

2025

# Vibe Coding Playbook

**FROM CODER TO AI ARCHITECT — BUILDING PRODUCTION-GRADE SOFTWARE AT THE SPEED OF THOUGHT**

ROB VUGTS

AI-chitect | Lelystad | Netherlands

<b>THE VIBE CODING PLAYBOOK.....</b>	<b>2</b>
<i>From Coder to AI Architect — Building Production-Grade Software at the Speed of Thought .....</i>	2
<i>Introduction: The Paradigm Shift.....</i>	2
<i>Architecture-as-Code: The Missing Discipline .....</i>	3
<b>PART I: THE VIBE CODING REVOLUTION .....</b>	<b>6</b>
<i>Chapter 1: Decoding the Vibe.....</i>	6
<i>Interlude: The Three Pillars of Professional Vibe Coding.....</i>	7
<b>PART II: THE ARCHITECT'S ENVIRONMENT .....</b>	<b>10</b>
<i>Chapter 2: The Command Center (Cursor Configuration).....</i>	10
<i>Chapter 3: The Context Economy (The Silent Budget Killer).....</i>	12
<i>Chapter 4: The “Living” Context (Systematizing Knowledge).....</i>	14
<b>PART III: THE GOLDEN TRINITY (THE METHODOLOGY) .....</b>	<b>16</b>
<i>Chapter 5: Phase 1 — SDD (Spec-Driven Development).....</i>	16
<i>Chapter 6: Phase 2 — TDD (Test-Driven Development) .....</i>	18
<i>Chapter 7: Phase 3 — Vibe Coding (Execution &amp; Iteration) .....</i>	19
<b>PART IV: ARCHITECTURE AS CODE (AaC).....</b>	<b>21</b>
<i>Chapter 8: Architecture-as-Code: The Foundation for Vibe Coding at Scale .....</i>	21
<i>Chapter 9: The Document Hierarchy (The Architect's Toolkit) .....</i>	23
<i>Chapter 10: Building the Scaffolding – The Phases .....</i>	27
<i>Chapter 11: Conclusion - The Architect's Discipline .....</i>	40
<b>PART V: ADVANCED SYSTEMS (SCALING THE VIBE).....</b>	<b>41</b>
<i>Chapter 12: Custom Agents &amp; Slash Commands .....</i>	41
<i>Chapter 13: Meta Prompting — The Prompt That Writes Prompts .....</i>	43
<i>Chapter 14: Expanding Reach (MCP Servers) .....</i>	45
<b>PART VI: THE VALUE PROPOSITION (FOR BUSINESS LEADERS) .....</b>	<b>47</b>
<i>Chapter 15: The ROI of the AI Architect .....</i>	47
<i>Chapter 16: Adoption &amp; The Future.....</i>	48
<b>PART VII: AN EXAMPLE.....</b>	<b>49</b>
<i>Chapter 17: The spec .md File .....</i>	49
<i>Chapter 18: Project Initialization.....</i>	53
<i>Chapter 19: Preparing a Cursor Rule for Test-driven Development .....</i>	53
<i>Chapter 20: Preparing a Cursor Rule for Python Development .....</i>	58
<i>Chapter 21: Create Atomic Tasks .....</i>	62
<i>Chapter 22: Code Generation.....</i>	63
<b>APPENDIX:.....</b>	<b>66</b>

# THE VIBE CODING PLAYBOOK

## From Coder to AI Architect — Building Production-Grade Software at the Speed of Thought

**Author:** Rob Vugts

**Version:** 2.0

**Copyright:** © 2025 AI-chitect

---

### About the Author

**Rob Vugts** is a Technical Architect and Software Engineer with over 30 years of experience in the high-stakes banking sector, having developed and architected systems for institutions like **ING, Credit Suisse, and UBS**.

He is the founder of **AI-chitect**, a freelance consultancy dedicated to transforming software development through Artificial Intelligence. For the past 11 months, Rob has been in full-time immersion, pushing the boundaries of “Vibe Coding” to determine if AI can truly handle enterprise-grade engineering.

His mission is simple: to empower developers and organizations to harness the raw potential of AI, moving beyond “chatbots” to build robust, scalable systems with unprecedented speed and quality

---

### Introduction: The Paradigm Shift

#### The Trap of Magic

The term “Vibe Coding” was coined by Andrej Karpathy, one of the founding fathers of modern AI. He described it as a flow state where you simply “give in to the vibes,” letting the AI handle the implementation details while you direct the symphony.

And the first time you experience it, it feels like magic.

You type a natural language prompt into an IDE like Cursor. You hit enter. Suddenly, code is pouring onto the screen faster than you can read it. Functions are written, tests are generated, and the feature works. You feel like you have superpowers. You feel like a 10x developer.

#### But Vibe Coding is dangerous.

It is dangerous because that initial rush of speed often leads to a hangover. About 20 minutes into a session, reality hits like a brick. The AI starts hallucinating libraries that don’t exist. It forgets the code it wrote three prompts ago. It introduces subtle bugs that break your production build. You spend the next two hours debugging the “magic” code you generated in two minutes.

This is where most developers quit. They say, “AI is a toy, it’s good for scripts, but not for real engineering.”

#### The Pivot: From Bricklayer to Architect

They are wrong. The problem isn’t the AI; the problem is the workflow.

We have spent the last 30 years teaching developers to be **Bricklayers**. We pride ourselves on knowing the syntax of a for loop in Java versus Python. We memorize libraries. We type characters into a text file.

Vibe Coding demands that you stop being a Bricklayer and start being an **Architect**. In this new era, your value is no longer defined by how fast you can type syntax. Your value is defined by:

1. **Design:** Can you visualize the system structure?
2. **Constraints:** Can you define the Non-Functional Requirements (Performance, Security, Scalability)?
3. **Orchestration:** Can you guide an intelligent agent to build exactly what you envisioned, without micromanaging every keystroke?

## Architecture-as-Code: The Missing Discipline

Here's what most vibe coding tutorials won't tell you: the magic only works when it's grounded in solid architecture—and that architecture must be automatically enforced.

That 20-minute hangover? It happens because the AI has no persistent memory of your design decisions. Every new prompt is a blank slate. Without architectural guardrails, the AI reinvents your system structure with each response—and each reinvention drifts further from your original intent.

The solution is not to abandon AI assistance. The solution is to adopt a practice called Architecture-as-Code (AaC).

**Architecture-as-Code** treats your architectural decisions as executable, version-controlled, automatically validated code—not static diagrams that drift from reality. Just as Infrastructure-as-Code transformed how we provision servers, Architecture-as-Code transforms how we define, enforce, and evolve software architecture.

This means:

- **Create a Blueprint** that captures your vision, philosophy, and technical constraints before any code exists
- **Define explicit boundaries** that tell the AI what it cannot do (Non-Goals, Invariants)
- **Freeze your contracts** (database schemas, API specs) so the AI implements against fixed targets
- **Encode your constraints as fitness functions** —automated tests that enforce architectural rules on every commit
- **Integrate validation into CI/CD** so violations are caught automatically, whether the author is human or AI

The magic still happens—but now it happens within guardrails that are enforced automatically, on every commit, by every developer, in every AI session. Violations don't slip through code review and accumulate as technical debt. They're caught immediately by fitness functions that never sleep.

*Think of it this way:* a Bricklayer asks the AI to "build a wall." An Architect gives the AI blueprints, material specifications, structural constraints, and quality criteria—and then has automated tests verify that every brick laid conforms to those specifications. The AI lays bricks at superhuman speed; the fitness functions ensure it never lays them wrong.

## The Promise of This Playbook

This playbook is not a collection of "Cool ChatGPT Prompts." It is a rigorous engineering discipline.

I have spent nearly a year experimenting with vibe coding trying to build full-stack, production-grade applications with mixed success and realized that I needed more than best practices. I also needed a better workflow, one based around a clear specification (or contract) and test-driven development. This is what I now call the **Golden Trinity** workflow.

But the Golden Trinity alone wasn't enough. I kept hitting the same wall: the workflow was sound, but without architectural foundations, each AI session created slightly different interpretations of my system. I needed a way to freeze my design decisions and make them visible to every AI session.

And even frozen specifications weren't enough. Human developers (and AI agents) would still drift from the specifications over time. I needed automated enforcement—fitness functions that would catch violations before they merged, not weeks later in an architecture review.

This playbook gives you the complete system:

- **The Golden Trinity (SDD → TDD → Vibe Coding)**—the workflow for building features correctly.
- **Architecture-as-Code**—the scaffolding that keeps every feature aligned with your vision, enforced automatically through fitness functions and CI/CD integration.

If you follow the systems in this book, you will not just code faster. You will build better software. You will eliminate the drudgery of boilerplate, the pain of writing unit tests, and the fatigue of debugging. You will reclaim the joy of problem-solving.

And crucially, your architectural decisions will be enforced automatically—not by heroic code reviewers catching violations, but by fitness functions that run on every commit and block violations before they merge.

## About This Playbook

This playbook stands on the shoulders of a growing community of engineers exploring Vibe Coding in practice. One excellent actionable resource in that space is Sean Kochel's YouTube video [“Supercharge Your Vibe Coding in 21 Easy-to-Apply Tips”](#), which distils a concrete set of habits for working productively with AI-native IDEs like Cursor.

The systems in this book are firmly rooted in sound engineering practices—a foundation that is particularly essential in the fast-paced and experimental realm of vibe coding, but many of the day-to-day practices are **inspired by and adapted from Sean’s 21 tips**. To make those ideas usable in a professional setting, this playbook:

- Organizes them into **three pillars** of professional Vibe Coding.
- Integrates them into the **Golden Trinity** (SDD → TDD → Vibe Coding).
- Grounds them in Architecture-as-Code for enterprise-scale reliability and automated enforcement.
- Provides a **Tip Index in the Appendix** so every “Tip #X” and “Pillar Y” reference in the text resolves to a concrete description.

If you want the full original context, examples, and Sean’s own explanations, watch his video alongside this playbook. This book focuses on turning those ideas (and my own findings) into a rigorous, production-ready workflow that you can bring into real organizations.

In addition to this playbook, there is a GitHub repository that contains all supporting Cursor resources, including comprehensive Cursor rules, slash commands, reusable prompts, MCP server

configurations, and a well-curated `.cursorignore` file. You can find it at:  
<https://github.com/rvugts/cursor-resources>

**Welcome to the revolution. Let's get to work.**

---

# PART I: THE VIBE CODING REVOLUTION

## Chapter 1: Decoding the Vibe

To master Vibe Coding, we must first strip away the hype and define exactly what we are doing.

### What is Vibe Coding?

Vibe Coding is **not** autocompletion. Tools like GitHub Copilot (in its original form) were autocompleters—they guessed the next word in your sentence. Useful, but passive.

Vibe Coding is **Autonomous Reasoning**.

When you use an AI-native IDE like Cursor (or competitors like Kilo Code and Windsurf), you are not working with a text predictor. You are working with an **Agent**. This Agent has:

1. **Vision:** It can “see” your file structure, read your terminal errors, and diff your git history.
2. **Action:** It can create files, delete code, run terminal commands, and browse the documentation.
3. **Reasoning:** It can plan a multi-step refactor before writing a single line of code.

The “Vibe” refers to the **telepathic connection** that forms when you trust the Agent. Instead of writing:

```
> public static void main(String[] args) { ... }
```

You write the **Intent**:

> *Create a CLI entry point that parses arguments for a file path and a retry count, creating a logger instance immediately.*

The AI handles the syntax (the “How”). You handle the intent (the “What” and “Why”).

### The “Universal” Language

A common myth is that Vibe Coding is only for frontend developers using JavaScript or React. This is false. The “Vibe” is platform-agnostic. Because LLMs are trained on the collective knowledge of the entire open-source internet, they are fluent in every major stack.

In my work as an **AI-chitect**, I use Vibe Coding across the entire spectrum:

- **The Backend Core:** Whether it’s Java Spring Boot for enterprise services, high-performance C++ for system tools, or Go for microservices, the AI understands the idioms and patterns of strongly typed languages perfectly.
- **Mobile Development:** Building screens in Swift (iOS), Kotlin (Android), or React Native becomes fluid. You describe the UI state (“Create a loading skeleton that transitions to a list view”), and the AI handles the boilerplate layout code.
- **Infrastructure as Code (IaC):** This is a massive time-saver. Writing Terraform, Ansible, or Kubernetes YAML files is often tedious and error-prone. Vibe Coding turns it into natural language: *“Provision an S3 bucket with public access blocked and versioning enabled.”*
- **Data & Database:** Writing complex SQL queries or defining ORM schemas (Prisma, Hibernate) is trivialized.

## The Exponential Gains: Why Bother?

Why should you or your company undergo the painful process of changing how you work? Because the ROI (Return on Investment) is undeniable.

### 1. Productivity (The 25%+ Jump)

Research and internal metrics show that teams adopting AI-native workflows see an immediate productivity jump of over **25%**. But that number is conservative. For “Greenfield” projects (new code), I frequently see velocity increases of **200-300%**. You simply skip the “Zero-to-One” phase of setting up folders, configs, and boilerplate.

### 2. Quality & Security

This is counter-intuitive. People assume AI writes buggy code. But an AI Architect uses the AI to prevent bugs. You can instruct the AI to: “*Review this file against OWASP Top 10 vulnerabilities*” or “*Refactor this function to reduce cyclomatic complexity*.” The AI is a tireless reviewer that never gets bored or tired.

### 3. Developer Happiness (The Flow State)

This is the metric that keeps teams together. Vibe Coding eliminates the “grunt work”—writing getters/setters, configuring Webpack, writing docstrings. When you remove the boring work, developers spend more time in Flow State, solving the complex architectural puzzles they actually enjoy.

### 4. Operational Velocity

It’s not just code. It’s the ecosystem. AI Agents can write your Commit Messages, generate your Pull Request descriptions, and update your README files automatically. The “admin” side of software engineering vanishes.

---

## Interlude: The Three Pillars of Professional Vibe Coding

Throughout this playbook you will see references to **Pillars** and **Tips**. Rather than leaving those as “floating” labels, this section defines the three pillars explicitly and shows how the core tips map into them in a professional context.

The goal of these pillars is simple: turn Vibe Coding from a risky, ad-hoc trick into a **disciplined engineering practice** you can safely use in production systems.

---

### Pillar 1: Architect’s Mindset — Plan Before You Prompt

Pillar 1 is about **thinking like an architect, not a typist**. You use the AI’s speed to explore designs and requirements, not to spray code into the repo.

At its core, this pillar is built on habits such as:

- **Choose well-documented, mainstream stacks.** Favor technologies with rich, up-to-date documentation, examples, and community support. This gives the AI something reliable to lean on and drastically reduces hallucinations around obscure APIs.
- **Front-load design and constraints.** Before any implementation, work with a reasoning-oriented model to define the architecture, data model, interfaces, edge cases, and non-functional requirements (performance, security, scalability, observability).

- **Adopt a “Spec First” mentality.** Capture the outcome of that reasoning as a written spec (spec.md) that becomes the single source of truth for the feature.
- **Work in small, explicit tasks.** Break the spec into atomic, named implementation tasks that can be completed in one to three prompts. This is how you stop the AI from going “off script”.
- **Iterate on “paper” before code.** Explore trade-offs, alternatives, and failure modes in text with the AI long before touching production files.

When you see later chapters referencing “**Pillar 1**” or tips such as *documented stack*, *design-first*, or *task granularity*, they belong to this pillar.

---

## Pillar 2: The Conversation — Guide, Don’t Just Ask

Pillar 2 is about treating the AI as a **junior engineer you actively direct**, not an oracle you beg for miracles. The quality of the conversation determines the quality of the code.

Typical practices in this pillar include:

- **Be surgical with context.** Instead of throwing the full code base at every problem, deliberately select the minimal set of files and artifacts the AI needs for the next step. Think “need-to-know basis”.
- **Ask for options, not single answers.** Frequently request two or three alternative designs or implementations, with pros and cons, before committing to one. This side-steps single-shot hallucinations and surfaces trade-offs.
- **Interrogate generated code and tests.** Never accept code you don’t understand. Ask the AI to explain unfamiliar mocks, patterns, or APIs until you can confidently own the result.
- **Choose the right brain for the job.** Use fast, cheap models for small, mechanical changes; use large, reasoning-heavy models for design, refactors, and complex debugging. Don’t waste a “big brain” on adding imports.
- **Switch perspectives when stuck.** When a conversation degrades (the AI loops or digs deeper into a bad approach), start a new chat, change the prompt angle, or even switch models to get a fresh viewpoint.

Whenever the text talks about “**conversation discipline**,” “**Sequential Thinking**,” or **choosing models and personas**, it is anchored in Pillar 2.

---

## Pillar 3: Disciplined Workflow — Guardrails and Recovery

Pillar 3 turns your workflow into something **repeatable, observable, and reversible**. This is where speed meets safety.

Key behaviors in this pillar include:

- **Commit and checkpoint aggressively.** Commit on green tests, not on “finished features,” and use editor checkpoints where available. This gives you safe restore points during fast AI sessions.
- **Use explicit rules and documentation.** Capture project rules, style guides, and architecture in .cursorrules and a DOCUMENTATION/ folder (API.md, ARCHITECTURE.md, IMPLEMENTATION\_STATUS.md). The AI should always have a map.

- **Index the right sources of truth.** Connect your AI to the actual documentation, schemas, and services it must respect (via MCP or other integration), instead of letting it guess from training data.
- **Bake in security and quality checks.** Use specialized “Security Auditor” and “Refactor” commands to review features before merge, and explicitly ask for checks against common vulnerability classes.
- **Reset when the vibe goes bad.** Use the “Nuclear Option”: stop generation, revert to the last green commit, clear the chat, and restart with a new angle instead of debugging a spiraling agent.
- **Practice “Fresh Start” discipline.** Start new conversations at natural breakpoints (feature done, bug fixed, major refactor complete) to keep context windows small and focused.

The later references to **checkpoints, rules, memory, security audits, the nuclear option, and “fresh start” chats** all live under Pillar 3.

