# Software Engineering Applied to AI Software Development

*Castor canadensis*
*(The North American Beaver)*

**Lee Harrington**

# Software Engineering with AI

*Claude Code Edition*

Lee Harrington

<!-- Generated from the canonical Book 1 source using the `claude-code` harness profile. -->

# Introduction

You have described what you want. The AI built something. It was close — but not quite right. So you described it again, differently. The AI built something else. Closer, maybe. You kept going. Three prompts in, you realized the foundation was wrong, and you were either going to fight it or start over.

This is not a failure of the AI. The AI did exactly what a capable colleague does when handed a vague request: it made reasonable assumptions and produced something that compiles, runs, and looks reasonable. The problem is that the assumptions were not yours. And because AI builds so fast, those assumptions became structural before you had a chance to question them.

The faster the execution, the more expensive the unclear thinking. This is the central challenge of AI-assisted development, and it does not get easier as the tools improve. It gets harder. Every speed increase in AI output raises the cost of not knowing what you want before you start.

## The Discipline Problem

Software development has always required clarity of thought. Before AI, the writing of code was slow enough that bad assumptions surfaced during implementation — you ran into them, you adjusted, the friction kept you honest. AI removes that friction. The cost of starting is nearly zero. The cost of being wrong does not change.

What changes is when you pay it.

In a pre-AI workflow, unclear requirements showed up as slow progress, lots of dead ends, and gradual course-correction. In an AI-assisted workflow, unclear requirements show up as a codebase full of someone else's decisions — the AI's reasonable interpretations of

what you probably meant — that you now have to maintain and extend.

The software engineering disciplines that address this are not new. Plan before you build. Define requirements precisely. Test against what you specified, not against what was implemented. Review code as if you have to own it forever. Iterate in controlled increments. Manage scope explicitly. Document the decisions that are not in the code.

These disciplines predate AI coding tools by decades. They are not more important now than they were before. But the consequences of skipping them have changed. The penalty for skipping planning used to be a slow start. The penalty now is thirty seconds of confident execution in the wrong direction.

**Software engineering discipline does not disappear in the AI era. It moves up a level.** The AI writes the code. You are responsible for everything the code is supposed to do — and that responsibility requires the same clarity, precision, and rigor it always did. More, in fact, because the AI will not push back on a bad idea. It will implement it.

## This Book

This book teaches you to apply seven foundational SE disciplines in an AI-assisted workflow, then shows you how to make those disciplines systematic through a methodology called AgentFlow.

**Part 1** covers the disciplines through hands-on practice. You will plan before you prompt. You will write verifiable requirements before you build. You will design test cases from your requirements — not from the implementation. You will review AI-generated code as if you are approving a pull request. You will iterate in explicit, verified increments. You will manage scope by naming what you are not building. You will maintain a decision log so that every new session — and every new agent — starts with context instead of guesswork.

**Part 2** shows you how these disciplines scale. AgentFlow is a documentation-driven methodology for human-AI collaboration. It formalizes the practices from Part 1 into a system that works across

sessions, tools, and — when you are ready — parallel AI agents. You will learn to build skills that encode your working style, context files that survive session resets, sprint plans that coordinate work across multiple AI runs, and autonomy calibration so you know how much to delegate and when to stay in the loop.

The two parts are designed to be read in order. Part 1 establishes the practice. Part 2 builds the system. You cannot build a reliable system around practices you have not internalized.

## The Project Thread

Every chapter in this book is grounded in a single running project: a personal task manager CLI written in Python.

You will not start with a finished product. You will build it from scratch in Chapter 1 and add to it in every chapter that follows. By the time you reach Chapter 14, the task manager will have grown from four commands and zero tests to more than ten commands and twenty tests — and you will have applied every discipline in this book to real, running code.

Here is what you start with at the end of Chapter 1:

- `python tasks.py add "Buy groceries"` — adds a task
- `python tasks.py list` — shows all tasks
- `python tasks.py done 1` — marks task 1 complete
- `python tasks.py delete 1` — removes task 1

That is it. Simple enough that the structure is obvious. Complex enough that all seven disciplines have real work to do on it.

Each chapter adds a feature or applies a discipline to the existing code. The project thread is continuous — what you build in Chapter 3 is what you test in Chapter 4 and review in Chapter 5. By Part 2, the same codebase is the vehicle for multi-session AgentFlow sprints.

If you want to substitute your own project, it needs to be a CLI tool you can build incrementally, with testable behavior and a growing feature set. Everything in the book applies regardless of what you are building.

