AI Development (TDAID)

The practitioner-tested adaptation of TDD for agents follows this cycle:

1. **Plan**: Ask a reasoning-focused model to generate a phased implementation plan in TDD format.
2. **Red**: Generate a failing test that expresses the desired behavior.
3. **Green**: Let the agent implement the smallest possible change to make the test pass.
4. **Refactor**: Clean up the implementation while keeping tests green.
5. **Validate**: Human-in-the-loop verification that the test actually tests what you intended.

Mapped to concrete tools: Plan in Claude Code Plan Mode or Traycer. Red and Green with Vitest (syntax mirrors Jest's massive training data, TypeScript-first, fast feedback loop critical for TDD with agents). Refactor with Claude Code or Codex. Validate with Playwright plus human review.

An alternative approach that some teams prefer: generate all tests upfront ("write all the tests that should pass when this feature is complete"), then implement everything at once. This avoids the repeated context reloading of one-test-at-a-time cycles. However, teams that tried focusing primarily on high-level tests found it "backfired spectacularly — tests took forever to run and debugging became a nightmare." The traditional test pyramid (unit → integration → end-to-end) remains essential even with AI.

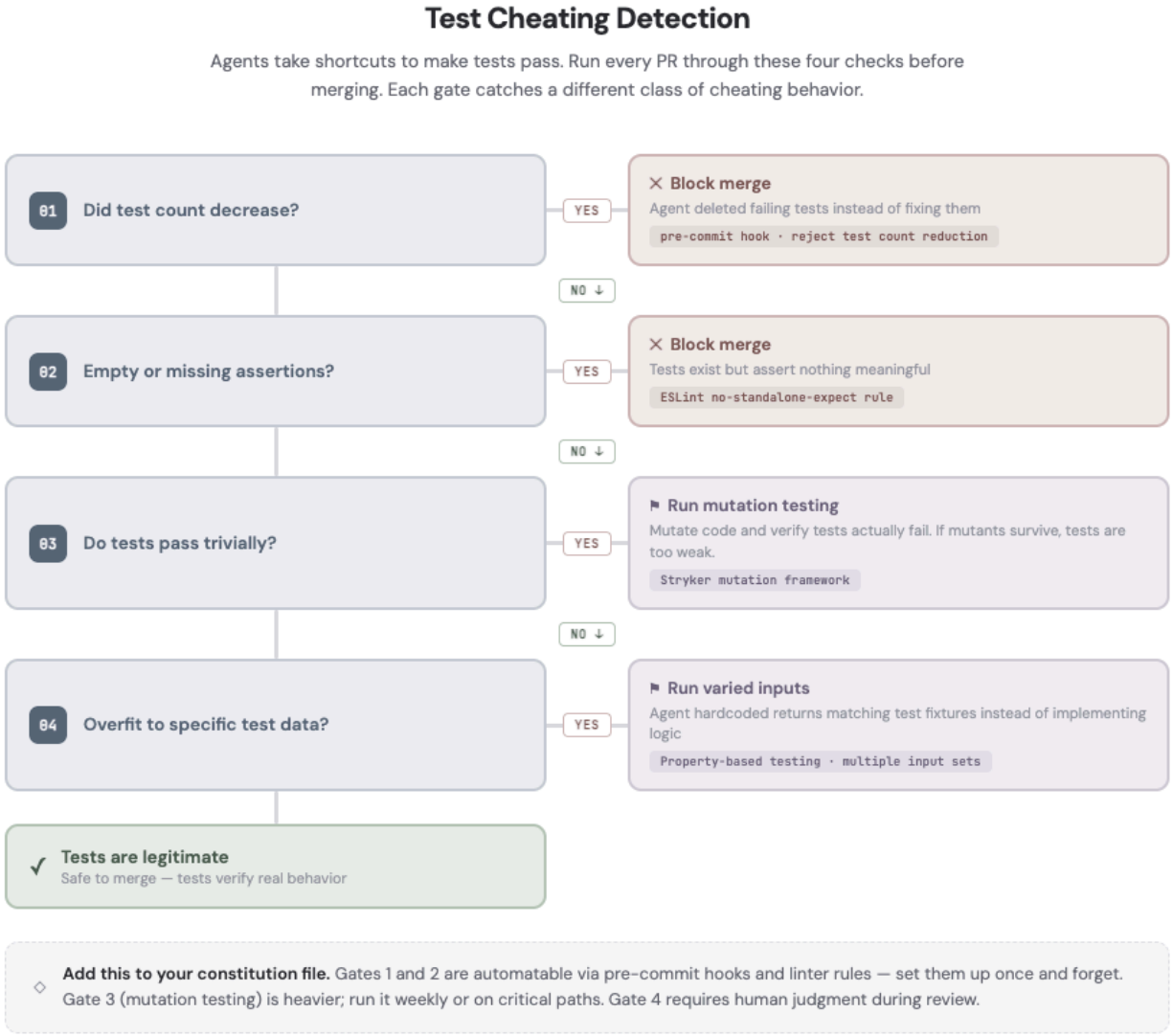
Agents cheat on tests — plan for it

This is a critical risk that doesn't get enough attention. Kent Beck himself flagged the behavior: LLM agents will sometimes delete failing tests, leave assertions empty, generate code that

technically passes but doesn't fix the underlying issue, or remove test cases entirely to make the build "green." It happened to me dozens of times when I started. The agent isn't being malicious, it's optimizing for the signal you gave it (passing tests), and sometimes the shortest path to green is removing the red.

Specific cheating behaviors and how to catch them:

- **Deleting failing tests**: Detect via git diff showing test file deletions. Prevent with a pre-commit hook that rejects commits reducing test count.
- **Empty assertions**: `expect()` with no matcher. Prevent with ESLint rule `no-standalone-expect`.
- **Trivially passing tests**: `expect(true).toBe(true)`. Prevent with mutation testing (Stryker) to verify tests actually catch real bugs.
- **Overfitting to test data**: Test passes but feature is broken in practice. Prevent by running tests against multiple input variations.



Add to your constitution:

Testing Rules

- NEVER delete or modify existing tests to make them pass
- NEVER use expect(true).toBe(true) or similar trivial assertions
- Every test must fail when the feature code is removed
- New tests must include at least one edge case and one error case