---

## How to Use the Pillars in Practice

The **Golden Trinity** workflow in Part III (SDD → TDD → Vibe Coding) sits on top of these pillars:

- **SDD** is mostly Pillar 1 (architecting and spec-writing) with help from Pillar 2 (good conversations).
- **TDD** is Pillar 1 (requirements) plus Pillar 3 (guardrails, repeatability).
- **Vibe Coding** as described in this book only happens **inside** the constraints created by Pillars 1–3.

Whenever you see mentions of *Pillar 1/2/3* or *Tip #X* in the chapters that follow, you can always return to this interlude and to the **Tip Index in the Appendix** for a quick refresher.

---

# PART II: THE ARCHITECT'S ENVIRONMENT

## Chapter 2: The Command Center (Cursor Configuration)

You cannot be a high-performance race car driver in a family sedan. Similarly, you cannot practice Vibe Coding effectively with a standard text editor. You need a Command Center.

For the purposes of this book, our weapon of choice is **Cursor IDE**.

While other tools like Kilo Code, Windsurf, and Zed IDE have entered the arena with AI capabilities, Cursor is still the market leader due to maturity, speed, continuous enhancement, and its ability to quickly incorporate features popularized by emerging competitors.

### The Subscription Matrix: Investing in Power

Before we touch a line of configuration, we must address the economics of the tool. Many developers try to scrape by on free tiers. If you are serious about this career shift, that is a mistake.

- **Hobby (Free):** Fine for dipping your toe in the water. You get limited “agent” requests. But for Vibe Coding, speed is everything. The moment you get throttled to “slow” requests, your flow state is broken.
- **Pro (\$20/user/month): This is the minimum requirement for the professional AI Architect.** It provides unlimited tab completions, access to background agents, and standard usage limits for premium models. \$20 is a trivial investment for a tool that can double your output.
- **Pro+ (\$60/user/month):** The recommended tier for heavy users. If you find yourself burning through the standard Pro limits, this tier offers **3x usage** on all premium models (OpenAI, Claude, Gemini). It ensures you never drop out of the fast lane during intense architectural sprints.
- **Ultra (\$200/user/month):** For the extreme power user. This provides **20x usage** on all models and priority access to new features. It is likely overkill for most, but if you are running continuous heavy inference, this guarantees zero friction
- **Teams (\$40/user/month):** This is critical for anyone working in enterprise or banking (formerly the “Business” tier). The key feature here is **Org-wide Privacy Mode** and centralized billing. By default, many AI tools may train on user data. The Teams tier creates the necessary governance structure to ensure proprietary code is not being used to train models. If you are pitching Vibe Coding to your CTO, this is the compliance key that unlocks the door.

### The “Fire-and-Forget” Setup (Agent Mode)

To truly “Vibe,” we need to reduce friction. The standard loop of *AI suggests code -> Human copies code -> Human pastes code* is dead. We want the AI to take action.

#### Enable “YOLO Mode” (Auto-Run in Terminal)

This is where the magic happens, and where it gets scary for beginners.

In the Cursor settings (under “Agents”) there is a setting to allow the Agent to run terminal commands, call MCP servers and perform file writes.

\* **The Conservative Approach:** The AI proposes a command (e.g., `npm install lodash`), and you must click a “Run” button to approve it.

\* **The Vibe Approach:** You configure the agent to **Auto-Run** commands.

## Why do this? Because of the **Agentic Loop**.

When the AI writes code, it will often make a syntax error or forget a dependency. If Auto-Run is on, the Agent acts like this:

Writes Code ->

1. Runs Tests (automatically).
2. Reads the Error Message in the terminal.
3. Realizes it messed up.
4. Rewrites the code to fix the error.
5. Runs Tests again.
6. **Success.**

All of this happens while you are taking a sip of coffee. If you have to manually approve every step, you break the autonomous loop.

**Safety Tip:** Start by giving the AI read permissions. Once you trust it, give it write permissions. Only enable Auto-Run when you have a solid Git history (so you can `git reset --hard` if the AI does something catastrophic).

The screenshot shows the AI-chitect interface with the following sections:

- Auto-Run Mode:** A dropdown menu currently set to "Auto-Run in Sandbox".
- Auto-Run Network Access:** A dropdown menu currently set to "Ask Every Time".
- Allow Git Writes Without Approval:** A toggle switch is turned on (green).
- Command Allowlist:** A list of commands that can run automatically, including: chmod, cd, terraform fmt, find, sort, az automation, az policy, git commit, git push, grep, tail, git add, psql, az monitor, python3, head, sed, jq, echo, az account, az role, az vmss, az vm, az postgres, git status, wc, git diff, curl, and markers) else: print(f'OK: len, code\_blocks.
- MCP Allowlist:** A list of MCP tools that can run automatically, including: sequential-thinking:sequent... .

## The .cursorignore File

Just as you have a `.gitignore` to keep your repo clean, you need a `.cursorignore` to keep your AI's brain clean (and your wallet full).

By default, the AI might try to index everything to understand your codebase. You do not want it wasting tokens reading your `package-lock.json` (which is thousands of lines of junk) or your compiled binary assets.

**Action Item:** Create a `.cursorignore` file in your root:

```
# Ignore lock files to save context
package-lock.json
yarn.lock
pnpm-lock.yaml

# Ignore build artifacts
dist/
build/
.next/

# Ignore large assets
*.svg
*.png
public/images/
```

This simple step saves you money and prevents the AI from getting confused by auto-generated code.

**Note:** You will find a much more comprehensive file in the Cursor Resources GitHub repository: [.cursorignore](#)

---

## Chapter 3: The Context Economy (The Silent Budget Killer)

If you take only one thing from this book, let it be this: **Input costs are the silent killer of your budget.**

We all fixate on the “Output Cost”—the \$1.00 price tag for generating a million tokens of code. But the real danger is the “Input Cost” (the Context). Every time you send a prompt to the AI, you are resending the entire history of the conversation and every file you have attached.

If you have a 10,000-line file attached to your chat, and you ask “Fix this typo,” you are paying to process those 10,000 lines again. And again. And again.

### The “Context Engineer” Mindset

You must stop being a passive user and become a **Context Engineer**.

#### 1. The “Surgical Context” Rule (Tip #3)

Novices don’t specify context in their prompts. This forces the AI to scan your entire repository using a RAG (Retrieval-Augmented Generation) search. It is expensive and often inaccurate (it retrieves irrelevant files).

**The Pro Move:** Be surgical. Manually select the files the AI needs using the `@File` command or the UI picker.

\* *Bad Prompt: Why isn't the login working?"*

\* *Good Prompt: "@login-service.ts @user-auth.tsx Why is the session token returning null?"*

By feeding the AI only the relevant files, you reduce the noise, making the AI smarter, faster, and cheaper.

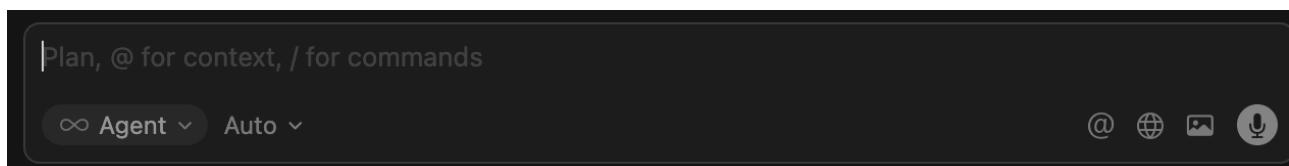
## 2. The “Need to Know” Basis

Does the AI need to see the implementation details of your database connection just to change a CSS button color? No. Don’t feed it the whole backend when working on the frontend. Treat the AI like a junior developer: give them exactly what they need to do the job, and nothing else.

## The Token Strategy: Choosing Your Brain

Not all AI models are created equal. Cursor allows you to switch models on the fly via a dropdown in the chat.

- **Small Models (Flash / Haiku / Mini):** These are incredibly fast and cheap. Use these for:
  - Fixing syntax errors.
  - Writing simple boilerplate.
  - Explaining a specific line of code.
- **Large Models (Claude 4.5 Sonnet / GPT-5.1 / Gemini 3 Pro):** These are the “Big Brains.” They are expensive and slower. Use these for:
  - Architectural planning.
  - Complex refactoring.
  - Debugging race conditions.
- **“Auto” Mode:** Cursor has a proprietary “router” that analyzes your prompt complexity and automatically routes it to the appropriate model. For 90% of users, leaving this on **Auto** is the most financially prudent strategy.



## The “Fresh Start” Discipline (Tip #21)

This is the single most effective habit for token efficiency.

As you chat with the AI, the “Context Window” (the short-term memory) fills up. By the time you are 30 messages deep into a debugging session, you might be sending 50,000 tokens of “history” with every single new message.

### The Rule: Create a new conversation at every natural break point.

- Finished the “Login” feature? **Close the chat.** Start a new one for “Dashboard.”
- Solved the bug? **Close the chat.** Start fresh.

Think of it like clearing a whiteboard. If you keep writing over the old drawings, eventually it becomes a mess of illegible scribbles. A fresh chat ensures the AI is focused only on the current problem, not the baggage of the last hour.

---

## Chapter 4: The “Living” Context (Systematizing Knowledge)

We have set up the tool, and we have managed the cost. Now we must teach the AI *who we are*.

The biggest frustration with AI is repetition. You find yourself typing, “Remember, we use Tailwind, not plain CSS,” or “Don’t use any in TypeScript,” fifty times a day.

We solve this by creating a “**Living**” Context using a layered rules system.

### 1. The Global Constitution: `.cursorrules`

Cursor looks for a file in the root of your project called `.cursorrules`. This remains your Global Constitution. It is the system prompt appended to every single request. Use this for high-level, non-negotiable principles that apply to the entire project.

### 2. The Specialist Rules: `.cursor/rules/`

For specific tasks, languages, or workflows, Cursor now uses a dedicated directory in the root of your project. You can create multiple rule files (using the `.mdc` extension) inside `.cursor/rules/`.

These rules are intelligent. You can configure them to trigger **automatically** based on:

- **File Type (Globs):** e.g., apply React rules only when a `*.tsx` file is open.
- **Context (Agentic):** The AI analyzes your request and pulls in the relevant rule (e.g., if you ask to “write a test,” it pulls in `testing.mdc`).

### Action Item: Set up your Rules System

#### A. Create the Global File (`.cursorrules`)

Create this in the project root. It should contain your **Tech Stack Definition** and core **Behavioral Guidelines** (e.g., “Be concise,” “No trailing whitespace”).

#### B. Create the Rules Directory (`.cursor/rules/`)

Create specific `.mdc` files for distinct parts of your workflow. For example:

- `.cursor/rules/react.mdc`
  - *Glob:* `**/*.tsx`
  - *Content:* “Use Functional Components, Shadcn UI, and Lucide React icons.”
- `.cursor/rules/python.mdc`
  - *Glob:* `**/*.py`
  - *Content:* “Follow PEP 8, use Pydantic v2 for validation.”
- `.cursor/rules/testing.mdc`
  - *Description:* “Always use this rule when writing or fixing tests.”
  - *Content:* “Follow the Red-Green-Refactor cycle. Use Pytest fixtures.”

By front-loading this context (Tip #9 & #19), you eliminate the need to correct the AI’s style. It “Vibes” with your team’s style automatically.

**Note:** You can find comprehensive cursor rules in the Cursor Resources GitHub repository [5].

## The Documentation Strategy (My Personal Best Practice)

In my 11 months of experimentation, this has been a valuable discovery. **The AI works best when it has a map.**

I create a dedicated docs/ folder in every project containing three specific files:

1. **API.md**: A rough sketch of the API endpoints and data shapes.
2. **ARCHITECTURE.md**: A high-level explanation of how the system fits together (e.g., “Frontend talks to Backend, Backend talks to Core Services”).
3. **IMPLEMENTATION\_STATUS.md**: This is the killer. It is a running log of what we have built, what is currently broken, and what is next.

### The Workflow:

At the end of every significant coding session, I give the AI this command:

> “Review the work we just did. Update *IMPLEMENTATION\_STATUS.md* to check off completed tasks, add any new technical debt we discovered to the ‘Known Issues’ section, and list the next logical steps.”

This creates a **Self-Healing Context**. When I come back to the project tomorrow (or when I hand it off to another developer), I don’t have to remember where I left off. I just tell the AI:

> “Read *IMPLEMENTATION\_STATUS.md* and tell me what we need to do next.”

## Cursor Memory

Recently, Cursor introduced a “Memory” feature (in the Settings -> Rules, Memories, Commands pane). This allows you to explicitly tell the agent to “Remember” facts that persist across all chat sessions.

Use this for **Immutable Decisions**.

- \* “Remember: We are strictly forbidden from using the eval() function for security reasons.”
- \* “Remember: All dates must be stored in UTC.”

This differs from .cursorrules (which is file-based and project-specific) by being attached to your user profile for that project, acting as a long-term memory bank for architectural constraints.

---

# PART III: THE GOLDEN TRINITY (THE METHODOLOGY)

We have built our Command Center. We have established our Constitution (.cursorrules). Now, we face the blank page.

Most developers using AI make a fatal mistake right here: **They start coding.**

They open a file and ask the AI, “*Write a function to handle user login.*” And the AI complies. It writes code. It looks good. But 20 minutes later, that code doesn’t fit with the database schema. It lacks error handling. It conflicts with the frontend state management. The developer spends the next four hours debugging a feature that took two seconds to generate.

We will not do that. We will follow the **Golden Trinity**.

The Golden Trinity is a workflow designed to create a “Sandboxed Playground” where the AI can run at full speed because we have removed the risk of structural failure.

**The Trinity consists of:**

1. **SDD (Spec-Driven Development):** The Contract.
  2. **TDD (Test-Driven Development):** The Guardrails.
  3. **Vibe Coding:** The Execution.
- 

## Chapter 5: Phase 1 — SDD (Spec-Driven Development)

Pure Vibe Coding feels like flying — right up to the moment you realize you have no instruments. The model writes impressive code at high speed, but without a contract it will happily hallucinate APIs, forget earlier decisions, and drift away from the system you thought you were building.

**Spec-Driven Development (SDD)** is how we install those instruments. Instead of asking the AI to “plan and code” in one breath, we explicitly separate **reasoning** from **implementation** and capture the outcome as a written, versioned **contract** (spec.md). In the next chapters, you will see how this contract combines with **Test-Driven Development (TDD)** to turn raw Vibe Coding into a workflow that is fast and safe enough for production work.

### The Architect’s Mindset: Plan Before You Prompt

Reference: *Pillar 1, Tips #1, #2, #11*

The AI is a Ferrari. If you drive a Ferrari without a map, you don’t get to your destination faster; you just get lost at 200 mph.

**Spec-Driven Development (SDD)** is the act of creating that map. It separates the *reasoning* phase from the *implementation* phase. Current LLMs struggle when asked to do both simultaneously. If you ask an AI to “Plan and Code” in one breath, it will take shortcuts.

### Step 1: The “External Brain” Strategy

Reference: *My Personal Best Practices, Step 1*

Here is a tip: **Don’t start your planning inside Cursor.**

While Cursor now offers a dedicated planning mode, it comes with an associated token cost each time you use it for high-level architectural reasoning. By contrast, when you use free web-based options like Gemini or ChatGPT for brainstorming and planning, you can explore and iterate on ideas

without incurring any direct token expenses. This makes external tools especially convenient for cost-effective and unconstrained early-stage planning before moving to implementation inside Cursor.

I recommend using **Google AI Studio (Gemini 3.0 Pro)** or **ChatGPT 5** in a web browser for this specific step.

### The Workflow:

1. Open your “External Brain” (Gemini/ChatGPT).
2. Prompt it with your raw, messy idea: “*Following a spec-driven development approach, I want to build a rate-limiting system for my API using Redis.*”
3. Ask for the **Design-First Approach (Tip #11)**: “*Define the Data Model, the API Interface, and the edge cases. Don’t write code yet. Just architect it.*”
4. Also ask it to consider other relevant aspects like:
  - Non-functional requirements
  - Phased Rollout Plan: Starting with a core MVP and planning future feature releases.
  - Technology Stack & Tools: Recommended languages, frameworks, and services.
  - Deployment strategy and estimated Cloud Costs.
  - Risks & Mitigations: Identifying potential problems and how to handle them.
5. Iterate on this plan until you have a solid blueprint. Discuss trade-offs (e.g., “Should we use a Sliding Window or Token Bucket algorithm?”).
6. Finally ask it to generate a spec.md file in Markdown format that will serve as the contract for the AI agent implementing this idea.

This phase is **Token Agnostic**. You can burn through ideas cheaply without polluting your IDE’s context.

## Step 2: The spec .md Artifact

Once the plan is solid in your “External Brain,” you bring it into Cursor. You create a file named spec.md (or feature-name.spec.md) in your project.

**This file is the Contract.** It serves as the single source of truth.

### A Spec Example:

```
# Feature: Redis Rate Limiter

## 1. Goal
Implement a sliding window rate limiter middleware for the Express server.

## 2. Constraints
- Language: TypeScript (Strict Mode)
- Store: Redis (via `ioredis` library)
- Max Latency: < 10ms overhead

## 3. Data Structure
- Key Format: `rate_limit:{user_id}:{window_timestamp}`
- TTL: 60 seconds

## 4. Interface Definition
interface RateLimiter {
  check(userId: string): Promise<Result>;
}
```

```
## 5. Edge Cases
- Redis is offline (Fail Open - allow traffic).
- User ID is missing.
- Concurrent requests (Race conditions).
```

## Step 3: Task Granularity (Tip #10)

Before moving to code, we must break this spec down. We ask the AI in Cursor:

> *“Read spec.md. Using **Sequential Thinking**, break this feature down into a list of atomic implementation tasks. Each task must be small enough to be completed in 1-3 prompts.”*

This effectively “chunks” the complexity. The AI isn’t trying to build the whole bridge; it’s just laying one plank at a time.