## How Part 1 Works

Each chapter in Part 1 introduces one discipline through a failure path and a success path.

The failure path shows what happens when the discipline is absent. You will do it — not read about it. You will make the vague prompt, watch the AI make assumptions, and feel the consequence. These are not constructed cautionary tales. They are the actual workflow most developers default to. Experiencing the failure is what makes the discipline legible.

The success path shows the same scenario with the discipline applied. Same feature, same tool, completely different outcome — because you changed what you did before opening the tool.

Each chapter ends with a debrief. The debrief names the discipline you just practiced, explains why it worked, and connects forward to the next chapter. Part 1 chapters build on each other. The requirements you write in Chapter 2 are the source of the tests you design in Chapter 3. The tests you write in Chapter 3 catch the scope drift you will see in Chapter 6. Each discipline compounds.

## How Part 2 Works

Part 2 assumes you can apply the Part 1 disciplines. It does not re-teach them. It shows you how to make them systematic.

AgentFlow introduces five core components: a skill system that encodes your working style into reusable prompts, context files that preserve project state across sessions, a sprint cadence for planning and executing work in coordinated increments, a multi-agent coordination model for parallel work, and an autonomy calibration framework for deciding how

much to delegate.

Each Part 2 chapter introduces one component through the same task manager project thread. By Chapter 14, you will run a complete AgentFlow sprint from scratch — using every component you learned — on a feature you choose from the backlog you built across Part 1.

Part 2 is where the practice becomes a system. If Part 1 is the discipline, Part 2 is the infrastructure that makes the discipline automatic.

## Before You Begin

**What you need:**

• **Python 3.10 or later.** The task manager is pure Python with no external libraries. Any modern Python installation works.

• **Claude Code installed.** Use Anthropic's current [Claude Code quickstart](https://code.claude.com/docs/en/quickstart) to install it on your machine, then run `claude` inside your project directory. Claude Code is terminal-first, so the exercises in this edition assume you are comfortable moving between your shell and editor.

• **A terminal you are comfortable with.** You will run Python commands from the terminal throughout. macOS Terminal, Windows PowerShell, or any Linux shell all work.

• **2–10 years of development experience.** This book is not for beginners. It assumes you know what a function is, you have written tests before, and you have shipped code in a production environment at least once. The disciplines here are not novel theory — they are practices you probably know exist and have not applied rigorously because the workflow was slow enough to get away with it. It is no longer slow enough.

**What you do not need:**

• Prior experience with AI coding tools. This edition starts from scratch with Claude Code as the harness.

• Experience with AgentFlow, AgentFlow-adjacent tools, or any specific AI methodology. AgentFlow is introduced in Part 2 and built up from first principles.

• Python expertise. The task manager code is deliberately simple — under 200 lines by Part 1's end. Understanding it requires reading, not fluency.

## Let's Start

Chapter 1 opens with the failure path. You will start a fresh Claude Code session, describe what you want, and watch what happens. By the time you finish that chapter, you will understand exactly why the experience felt familiar — and what to do instead.

*The AI is fast. What it builds is only as good as what you specified. This book is about the specifying.*

# Chapter 1: Plan Before You Prompt

You have used AI coding tools. You have described what you want, watched the AI generate something, looked at it, and thought: *that's not quite right*. So you described it again, differently. The AI generated something else. Closer, maybe. Or maybe not.

This is the most common frustration in AI-assisted development, and it has a simple cause: you did not have a plan before you opened the tool.

This chapter is about fixing that. Not with a complicated framework. With five questions you answer before you type your first prompt.

## Why "Just Describe It" Doesn't Work

Before AI, writing code yourself was slow. That slowness was secretly useful. When you had to write every line, you had to think about every line. Bad assumptions surfaced during implementation, because you ran

into them. You adjusted. The friction of the work kept you honest.

AI removes that friction. You describe something, and thirty seconds later you have code. The speed feels like productivity. It often isn't.

Here is what actually happens: the agent makes assumptions. It has to — you did not give it enough information to do otherwise. It produces something that compiles, runs, and looks reasonable. You think it is mostly right. You move on.

Three prompts later, you realize the foundation is wrong. The AI built a task manager with SQLite persistence and a full CLI argument parser, and you just wanted a simple JSON file and four commands. Now you are either fighting the existing structure or starting over.

The AI did not fail you. It did exactly what a capable colleague would do when handed a vague request: it made reasonable assumptions and built something. The assumptions just were not yours.