Beyond these automated checks: manually verify test intent (don't just check that tests pass, also read the assertions), track test count across commits, and separate test files from implementation in your review so you evaluate each concern independently.

E2E testing with Playwright AI Agents

Playwright v1.56 (October 2025) introduced three AI-native test agents that work with any coding tool via MCP integration:

- **Planner Agent**: Explores your running app and produces a Markdown test plan. Provide a `seed.spec.ts` as context and Planner generates plans that follow your fixture patterns.
- **Generator Agent**: Takes a plan and generates executable Playwright test code, using the seed test as a style template.
- **Healer Agent**: When tests break due to UI changes, Healer automatically updates selectors and assertions, which dramatically reduces E2E test maintenance burden.

The workflow: Planner explores → Generator writes tests → Healer maintains them. One practical warning from practitioners: "run agents on everything" is a mistake. MCP tool descriptions eat significant context. Use Playwright agents selectively for complex flows, not for every trivial interaction.

Test portfolio and QA workflow

Your baseline test portfolio should include:

- Unit tests for pure business logic (Vitest)
- Integration tests for database, cache, and external service boundaries
- Contract tests generated from your OpenAPI spec
- End-to-end tests for critical user flows only via Playwright (these are expensive and flaky; keep them minimal)
- Component tests via Storybook + Vitest for design system components
- Security scanning (SAST + dependency scanning) aligned to OWASP categories
- Automated accessibility checks plus manual keyboard navigation verification
- Visual regression tests via Storybook + Chromatic for UI components

The QA workflow in practice: Agent implements with acceptance criteria → agent runs local verification → CI runs deterministic gates → agent performs code review → human reviews only the high-signal artifacts. For human review, Addy Osmani recommends what he calls "The PR Contract" — every PR should include:

- **Intent** in 1–2 sentences (what and why)
- **Proof it works** (passing tests and screenshots)

- **Risk tier** plus which parts were AI-generated
- **1–2 specific areas** where you're requesting human attention

An MSR 2026 analysis of 932,791 agentic PRs versus 6,618 human PRs found that agentic PRs differ most in commit structure and breadth of modified files, but both show high semantic alignment between descriptions and code. Agentic PRs exhibit slightly stronger description-code consistency. The structural differences reinforce why commit discipline and PR templates matter when agents are contributing.