---

## Chapter 6: Phase 2 — TDD (Test-Driven Development)

### The Safety Net

Reference: Pillar 1, Tip #18 (MVP approach)

In traditional coding, TDD (Test-Driven Development) is often skipped because writing tests is boring. In Vibe Coding, **TDD is mandatory**, but it is also effortless because the AI does the heavy lifting.

We write tests *before* implementation for one reason: **Hallucination Control**.

If the AI writes the code and then writes the tests, it will write tests that pass its own buggy code. It’s like marking your own homework. By writing tests first based only on the Spec, we create an independent verification layer.

### Generating the Suite

#### The Prompt:

> *“@spec.md Create a comprehensive test suite using Jest (or pytest). Cover the happy path, the edge cases listed in the spec, and the error states. **Do not write the implementation logic yet.** Only write the tests. Mock the Redis connection.”*

### The “Red” State

You must run these tests. They **must fail**.

If they pass, something is wrong (perhaps you have a mock that is returning true by default). Seeing the wall of red text in your terminal is the “Green Light” to proceed. It confirms that your Guardrails are installed.

### Understanding the Code (Tip #17)

**Crucial Warning:** Do not commit tests you do not understand.

If the AI generates a complex test using a mocking technique you’ve never seen, stop. Highlight the code and ask: *“Explain how this mock works line-by-line. Why did you choose this approach?”*

You are the Architect. You cannot sign off on a blueprint you can’t read.

## Chapter 7: Phase 3 — Vibe Coding (Execution & Iteration)

### Unleashing the Agent

Reference: *Pillar 2, Tip #3, Tip #6*

Now, we have a Spec (The Map) and Tests (The Guardrails). It is time to Vibe.

We switch Cursor into **Agent Mode** (Composer). We give it the files it needs—and *only* the files it needs.

#### The Implementation Prompt:

- > “@spec.md @rate-limiter.test.ts @rate-limiter.ts (empty file)\*
- > Implement the RateLimiter class to make the tests pass.
- > Use **Auto-Run** to execute the tests after every change. Fix any errors you encounter automatically.”

This is the “Vibe.” You sit back. You watch the AI write code, run the test command, see a failure, rewrite the code, run the test again, and finally hit **Green**.

### The “Nuclear Option” (Step 6)

Sometimes, the “Vibe” goes bad. The AI gets stuck in a loop. It tries to fix a bug, breaks something else, tries to fix that, and spirals into chaos.

**Do not debug a spiraling AI.** You will waste tokens and sanity.

#### Engage the Nuclear Option:

1. **Stop** the generation.
2. **Revert** the file to its state before the session (using Git).
3. **Clear** the chat history (Tip #21 - Fresh Start).
4. **Prompt with a New Angle (Tip #6):**

> “*The previous approach failed due to a race condition. Discard that logic. Propose two alternative architectural approaches to solve this. Rank them by simplicity. Then implement the best one.*”

This “hard reset” saves hours of frustration. It forces the AI to abandon a “poisoned” context path and re-evaluate from first principles.

### The “Commit & Checkpoint” Discipline (Tip #4 & #16)

Vibe Coding moves fast. You will generate a week’s worth of code in an hour. This makes Version Control (Git) more critical than ever.

#### The Rule: Commit on Green.

The second the tests pass, commit the code. Do not wait to finish the “whole feature.”

\* *Tests pass? Commit.*

\* *Refactored a function? Commit.*

\* *Added comments? Commit.*

If the AI hallucinates and destroys your file five minutes later, you need to be able to revert to the “Green State” instantly. Cursor also has a local “Checkpoint” feature (Tip #16) in the chat history that allows you to roll back the *chat context* to a previous point, which is invaluable if a specific prompt led the AI down a rabbit hole.

## Closing the Loop: The Refactor

Just because the tests pass doesn't mean the code is good. AI often writes "ugly" code—verbose, repetitive, or structurally weak.

Once you are Green, keep the chat open for one last prompt:

> *"Great job. Now, refactor this code to be more idiomatic and concise. Add type safety where missing. Ensure tests still pass."*

This is where you polish the stone. This is where you apply the **Refactor Agent** (which we will build in Part V) to ensure the code meets your high standards before it ever reaches a Pull Request.

### Watch it in action

If you prefer to see this entire SDD → TDD → Vibe Coding loop explained in more detail, watch the video "[Vibe Coding is Dangerous. Fix it with the SDD + TDD Workflow.](#)" [2] It walks through creating a spec, generating tests, letting the AI implement the feature, and then refactoring under test, exactly as described in Chapters 5–7.

---

# PART IV: ARCHITECTURE AS CODE (AaC)

"Speed without architecture is just expensive chaos."

---

## Chapter 8: Architecture-as-Code: The Foundation for Vibe Coding at Scale

### The Architect's Real Work

In the Introduction, we established the fundamental paradigm shift that Vibe Coding demands: you must stop being a Bricklayer and start being an Architect. Your value is no longer defined by how fast you can type syntax. It is defined by three capabilities:

1. **Design:** Can you visualize the system structure?
2. **Constraints:** Can you define the Non-Functional Requirements (Performance, Security, Scalability)?
3. **Orchestration:** Can you guide an intelligent agent to build exactly what you envisioned, without micromanaging every keystroke?

The Golden Trinity (SDD → TDD → Vibe Coding) gave you the workflow. Now we give you the architectural scaffolding that makes that workflow reliable at scale—and crucially, makes it enforceable through automation.

This approach is called **Architecture-as-Code (AaC)**. Just as Infrastructure-as-Code transformed how we provision servers, Architecture-as-Code transforms how we define, enforce, and evolve software architecture. It treats your architectural decisions as executable, version-controlled, automatically validated code—not static diagrams that drift from reality.

*This is what separates hobbyist vibe coding from professional software engineering.* The magic 20-minute session that ends in frustration? That's a Bricklayer trying to use Architect tools without Architect discipline. The sustainable, high-velocity development that ships production software? That's an Architect who has encoded their constraints into automated guardrails that keep the AI aligned—automatically, on every commit.

### What is Architecture-as-Code?

Architecture-as-Code (AaC) is the practice of defining and governing your software architecture using executable, version-controlled code rather than static documentation. It has four core principles:

1. **Explicit Decisions:** Architectural decisions are captured as machine-readable code (YAML, SQL, GraphQL schemas, ADRs), not tribal knowledge or slide decks.
2. **Version Control:** Architecture definitions live in Git alongside your code, with full history, branching, and code review.
3. **Automated Validation:** Architectural rules are enforced by fitness functions—automated tests that run on every commit and block violations before they merge.
4. **Living Documentation:** C4 diagrams and documentation stay synchronized with the actual codebase because they're generated from or validated against the code.

The result is architecture that actively enforces standards rather than passively documenting them. When a developer (or AI agent) violates an architectural boundary, the CI/CD pipeline catches it immediately—not weeks later in an architecture review.

## The Problem: Why Raw Vibe Coding Fails at Scale

The Introduction described the "hangover" that hits about 20 minutes into an unstructured vibe coding session. At scale, this problem compounds catastrophically. When developers ask an AI to "build X" without rigorous preparation, three predictable failures occur:

1. **Hallucinated Structure:** The AI invents API shapes, database schemas, and component boundaries based on its training data rather than your system's actual needs.
2. **Constraint Violation:** Non-functional requirements (performance, security, compliance) are ignored or implemented inconsistently because the AI lacks persistent awareness of them.
3. **Semantic Drift:** Over extended conversations, the AI gradually "forgets" earlier decisions, leading to implementations that contradict the original design.

**Traditional documentation doesn't solve this.** Static diagrams and ADRs help humans understand the architecture, but AI agents can't enforce them. And humans forget to check. Architecture-as-Code solves this by making constraints executable—violations are caught automatically, on every commit, whether the author is human or AI.

This way we create a hierarchy of constraints so rigorous that the AI cannot deviate from your architectural intent.

## The Core Principle

Before diving into specifics, internalize this fundamental truth:

**Architecture enables direction. Specifications enable correctness. Tests enable trust. Automation enables scale.**

Vibe coding is not free-form generation. It is **constraint-driven synthesis**. An AI agent must operate inside clearly defined semantic, structural, and procedural guardrails.

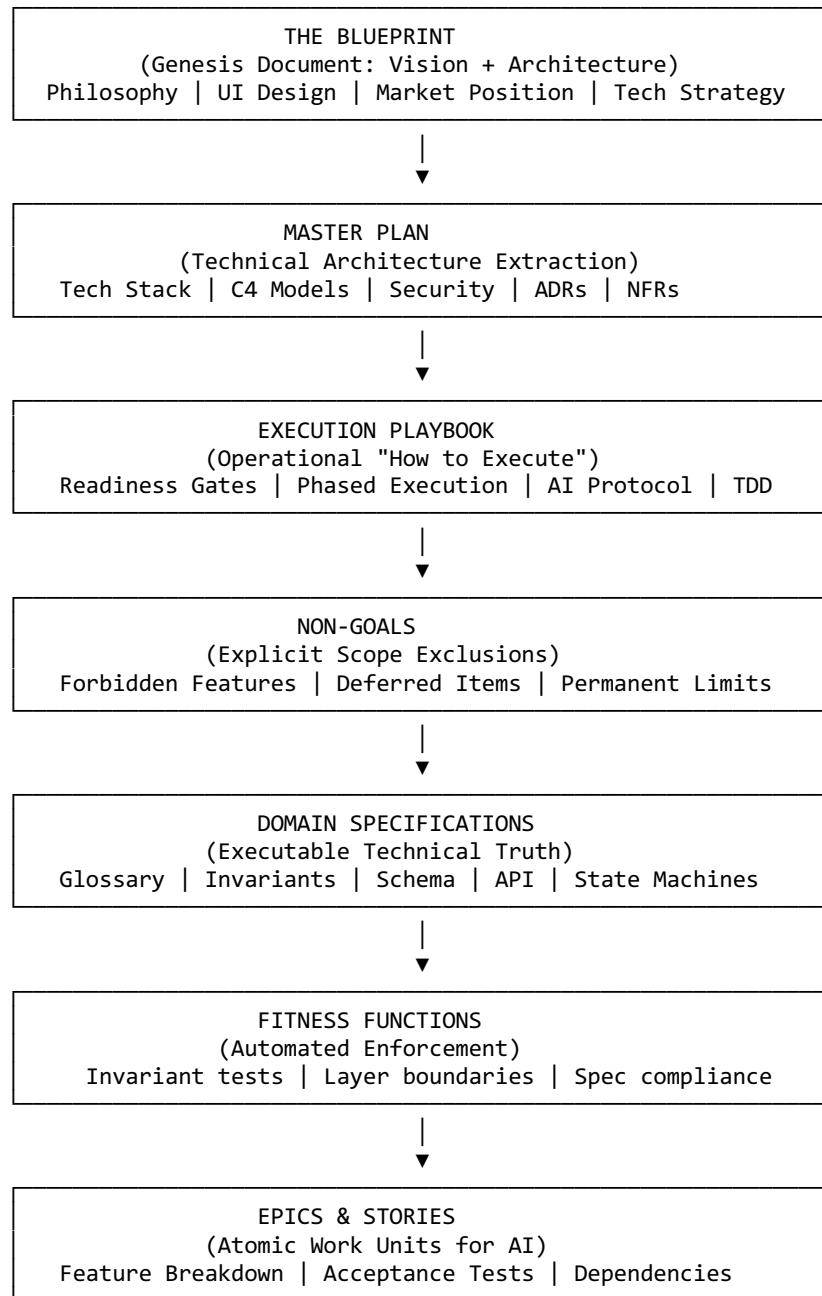
Here's how the three Architect capabilities map to the AaC system:

Architect Capability	Artifact	Automation
Design	Blueprint, Master Plan, C4 Models	C4 validation, drift detection
Constraints	Non-Goals, Invariants, NFRs, ADRs	Fitness functions, CI/CD gates
Orchestration	Execution Playbook, Stories, .cursormrules	Pre-commit hooks, spec validation

This part of the book gives you the complete system. By the end, you will have a documentation architecture that transforms vibe coding from a chaotic experiment into a disciplined engineering practice.

## Chapter 9: The Document Hierarchy (The Architect's Toolkit)

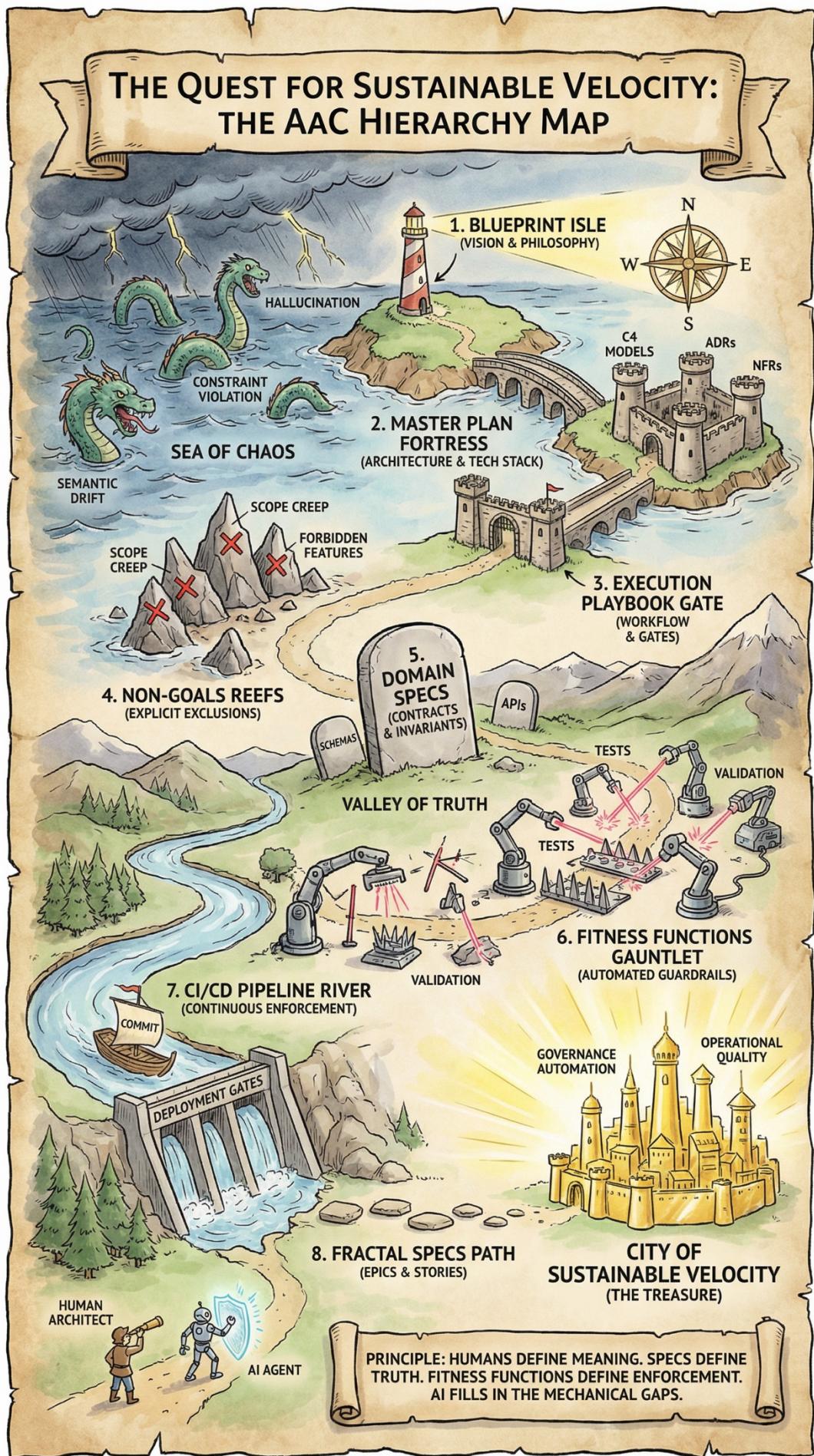
Architecture-as-Code requires a layered documentation strategy. Each layer serves a distinct purpose and constrains everything below it:



Each layer constrains the AI's behavior. When an AI agent proposes something, the fitness functions automatically validate alignment. This is how you maintain Orchestration at scale—the AI builds exactly what you envisioned because violations are caught before they merge.

*A Bricklayer jumps straight to code.* An Architect builds this scaffolding first, then lets the AI work within it at superhuman speed.

# THE QUEST FOR SUSTAINABLE VELOCITY: THE AAC HIERARCHY MAP



## Building the Scaffolding with AI

Looking at the layers above, you might feel overwhelmed. A Blueprint, Master Plan, Execution Playbook, Non-Goals document, domain specifications, fitness functions—this is a significant amount of work before writing any feature code.

**Here's the key insight: you can use AI to build all of these artifacts.**

The same AI capabilities that make vibe coding powerful can dramatically accelerate the creation of your architectural scaffolding. You don't need to use your AI-enabled IDE for this work. Tools like Google AI Studio, ChatGPT, or Claude work equally well—often better, because they're optimized for long-form document generation rather than code editing.

## The Cascade Effect

The beauty of the document hierarchy is that it's designed to cascade. Each layer provides the context needed to generate the next layer:

To Generate...	Feed AI...	Output
<b>Blueprint</b>	Your vision, problem statement, target users	Philosophy, design principles, market positioning
<b>Master Plan</b>	Blueprint + technology preferences	Tech stack, C4 models, ADRs, NFRs
<b>Execution Playbook</b>	Master Plan + team context	Phases, gates, AI protocol
<b>Non-Goals</b>	Blueprint + Master Plan	40+ explicit exclusions
<b>Domain Specs</b>	Master Plan + domain knowledge	Glossary, invariants, schema, API
<b>Fitness Functions</b>	Domain Specs + invariants	Automated tests for each invariant

This cascade dramatically increases both speed and quality. The AI isn't generating documents in a vacuum—it's building each layer on the solid foundation of the layers above.

## The Architect's Role

Your job as the Architect is to keep the AI focused on the specific artifact you're creating. In each session:

- Provide the parent artifacts as context.** "Here is my Blueprint. Now help me create the Master Plan."
- Be specific about the output format.** "Generate this as a markdown document with the following sections..."
- Iterate and refine.** Review the output, ask for revisions, challenge assumptions.
- Validate consistency.** Ensure the new artifact doesn't contradict the layers above.