**The problem is not the AI's ability to generate code. It is your failure to specify what code to generate.**

This gets worse with AI than with human developers for one specific reason: speed. A human colleague who receives a vague brief will ask clarifying questions before starting. An AI agent starts immediately. The faster the execution, the faster the wrong assumption becomes load-bearing architecture.

## The Project You'll Build

Across Part 1 of this book, you will build a personal task manager CLI. Every chapter adds something to it — a feature, a test suite, a code review, a documentation pass. By Part 1's end, you will have a real, working tool and seven chapters of SE discipline applied to it.

Here is what "done" looks like at the end of this chapter:

• `add` — adds a task by description

• `list` — shows all tasks

• `done` — marks a task complete by ID

• `delete` — removes a task by ID

Tasks are stored in a JSON file. The whole thing runs with `python tasks.py`. No database. No external services. One command to run it.

If you want to use your own project instead, that is fine — see the appendix for the criteria it needs to meet. Everything in this chapter applies regardless of what you are building.

## The Failure Path: Prompting Without a Plan

Let's start by doing it wrong. Not to be perverse — because you need to feel what goes wrong before the fix makes sense.

**Create a new project folder and start Claude Code inside it.** Make a new folder for your project, open a terminal in that folder, and run `claude`. If you have not installed Claude Code yet, use Anthropic's current [quickstart guide](https://code.claude.com/docs/en/quickstart).

**Start a fresh session.** Claude Code runs in your terminal. When you launch `claude` in the project directory, you start with a fresh session and an empty conversation history.

Now type this prompt — exactly this, nothing more:

```
Build me a task manager CLI in Python.
```

> **What the Agent Will Do:** The agent will begin working autonomously. You will see it creating files in your project directory, running commands in the terminal, and occasionally pausing to report what it has done. Do not interrupt it. Let it finish.

> **Watch For:** How many files it creates. Look at the project structure that appears in the editor. Look at what it built — open `README.md` if it created one, or the main Python file.

Take a moment to look at what you got.

The agent probably built something that works. It might have a full argument parser. It might use SQLite for persistence. It might have categories, priorities, or due dates. It might have color output. It is probably more than you wanted, in a structure you did not choose, organized in a way that reflects the agent's assumptions, not yours.

Now try to correct it. Type something like:

```
Make it simpler — just a flat JSON file, four commands: add, list, done
```

> **What the Agent Will Do:** The agent will attempt to refactor what it built. You will see it editing existing files and possibly creating or deleting others.

> **Watch For:** Whether the result is actually simpler, or whether the agent made a different set of decisions that are also not quite what you wanted. Depending on how far the first version diverged from what you want, this may produce something close, or it may produce a different set of assumptions that are also not quite right.

> **Credit Note:** Each prompt to the agent consumes from your rate limit budget. If you have been iterating — trying to correct the first version — you may see a rate limit warning after three or four exchanges. This is the cost of planning failures made visible.

After two or three correction attempts, stop. Do not fix it further.

Notice what happened. You spent multiple prompts and have something that is approximately what you wanted. You do not fully understand its structure because you did not design it. And if you keep building on this foundation in later chapters, every chapter will inherit its unexamined assumptions.

This is the failure mode. Not a crash. Not an error message. A gradual accumulation of someone else's decisions about your project.

## The Planning Brief

Before you open Claude Code for any feature or project, answer five questions in writing. Not in your head — in writing, in a file. Vague thinking survives in your head. It does not survive contact with a blank document.

**The five questions:**

**1. What does it do?**

One sentence. Core function only. No features that are "nice to have."

**2. What does it NOT do?**

This is as important as question one. Constraints eliminate the AI's assumption space. Every constraint you leave unspecified is a choice the agent will make for you.

**3. How does the user interact with it?**

List the exact commands or interface elements. If it is a CLI, list every command and what it takes as input.

**4. What does the data look like?**

How is information stored? What format? What fields? If you do not specify this, the agent will choose — and its choice may not match your mental model.

**5. What does "done" mean for this session?**

Scope the work to what you actually want to accomplish right now. Not the full vision — this session's deliverable.

Here is the planning brief for Chapter 1's task manager:

*What it does:* A CLI tool to manage a personal to-do list from the terminal.

*What it does NOT do:* No categories, no priorities, no due dates, no colors, no database, no network requests, no configuration files.