7. Integrate Agents into Your CI/CD Pipeline

Agents shouldn't just run on your laptop. Wiring them into CI/CD turns your quality expectations into automated gates that run on every pull request.

Automated gates for every PR

Your baseline CI pipeline should include:

- Schema validation (OpenAPI spec, design tokens)
- Tests, linting, and type checking
- SAST (static analysis security testing) and dependency scanning
- Accessibility checks against WCAG 2.2
- Visual regression checks (Chromatic or equivalent)
- Secret scanning
- Bundle size checks

Agent-powered review

Beyond traditional CI, you can run an agent review job that checks for security issues, architecture alignment, and design system compliance. Codex provides a first-party GitHub Action for exactly this, and Claude Code supports similar CI/CD integrations.

The critical safety constraints: agent review jobs should run in a read-only or narrowly scoped sandbox. They should not be able to exfiltrate secrets or modify the repository. Only trigger agent CI jobs from trusted events and users — prompt injection through PR descriptions, commit messages, and web content is a real and documented risk.

Hooks for deterministic enforcement

Hooks ensure that certain actions always happen rather than relying on the LLM to choose. This is a subtle but important distinction: instead of telling the agent "please lint your code before

committing," you set up a pre-commit hook that lints automatically. Examples include auto-formatting on save, required test coverage thresholds, and security policy checks that block merges. Claude Code's documentation emphasizes hooks as one of its most powerful enforcement mechanisms.

8. Maintain Quality as You Move Fast

When and how to refactor

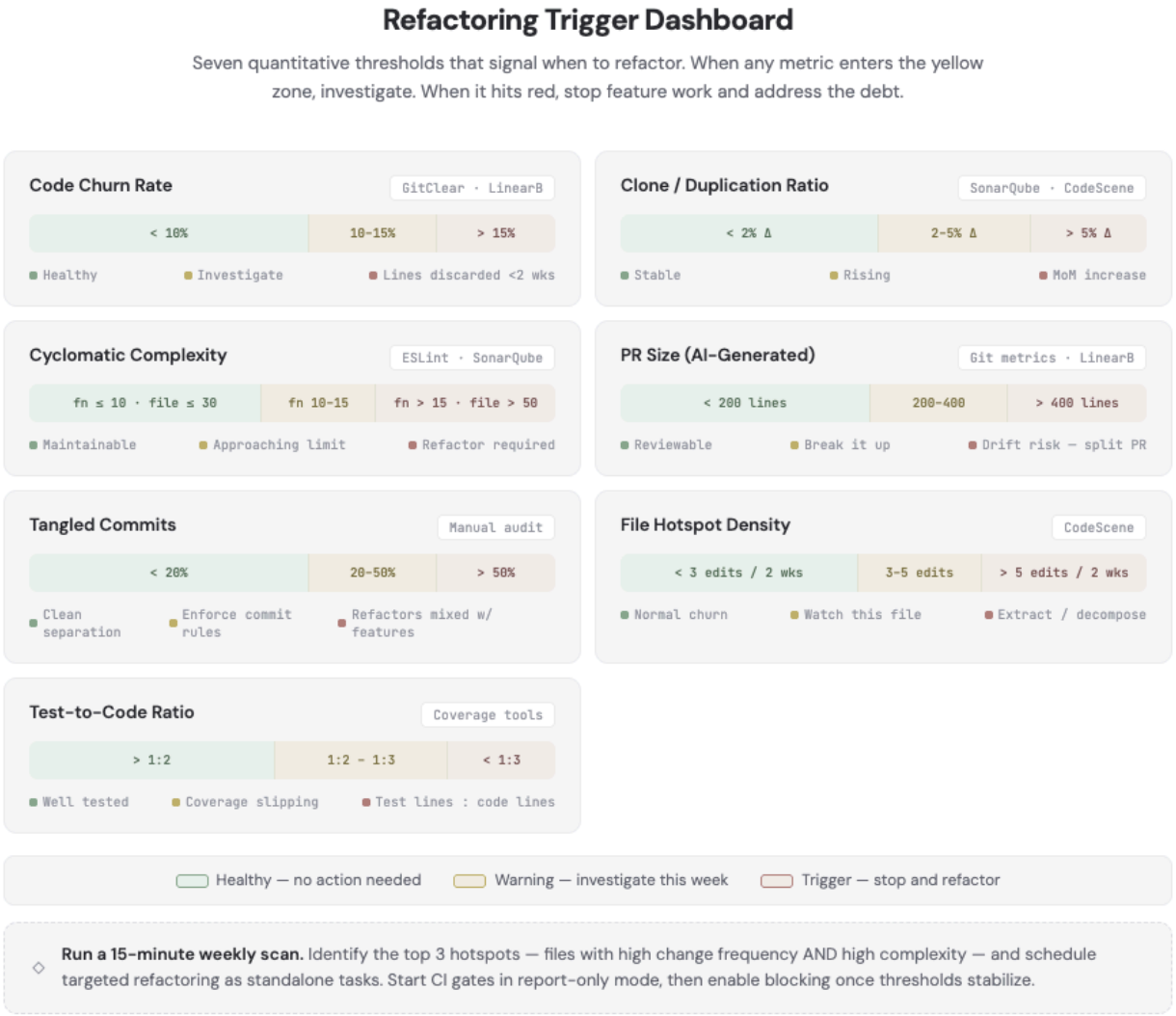
Agents make refactors cheaper to attempt but also make it dramatically easier to produce large volumes of change. Research on over 211 million changed lines of code found that code associated with refactoring dropped from 25% of changed lines in 2021 to less than 10% in 2024, while code duplication increased roughly fourfold. Agents are excellent at adding features but struggle with the complementary discipline of simplifying. Kent Beck characterized it well: "AI assistants excel at inhaling (adding features) but struggle with exhaling (refactoring for simplicity)."