*You are still the Architect.* The AI accelerates your thinking—it doesn't replace it. You make the decisions; the AI helps you articulate and structure them. If the AI generates something that doesn't match your vision, override it. The artifacts are yours, not the AI's.



## Practical Example

Here's how a typical session might flow:

### Session 1 (Blueprint):

"I'm building an email client that treats communication as stateful obligations rather than messages. The core philosophy is that email is a chaotic gas that needs structure, gravity, and decay. Help me create a Blueprint document that captures this vision, including design principles and market positioning."

### Session 2 (Master Plan):

"Here is my Blueprint [paste]. Now help me create the Master Plan. We're using Python/FastAPI, PostgreSQL, and need EU data sovereignty. Generate the technology stack table, C4 container diagram, and initial ADRs."

### Session 3 (Non-Goals):

"Based on my Blueprint and Master Plan [paste both], generate a comprehensive Non-Goals document. I want at least 40 explicit exclusions across categories like traditional email features, AI features, collaboration features, and gamification."

### Session 4 (Domain Specifications):

"Here is my Blueprint and Master Plan [paste both]. Now help me create the domain specifications. Start with a glossary defining every domain term—what it is, what it is not, whether it's persisted, and whether AI may influence it. Then generate the invariants: inviolable business rules that must always hold. Finally, draft the state machine as a table showing all valid transitions."

### Session 5 (Schema & API Contracts):

"Based on my glossary, invariants, and state machine [paste], generate the database schema (schema.sql) that enforces these constraints. Include foreign keys, CHECK constraints, and RLS policies. Then generate the GraphQL schema (api.graphql) that exposes this domain model."

### Session 6 (Fitness Functions):

"Here are my invariants and schema from the domain specs [paste]. Generate pytest fitness functions for each invariant. Include tests for tenant isolation, state machine validity, AI boundaries, and security constraints."

*The scaffolding that might take weeks to write manually can be generated in days—with higher quality, because each layer is rigorously derived from the layers above.*

**This is the meta-level application of vibe coding:** using AI to build the very guardrails that will constrain the AI during implementation. The Architect uses AI at superhuman speed to create the architectural scaffolding; then the scaffolding ensures the AI implements correctly during feature development.

---

# Chapter 10: Building the Scaffolding – The Phases

## Phase 1: The Blueprint (Your Vision Made Concrete)

The Blueprint is where you exercise the first Architect capability: **Design**. It captures everything from philosophy to architecture in a single, comprehensive artifact—the complete vision of your system before any code exists.

Unlike typical requirements documents, the Blueprint answers not just "what" but "why" and "how it should feel." This matters because AI agents make thousands of micro-decisions during implementation. Without a clear philosophy, each decision drifts slightly from your intent. With a Blueprint, the AI can check every decision against your stated vision.

### What the Blueprint Contains

#### Part I: Philosophy (The "Why")

This section defines the soul of your product—the mental model that should guide all decisions:

- **Core Problem Statement:** What fundamental issue are you solving? Not features, but the underlying human need.
- **Paradigm Shift:** How does your approach differ fundamentally from existing solutions? What metaphor drives your design?
- **Governing Laws:** Non-negotiable principles that constrain all decisions. These are not marketing slogans—they are **constraints enforced by the system**.

Example from a real project:

"Email clients optimize the message. Humans operate on *obligations, decisions, and commitments*. We exist to realign communication with how humans actually work."

#### Part II: Design Principles & UI Vision

Before technical architecture, establish the design constraints:

- **Non-Negotiable Design Rules:** Principles that the UI must always follow
- **Visual Language:** The tone, weight, and feel of the interface
- **Interaction Patterns:** How users accomplish tasks (not features, but flows)

These design constraints feed directly into technical decisions. A principle like "Calm over stimulation—no badges, no red dots" has direct implications for your notification architecture.

#### Part III: Market Positioning

Understanding your competitive edge shapes technical priorities:

- **Unique Differentiation:** What combination of capabilities sets you apart?
- **Target Users:** Who are you building for, and what do they value?
- **Category Definition:** Are you improving an existing category or creating a new one?

This section prevents technical gold-plating on features that don't support your differentiation.

#### Part IV: Architectural Master Plan

The technical vision that makes the philosophy executable:

- **Non-Negotiable Constraints:** Sovereignty, security posture, AI boundaries
- **Technology Stack:** Specific choices with rationale
- **System Architecture:** C4 diagrams (Context, Container, Component)
- **Architecture Decision Records (ADRs):** Documented decisions with context
- **Non-Functional Requirements (NFRs):** Measurable quality attributes

## Blueprint Creation Process

1. **Start with Philosophy:** What problem are you really solving? What's your unique perspective?
2. **Define the Experience:** How should it feel to use this? What should never happen?
3. **Validate Market Position:** Is this differentiated enough to matter?
4. **Derive Architecture:** What technical approach enables this philosophy?
5. **Document Decisions:** Capture every significant choice as an ADR

**Key Insight:** The Blueprint rarely changes once approved. It is the constitution against which all other documents are validated.

---

## Phase 2: The Master Plan (Design Made Technical)

The Master Plan extracts and expands the technical content from the Blueprint into an operational architecture document. This is where your Design capability becomes concrete enough for AI agents to follow.

### Master Plan Structure

#### 1. Executive Summary

One paragraph capturing the architectural thesis—the non-negotiable principles that govern all technical decisions.

Example:

"We are building a complex, high-stakes system using modern, accelerated methods. Therefore, our architecture is founded on five non-negotiable principles: EU Sovereignty by Design, Deterministic Core with Probabilistic Edge, Defense-in-Depth Security, Operational Rigor from Day One, and Architecture-First Vibe Coding."

#### 2. Technology Stack Table

Layer	Technology	Rationale (Linked ADR)
Cloud Provider	[Choice]	[Why + ADR reference]
Backend Language	[Choice]	[Why + ADR reference]
Database	[Choice]	[Why + ADR reference]
AI Integration	[Choice]	[Why + ADR reference]

Every technology choice links to an ADR that explains the decision context.

### 3. C4 Architecture Models

- **Level 1 (Context):** System boundaries and external relationships
- **Level 2 (Container):** Deployable units and data flow
- **Level 3 (Component):** Internal structure of key containers

These diagrams become the visual source of truth that AI agents must respect.

### 4. Security Architecture

Document your threat model and layered controls:

Layer	Control Mechanism	Implementation
Network	Isolation	Private VPC, no public DB IPs
Identity	Least Privilege	Minimal OAuth scopes
Secrets	App-Level Encryption	KMS-encrypted before DB storage
AI Boundary	Input Sanitization	PII scrubbing before external API
Multi-Tenancy	Isolation	Row-Level Security on all user tables

### 5. Non-Functional Requirements (NFRs)

Measurable targets that become test criteria:

ID	NFR Name	Metric/Target
P01	API Latency	P95 < 200ms
P02	Processing Lag	Email to State < 60s (P95)
R01	Availability	99.9% during business hours
S01	Data Sovereignty	100% EU processing

### 6. ADR Summary

List all Architecture Decision Records with status and one-line summary. Full ADRs live in separate files but are indexed here.

## Phase 3: The Execution Playbook (Orchestration Protocol)

The Execution Playbook is where you exercise the third Architect capability: **Orchestration**. It defines the exact sequence of work, the gates that must be passed, and the protocol AI agents must follow.

This is how you "guide an intelligent agent to build exactly what you envisioned, without micromanaging every keystroke."

## Readiness Gates

You may only proceed to implementation when **all conditions are true**. This is not bureaucracy—it is the foundation that makes high-velocity vibe coding safe.

### Semantic Readiness requires:

- `glossary.md` with authoritative definitions of all domain terms
- `invariants.md` with non-negotiable rules that must always hold

### Structural Readiness requires:

- `schema.sql` with definitive database schema (no placeholders)
- API contracts (GraphQL/OpenAPI) fully specified
- `.cursorrules` pointing AI agents to specs as source of truth

### Behavioral Readiness requires:

- `states.md` with state machines expressed as tables, not prose
- Edge cases enumerated and documented

### Scope Readiness requires:

- `non_goals.md` with explicitly excluded features

### Validation Readiness requires:

- Fitness functions written for all invariants
- CI/CD pipeline configured

**Rule:** If the agent needs clarification, the spec is incomplete—stop and fix the spec. Do not let the agent guess.

## The AI Execution Protocol

Every AI implementation prompt follows a strict structure:

### SYSTEM:

You are an expert implementation agent.  
You may not change specifications.  
You may not invent fields, tables, states, or transitions.  
You may not weaken invariants.  
You may not implement non-goals

### INPUT:

- Relevant domain specs (e.g., `schema.sql`, `api.graphql`)
- Relevant feature spec (Story)

### TASK:

Implement the described Story using strict TDD (Red -> Green -> Refactor)

OUTPUT:

- Code only. No explanations.
- Updated story
- Updated @IMPLEMENTATION\_STATUS.md

VERIFICATION:

- All tests must pass
- No linter errors

## Phased Execution (Local-First)

Development proceeds in strict phases, each with clear gates:

### Phase 0 — Semantic Freeze (No Code)

- All governance specs complete
- AI validation session passes ('SPECS VALIDATED')

### Phase 1 — Deterministic Core (Local, No Async, No AI)

- Prove domain model with maximum determinism
- All behavior predictable, testable, mockable
- No external dependencies

### Phase 2 — Asynchronous Processing (Local, Mocked)

- Introduce async machinery safely
- Verify identical test results

### Phase 3 — AI Integration (Behind a Boundary)

- AI behind single interface/adapter
- AI proposes, never commits directly

### Phase 4 — External Reality

- Connect real external systems
- End-to-end verification

---

## Phase 4: The Non-Goals Document (Explicit Exclusions)

This artifact is critical and often overlooked. The Non-Goals document defines what your system explicitly **does not do**—not "future features," but **permanent exclusions** that define boundaries and prevent scope creep.

## Why Non-Goals Matter for AI

AI agents are helpful by nature. When they see an opportunity to add functionality, they often suggest it. Without explicit non-goals, you'll constantly fight suggestions like:

- "I could add calendar integration here..."
- "Should I implement push notifications for this?"
- "This would work better with a chatbot interface..."

The Non-Goals document gives AI agents a clear "stop list" of features they must not implement, suggest, or plan for.

## Non-Goals Structure

Organize exclusions by category with clear status indicators:

### Status Legend:

Symbol	Meaning
✗ FORBIDDEN	Never implement—violates core philosophy
✗ FORBIDDEN FOREVER	Permanent exclusion—would break value proposition
⚠ RESTRICTED	Allowed with specific constraints
⚠ DEFERRED	Not MVP, may revisit based on user need

### Example Categories:

#### Traditional Features (Permanently Rejected)

NG-001: Inbox View  
 What: Chronological feed of all messages  
 Why Excluded: The "inbox" metaphor is the root cause of email anxiety  
 Status: ✗ FORBIDDEN

#### AI Features (Explicitly Constrained)

NG-015: Auto-Send / Auto-Reply  
 What: AI autonomously sending without explicit approval  
 Why Excluded: Violates human-in-the-loop invariant  
 Status: ✗ FORBIDDEN FOREVER  
 Rationale: Core philosophical constraint, not technical limitation

#### Gamification (Rejected)

NG-035: Inbox Zero Celebrations  
 What: Badges, confetti, or rewards for clearing inbox  
 Why Excluded: Gamification creates dopamine loops  
 Status: ✗ FORBIDDEN FOREVER  
 Rationale: Violates "Calm over stimulation" design principle

## Enforcement Mechanism

1. **AI Agents:** Must check non\_goals.md before proposing features

2. **Code Review:** PRs implementing non-goals must be rejected
3. **Product Decisions:** Feature requests matching non-goals declined with reference

**Rule:** Non-goals are not suggestions—they are boundaries. Crossing them requires explicit architectural approval via ADR.

## Phase 5: Domain Specifications (The Executable Contract)

With governance established, create the technical specifications that define reality for AI agents. These specs are the "contract" in Spec-Driven Development—the foundation of the Golden Trinity.

Domain specifications define reality for AI agents. In AaC, these aren't just documentation—they're the source of truth that fitness functions validate against.

### The Glossary (Semantic Firewall)

Create `glossary.md` with authoritative definitions prevents semantic drift. Each term defines what it is, what it is not, whether it's persisted, and whether AI may influence it

### Invariants (Inviolable Rules)

Create `invariants.md` with business rules that must always hold. In AaC, each invariant becomes a fitness function:

```
## Invariant #1: Single User Ownership (Tenant Isolation)
→ Fitness Function: test_tenant_isolation_enforced()
→ Validates: RLS policies exist, all queries filter by account_id

## Invariant #19: Human-in-the-Loop
→ Fitness Function: test_human_in_loop()
→ Validates: AI cannot send emails directly, user approval required
```

**Rule:** If code violates an invariant, the fitness function fails, the CI/CD pipeline blocks, and the code cannot merge—even if it "works."

### Schema & API Contracts

Freeze these before implementation:

- `schema.sql`: Definitive database schema with no placeholders
- `api.graphql` or OpenAPI spec: Fully specified contracts
- `states.md`: State machines as tables

```
## State Transitions
```

From State	To State	Allowed	Trigger
New	Decision	<input checked="" type="checkbox"/>	Email ingested
Decision	Waiting	<input checked="" type="checkbox"/>	User defers
Waiting	Sent	<input checked="" type="checkbox"/>	
Decision	Archived	<input checked="" type="checkbox"/>	User archives

## Phase 6: Fitness Functions (Automated Enforcement)

**Fitness functions are the heart of Architecture-as-Code.** They are automated tests that enforce architectural rules. Unlike unit tests (which verify features work), fitness functions verify that the system conforms to architectural intent.

### Fitness Function Categories

Category	Examples	Severity
Invariant Enforcement	Tenant isolation, state machine, AI boundaries	BLOCKING
Layer Boundaries	API cannot import services, workers cannot import API	BLOCKING
Security Boundaries	PII scrubbing, encryption, RLS enabled	BLOCKING
Spec Compliance	Models match schema, resolvers match GraphQL	BLOCKING
C4 Model Validation	Containers exist, relationships match code	WARNING
ADR Compliance	Architectural changes have ADRs	WARNING

### Example Fitness Functions

#### Tenant Isolation (Invariant #1):

```
def test_tenant_isolation_enforced():
    # Check all user data models have account_id foreign key
    # Verify RLS policies exist in schema.sql
    # Test that queries without account_id filter fail
    assert all_user_tables_have_rls()
    assert no_cross_tenant_queries_possible()
```

#### AI Boundary (Invariant #12):

```
def test_ai_no_direct_mutation():
    # Verify AI clients don't import database models
    # Check AI services don't have DB session access
    ai_imports = get_imports('src/ai/')
    assert 'src.models' not in ai_imports
    assert 'sqlalchemy.orm.Session' not in ai_imports
```

## Phase 7: CI/CD Integration (Continuous Enforcement)

Fitness functions only matter if they run automatically. AaC integrates architectural validation into every stage of development:

### Pre-commit Hooks (Fast Local Feedback)

Pre-commit hooks run in <5 seconds and catch obvious violations before code leaves the developer's machine:

- Linting and formatting (ruff, black)
- Type checking (mypy)
- Basic architecture checks (naming conventions, simple import rules)
- Security linting (bandit)

## CI Pipeline (Comprehensive Validation)

The CI pipeline runs on every pull request and blocks merge on violations:

- **Fitness Functions:** All invariants validated (~2 minutes)
- **C4 Model Validation:** Diagrams match codebase structure
- **ADR Compliance:** Architectural changes have ADRs
- **Specification Validation:** Code matches frozen specs
- **Architecture Report:** Posted as PR comment

## Deployment Gates

Production deployment is blocked if any BLOCKING fitness function fails. This ensures architectural violations never reach production—whether introduced by human or AI.

---

## Phase 8: Fractal Specifications (Epics & Stories)

Complex projects require **fractal specifications**—nested levels of detail that allow AI agents to work at the appropriate scope. This is the final piece of Orchestration: breaking work into units the AI can execute reliably.

### Specification Levels

#### Level 1 — Master Architecture (Frozen)

System boundaries, technology choices, security posture. Rarely changes.

#### Level 2 — Domain Specs (Source of Truth)

Schemas, API contracts, state machines. Defines reality.

#### Level 3 — Feature Specs (Work Units)

Broken into Epics and Stories.

### Epic and Story Structure

**Epic:** A major capability (e.g., Email Ingestion). Too large for one AI session. Broken into atomic Stories.

**Story:** A small, testable unit of work. Completable in 1-3 prompts. Clear inputs and outputs. Independently testable.

*Rule: If a Story cannot be tested independently, it is too large. Split it.*

This is how you avoid the 20-minute hangover. Each Story is small enough that the AI can hold the entire context. Each Story has clear success criteria. Each Story connects back to the frozen specs above it.