*How the user interacts with it:*

• `python tasks.py add "Buy groceries"` — adds a task, prints the assigned ID

• `python tasks.py list` — prints all tasks with their IDs and completion status

• `python tasks.py done 1` — marks task 1 as complete

• `python tasks.py delete 1` — removes task 1

*What the data looks like:* A `tasks.json` file in the project directory. An array of task objects. After adding two tasks, the file should look like this:

```
[
  {"id": 1, "description": "Buy groceries", "done": false},
  {"id": 2, "description": "Write chapter 1", "done": false}
]
```

IDs are sequential integers starting at 1.

*Done when:* All four commands work. `tasks.json` is created on first run if it does not exist. Running the commands in sequence produces the expected output.

That brief took five minutes to write. It eliminates every assumption the agent made in the failure path.

Notice what the brief actually is: a written record of your decisions, made before implementation started. In Part 2 of this book, you will see this idea formalized into a document system that persists across sessions, tools, and collaborators. For now, a text file is enough.

## The Success Path: Prompting From a Plan

Delete the project folder from the failure path — or simply create a new empty folder and start a fresh `claude` session in that directory instead. You want a clean slate with no files from the previous attempt.

Start a fresh Claude Code session. This time, your prompt will be the planning brief, prefaced with a one-sentence instruction:

```
Build a Python CLI task manager matching this specification exactly.
```

```
What it does: A CLI tool to manage a personal to-do list from the termi

What it does NOT do: No categories, no priorities, no due dates, no col
no database, no network requests, no configuration files.

Commands:
- python tasks.py add "Buy groceries" — adds a task, prints the assigne
- python tasks.py list — prints all tasks with IDs and completion statu
- python tasks.py done 1 — marks task 1 complete
- python tasks.py delete 1 — removes task 1

Storage: tasks.json in the project directory. Array of objects:
{"id": 1, "description": "Buy groceries", "done": false}
IDs are sequential integers. Create tasks.json on first run if missing.

Done when: All four commands work correctly in sequence.
```

**What the Agent Will Do:** The agent will create `tasks.py` and likely a short `README.md`. It should not create additional files, a database layer, or a complex directory structure. The implementation should be direct — a single Python file under 100 lines.

**Watch For:** The agent completing the task in one pass without asking clarifying questions. When a prompt is specific enough, the agent does not need to guess.

When the agent finishes, test it:

```
python tasks.py add "Buy groceries"
python tasks.py add "Write chapter 1"
python tasks.py list
python tasks.py done 1
python tasks.py list
python tasks.py delete 2
python tasks.py list
```

**Watch For:** Each command producing exactly the output you specified. `list` showing IDs, descriptions, and completion status. `done` marking the correct task. `delete` removing the correct task. `tasks.json` appearing in your project folder and updating with each command.

If anything is off, do not issue a vague correction. Go back to the planning brief, identify the specific thing that is wrong, and issue a

targeted correction.

Vague: *"The list command doesn't look right."*

Specific: *"The list command should print each task as* `[1] Buy groceries` *with a* `[x]` *prefix for completed tasks. Currently it prints the raw JSON object."*

One specific problem. One specific fix. This is the same discipline as the planning brief — specificity is what makes the agent useful.

> **Credit Note:** Notice how many prompts this approach took compared to the failure path. Planning before prompting is not just good practice — it is how you make your rate limit budget last.

## Debrief

Look at what you have. A working CLI tool, built in one focused Claude Code session, that does exactly what you specified — because you specified it.

The agent's job was execution. Your job was definition. When you handed the definition work to the agent in the failure path, the output reflected the agent's judgment, not yours. When you did the definition work yourself first, the output reflected your judgment.

**What you own: the plan. What the AI owns: the implementation.** Keep those roles clear and AI-assisted development becomes predictable. Blur them and you are debugging someone else's assumptions.

The task manager you just built is the foundation for the next six chapters. In Chapter 2, you will add a feature to it — but before you open Claude Code, you will define exactly what that feature is.

*Planning is not a speed bump between you and the AI. It is the thing that makes the AI fast.*

# Chapter 2: Define Requirements Before You Build

In Chapter 1, you wrote a planning brief that answered: *what are we building?* Now you need to add a feature, and that question has an answer — the task manager exists. The question changes: *what must this specific feature do?*

That is a requirement. It is not the same thing as a plan.

## A Plan and a Requirement Are Different Things

A plan tells the AI what to build. A requirement tells the AI how to verify that it built it correctly.

Without a requirement, "working" is undefined. The AI will implement something that runs and looks reasonable. You look at the result and think: *that is not quite what I meant*. But you cannot articulate why, because you never articulated what you did mean.