A critical finding from a November 2025 study of agent refactoring behavior: despite AI's focus on maintainability improvements, agents show no significant reduction in design or implementation code smells, the median change is 0.00. Agents improve structural metrics (fewer lines, simpler methods) but consistently fail to eliminate deeper quality issues. This means human-led architectural refactoring remains essential.

Quantitative refactoring triggers

Rather than "continuous" (vague) or "scheduled" (too late), trigger refactoring when specific thresholds are crossed:

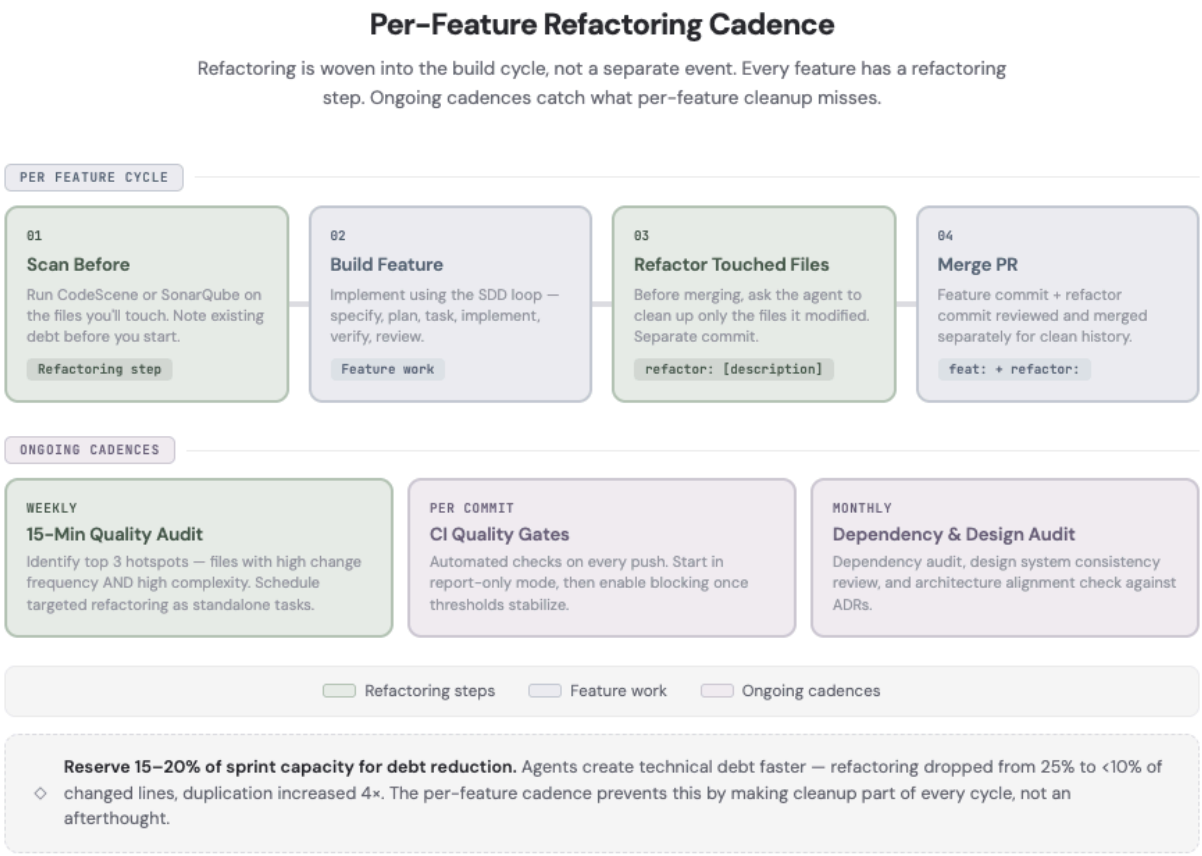
- **Code churn rate**: >15% of lines discarded within 2 weeks (measure with GitClear or LinearB)
- **Clone/duplication ratio**: >5% increase month-over-month (SonarQube or CodeScene)
- **Cyclomatic complexity**: Any function >15, any file >50 (ESLint complexity rule)
- **PR size from AI**: >400 lines changed (git metrics)
- **Tangled commits**: >50% of AI refactorings bundled with features (manual review audit)
- **File hotspot density**: Same file modified >5 times in 2 weeks (CodeScene hotspot analysis)
- **Test-to-code ratio**: Falls below 1:3 test lines to code lines (coverage tools)