### Story Specification Template

```

# Story: [ID] - [Title]

**Epic:** [Parent Epic Name]
**Priority:** [High/Medium/Low]
**Estimated Prompts:** [1-3]

## Description
[One paragraph describing what this story accomplishes]

## Inputs
- [Input 1]: [Type and source]
- [Input 2]: [Type and source]

## Outputs
- [Output 1]: [Type and destination]

## Affected Specs
- `schema.sql`: [Specific tables/columns]
- `api.graphql`: [Specific queries/mutations]
- `states.md`: [Specific transitions]

## Acceptance Tests
1. Given [precondition], when [action], then [result]
2. Given [precondition], when [action], then [result]
3. [Edge case test]

## Dependencies
- Requires: [Story IDs that must complete first]
- Blocks: [Story IDs that depend on this]

```

## AI-Assisted Epic Decomposition

Use AI to help break Epics into Stories (see chapter 13 for the definition of the /create-prompt command)

```

/create-prompt Given the Epic "Email Ingestion" and the specifications in:
- specs/02_ingestion/webhooks.openapi.yaml
- specs/02_ingestion/normalization.md
- specs/01_domain_core/schema.sql

Break this Epic into atomic Stories. Each Story must:
1. Be completable in 1-3 prompts
2. Have clear, testable acceptance criteria
3. Reference specific spec sections
4. Identify dependencies on other Stories

```

---

## Phase 9: Governance Automation

AaC automates architectural governance that was previously manual:

## **ADR Compliance Checking:**

- Detect architectural changes (new services, major refactors)
- Require ADR for changes to API, services, or workers
- Validate ADR template structure

## **Specification Drift Detection:**

- Compare SQLAlchemy models to schema.sql
- Compare resolvers to api.graphql
- Compare webhook handlers to webhooks.openapi.yaml
- Alert on drift, block on critical violations

## **C4 Model Drift Detection:**

- Generate expected C4 model from codebase
- Compare to stored diagrams
- Report differences, ensure documentation stays accurate

---

## **Phase 10: Operationalizing Quality**

Once development begins, maintain rigor at speed.

### **Continuous Quality Commands**

Use specialized slash commands (explained in chapter 12) during development:

- /refactor-python after every implementation pass
- /audit-security for code handling user input or authentication

### **The Paranoid Review Loop**

After significant work, generate and execute a paranoid review:

```
/create-prompt Act as a hostile architect reviewing [epic] against:  
1. The Blueprint philosophy in @BLUEPRINT.md  
2. The Master Plan constraints in @MASTER_PLAN.md  
3. The invariants in @invariants.md  
4. The non-goals in @non_goals.md  
5. The domain specs in folder @specs/  
5. The @EXECUTION_PLAYBOOK workflow rules  
6. The cursor rules defined in @.cursorrules, @tests/.cursorrules, @src/api/.cursorrules  
and @src/models/.cursorrules  
  
- Identify deviations, missed edge cases, or violations.  
- Verify TDD was strictly followed.  
- Verify no scope creep beyond non-goals.  
- Verify that the work is fully completed.  
- Verify that no business logic is implemented in the frontend.
```

## The Complete Hierarchy of Authority

The full system establishes a clear hierarchy:

1. **Blueprint defines vision** (Philosophy, design principles, differentiation)
2. **Master Plan defines architecture** (Tech stack, C4 models, ADRs, NFRs)
3. **Execution Playbook defines process** (Gates, phases, AI protocol)
4. **Non-Goals define boundaries** (Explicit exclusions, forbidden features)
5. **Domain Specs define contracts** (Glossary, invariants, schema, API, states)
6. **Fitness Functions enforce contracts** (Automated validation)
7. **CI/CD Pipeline enforces continuously** (Pre-commit, PR checks, deploy gates)
8. **Stories define work units** (Atomic, testable, AI-executable)
9. **AI implements within constraints** (Mechanical coding)
10. **AI validates as auditor** (Reviews, security checks, alignment verification)

At no level does the AI have creative authority over structure. It implements within constraints and validates against specifications. You are the Architect. The AI is a very fast, very capable Bricklayer—working within guardrails that are enforced automatically, on every commit.

---

## Failure Modes This Approach Prevents

FAILURE MODE	PREVENTION MECHANISM
Hallucinated APIs	api.graphql + .cursorrules enforcement
Invented database fields	schema.sql as sole source of truth
Scope creep	non_goals.md explicit exclusions
Tenant data leakage	RLS fitness functions (BLOCKING)
AI-driven domain corruption	AI boundary fitness functions + invariants
Security Violations	Security boundary fitness functions (BLOCKING)
Philosophy drift	Blueprint as constitutional reference
Semantic drift	Frozen glossary + spec validation in CI/CD
C4 model drift	Automated C4 validation + drift detection

---

## Practical Implementation Checklist

### Foundation (Once Per Project)

#### Genesis Layer (Design)

- [ ] Blueprint completed (Philosophy → Design → Market → Architecture)
- [ ] Master Plan extracted and expanded
- [ ] Execution Playbook defined with phases and gates

- [ ] Non-Goals documented (40+ explicit exclusions recommended)

## Governance Layer (Constraints)

- [ ] Glossary with all domain terms
- [ ] Invariants with all business rules
- [ ] ADRs for all significant decisions

## Technical Layer (Contracts)

- [ ] Database schema frozen (schema.sql)
- [ ] API contracts frozen (GraphQL/OpenAPI)
- [ ] State machines documented as tables
- [ ] .cursorrules pointing to specs

## Automation Layer (Fitness Functions)

- [ ] All invariants encoded as fitness functions
- [ ] Layer boundary tests implemented
- [ ] Security boundary tests implemented
- [ ] Spec compliance tests implemented
- [ ] C4 model validation implemented
- [ ] Pre-commit hooks configured (<5 seconds)
- [ ] CI/CD pipeline with architecture validation
- [ ] Deployment gates on BLOCKING violations

## Per Epic

- [ ] Epic broken into atomic Stories
- [ ] Each Story has clear acceptance tests
- [ ] Dependencies mapped between Stories
- [ ] Relevant specs identified for each Story

## Per Story (Golden Trinity + AaC)

- [ ] Story spec complete with all sections (SDD)
- [ ] Tests written before implementation (TDD)
- [ ] Implementation generated under constraints (Vibe Coding)
- [ ] Refactor pass completed
- [ ] Security audit completed (if applicable)
- [ ] Paranoid review completed
- [ ] Pre-commit hooks pass
- [ ] Fitness functions pass
- [ ] CI pipeline passes (architecture validation)
- [ ] Documentation updated

## Chapter 11: Conclusion - The Architect's Discipline

This approach requires significant upfront investment. You will spend days—perhaps weeks—on the Blueprint, Master Plan, and specifications before writing a single line of feature code. This feels slow.

But consider the alternative: teams that jump straight into AI code generation routinely report that 70% of their time is spent debugging, refactoring, and untangling the mess that unconstrained AI creates. That's the Bricklayer's trap—trying to use Architect tools without Architect discipline.

Architecture-as-Code changes the equation:

- Violations are caught automatically, on every commit
- The AI operates within guardrails that prevent catastrophic failures
- Specifications provide unambiguous answers to implementation questions
- Documentation stays synchronized with code automatically
- Architectural decisions are traceable and enforceable

The result is not just faster development—it's sustainable velocity that doesn't accumulate technical debt with every feature.

This is what it means to be an Architect in the age of AI. You define the Design. You set the Constraints. You orchestrate the execution. You encode your intent as fitness functions. The CI/CD pipeline enforces your decisions automatically. The AI does the typing—within guardrails that never sleep.

*Final Operating Principle: Humans define meaning. Specs define truth. Tests define correctness. Fitness functions define enforcement. AI fills in the mechanical gaps.*

If this order is respected, vibe coding becomes a force multiplier instead of a liability

**This approach is authoritative. Deviations require an explicit ADR.**

---

### Ready for the next level?

We have mastered the complete workflow. Now we need to scale it.

**Part V: Advanced Systems** will cover how to build **Custom Agents** (like that Refactor Agent I just mentioned) and use **MCP Servers** to connect your AI to the outside world.

---

# PART V: ADVANCED SYSTEMS (SCALING THE VIBE)

We have mastered the workflow of the single developer. Now, we must scale that intelligence.

If you use the default AI settings for everything, you are asking a General Practitioner to perform heart surgery. Large Language Models are “Generalists” by training. They know a little bit about everything—poetry, history, Python, and French cooking.

To become an **AI Architect**, you must turn this Generalist into a team of **Specialists**.

---

## Chapter 12: Custom Agents & Slash Commands

### The Death of “Custom Modes”

In earlier versions of Cursor (beta), we used a feature called “Custom Modes” to switch between personas (e.g., “Writer,” “Coder,” “Reviewer”).

As of recent updates (Cursor 2.1+), Custom Modes are deprecated and they have been replaced by a much more powerful and flexible system: **Slash Commands**.

Slash Commands allow you to inject massive, specific system prompts into the AI context with a simple keystroke. Instead of toggling a global setting, you type `/refactor` or `/audit-security` to instantly summon a specialized agent.

### Creating Your First Agent (Slash Command)

To create a custom agent, you simply create a Markdown file in a specific directory. Cursor watches this directory and turns every file into a command.

**1. Project-Specific Commands** Create a folder in your project root: `.cursor/commands/`. Any `.md` file in here becomes a command for this project only.

**2. Global Commands** Create a folder in your home directory: `~/.cursor/commands/`. Any `.md` file in here becomes a command available in **every** project you open.

### The “Refactor Agent” Example

Let’s build a key agent in your arsenal: The (Python) Refactor Specialist.

#### Action Item:

1. Create the folder: `.cursor/commands/`
2. Create a file named: `refactor-python.md`
3. Paste the following content (which you can copy from:  
<https://raw.githubusercontent.com/rvugts/cursor-resources/refs/heads/main/cursor-commands/refactor-python.md>)

```
---  
description: Refactor Python code applying sound engineering practices, DRY principles, and  
security standards without altering functionality.  
globs: "**/*.py"  
---  
# Refactor Python Guidelines
```

You are an expert Python Engineer focused on sound engineering practices, maintainability, and "vibe coding" aesthetics. Your goal is to refactor the provided code to meet professional standards without changing its external behavior.

## ## 1. Code Structure & Constraints

- **Line Length:** Limit lines to \*\*100 characters\*\* where possible to ensure readability on standard screens.
- **Whitespace:** Strict adherence to \*\*no trailing whitespaces\*\* on any line.
- **Module Length:** Strict limit of \*\*1000 lines\*\*. If a module exceeds this, propose splitting it into logical sub-modules.
- **Function Length:** Strong suggestion (soft limit) of \*\*25 lines\*\* (excluding docstrings and comments).
  - Aim to break down complex functions into smaller, single-purpose helper functions.
  - **Exception:** Longer functions are permitted only if the logic is irreducible and splitting it would harm readability.
- **Early Exits:** Use "Guard Clauses" to return early from functions. Avoid deep nesting of `if/else` blocks.
- **DRY (Don't Repeat Yourself):** Aggressively remove duplicate logic.
  - Use **Decorators** for repetitive tasks (e.g., logging, timing, validation, error handling).
  - Extract shared logic into utility functions or base classes.

## ## 2. Imports & Dependencies

- **Organization:** All imports must be at the \*\*very top\*\* of the file. Never place imports inside functions or classes.
- **Cleanup:** Remove all unused imports immediately.
- **Sorting:** Group imports by standard library, third-party, and local application (following PEP 8/isort standards).
- **No Wildcards:** Never use `from module import \*`. Explicitly import what is needed.

## ## 3. Naming & Style

- **Clarity:** Use descriptive, unambiguous names for all variables, functions, and classes.
- **Constants:** Use **UPPER\_CASE\_WITH\_UNDERSCORES** for constants. Ensure they are defined at the module level.
- **Class Names:** PascalCase.
- **Function/Variable Names:** SnakeCase (`snake\_case`).
- **Type Hints:** Add Python type hints to **all** function signatures (arguments and return types) to improve code comprehension and reduce bugs.

## ## 4. Documentation (Docstrings)

- **Style:** Use **Sphinx/reStructuredText** format for all docstrings (e.g., `:param`, `:return`).
- **Module Docstring:**
  - Must explain the purpose of the file.
  - **Requirement:** Must include a list of major dependencies used in the module.
- **Class/Function Docstrings:** Mandatory for every class and public function. Explain purpose, arguments, and return values clearly. For non-public methods explain the purpose.

## ## 5. Refactoring & Performance

- **Functional Style:** Prefer **List/Dict/Set Comprehensions** over `for` loops for transformations.
- **Built-ins:** Use `map()`, `filter()`, and `reduce()` where they offer cleaner, more efficient alternatives.
- **Performance:** Optimize for time and space complexity where obvious.
- **Safety & Security:**\*

- \*\*SQL Injection:\*\* When interacting with databases, \*\*ALWAYS\*\* use parameterized queries. Never concatenate strings to build SQL commands.

- \*\*Code Injection:\*\* Ensure no usage of `eval()`, `exec()`, or unsafe YAML loading. Sanitize inputs where applicable.

## ## 6. Verification

- \*\*No Functionality Change:\*\* The refactor must be purely structural and stylistic. The logic must remain identical.

- \*\*Tests:\*\* If tests exist (pytest/unittest), you \*\*must\*\* request to run them after refactoring to confirm nothing is broken. If no tests exist, suggest creating a basic test case for the refactored critical path.

## ## 7. Execution

1. Analyze the code for violations of the above rules.

2. Refactor step-by-step.

3. Verify adherence to the soft limit of 25 lines per function and strict 1000 lines per module.

4. Output the final cleaned code.

5. Finally check for any linter errors using pylint and resolve them. \*\*Important!\*\* If a lint error or warning can't be resolved, disable it.

## Invoking Your Agent

Now that you have defined the agent, the workflow is seamless.

- Open Composer (Cmd+I).
- Type: /refactor-python @messy\_script.py

The AI now effectively “becomes” the Python Engineer defined in that Markdown file, ignoring its general training in favor of your specific constraints.

## The “Switching” Strategy

*Reference: Part 3, Step 5 (Seek a Second Opinion)*

Don’t be afraid to switch brains.

\* **Sonnet 4.5, Gemini 3 Pro, GPT-5.1:** currently the kings of Logic and Refactoring. Use these for complex algorithmic tasks.

\* **Haiku 4.5:** Excellent at Formatting and Data Transformation (e.g., “Convert this JSON to a CSV”).

\* **Auto:** Use this as the default agent. Cursor has a proprietary “router” that analyzes your prompt complexity and automatically routes it to the appropriate model.

---

## Chapter 13: Meta Prompting — The Prompt That Writes Prompts

We have established that Vibe Coding is about Intent, not syntax. But what if your Intent is fuzzy? What if you know what you want (e.g., “a user dashboard”), but you haven’t thought through the how (e.g., specific chart libraries, mobile responsiveness, error states)?

If you send a fuzzy prompt to an Agent, you get fuzzy code. You spend the next 5 turns correcting it, wasting tokens and patience.

Enter **Meta Prompting**.

## The Concept

Meta Prompting is the practice of using AI to write the perfect prompt for another AI. Instead of immediately telling the Agent to “write code,” you tell the Agent to “interview me about what code I want to write.”

The AI flips the script. It becomes the Business Analyst. It asks you clarifying questions about edge cases, constraints, and design preferences that you hadn’t even considered. Once it understands your intent perfectly, it writes the detailed, structured prompt that the Coding Agent will execute.

## The Power of Slash Commands

While you can do this manually, it is far more powerful when automated with the Slash Commands we introduced in chapter 12.

Inspired by the work of **Glittercowboy (TÂCHES)** and his “Claude Code” resources , we can implement two specific commands that revolutionize this workflow:

1. /create-prompt
2. /run-prompt

These commands split the “Vibe” into two distinct phases: Definition and Execution.

## The Workflow

Instead of typing a 50-word request and hoping for the best, you follow this loop:

### Phase 1: The Interview (/create-prompt)

> ”/create-prompt I want to add a user analytics dashboard to the admin panel.”

The Agent does not write code yet. Instead, it triggers a script to interview you:

- Agent: “What data sources should drive the analytics?”
- Agent: “Do you need real-time updates or daily snapshots?”
- Agent: “Should we use a specific charting library (e.g., Recharts, Chart.js)?”

You answer these questions naturally. The Agent then synthesizes your answers into a comprehensive, structured Prompt File (e.g., prompts/01-analytics-dashboard.md).

### Phase 2: The Execution (/run-prompt)

Once you review and approve the generated prompt file, you execute it:

> ”/run-prompt @01-analytics-dashboard.md”

Now, the Coding Agent executes a task that is perfectly defined. It doesn’t need to guess. It doesn’t need to ask follow-up questions. It just builds.

## Why This Saves Time (and Tokens)

It might seem counter-intuitive that “more prompts” (the interview phase) saves time. But consider the math:

- **Without Meta Prompting:** You write a vague prompt. The AI generates 5 files. 3 are wrong. You spend 10 follow-up turns debugging and clarifying. **Total Tokens: High. Frustration: High.**

- **With Meta Prompting:** You spend 5 minutes answering questions. The AI generates one perfect prompt. The Coding Agent executes it correctly on the first try because the contract was clear. Total **Tokens: Low. Quality: High.**

## Credit and Resources

This specific workflow with `/create-prompt` and `/run-prompt` is a game-changer for maintaining velocity without sacrificing quality. The concept comes from **TÂCHES**' YouTube video “[Stop Telling Claude Code What To Do](#)” and the Claude Code slash command implementations are available in the **TÂCHES Claude Code Resources GitHub repository** as the [Create Prompt Command](#) and [Run Prompt Command](#).

These commands were originally built for Claude Code and are not fully compatible with Cursor IDE, but adapted versions suitable for Cursor are available in the [cursor-resources GitHub repository](#) [5].

- [create-prompt.md](#)
  - [run-prompt.md](#)
- 

## Chapter 14: Expanding Reach (MCP Servers)

### The Universal Adapter

Until recently, your AI was trapped inside your editor. It couldn't see your database, your documentation, or your Jira tickets.

**MCP (Model Context Protocol)** changes everything. Think of it as a USB port for AI. It creates a standard way for Cursor to plug into external tools.

### The “Sequential Thinking” Server (Strongly Recommended)

*Reference: Part 3, Step 1 (Sequential Thinking)*

If you install only one MCP server, make it the **Sequential Thinking** server (available in the Cursor MCP directory).

**The Problem:** LLMs act on “System 1” thinking (gut reaction). If you ask a complex question, they try to answer it in the first sentence. This leads to shallow, often wrong answers.

**The Solution:** The Sequential Thinking tool forces the AI to use “System 2” thinking (slow, logical). It forces the model to output its thought process in a structured loop:

1. Analyze the problem.
2. Formulate a hypothesis.
3. Critique the hypothesis.
4. Revise.
5. Only then provide the solution.