The mistake is not the AI's implementation. The mistake is evaluating the output against a mental model you never made explicit. You cannot test a requirement you never wrote.

**A requirement is a statement you can test as true or false.**

Non-verifiable: "Users should be able to set task priority." You cannot test this. There are a dozen ways to implement it, and any of them satisfies the statement.

Verifiable: "Running `python tasks.py add "Buy groceries" --priority high` adds a task and `list` displays it as `[high] Buy groceries`." Either it does or it does not. No interpretation required.

## The Failure Path: Adding Priority Without a Requirement

Your task manager from Chapter 1 has four commands. A reasonable next feature: let users mark tasks as high, medium, or low priority.

Open Claude Code and open your task manager project from Chapter 1. In the Claude Code session, type:

```
Add priority levels to tasks.
```

> **What the Agent Will Do:** The agent will modify `tasks.py` and update `tasks.json` handling. It will make a series of decisions you did not make: whether priority is set at creation or edited later, what the valid values are, what the default is, how priority appears in the list output, whether the list sorts by priority, and how existing tasks without priority are handled.

> **Watch For:** How the agent chose to implement the `add` command. Does it use a flag? A positional argument? A prompt? Look at the `list` output format — what does priority look like? Open `tasks.json` and look at the schema it chose.

When the agent finishes, run a few commands and examine what you got:

```
python tasks.py add "Buy groceries"
python tasks.py add "Fix critical bug"
python tasks.py list
```

The agent probably built something functional. But look carefully at the decisions embedded in that implementation:

• Did it use `--priority` as a flag, or something else?

• What is the default priority for a task added without specifying one?

• How are existing tasks (from before priority was added) handled?

• Does `list` sort by priority, or just display it?

• What happens if you pass an invalid priority value?

You did not make any of these decisions. The agent did. You will only discover the mismatches later, when the feature does not behave as you expected.

> **Credit Note:** At this point you might be tempted to issue corrections — "actually, use a flag," "sort the list by priority." Each correction is another prompt, another rate limit draw, and another opportunity for the agent to introduce new assumptions. You are debugging a mental model you never wrote down.

Stop here. Do not fix it. You are going to start over with a requirement.

## What Makes a Requirement Verifiable

A verifiable requirement has four properties:

**1. Specific input.** What exactly does the user type or provide? Not "the user sets a priority" — `python tasks.py add "description" --priority high`. Exact syntax.

**2. Specific output.** What exactly does the user see? Not "priority is displayed" — `[high] Buy groceries` as the list format. Exact format.

**3. Defined edge cases.** What happens at the boundaries? What if no priority is specified? What happens with an invalid value? What about tasks created before this feature existed?

**4. Defined scope boundary.** What is explicitly *not* part of this feature? This prevents the AI from adding related things you did not ask for.

A requirement that has all four properties is testable before the agent writes a single line. You can read it and answer: "If the agent implements exactly this, will I be satisfied?" If yes, it is a good requirement. If you are still unsure, the requirement is not specific enough yet.

## Writing the Priority Requirement

Apply the four properties to the priority feature:

*Feature:* Task priority levels

*Specific input:*

• `python tasks.py add "description" --priority high` sets priority to high

• `python tasks.py add "description" --priority medium` sets priority to medium

• `python tasks.py add "description" --priority low` sets priority to low

• `python tasks.py add "description"` (no flag) sets priority to medium by default

• Any value other than `high`, `medium`, or `low` prints an error and exits without adding the task

*Specific output:*

• `list` displays each task as: `[ID] [priority] description` — for example: `[1] [high] Buy groceries`

• Completed tasks display as: `[1] [high] [done] Buy groceries`

*Edge cases:*

• Tasks in `tasks.json` that have no `priority` field (created before this feature) display as `[medium]` without modifying the stored data

*Scope boundary:*

• `list` does NOT sort by priority — tasks appear in creation order

• No command to change a task's priority after creation

• No filtering tasks by priority

That requirement took about ten minutes to write. Every decision the agent made for you in the failure path is now made by you in writing. The agent's job is to implement this — not to interpret it.

Notice also the edge case: existing tasks must display as medium priority without rewriting their stored data. That is a backward

compatibility requirement. If you do not specify it, the agent will handle it some way. Maybe correctly. Maybe not. Now you control the outcome.