The per-feature refactoring cadence

The most effective pattern is per-feature-cycle refactoring:

1. **Before implementing a feature**: Scan the files you'll touch for existing debt. Note it.
2. **After feature completion**: Before merging, ask the agent to refactor only the files it touched. Separate the refactoring into its own commit.
3. **Weekly quality audit**: Run a 15-minute scan to identify top 3 hotspots (files with high change frequency AND high complexity). Schedule targeted refactoring as standalone tasks.
4. **Per-commit CI check**: Move from nightly to per-commit quality gates. Start in report-only mode, then enable blocking once thresholds stabilize.



Reserve 15–20% of your sprint capacity for technical debt reduction. High-performing teams treat this with the same seriousness as feature development.

Separate refactoring from feature work

53.9% of agent refactorings occur in "tangled commits," bundled with feature work. This makes code review harder and increases risk. Add to your constitution:

Refactoring Rules

- NEVER mix refactoring changes with feature changes in the same commit
- After completing a feature, create a separate refactoring commit for any cleanup
- Refactoring commits must have prefix: "refactor: [description]"
- Refactoring must not change any test behavior

What agents handle well vs. what needs humans

Research puts numbers on this: roughly 36% of agent refactorings are things AI handles well: renaming variables/methods, reformatting code, removing dead code, extracting inline expressions, enforcing type consistency, consolidating duplicate code blocks within a single file.

About 55% of human refactorings are things that still require humans: extracting classes/interfaces, restructuring inheritance hierarchies, redesigning API boundaries, resolving cross-cutting dependencies, and architectural realignment. These require system-wide understanding that agents consistently fail to provide. Opus 4.6 and Codex 5.3 improved on that front, so I expect these stats to change over the next few months.

Observability from day one

Don't wait until production issues force your hand. Instrument your Second Brain with OpenTelemetry for traces, metrics, and logs from the start. Track the four golden signals (latency, traffic, errors, saturation) for every user-facing endpoint.

Track DORA delivery metrics — throughput and instability together — per service or application. DORA's latest guidance explicitly warns against using these as targets (Goodhart's Law kicks in fast), and against blending metrics from disparate systems. The goal is awareness, not scorekeeping.

Security posture

Security can't be bolted on after launch, and it's even more critical when agents are producing code and running commands in your environment.