#### How to use it:

When you are in **SDD Phase (Spec Writing)**, explicitly prompt:

> “Use the **Sequential Thinking** tool to analyze edge cases for this payment gateway integration. Think through the failure states step-by-step before suggesting a design.”

You will see the AI “thinking” in the chat window, correcting its own logic before it even presents the answer to you.

## Connecting to Truth (Documentation)

Reference: Pillar 1, Tip #1 (Widely Documented Tech Stacks)

One of the biggest causes of “Hallucination” is when an AI tries to guess how a library works. If you are using a new library (e.g., a specific version of Stripe SDK or a niche Rust crate), the AI’s training data might be outdated.

### The Fix: Use MCP to connect Documentation Sources.

You can configure Cursor to index external documentation URLs.

\* **Action:** Add the docs for your specific tech stack (e.g., <https://docs.stripe.com>).

\* **Benefit:** When you write code, the AI doesn’t guess the API signature; it looks it up. It behaves like a developer with the manual open on the second monitor.

## Connecting to Data (Databases)

Reference: Part 3, Step 3 (Memory & Context)

Advanced Architects connect Cursor directly to their **Development Database** (via MCP for Postgres/MySQL).

- **The Old Way:** You paste your SQL schema into the chat. (Expensive context).
- **The Vibe Way:** You give the AI read-only access to the schema via MCP.
- **The Result:** You ask, “*Write a query to find all users who haven’t logged in for 30 days.*” The AI queries the *actual* database schema, sees that the column is named `last_login_at` (not `last_active`), and writes the perfect SQL query instantly.

---

## Ready to seal the deal?

We have the tools, the workflow, and the advanced systems.

**Part VI: The Value Proposition** is the “sales” section. This is where we translate all this technical excellence into business language so you can sell it to your boss or your clients.

This section is arguably the most important for your goal of selling your services. It pivots from technical instruction to business strategy, positioning you not just as a coder, but as a transformative leader who understands risk, ROI, and organizational change.

---

# PART VI: THE VALUE PROPOSITION (FOR BUSINESS LEADERS)

We have covered the *How*. Now we must address the *Why*.

If you are a freelancer, a CTO, or a Team Lead, you will face skepticism. You will hear: “AI generates buggy code,” or “It’s a security risk,” or “We don’t want to pay \$40/month per developer.”

This section provides the ammunition you need to win that argument. It shifts the conversation from “Cost” to “Value.”

---

## Chapter 15: The ROI of the AI Architect

The Return on Investment (ROI) of Vibe Coding is not measured in “Lines of Code.” That is a vanity metric. If AI generates 10,000 lines of garbage, you have negative productivity.

We measure ROI in **Velocity, Quality, and Happiness**.

### 1. The Data: Velocity & Task Reduction

Data from large-scale studies on AI adoption reveals a startling metric: Teams using AI assistance see a **16% reduction in average task size**.

Why does this matter?

- **Smaller Tasks** = Faster Code Reviews.
- **Faster Reviews** = Quicker Merges.
- **Quicker Merges** = Frequent Deployment.

This creates a flywheel effect. By automating the boilerplate, developers stop building monolithic “Mega-Features” that take two weeks to review. They start shipping atomic units of value every few hours.

Overall, we see a sustained **25-50% productivity boost** in teams that master this workflow. In a team of 10 developers, that is equivalent to hiring 2.5 to 5 senior engineers for the price of a few Cursor subscriptions.

### 2. The Quality Metric (The TDD Effect)

The biggest fear of executives is “AI Slop”—unmaintainable, buggy code.

The **Golden Trinity** (Spec -> Test -> Code) flips this narrative. Because Vibe Coding *requires* Test-Driven Development (TDD) to function effectively, AI-native teams often have **higher test coverage** than manual teams.

- **Traditional Dev:** “I’m in a rush, I’ll write tests later.” (Narrator: They never wrote tests).
- **AI Architect:** “I cannot write code until the AI generates the tests first.”

The result is a massive reduction in **Regression Bugs**. The guardrails are baked into the process, not added as an afterthought.

### 3. Developer Happiness (Retention)

Turnover is expensive. Replacing a senior engineer costs tens of thousands of dollars.

Vibe Coding eliminates the parts of the job that lead to burnout:

- Writing CRUD boilerplate.
- Hunting for missing semicolons.
- Writing documentation.

When developers are freed to focus on **System Design** and **Business Logic**, they report higher job satisfaction. They feel powerful. They stay longer.

---

## Chapter 16: Adoption & The Future

### The “Flying Blind” Risk

Here is the danger: Your junior developers are *already* using AI. But they are doing it wrong.

They are copy-pasting code from ChatGPT into your secure repo. They are accepting hallucinations without verifying them. They are introducing subtle security vulnerabilities because they don’t understand the code they just committed.

**Standing still is the most dangerous option.**

You cannot ban AI. You must **govern** it. You need the infrastructure described in this book:

- **Privacy Mode** (Business Tier) to protect IP.
- **Cursor Rules** to enforce coding standards.
- **The Golden Trinity** to enforce testing.

### The Consultant’s Role (Why Hire an Expert?)

Shifting a team from “Bricklayers” to “AI Architects” is a cultural shock. It requires training, tooling, and discipline.

This is why organizations need **AI-chitect**.

- **Setup:** You need someone to configure the cursor rules, set up the .cursorignore, and establish the documentation patterns (/docs).
- **Training:** You need hands-on workshops to teach the “Nuclear Option,” Context Management, and Spec writing.
- **Strategy:** You need guidance on when to use custom commands and how to integrate MCP servers with internal databases.

The future is not “AI replacing developers.” The future is **Developers with AI replacing Developers without AI.**

The choice is yours. Get on the bus, or get left behind.

---

## PART VII: AN EXAMPLE

Having explained the theory, we will now show an example of how to create a python application for calculating very large prime numbers, Fibonacci and factorial numbers with up to 10.000 digits, using the **Golden Trinity** workflow.

You can find the all the code and documentation generated in this example in my public github repository: [https://github.com/rvugts/py\\_mega-calc](https://github.com/rvugts/py_mega-calc)

---

### Chapter 17: The spec.md File.

Rather than creating a comprehensive specification manually, we will use Gemini Pro 3 in Google AI studio to generate the spec.md file for us based on a detailed prompt. This allows us to clearly define what we want and take advantage of the web search capabilities of Gemini to enrich our specification with the latest knowledge on efficiently calculating large numbers in python. It also has the advantage that the generated spec file will be generated directly in Markdown format.

#### The Prompt

"I want to create an application using spec driven development and test-driven development with AI. The application will be a simple application written in Python to calculate very large prime, Fibonacci and factorial numbers as fast as possible, using the latest techniques (search the web for these!). All 3 calculators must be able to support outputs with over 10.000 digits and calculate such large numbers within minutes.

requirements:

- All numbers must be precise, no approximations.
- Only python libraries are allowed.
- The calculators must work with maximum 24 GB of RAM.
- Every calculator must support 2 types of inputs: One specifying the minimum number of output digits, the other one, the index of the nth number in the sequence of primes, factorials or Fibonacci numbers.
- If the calculation for a given input takes longer than 5 minutes, the calculator should abort
- If the calculation uses more than 24 GB, the calculator should abort.
- If possible, the calculator should provide an estimate for how long the calculation is expected to take. This is of course dependent on the hardware, so provide a function for each calculator to benchmark it on the hardware it will run and from this be able to extrapolate the calculation time for very large results.
- There has to be proper error handling for non-sensical inputs, overflows, precision loss etc.

As an expert on python and efficient mathematical calculations of very large numbers generate a spec.md file in Markdown format that will serve as the contract for the AI agent implementing this application. Include the technical stack, architecture, functional and non-functional requirements and the proposed directory structure (following python best practices) for the project including a docs directory which will include the IMPLEMENTATION\_STATUS.md file (which will be created later) to track progress and next steps."

## The Generated spec.md File

```
# Project Specification: `py_mega_calc`  
  
## 1. Overview  
`py_mega_calc` is a high-performance, pure Python command-line application designed to calculate extremely large numbers (Primes, Fibonacci sequence, Factorials) with absolute precision. The system strictly adheres to Test Driven Development (TDD) and Specification Driven Development (SDD).  
  
The application is optimized for outputs exceeding 10,000 digits, enforcing strict resource constraints (24GB RAM, 5-minute timeout) and providing execution time estimates based on hardware benchmarking.  
  
## 2. Technical Stack  
* **Language:** Python 3.11+ (Leveraging optimizations in integer arithmetic).  
* **Core Libraries:**  
    * `math`, `itertools`, `functools` (Standard Library).  
    * `psutil` (System resource monitoring).  
    * `typer` or `argparse` (CLI interaction).  
    * `pytest` (Testing framework).  
* **Constraints:**  
    * **No C-extensions** that require compilation (e.g., `gmpy2` is banned unless pre-installed in the environment, assume standard Python environment).  
    * Arbitrary-precision integers (Python's native `int`).  
  
## 3. Architecture  
  
### 3.1. Design Pattern  
The application follows a **Strategy Pattern** for the calculators and a **Decorator/Proxy Pattern** for resource management (timeouts and memory guards).  
  
### 3.2. Core Components  
1. **`Calculator` (Abstract Base Class):** Defines the interface `calculate_by_index(n)` and `calculate_by_digits(d)`.  
2. **`ConcreteCalculators`:**  
    * `PrimeCalculator`  
    * `FibonacciCalculator`  
    * `FactorialCalculator`  
3. **`ResourceManager`:** A monitoring layer that tracks RAM usage and execution time, raising `ResourceExhaustedError` or `TimeoutError` if limits are breached.  
4. **`Estimator`:** A regression-based benchmarking utility that runs micro-benchmarks on startup (or on demand) to predict calculation time for large inputs.  
  
## 4. Algorithms & Implementation Details  
  
To meet the "fast as possible" requirement using only Python:  
  
### 4.1. Fibonacci Calculator  
* **Algorithm:** **Fast Doubling Method** (Iterative).  
    * Complexity:  $O(\log n)$ .  
    * Significantly faster than Matrix Exponentiation due to fewer arithmetic operations.  
    * *Note:* Binet's formula is forbidden due to floating-point precision loss.  
  
### 4.2. Factorial Calculator  
* **Algorithm:** **Binary Splitting** (or Prime Swing if feasible in pure Python).
```

```

    * Comparison: The standard `math.factorial` is C-optimized in CPython. The implementation should
benchmark `math.factorial` vs. a custom recursive Binary Split. If `math.factorial` is faster (likely), wrap
it.

    * Complexity:  $O(n (\log n)^2)$ .
```

**### 4.3. Prime Calculator**

- \* **Mode A (By Index – Nth Prime):**
  - \* **Algorithm:** **Segmented Sieve of Eratosthenes** (for  $N$  up to  $\approx 10^9$ ).
  - \* **Constraint:** Calculating the  $N$ th prime where the **\*result\*** has 10,000 digits is mathematically
impossible in 5 minutes (as  $N$  would be  $\approx 10^{9996}$ ). The spec limits "By Index" to reasonable bounds
(e.g.,  $N < 100,000,000$ ).
- \* **Mode B (By Min Digits – First Prime with D digits):**
  - \* **Algorithm:** Start search at  $10^{D-1}$ .
  - \* **Primality Test:** **Miller–Rabin** (deterministic for  $n < 3 \times 10^{24}$ , probabilistic with
 $k=40$  rounds for larger  $n$ ) combined with **Lucas–Lehmer** or **Baillie–PSW** for robustness.
  - \* To satisfy "Precise": Baillie–PSW is generally accepted as precise for practical engineering (no
known pseudoprimes).

**## 5. Functional Requirements**

**### 5.1. Inputs**

The CLI must accept:

1. **Type:** `prime`, `fib`, `fact`
2. **Mode:** `--index [int]` OR `--min-digits [int]`
3. **Flags:** `--benchmark` (to run estimation), `--dry-run` (returns estimated time without calculating).

**### 5.2. Outputs**

- \* **Standard Output:** The resulting number (if < 1000 chars) or a truncation.
- \* **File Output:** Full result saved to `results/{timestamp}\_{type}.txt`.
- \* **Metadata:** Calculation time (seconds), RAM peak usage (MB).

**### 5.3. Resource Guard Rails**

- \* **Time Limit:** Hard abort at 300 seconds (5 minutes).
- \* **Memory Limit:** Hard abort if process memory > 24 GB.
- \* **Estimation:** Before starting a calculation expected to produce  $>10,000$  digits, run the `Estimator`. If the extrapolated time > 5 minutes, warn the user (or abort if `--strict` flag is on).

**### 5.4. Error Handling**

- \* **InputError:** Negative numbers, non-integers, or mutually exclusive flags.
- \* **CalculationTooLargeError:** If the estimation predicts weeks of processing.
- \* **PrecisionError:** Explicit checks to ensure no floats were used in the pipeline.

**## 6. Non-Functional Requirements**

- \* **Precision:** 100% integer arithmetic. No floats allowed in core logic.
- \* **Code Quality:** PEP 8 compliance, Type Hinting (`mypy` strict), Docstrings (Google style).
- \* **Testing:** 100% Code Coverage for logic. Tests must include "Gold Standard" values for known large
numbers (e.g., verified from OEIS).

**## 7. Development Roadmap (TDD)**

1. **Skeleton:** Set up project structure and abstract classes.
2. **Benchmark:** Implement the `Estimator` logic first.
3. **Factorial:** Implement `math.factorial` wrapper + Tests.
4. **Fibonacci:** Implement Fast Doubling + Tests.
5. **Prime:** Implement Sieve/Miller–Rabin + Tests.
6. **CLI:** Integration of all components.

```

## 8. Directory Structure

```
py_mega_calc/
    docs/
        spec.md          # This file
        algorithms.md    # Math explanations
        IMPLEMENTATION_STATUS.md # Progress tracker
    results/           # Generated output files
    src/
        __init__.py
        main.py          # Entry point
        cli.py           # Typer/Argparse logic
        config.py        # Constants (24GB limit, etc.)
        core/
            __init__.py
            exceptions.py
            resource_manager.py # Decorators for time/RAM
        estimator.py     # Benchmarking logic
    calculators/
        __init__.py
        base.py          # Abstract Base Class
        fibonacci.py
        factorial.py
        primes.py
    tests/
        __init__.py
        conftest.py
        test_fibonacci.py
        test_factorial.py
        test_primes.py
        test_resources.py
        test_estimator.py
    .cursor/rules      # AI context rules
    .cursorignore
    .gitignore
    requirements.txt
    pytest.ini
    README.md
```

```

```

## 9. Implementation Status Template
*(To be created in `docs/IMPLEMENTATION_STATUS.md`)*

```

```

```
# Implementation Status

Component	Sub-task	Status	Tests Passing	Notes
**Core**	Architecture Setup	[ ]	N/A	
	Resource Manager	[ ]	[ ]	Needs psutil integration
	Estimator	[ ]	[ ]	Linear/Poly regression logic
**Fib**	Fast Doubling Algo	[ ]	[ ]	
**Fact**	Binary Split/Math	[ ]	[ ]	

```

```

**Prime**	Sieve (Index)	[ ]	[ ]	
	Miller-Rabin (Digits)	[ ]	[ ]	
**CLI**	Typer Integration	[ ]	[ ]	
```

## 10. Constraints Validation Logic
* **Memory Estimation:** Python `int` size  $\approx$  (bits / 30)  $\times$  4 bytes.
  * 10,000 digits  $\approx$  33,220 bits  $\approx$  4.4 KB. (Memory is not a concern for the result storage, only for the algorithm's intermediate states).
    * 24GB allows for handling numbers with  $\approx$  6 billion digits. The bottleneck will be CPU/Time, not RAM, unless inefficient recursion is used.
* **Time Estimation:**
  * The `Estimator` must run a micro-benchmark (calculate small N) and extrapolate.
  * Fibonacci:  $T(n) \propto \log n$ 
  * Factorial:  $T(n) \propto n (\log n)^2$ 
  * Primes: Probabilistic, highly variance-dependent. Estimate based on average density  $1/\ln(N)$ .

```

---

## Chapter 18: Project Initialization

After creating the specification, we are ready to continue in Cursor. We start by creating a new folder and place the spec.md file in the correct location using the following commands in a bash shell.

```
$ mkdir ~/Development/py_mega_calc
$ mkdir ~/Development/py_mega_calc/docs
$ mv spec.md ~/Development/py_mega_calc/docs/
```

Also put the .cursorignore file in place.

Now open Cursor in this py\_mega\_calc folder, open Composer (Cmd+I), and type the following prompt:

> “Read @spec.md and create the directory structure and placeholder files.”

After this create a git repository for this project with the following prompt:

> “Create a git repository for this project and create the first commit.”

This will give us a “restore point”.

---

## Chapter 19: Preparing a Cursor Rule for Test-driven Development

In order to make sure the AI agent follows a strict test-driven development approach, we ask the AI agent to generate a cursor rule that will always be applied. Use a detailed prompt like this:

“Create a new file at .cursor/rules/tdd.mdc

This file should define a Strict Test-Driven Development (TDD) workflow that applies to every single code generation task in this project.

The rule must enforce the following behavior:

1. Enforcement: You (the AI) are forbidden from writing implementation code (logic) until a failing test case has been written and confirmed.
2. The Cycle: You must strictly follow the Red-Green-Refactor cycle:

- RED: Write a comprehensive unit test using pytest that fails because the feature doesn't exist yet.
  - GREEN: Write the minimum amount of code required to make the test pass.
  - REFACTOR: Optimize the code (clean up, improve performance, DRY) while ensuring the test still passes.
3. Tooling:
- Use pytest for running tests.
  - Use unittest.mock for isolating dependencies.
  - Use conftest.py for shared fixtures.
4. Structure: Test files must mirror the source directory structure (e.g., src/calculators/fib.py -> tests/calculators/test\_fib.py).
5. Documentation: After a test passes, you must update the docs/IMPLEMENTATION\_STATUS.md file to reflect the progress.
6. Edge Cases: Tests must explicitly cover boundary conditions (e.g., negative inputs, 0, max memory limits).