## The Success Path: Building to a Requirement

Stay in the same project folder — your `tasks.py` and `tasks.json` from Chapter 1 should still be there. Start a fresh Claude Code session. This keeps the existing code but gives the agent a clean context with no memory of the failure path session.

> **Credit Note:** Starting a new agent session resets the agent's context. This is intentional — you are giving the agent a complete requirement, so it does not need the history of the failure path session.

Your prompt is the requirement, prefaced with context. Copy this exactly — the precision is the point:

```
I have an existing Python CLI task manager in tasks.py. The current sch
tasks.json is: {"id": 1, "description": "Buy groceries", "done": false}

Add priority levels to this existing implementation with the following
specification exactly:

Feature: Task priority levels

Input:
- python tasks.py add "description" --priority high/medium/low
- Default priority when flag is omitted: medium
- Invalid priority values print an error and exit without adding the ta

Output:
- list displays: [ID] [priority] description
  Example: [1] [high] Buy groceries
- Completed tasks: [1] [high] [done] Buy groceries

Edge cases:
- Existing tasks.json entries with no priority field display as [medium
  without modifying the stored data

Out of scope:
- No sorting by priority
- No changing priority after creation
- No filtering by priority
```

**What the Agent Will Do:** The agent will read `tasks.py`, understand the existing structure, and modify it to add the `--priority` flag to the `add` command and update the `list` display. It should not restructure the file or change other commands.

**Watch For:** The agent asking clarifying questions — if your requirement is complete, it should not need to. An agent that asks no questions on a well-specified requirement is confirmation that the requirement was specific enough.

When the agent finishes, verify each property of your requirement:

```
# Test default priority
python tasks.py add "Buy groceries"
python tasks.py list
# Expected: [1] [medium] Buy groceries

# Test explicit priorities
python tasks.py add "Fix critical bug" --priority high
python tasks.py add "Update README" --priority low
python tasks.py list
# Expected: three tasks with correct priority labels, creation order

# Test invalid value
python tasks.py add "Test task" --priority urgent
# Expected: error message, task not added

# Test done display
python tasks.py done 1
python tasks.py list
# Expected: [1] [medium] [done] Buy groceries

# Test backward compatibility
# In Claude Code's editor (left panel), click tasks.json to open it.
# Remove the "priority" field from one task entry so it looks like the
# original Chapter 1 format: {"id": 1, "description": "Buy groceries",
# Save the file (Cmd+S / Ctrl+S), then:
python tasks.py list
# Expected: that entry displays as [medium] without you having added pr
```

**Watch For:** Each test matching the requirement exactly. If something does not match, you now have a specific, testable mismatch — not a vague "it seems off." Go back to the requirement, find the property that failed, and issue a targeted correction quoting the requirement and the observed behavior.

## Debrief

Look at what just happened. You made a product decision — priority is set at creation, not editable after; list is unsorted; invalid values error — and the AI implemented your decision. In the failure path, the AI made those same decisions. You just did not know what they were until you looked.

This is the shift that requirements create. Without them, you are a tester discovering someone else's design decisions after they are already built. With them, you are the product owner verifying your own decisions were implemented correctly.

**A requirement is not documentation. It is specification.** You are not writing it to record what was built. You are writing it to define what will be built — before the agent writes a line.

Every feature you add to this task manager from here will follow this pattern: write the requirement first, then prompt. Not as bureaucracy. As precision.

In Part 2, you will see how requirements connect to a larger document system that persists across sessions and lets any AI — or any collaborator — pick up exactly where you left off. The requirement you just wrote is a backlog item in everything but name. The reason AgentFlow works across tools and sessions is that the decisions live in files, not in your head or the AI's memory. You have already started doing this — you just did not have a name for it yet.

*The AI executes. You decide what done means. A requirement is how you make that decision stick.*

# Chapter 3: Test What the AI Produces

You have a task manager that runs. You have verified that the main commands work — you ran them manually after each chapter and checked the output. That is not testing.

Testing is systematic verification of every behavior you specified. There is a meaningful difference between "it worked when I tried it" and "it works for every case it is supposed to handle."

That difference matters more with AI-generated code than with code you wrote yourself. When you write code, you understand it — you made the decisions, you know the edge cases you considered and the ones you skipped. AI-generated code has no such intuition attached to it. The code exists. It does things. What those things are, precisely, requires verification.

This chapter is about building that verification systematically — and about why you should design your tests from your requirements, not from your implementation.

## The Trust Problem

Here is a question worth sitting with: how confident are you that your task manager handles every case in the requirements you wrote in Chapter 2?