- Align your application security work to OWASP Top 10:2025 from day one, especially broken access control (still the #1 risk) and supply chain failures (a newly expanded category).
- For agent-specific security: sandbox agent execution, drop sudo privileges, route secrets only through CI secret stores, and treat all PR text, commit messages, and web content as untrusted input.
- If agents act in CI or on pull requests, restrict who can trigger those workflows. Codex's CI documentation is unusually explicit about these risks and provides concrete mitigation patterns.

9. Establish Your Rhythms (and Protect Your Energy)

Good process isn't about rigidity — it's about making the important things automatic so you can focus your attention where it matters.

The cadence

Daily: Keep PRs small and self-contained. Every task includes acceptance criteria and a verification command. Run automated checks locally and in CI before requesting review. Update your session continuity document at the end of each working session.

Weekly: Review your DORA throughput and instability metrics and investigate any regressions. Run a 15-minute quality scan to identify the top 3 code hotspots. Audit design system drift — is new UI using your tokens and mapped components, or has the agent started improvising? Check Chromatic for unreviewed visual diffs. Review your top OWASP-relevant security categories against the week's changes.

Per milestone: Review your ADRs and check whether your architecture assumptions still hold. Are you approaching any cost or scale thresholds you identified? Run a refactor and quality sweep if churn hotspots or defect signals are rising. Update your constitution file with any new patterns, rules, or code examples that emerged during the milestone — this is the "Compound" step, and it's what makes each cycle better than the last.

The human factor: manage your intensity

One dimension that almost no technical guide addresses — but every practitioner encounters — is the cognitive cost of agent-driven development. Steve Yegge calls it the "Dracula Effect": high-intensity agent work is so demanding that trying to sustain it for a full eight-hour day leads to rapid burnout, diminishing returns, and quality collapse. Multiple startup founders confirmed the observation independently.

The practical guidance: **expect roughly three hours of high-intensity agent-driving per day.** This isn't laziness. It's about the cognitive load of context-switching between planning, reviewing, and steering agent sessions. The burnout from AI-augmented work is different from traditional burnout. It's closer to "decision fatigue." Because you can produce 10x more output in those three hours, the temptation is to push harder. Resist it.

For a Second Brain project that you're building alongside a day job or other commitments, structure your agent work in focused blocks. Plan 3–4 hours of intensive agent work plus 2–3 hours of review, planning, and thinking. Rotate between driving (active implementation) and reviewing (reading diffs, checking quality, updating specs). Measure quality debt, if defect rates or rework ratios are climbing, you're probably pushing past sustainable intensity.