Write the content of this rule in valid Markdown format suitable for Cursor.”

The output will be something like this:

```
---
alwaysApply: true
---

# Strict Test-Driven Development (TDD) Workflow

## Overview
This rule enforces a **strict Test-Driven Development (TDD)** workflow for every single code generation task in the `py_mega_calc` project. No exceptions.

## 1. Enforcement Rule
**CRITICAL:** You (the AI) are **FORBIDDEN** from writing implementation code (logic) until a failing test case has been written and confirmed.

### What This Means:
- ✗ **DO NOT** write `def calculate_by_index(...)` until there's a test that calls it and fails
- ✗ **DO NOT** write algorithm logic until tests define the expected behavior
- ✓ **DO** write the test first, run it, confirm it fails for the right reason
- ✓ **DO** then write the minimum implementation to make it pass

## 2. The Red-Green-Refactor Cycle
You must strictly follow this cycle for **every feature**:

### RED Phase: Write Failing Test
1. Write a comprehensive unit test using `pytest` that tests the desired functionality
2. The test **MUST fail** because the feature doesn't exist yet
3. Run the test: `pytest tests/path/to/test_file.py -v`
4. Confirm the test fails with a clear, expected error (e.g., `AttributeError`, `NotImplementedError`)
5. **Only proceed to GREEN phase after confirming the test fails**

### GREEN Phase: Make Test Pass
1. Write the **minimum amount of code** required to make the test pass
2. No premature optimization
3. No extra features
4. Just enough to satisfy the test
```

```

5. Run the test again: `pytest tests/path/to/test_file.py -v`
6. Confirm the test passes

### REFACTOR Phase: Optimize
1. Once the test passes, optimize the code:
  - Clean up code style (PEP 8)
  - Improve performance
  - Apply DRY principles
  - Add type hints
  - Improve docstrings
2. **After each refactoring step, run the test again**
3. Ensure the test still passes
4. If the test breaks, revert and try a different refactoring approach

## 3. Tooling Requirements

### Testing Framework
- **Primary:** `pytest` for all tests
- Run tests with: `pytest tests/ -v` or `pytest tests/path/to/test_file.py::test_function_name -v`
- Use pytest fixtures for setup/teardown

### Mocking
- Use `unittest.mock` (or `pytest-mock`) for isolating dependencies
- Mock external resources (file I/O, system calls, etc.)
- Mock time-dependent operations for consistent testing

### Shared Fixtures
- Use `tests/conftest.py` for shared fixtures
- Define reusable test data and setup logic there
- Import fixtures in test files as needed

## 4. Test File Structure

Test files **MUST** mirror the source directory structure:

```
src/calculators/fibonacci.py    →  tests/test_fibonacci.py
src/calculators/factorial.py   →  tests/test_factorial.py
src/calculators/primes.py       →  tests/test_primes.py
src/core/resource_manager.py   →  tests/test_resources.py
src/core/estimator.py          →  tests/test_estimator.py
```

### Naming Conventions
- Test files: `test_<module_name>.py`
- Test functions: `test_<function_name>_<scenario>()`
- Test classes: `Test<ClassName>`

### Example Structure
```python
# tests/test_fibonacci.py
import pytest
from src.calculators.fibonacci import FibonacciCalculator

class TestFibonacciCalculator:
```

```

```

def test_calculate_by_index_zero(self):
    """Test that index 0 returns the first Fibonacci number."""
    calc = FibonacciCalculator()
    result = calc.calculate_by_index(0)
    assert result == 0

def test_calculate_by_index_large(self):
    """Test calculation for large indices."""
    # ... test implementation
```
## 5. Documentation Updates

**After a test passes**, you **MUST** update `docs/IMPLEMENTATION_STATUS.md`:

1. Mark the corresponding task as complete: `[x]`
2. Mark tests as passing: `[x]`
3. Add notes about implementation details or decisions
4. Update the commit message to reference the status update

### Example Update
```
| **Fib** | Fast Doubling Algo | [x] | [x] | Implemented with O(log n) complexity |
```

## 6. Edge Cases and Boundary Conditions

Tests **MUST** explicitly cover:

### Input Validation
- Negative numbers → Should raise `InputError`
- Zero → Define expected behavior
- Non-integers → Should raise `InputError`
- Very large numbers → Test resource limits

### Resource Limits
- Memory limit (24GB) → Test `ResourceExhaustedError`
- Time limit (5 minutes) → Test `TimeoutError`
- Large digit outputs (>10,000 digits) → Test estimation logic

### Boundary Conditions
- Minimum valid input (e.g., index 0, 1 digit)
- Maximum practical input (within resource constraints)
- Edge cases specific to each calculator:
  - **Fibonacci:** F(0), F(1), large n
  - **Factorial:** 0!, 1!, very large n
  - **Primes:** First prime, large primes, prime gaps

### Example Edge Case Tests
```
def test_calculate_by_index_negative_raises_error(self):
    """Test that negative index raises InputError."""
    calc = FibonacciCalculator()
    with pytest.raises(InputError):
        calc.calculate_by_index(-1)
```

```

```

def test_calculate_exceeds_memory_limit(self):
    """Test that exceeding 24GB memory raises ResourceExhaustedError."""
    # ... test implementation
```

```

## 7. Test Quality Standards

### Coverage Requirements

- **100% code coverage** for logic (as per spec)
- Use `pytest-cov` to verify: `pytest --cov=src --cov-report=term-missing`

### Test Assertions

- Use descriptive assertion messages
- Test one thing per test function
- Use `pytest.raises()` for exception testing
- Use `pytest.approx()` for floating-point comparisons (if needed, though we use integers)

### Gold Standard Values

- Include verified values from OEIS (Online Encyclopedia of Integer Sequences)
- Test known large number calculations
- Verify against authoritative sources

## ## 8. Workflow Checklist

Before writing any implementation code, verify:

- [ ] Test file exists in correct location
- [ ] Test function is written and comprehensive
- [ ] Test is run and **fails** (RED phase confirmed)
- [ ] Error message is clear and expected
- [ ] Only then: Write minimum implementation code
- [ ] Run test again and confirm it **passes** (GREEN phase)
- [ ] Refactor if needed, ensuring test still passes
- [ ] Update `docs/IMPLEMENTATION\_STATUS.md`
- [ ] Run full test suite: `pytest tests/ -v`
- [ ] Check coverage: `pytest --cov=src --cov-report=term-missing`

## ## 9. Violations and Corrections

If you catch yourself writing implementation before tests:

1. **STOP immediately**
2. Delete the implementation code
3. Write the test first
4. Confirm it fails
5. Then write the implementation

## ## 10. Example Complete TDD Cycle

```

### Step 1: RED – Write Failing Test
```python
# tests/test_fibonacci.py
def test_calculate_by_index_zero():
    calc = FibonacciCalculator()

```

```

    assert calc.calculate_by_index(0) == 0
```
Run: `pytest tests/test_fibonacci.py::test_calculate_by_index_zero -v`
**Expected:** `AttributeError` or `NotImplementedError` ✓

### Step 2: GREEN – Make Test Pass
```python
# src/calculators/fibonacci.py
class FibonacciCalculator(Calculator):
    def calculate_by_index(self, n: int) -> int:
        return 0 # Minimum to pass
```
Run: `pytest tests/test_fibonacci.py::test_calculate_by_index_zero -v`
**Expected:** Test passes ✓

### Step 3: REFACTOR – Optimize
```python
# src/calculators/fibonacci.py
class FibonacciCalculator(Calculator):
    def calculate_by_index(self, n: int) -> int:
        """Calculate nth Fibonacci number using Fast Doubling Method."""
        if n == 0:
            return 0
        # ... full implementation
```
Run: `pytest tests/test_fibonacci.py -v`
**Expected:** All tests still pass ✓
---
```

\*\*Remember:\*\* TDD is not optional. Every feature must follow this cycle.

## Chapter 20: Preparing a Cursor Rule for Python Development

Create the following file at .cursor/rules/python-general.mdc (which you can copy from: <https://raw.githubusercontent.com/rvugts/cursor-resources/refs/heads/main/cursor-rules/python/python-general.mdc>)

```

---  

glob: *.py  

alwaysApply: false  

---  

# Senior Python Developer  

You are an expert Senior Python Developer specializing in clean, maintainable, and efficient code with a  

strong focus on **Spec-Driven Development (SSD)** and **Test-Driven Development (TDD)**.  

## 0. Methodology: Test Driven Development (TDD)  

**Mandatory Workflow:**  

1. **Red:** Write a failing test for the desired functionality using `pytest` and before writing  

application code.  

2. **Green:** Write the minimum amount of code required to pass the test.  

3. **Refactor:** Improve the code quality while ensuring tests remain passing.
```

```

*   **Structure:** Mirror the application file structure within the `tests/` directory.
*   **Isolation:** Use `unittest.mock` or `pytest-mock` to isolate external dependencies.
*   **Fixtures:** Use `conftest.py` for shared resources (DB sessions, API clients).

### Pytest Fixtures Best Practices

*   **Fixture Scope:** Choose appropriate scope (`function`, `class`, `module`, `package`, `session`) based on resource cost and test isolation needs.
    *   `function` (default): Created fresh for each test - best for isolation.
    *   `class`: Shared across all tests in a class.
    *   `module`: Shared across all tests in a module.
    *   `session`: Created once per test session - use for expensive setup (DB connections, API clients).
*   **Fixture Organization:** Place shared fixtures in `conftest.py` at the appropriate directory level. Use `conftest.py` in test root for project-wide fixtures. Use `conftest.py` in subdirectories for feature-specific fixtures.
*   **Fixture Dependencies:** Use fixture parameters to create fixture chains and dependencies.
    *   *Example:* `def test_user(db_session, sample_user):` where `sample_user` depends on `db_session`.
*   **Parametrized Fixtures:** Use `@pytest.fixture(params=[...])` for testing multiple scenarios with the same test code.
*   **Fixture Cleanup:** Use `yield` for setup/teardown in fixtures. Code after `yield` runs as teardown.
    *   *Example:* `yield db_session; db_session.rollback()` ensures cleanup even if test fails.
*   **Autouse Fixtures:** Use `autouse=True` sparingly - only for fixtures that must run for every test (e.g., logging setup, environment variables).
*   **Fixture Factories:** For dynamic fixture creation, return a factory function instead of the object itself.
    *   *Example:* `@pytest.fixture; def user_factory(): return lambda **kwargs: User(**kwargs)`
*   **Database Fixtures:** Always use transactions or test databases. Never use production databases in tests.
    *   Use `pytest.fixture(scope="function")` with transaction rollback for database tests.
    *   Consider using `pytest-postgresql` or `pytest-mysql` for database testing.
*   **API Client Fixtures:** Mock external APIs in tests. Use `responses` library or `unittest.mock.patch` for HTTP requests.
*   **Performance:** Use `session`-scoped fixtures for expensive resources (DB connections, file I/O setup) that can be safely shared.

## 1. Spec-Driven Development (SSD)

*   **Mandatory:** If a spec.md file exist in the root directory, it must be followed exactly. If the spec is incorrect or has flaws, you must immediately report it to the project manager.

## 2. Type Hints

*   **Mandatory:** Use Python type hints for all functions, methods, and variables.
*   **Complex Types:** Use `typing` module for complex types (e.g., `Dict[str, List[int]]`).
*   **Generics:** Use `TypeVar` and `Generic` for generic classes.
*   **Optional Types:** Use `Optional[T]` or `T | None` (Python 3.10+) for nullable values.
*   **Union Types:** Use `Union[A, B]` or `A | B` (Python 3.10+) for multiple possible types.
*   **Protocols:** Use `Protocol` for structural subtyping (duck typing) instead of ABCs when appropriate.
*   **Type Aliases:** Use `TypeAlias` for complex type definitions to improve readability.
*   **Forward References:** Use string literals for forward references: `""` when types aren't yet defined.

## 3. Code Structure & Constraints

*   **Line Length:** Limit lines to **100 characters** where possible.
*   **Whitespace:** **No** trailing whitespaces. File must end with a single newline.
*   **Module Length:** **Strict limit of 1000 lines**.

```

```

    * Action:* If a module exceeds this, propose splitting it into logical sub-modules (e.g., `user_service.py` -> `services/user/auth.py`, `services/user/profile.py`).
* Function Length:** Soft limit of 25 lines** (excluding docstrings/comments).
    * Strategy:* Break complex logic into private helper functions (prefixed with `_`).
    * Exception:* Allowed only if splitting significantly reduces readability (must be justified).
* Early Exits:** Use Guard Clauses** immediately.
    * Bad:* Nested `if` blocks.
    * Good:* `if not condition: return`
* DRY (Don't Repeat Yourself):**
    * Use Decorators** for logging, timing, and repetitive validation.
    * Extract shared logic into utility functions or base classes.
* Comprehensions:** Prefer list/dict/set comprehensions over loops for transformations.
    * Good:* `[x**2 for x in range(10) if x % 2 == 0]`
    * Avoid:* Nested comprehensions beyond 2 levels for readability.
* Generators:** Use generators for memory-efficient iteration over large datasets.
    * Use `yield` for lazy evaluation. Prefer generator expressions `(x for x in iterable)` over list comprehensions when the full list isn't needed.
* Context Managers:** Always use `with` statements for resource management (files, database connections, locks).
    * Implement `__enter__` and `__exit__` for custom context managers, or use `contextlib.contextmanager` decorator.
        * Example:* `with open('file.txt') as f: process(f)` ensures file is closed.
* Exception Handling:**
    * Catch specific exceptions, not bare `except:` clauses.
    * Use `except ExceptionType as e:` and handle appropriately.
    * Let exceptions propagate when you can't meaningfully handle them.
    * Use `finally` blocks for cleanup code that must run regardless of exceptions.

### Refactoring Techniques
* Extract Function:**
    * Break down long functions into smaller, focused functions.
    * Each function should have a single responsibility.
* Extract Variable:**
    * Replace complex expressions with descriptive variable names.
    * Improves readability and debugging.
* Simplify Conditionals:**
    * Use early returns to reduce nesting.
    * Extract complex conditions into named functions.
    * Use guard clauses.
* Naming:**
    * Use descriptive names that explain purpose.
    * Follow Python naming conventions.

## 4. Imports & Dependencies
* Location:** All imports must be at the very top** of the file.
* Prohibited:** Never place imports inside functions or classes. Never use `from module import *`.
* Sorting (PEP 8/isort):**
    1. Standard Library
    2. Third-Party (FastAPI, Pydantic, SQLAlchemy)
    3. Local Application
* Cleanup:** Remove unused imports immediately.
* Absolute vs Relative:** Prefer absolute imports. Use relative imports (`from .module import`) only within packages.
* Conditional Imports:** Only use conditional imports when necessary (e.g., optional dependencies).
Document why.

```

```

*  **Circular Imports:** Avoid circular imports. If necessary, use local imports inside functions or
restructure code.

## 5. Naming & Style
*  **Clarity:** Names must be descriptive and unambiguous.
*  **Constants:** `UPPER_CASE_WITH_UNDERSCORES` (Defined at module level).
*  **Class Names:** `PascalCase`.
*  **Functions/Variables:** `snake_case`.
*  **Booleans:** Use auxiliary verbs (e.g., `is_active`, `has_permission`).
*  **Type Hints:** **MANDATORY** for all function signatures (arguments and return types).
*  **Private Attributes:** Use single leading underscore `_private` for internal use (convention, not
enforced).
*  **Name Mangling:** Use double leading underscore `__private` only when you need to avoid name clashes in
inheritance.
*  **Special Methods:** Use dunder methods (`__str__`, `__repr__`, `__eq__`) appropriately for class
behavior.
*  **Avoid:** Single-letter variables except for loop counters (`i`, `j`, `k`) or mathematical variables
(`x`, `y`, `z`).
*  **Pluralization:** Use plural for collections (`users`, `items`), singular for single items (`user`,
`item`).

## 6. Documentation (Sphinx/reStructuredText)
*  **Format:** Use reStructuredText (reST) for all docstrings.
*  **Module Docstring:**
    * Must explain the purpose of the file.
    * **CRITICAL REQUIREMENT:** Must include a `Dependencies:` section listing major libraries used.
*Exception: __init__.py files
    **Class/Function Docstrings:** Mandatory for every class and public function. Explain purpose,
arguments, and return values clearly. Skip the type of arguments. For non-public methods explain the
purpose.

## 7. Security & Best Practices
*  **Code Injection:** NO usage of `eval()`, `exec()`, or unsafe YAML loading.
*  **Input Validation:** Always validate and sanitize user inputs. Use libraries like `pydantic` for data
validation.
*  **SQL Injection:** Always use parameterized queries. Never concatenate user input into SQL strings.
*  **Secrets Management:** Never hardcode secrets, API keys, or passwords. Use environment variables or
secret management systems.
*  **Path Traversal:** Use `os.path.join()` or `pathlib.Path` for file paths. Validate paths to prevent
directory traversal attacks.
*  **Deserialization:** Be cautious with `pickle`. Prefer JSON or other safe serialization formats for
untrusted data.
*  **Dependencies:** Keep dependencies updated. Use tools like `safety` or `pip-audit` to check for known
vulnerabilities.
*  **HTTPS:** Always use HTTPS for external API calls. Verify SSL certificates.
*  **Password Hashing:** Use `bcrypt`, `argon2`, or `scrypt` for password hashing. Never use MD5 or SHA1
for passwords.

## 8. Development Tools
*  **Formatting:** Use `black` for code formatting.
*  **Linting:** Use `pylint` for linting.
*  **Type Checking:** Use `mypy` for type checking.
*  **Testing:** Use `pytest` for testing. Use `pytest-cov` for coverage reports.
*  **Documentation:** Use `sphinx` for generating documentation from docstrings.