Specifically:

• What happens when you pass `--priority urgent` (an invalid value)?

• What happens if `tasks.json` gets a task entry without a `priority` field?

• What happens if you run `python tasks.py done 99` when task 99 does not exist?

If you are not certain, you have not tested it — you have assumed. Assumptions about AI-generated code are exactly as reliable as assumptions about a colleague's code you have never read: sometimes fine, sometimes not, always unverified.

The common response at this point is: "I'll just ask Claude Code to write the tests." That seems efficient. It is not — for a specific reason.

## The Failure Path: Letting the AI Write Its Own Tests

Open Claude Code and your task manager project. Start a new agent session. Type:

```
Write a test suite for tasks.py using pytest.
```

> **What the Agent Will Do:** The agent will read `tasks.py`, understand its structure, and write tests based on what the code does. It will create a `tests/` directory and a `test_tasks.py` file. The tests will likely cover the main commands with typical inputs.

> **Watch For:** Which cases the agent tests. Specifically, look for whether it tests: invalid priority values, missing priority fields in existing JSON entries, commands with IDs that do not exist.

When the agent finishes, return to your shell in the same project directory. Install pytest if you have not already, then run the suite:

```
pip install pytest
pytest tests/ -v
```

> **Watch For:** All tests passing. Pytest's verbose output lists each test by name with `PASSED` or `FAILED` next to it. A fully green run ends with a summary line like `9 passed in 0.12s`. That is what you are looking for — or not looking for, in this case.

The tests probably pass. Congratulations — you have a false sense of security.

Open `test_tasks.py` and read what the agent wrote. You will likely find:

• Tests that add a task and check it was stored

• Tests that list tasks and check the output format

• Tests that mark tasks done and delete them

• Happy path coverage for the priority flag

What you will probably not find:

• A test for `--priority urgent` (invalid value)

• A test for a `tasks.json` entry with no `priority` field

• A test for `done 99` when task 99 does not exist

The agent tested what it implemented. It did not test what you required. Those are different things. The edge cases in your requirement — the ones that make the behavior predictable — are exactly the cases the agent's tests skip, because the agent derived its tests from the implementation, not the specification.

> **Credit Note:** If you followed the failure path and the agent's tests pass, resist the urge to move on. A green suite that does not cover your requirements is not a safety net — it is a false floor.

## Tests Derived From Requirements, Not Implementations

The correct approach is to write tests from the requirement before the agent writes a line of test code.

Not the implementation — the requirement. The requirement is what you specified the code should do. The tests are verification that it does those things. The implementation is irrelevant to the test design — it is what you are testing, not where you get your test cases.

This distinction matters because of how AI-generated code fails. It rarely fails at the happy path. It fails at boundaries, edge cases, and behaviors you specified but did not manually exercise. Those are the cases your requirement covers. Those are the cases your tests must cover.

The process:

• Read your requirement

• For each property of the requirement, write one or more test cases

• Give those test cases to Claude Code along with the requirement

• Let the agent implement the test code — not design it

You design the test cases. The agent writes the test code. This mirrors the same division from earlier chapters: you define what is correct, the AI executes.

## The Test Plan

Here is the test plan for the task manager, covering both Chapter 1 and Chapter 2 requirements. Write this before opening Claude Code.

**From Chapter 1 requirements:**

**From Chapter 2 requirements:**

Nine test cases. Each one maps to a specific property in your requirements. The test plan took about fifteen minutes to design.

## The Success Path: Requirement-Driven Tests

Start a fresh Claude Code session in your task manager project. Your prompt gives the agent the requirement, the test plan, and the implementation context:

```
I have a Python CLI task manager in tasks.py with this behavior:

Core commands: add, list, done, delete
Storage: tasks.json — array of {"id": int, "description": str, "done":
Priority values: high, medium, low. Default: medium.
Invalid priority: print error, exit non-zero, do not add task.
Missing priority field in tasks.json: display as medium without modifyi

Write a pytest test suite in tests/test_tasks.py that covers exactly th
nine test cases (do not add others):

1. add creates a task with correct id, description, done=false, priorit
2. list shows all tasks in creation order
```

```
3. done marks the correct task as done, leaves others unchanged
4. delete removes the correct task, leaves others
5. tasks.json is created on first run if it does not exist
6. --priority high stores and displays [high]
7. add with no --priority flag displays [medium]
8. --priority urgent prints an error and does not add a task
9. a tasks.json entry with no priority field displays as [medium]

Use pytest. Each test must be independent — use a temporary directory f
so tests do not affect each other.
```

> **What the Agent Will Do:** The agent will create `tests/test_tasks.py` with nine tests. It will use `tmp_path` to give each test a clean JSON file, avoiding test interdependencies. For CLI-level tests it may use `subprocess.run` to call `tasks.py` as a process.