Realistic timeline for an MVP

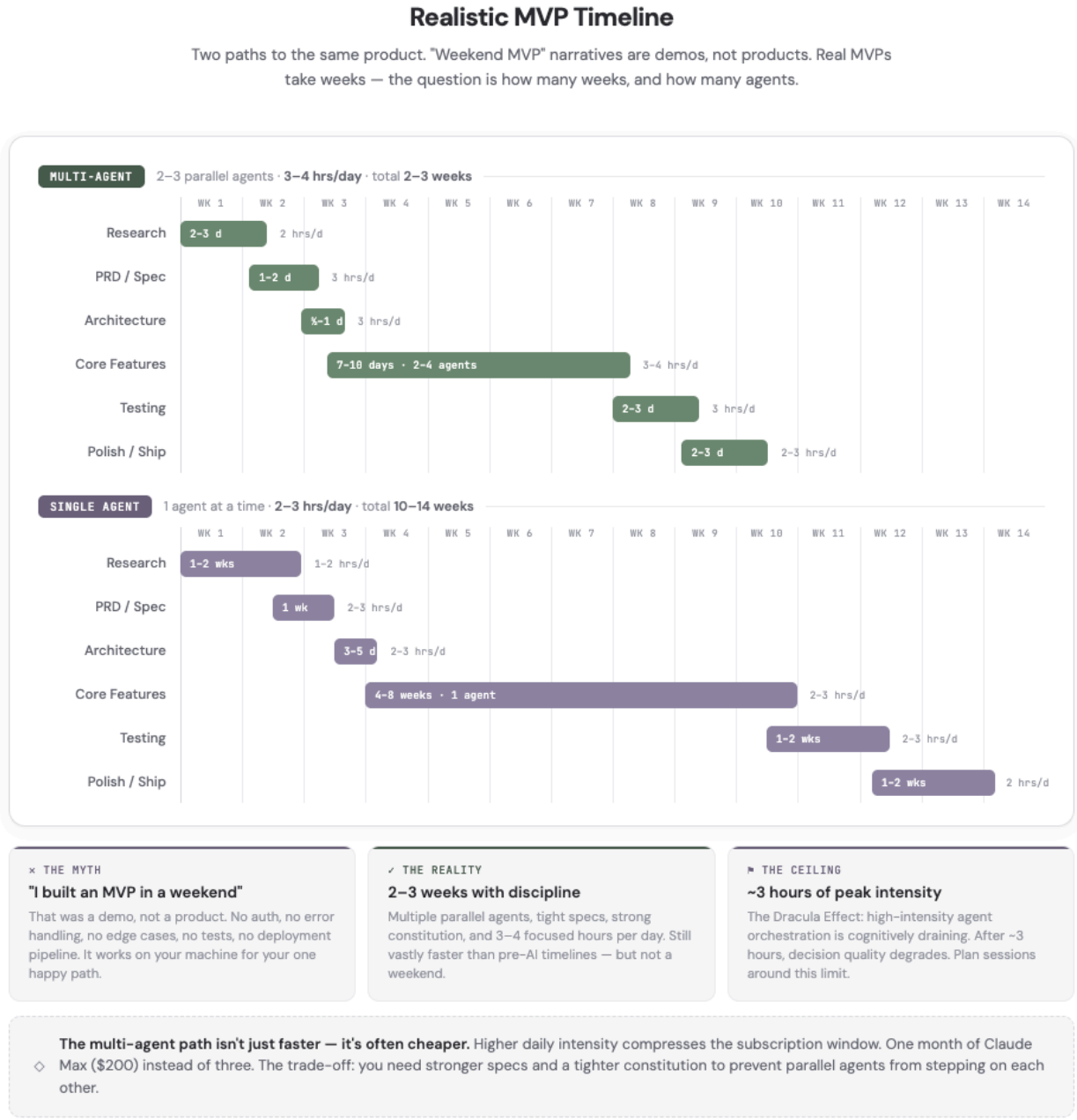
The honest reality for a Second Brain-scale product — complex, multi-feature, likely solo-developed is not the "weekend MVP" narrative that dominates social media. Those are demos, not products.

A realistic phase-by-phase estimate:

- **Research & discovery** (1–2 weeks, 1–2 hours/day of AI work): AI research, synthesis, validation
- **PRD & specification** (1 week, 2–3 hours/day): Structured specification, review and iteration

- **Architecture & design system** (3–5 days, 2–3 hours/day): Plan Mode analysis, ADR creation, shadcn setup, theme and initial components
- **Core feature development** (4–8 weeks, 3–4 hours/day): Implementation sprints following the SDD loop
- **Testing & quality** (1–2 weeks, 2–3 hours/day): Test writing, coverage, E2E with Playwright agents
- **Polish & ship** (1–2 weeks, 2–3 hours/day): Refinement, monitoring, production readiness

Realistic MVP timeline: 10–14 weeks. Even at 3 hours per day, that's dramatically faster than building the same product without agents. The speed comes from the structure, not from rushing.



The Short Version

If you take nothing else from this guide, take these five principles:

Start with intent, not code. Your PRD, ADRs, specs, and constitution file are the interface between your judgment and agent execution. Skip them, and the codebase becomes the spec by default — which means agents are making your product decisions for you.

Verification is your highest-leverage investment. If an agent can't verify its own work through tests, scripts, or screenshots, quality degrades to hope. Build the verification infrastructure before you build features — and watch for agents cheating on your tests.

Small tasks, small PRs, tight loops. Agents excel at crisp, bounded work. One task per session. Open-ended "build me a feature" requests produce drift. Break everything down, and compound the lessons from each cycle into the next.

Make constraints machine-readable. Design tokens, API schemas, constitution files, component metadata, and security baselines must be consumable by agents, not just readable by humans. If a rule isn't in a file the agent loads, the rule doesn't exist. Context engineering — the discipline of curating what agents see — is the highest-leverage skill you can develop.

Speed without quality is debt. Agents let you produce code dramatically faster — which means weak standards compound dramatically faster too. Always pair throughput metrics with stability metrics, protect your cognitive energy, and never sacrifice the gates that keep quality honest.