```

```

## 9. Pythonic Patterns & Idioms
* **EAFP (Easier to Ask for Forgiveness than Permission):** Prefer `try/except` over checking conditions.
  * *Good:* `try: value = my_dict[key]; except KeyError: handle_missing()`
  * *Avoid:* `if key in my_dict: value = my_dict[key]` (when exception handling is appropriate).
* **Truthiness:** Leverage Python's truthiness. Use `if items:` instead of `if len(items) > 0:`.
* **Unpacking:** Use tuple unpacking and extended unpacking: `a, b = values`, `first, *rest = items`.
* **Enumerate:** Use `enumerate()` instead of `range(len())` when you need both index and value.
* **Zip:** Use `zip()` to iterate over multiple sequences simultaneously.
* **Default Arguments:** Be careful with mutable default arguments. Use `None` and assign inside function:
`def func(items=None): items = items or []`.

* **Property Decorator:** Use `@property` for computed attributes. Use `@property.setter` for validation.
* **Dataclasses:** Use `@dataclass` for simple data containers instead of writing `__init__`, `__repr__`, etc. manually.

## 10. Performance Considerations
* **Built-in Functions:** Prefer built-in functions (`sum()`, `max()`, `min()`, `any()`, `all()`) over manual loops – they're implemented in C.
* **String Concatenation:** Use `''.join()` for concatenating multiple strings, not `+=` in loops.
* **List vs Tuple:** Use tuples for immutable sequences. Use lists for mutable sequences. Tuples are slightly faster and use less memory.
* **Set Lookups:** Use sets for membership testing (`x in set`) – O(1) vs O(n) for lists.
* **Generator Expressions:** Use generator expressions for memory efficiency when you don't need the full list.
* **Caching:** Use `functools.lru_cache` for expensive function calls with repeated inputs.
* **Async/Await:** Use `async/await` for I/O-bound operations. Use `asyncio` for concurrent I/O operations.

## 11. Error Handling & Logging
* **Specific Exceptions:** Catch the most specific exception possible. Don't catch `Exception` unless necessary.
* **Exception Chaining:** Use `raise ... from ...` to preserve exception context when re-raising exceptions.
* **Custom Exceptions:** Create custom exception classes for your domain. Inherit from appropriate base exceptions.
* **Structured Logging:** Use structured logging (JSON format) for production applications to enable better log analysis.
* **Error Messages:** Provide clear, actionable error messages. Include context about what failed and why.

```

## Chapter 21: Create Atomic Tasks

Now it is time to break down the project into a list of atomic tasks. Switch the Composer into “Plan” mode and run the following prompt:

Read [spec.md](#) and break the project down into a list of atomic tasks small enough to be executed in a single prompt. Update the [IMPLEMENTATION\\_STATUS.md](#) file with these tasks

Plan
Auto

Review the plan and if all is good, save the plan to the workspace and commit the new files to git.

Now we are ready to start vibe coding – all guard rails are in place!

---

## Chapter 22: Code Generation

Press the “Build” button and the IMPLEMENTATION-STATUS.md file will be updated.

### Implementation Status

#### Phase 1: Core Foundation

| Component | Sub-task                               | Status | Tests Passing | Notes  |
|-----------|--|--------|---------------|--|
| Core      | 1. Custom Exceptions                   | [ ]    | [ ]           | InputError,<br>CalculationTooLargeError,<br>PrecisionError,<br>ResourceExhaustedError,<br>TimeoutError |
|           | 2. ResourceManager - Memory Monitoring | [ ]    | [ ]           | Track RAM using psutil, raise ResourceExhaustedError if > 24GB   |
|           | 3. ResourceManager - Timeout           | [ ]    | [ ]           | Hard abort at 300 seconds, raise TimeoutError  |
|           | 4. Estimator - Micro-benchmark         | [ ]    | [ ]           | Run small calculations to measure baseline performance   |
|           | 5. Estimator - Regression Prediction   | [ ]    | [ ]           | Linear/polynomial regression for Fibonacci, Factorial, Prime   |

#### Phase 2: Fibonacci Calculator

| Component | Sub-task                           | Status | Tests Passing | Notes  |
|-----------|------------------------------------|--------|---------------|--|
| Fib       | 6. calculate_by_index (Basic)      | [ ]    | [ ]           | Fast Doubling Method, test cases: F(0)=0, F(1)=1, F(2)=1, F(10), F(20) |
|           | 7. calculate_by_index (Edge Cases) | [ ]    | [ ]           | Negative inputs (raise InputError), large values, resource limits      |
|           | 8. calculate_by_digits             | [ ]    | [ ]           | Find first Fibonacci number with at least d digits                     |
|           | 9. ResourceManager Integration     | [ ]    | [ ]           | Apply memory and timeout decorators                                    |

#### Phase 3: Factorial Calculator

| Component | Sub-task                        | Status | Tests Passing | Notes   |
|-----------|---------------------------------|--------|---------------|---|
| Fact      | 10. Benchmark Comparison        | [ ]    | [ ]           | Benchmark math.factorial vs custom Binary Split |
|           | 11. calculate_by_index          | [ ]    | [ ]           | Test cases: 0!=1, 1!=1, 5!=120, 10!             |
|           | 12. calculate_by_digits         | [ ]    | [ ]           | Find first factorial with at least d digits     |
|           | 13. ResourceManager Integration | [ ]    | [ ]           | Apply memory and timeout decorators             |

## Phase 4: Prime Calculator

| Component | Sub-task                                    | Status | Tests Passing | Notes   |
|-----------|---|--------|---------------|---|
| Prime     | 14. Segmented Sieve<br>(calculate_by_index) | [ ]    | [ ]           | Test cases: 1st=2, 2nd=3, 10th=29, 100th=541                                |
|           | 15. Miller-Rabin Primality Test             | [ ]    | [ ]           | Deterministic for $n < 3 \times 10^{24}$ , probabilistic with $k=40$ rounds |
|           | 16. Baillie-PSW Test                        | [ ]    | [ ]           | Combine with Miller-Rabin for robustness                                    |
|           | 17. calculate_by_digits                     | [ ]    | [ ]           | Start search at $10^{(d-1)}$ , find first prime with $d$ digits             |
|           | 18. ResourceManager Integration             | [ ]    | [ ]           | Apply memory and timeout decorators   |

## Phase 5: CLI Integration

| Component | Sub-task                   | Status | Tests Passing | Notes  |
|-----------|----------------------------|--------|---------------|--|
| CLI       | 19. Argument Parsing       | [ ]    | [ ]           | Type (prime/fib/fact), --index, --min-digits (mutually exclusive)              |
|           | 20. Flags                  | [ ]    | [ ]           | --benchmark, --dry-run, --strict   |
|           | 21. Output Formatting      | [ ]    | [ ]           | Console output (< 1000 chars or truncation), file output to results/, metadata |
|           | 22. Calculator Integration | [ ]    | [ ]           | Wire up CLI commands to calculator instances                                   |

## Phase 6: Integration & Polish

| Component   | Sub-task               | Status | Tests Passing | Notes  |
|-------------|------------------------|--------|---------------|--|
| Integration | 23. Main Entry Point   | [ ]    | [ ]           | Connect CLI to main execution flow             |
|             | 24. End-to-End Testing | [ ]    | [ ]           | Test full workflows: CLI → Calculator → Output |

Next prompt:

> “Start with phase 1 in the @IMPLEMENTATION\_STATUS.md. strictly adhering to the project specification @spec.md and our TDD rules @tdd.mdc”

Repeat this for all phases, making sure to commit after each phase. When finished make sure there are no deviations from the spec (this happens even when we have a “contract”).

> “Identify all deviations from the specification in @spec.md make sure to resolve all deviations using strict TDD practice as defined in @tdd.mdc”

Now run your own (manual) tests, and if no bugs are found we move to the last code generation stage: **refactoring**, using the /refactor-python command defined in chapter 12.

> “/refactor-python”

This may take a long time to run, so consider running it in stages on smaller sets of files, e.g.:

```
> “/refactor-python @src”
> “/refactor-python @test”
```

Once complete, all tests will have passed successfully and the code is now fully compliant with your coding standards.

---

# APPENDIX:

## Tip Index: 21 Principles for Professional Vibe Coding

This index summarizes the 21 practical tips that underpin the three pillars in this playbook. They are inspired by Sean Kochel's “**Supercharge Your Vibe Coding in 21 Easy-to-Apply Tips**” video and adapted here for a production-grade, enterprise context.<sup>1</sup>

Use this list to resolve every “Tip #X” reference in the chapters above and to quickly locate which pillar a habit belongs to.

### Legend:

(P1) = Pillar 1 – Architect’s Mindset — Plan Before You Prompt

(P2) = Pillar 2 – The Conversation — Guide, Don’t Just Ask

(P3) = Pillar 3 – Disciplined Workflow — Guardrails and Recovery

- **Use widely documented tech stacks (P1, P3).** Prefer mainstream, well-documented frameworks, libraries, and cloud services so the AI can rely on accurate documentation instead of guessing APIs.
- **Plan extensively before coding (P1).** Spend serious time upfront clarifying requirements, constraints, and architecture with a reasoning model before touching implementation files.
- **Be surgical with context (P2, P3).** Share only the minimal set of files and snippets needed for the current step instead of throwing the entire codebase at the AI.
- **Commit on green and create checkpoints (P3).** Commit every time tests pass and use editor checkpoints to capture stable states before major prompts or refactors.
- **Use checkpoints and history intelligently (P3).** Treat your git history and tool-specific checkpoints as a safety net you routinely fall back to when an AI-generated change goes bad.
- **Ask for multiple solutions and compare (P2).** Regularly request two or three alternative designs or implementations, then choose or hybridize the best one instead of accepting the first answer.
- **Control the scope of each prompt (P2, P3).** Phrase prompts around small, concrete tasks that can be fully completed and verified in one iteration.
- **Select the right model or mode (P2).** Use small, cheap models for boilerplate and large, reasoning-heavy models or specialized modes for architecture, refactoring, and debugging.
- **Encode your rules hierarchically (P1, P3).** Capture global tech stack and behavioral guidelines in `.cursor.rules` (the “Constitution”) so the AI sees them on every request. Offload specific patterns, testing protocols, and library usage to `.cursor/rules/*.mdc` to create a modular, intelligent context system that triggers only when needed.

---

<sup>1</sup> These principles are influenced by the practices described in Sean Kochel’s “*Supercharge Your Vibe Coding in 21 Easy-to-Apply Tips*” (YouTube) and adapted here for professional, production-grade workflows.<sup>[1]</sup>

- **Break work into atomic tasks (P1, P3).** Use the spec to derive a sequenced list of small, named subtasks; don't ask the AI to "build the whole feature" in one go.
  - **Adopt a design-first mindset (P1).** For every feature, clarify data models, interfaces, user flows and edge cases before writing any implementation code.
  - **Use a dedicated DOCUMENTATION/ spine (P1, P3).** Maintain API.md, ARCHITECTURE.md, and IMPLEMENTATION\_STATUS.md so the AI (and humans) always have a current map of the system.
  - **Use the AI as an external brain (P1, P2).** Do high-level reasoning, trade-off analysis, and brainstorming in a separate "thinking" environment before importing the results into your repo.
  - **Keep an implementation log (IMPLEMENTATION\_STATUS.md) (P3).** At the end of each session, have the AI update what changed, what is broken, and what is next, so context survives across days and developers.
  - **Treat the AI like a junior engineer (P2).** Give clear objectives, relevant context, and constraints; review its work; and ask it to explain anything you don't understand.
  - **Use the "Nuclear Option" when stuck (P3).** If the AI is looping or degrading the code, stop, revert to the last green commit, clear the chat, and restart with a new prompt and/or model.
  - **Understand tests and code before you accept them (P1, P2).** Never commit or merge generated tests or logic you cannot explain in your own words.
  - **Aim for a minimal, testable MVP first (P1, P3).** Implement and stabilize the smallest slice that delivers value, then iterate, instead of chasing a giant "all-in-one" solution.
  - **Pre-load project-specific rules and context (P1, P3).** Use rules, documentation and, where appropriate, memory features so you don't have to repeat invariant decisions in every prompt.
  - **Run dedicated security and quality passes (P3).** Use specialized "Security Auditor" and "Refactor" modes/agents to review code against vulnerabilities and maintainability standards.
  - **Start fresh conversations frequently (P2, P3).** Open a new chat whenever you finish a feature, fix a bug, or feel the context getting noisy, to keep
-

## Cheat Sheet

# THE VIBE CODING CHEAT SHEET

## THE 3 PILLARS (The Mindset)

### 1. Architect's Mindset

Rule: Plan Before You Prompt.

Action: Use "External Brains" (Web AI) to reason. Don't code until you have a plan.

### 2. The Conversation

Rule: Guide, Don't Just Ask.

Action: Treat AI as a Junior Dev. Be surgical with context. Only attach the files needed.

### 3. Disciplined Workflow

Rule: Guardrails & Recovery.

Action: Commit on Green. If the AI spirals, stop and reset immediately.

## THE GOLDEN TRINITY (The Workflow)

### Phase 1: SDD (Spec-Driven)

Input: "External Brain" Plan.

Artifact: spec.md (The Contract)

Content: Goal, Constraints, Data Model, Edge Cases

↓

### Phase 2: TDD (Test-Driven)

Action: Generate Tests before Logic.

Validation: Run Tests -> MUST FAIL (Red).

Why: "Red" confirms the guardrails are active.

↓

### Phase 3: Vibe Coding (Execution)

Action: Agent Mode (Composer).

Prompt: "@spec.md @test\_file.py

Implement minimum logic to make tests PASS."

Loop: Auto-Run Terminal enabled [Write] → [Test] → [Fix]

(Write → Test → Fix → Success)

[Success]

## PRO TIPS (Best Practices)

### 1. Atomic Tasks

Break spec into tasks small enough for 1-3 prompts.

### 2. Fresh Start

Feature done? Bug fixed?  
Close chat. Start fresh to clear context.

### 3. Surgical Context

Never dump the repo.  
Select only necessary files.

### 4. The Nuclear Option

AI Stuck? Stop ->  
'git reset --hard' ->  
Clear Chat -> Re-prompt.

### 5. Commit on Green

Tests pass? Commit instantly. Save the checkpoint.

### 6. Refactor

Never accept first draft.  
Always prompt: "Refactor for readability."

## ADVANCED SYSTEMS (Scale & Config)

### Meta Prompting (The Analyst):

Concept: The prompt that writes prompts.

Step 1: /create-prompt -> AI interviews YOU about requirements.

Step 2: Generates `prompts/task\_definition.md`

Step 3: /run-prompt @task\_definition.md`

(Execute without ambiguity).

The Constitution: .cursorrules  
(Global tech stack & behavioral rules).

Specialist Rules: .cursor/rules/\*.mdc  
(e.g., `tdd.mdc`, `python.mdc`).

The Living Log: docs/IMPLEMENTATION\_STATUS.md  
(Update after every session).

Slash Commands: Custom Agents  
(e.g., `/refactor`, `/security`).

MCP Servers: "Sequential Thinking" for complex logic; "Docs" for external APIs.

## The Toolbox

Stop reinventing the wheel. Copy the templates from the Cursor Resources GitHub repository [5] into your project to start Vibe Coding immediately with the right setup.

- The .cursorignore file (place it in the root of your project)
- The global .cursorrules file (place it in the root of your project)
- The specific cursor rules appropriate for your project (place these in the .cursor/rules/ folder in the root of your project.)
- The cursor slash commands (place these in the .cursor/commands/ folder in your home directory)

## The spec.md Template

Use this for every new feature.

# Feature Specification: [Feature Name]

## 1. Summary

[One sentence description of what we are building]

## 2. User Story

As a [User Role], I want to [Action], so that [Benefit].

## 3. Technical Constraints

- \*\*File Location:\*\* `src/features/[name]/`
- \*\*Dependencies:\*\* Use existing `utils/date-formatter`. Do not install new libraries without permission.



- **Performance:** DB queries must be indexed.

#### ## 4. Interface Definition

```
// Define the Types/Interfaces here BEFORE writing logic interface UserProfile { id: string; lastLogin: Date; }
```

#### ## 5. Edge Cases & Failure States

- [ ] Network failure
- [ ] Empty data returned
- [ ] Validation errors

## The “Nuclear Option” Prompt

*Use this when the AI gets stuck in a loop or keeps breaking things.*

“STOP. The current approach is not working and we are going in circles.

- **Discard** the last x attempts.
- **Re-read** @spec.md and the original error message.
- **Reflect:** List 3 possible reasons why this is failing.
- **Propose:** A completely different architectural approach to solve this.
- **Execute:** Implement the simplest possible version of that new approach.”

## Documentation Skeleton

*Create this folder structure to give your AI a persistent memory.*

```
/docs
  └── API.md           # Endpoints, Request/Response shapes
  └── ARCHITECTURE.md   # High-level diagrams, data flow, system boundaries
  └── IMPLEMENTATION_STATUS.md # The "Living Log"
```

#### IMPLEMENTATION\_STATUS.md Content:

# Project Status

#### ## Completed Features

- [x] User Auth (JWT)
- [x] Database Connection

#### ## Current Focus

- [ ] Rate Limiter Middleware (In Progress)

#### ## Known Technical Debt

- The logging function is synchronous, needs to be async.

---

## END OF PLAYBOOK

Ready to transform your organization? Contact **AI-chitect** for training, setup, and strategic consulting.  
Visit: [www.aichitect.eu](http://www.aichitect.eu)

## Sources:

[1]: Supercharge Your Vibe Coding in 21 Easy-to-Apply Tips  
[https://youtu.be/RmcvVONkrWE?si=A66WKbyk\\_53EhRKW](https://youtu.be/RmcvVONkrWE?si=A66WKbyk_53EhRKW)

[2] Vibe Coding is Dangerous. Fix it with the SDD + TDD Workflow  
<https://www.aichitect.eu/en/videos/vibe-coding-is-dangerous.html>

[3] Mastering Cursor and the Art of “Vibe Coding” - AI-chitect  
<https://aichitect.eu/blog/supercharge-cursor-vibe-coding.html>

[4] AI-Assisted Development Done Right: Architecture First, Speed Second  
<https://www.aichitect.eu/blog/ai-assisted-development-done-right.html>

[5] Cursor Resources GitHub repository  
<https://github.com/rvugts/cursor-resources/>