> **Watch For:** The agent creating exactly nine tests, not more. If it adds extra "bonus" tests beyond your plan, note it — that is scope expansion, which is the subject of Chapter 6.

Run the suite in the terminal panel:

```
pytest tests/ -v
```

> **Watch For:** One test failing. Pytest will show `FAILED tests/test_tasks.py::test_missing_priority_defaults_to_medium` (or similar name) with a traceback beneath it. Specifically, test 9 — the missing priority field case — commonly surfaces a bug: the agent may have used `task["priority"]` (which raises `KeyError` on a missing field) instead of `task.get("priority", "medium")`. The test catches it. The implementation did not.

When you see the failure, do not guess at the fix. Open `tasks.py`, find the line that reads `priority`, and look at whether it handles a missing key. Then issue a targeted correction:

```
In tasks.py, the list command crashes with KeyError when a task in task
has no "priority" field. Fix it to default to "medium" when the field i
Do not change any other behavior.
```

> **What the Agent Will Do:** The agent will locate the specific line and change `task["priority"]` to `task.get("priority", "medium")` (or equivalent). One line change.

Run the suite again:

```
pytest tests/ -v
```

> **Watch For:** All nine tests passing. The fix was targeted because the test was specific. The test was specific because the requirement was specific.

## Debrief

That failing test found a real bug. Not a hypothetical. A behavior your code had that violated your requirement — specifically the backward compatibility requirement you wrote in Chapter 2.

The agent's self-written tests missed it because the agent wrote tests that matched the implementation it produced, and the implementation had this bug. When the tests come from the requirement, they are independent of the implementation's blind spots.

This is why you design test cases and let the agent write test code. The test case design requires knowing what should be true — which is your job, derived from the requirement. The test code implementation is mechanical — which is the agent's job.

**A test suite that passes is not proof of correctness. It is proof of coverage.** The question is: coverage of what? If the tests come from the agent's interpretation of its own code, you are testing the agent's assumptions. If the tests come from your requirements, you are testing your decisions.

You now have a task manager with a requirement, an implementation, and a test suite that enforces the requirement against the implementation. Every feature you add in subsequent chapters gets the

same treatment: requirement first, tests second (designed from the requirement), implementation third.

In Part 2, the Development Loop formalizes this sequence into an explicit structure that scales from solo sessions to teams of parallel agents. The same requirement-first, test-second discipline applies whether you are the only developer or coordinating multiple AIs. You have been following the pattern here — Part 2 gives it a name and a system.

*Passing tests tell you the code does what you tested.*
*Requirements-derived tests tell you the code does what you specified.*

# Chapter 4: Review AI Output Like a PR

You have been running `tasks.py` for three chapters. You have verified its behavior. The tests pass. The commands work.

Here is a question: can you explain how the code assigns IDs to new tasks?

Not what it does — how. Open `tasks.py` right now and find the ID assignment logic. Read it. Could you explain it to a colleague in thirty seconds without looking at the file?

If you hesitated, you have an understanding gap. The code works. You do not fully understand it. And the next time you add a feature — which is next chapter — you will be building on a foundation you cannot fully explain.

This is the problem that code review solves. Not finding bugs. Not enforcing style. **Transferring understanding from the implementation to you**, so that you — not the AI — own the code going forward.

## The Understanding Gap

When you write code yourself, review happens continuously. You make a decision, write the code, read it back, adjust. By the time you are done, you understand the code because you built it incrementally.

AI-generated code arrives complete. You read it once to verify it runs, then move on. Three chapters in, you may have read `tasks.py` in fragments — checking one function when something seemed off — but probably not end-to-end with the goal of understanding the whole thing.

The gap between "verified behavior" and "understood implementation" matters for one specific reason: **you are the permanent maintainer**. The AI that generated this code has no memory of it. Next session, it will read the file fresh, just like you will. The difference is that you are responsible for what it contains and what it does going forward. The AI is not.

Code review is how you take that responsibility seriously. Not as a formality — as a transfer of ownership.

## The Failure Path: Running Without Reading

Open `tasks.py` right now. Not to check something specific — to read it, top to bottom, as if a colleague had submitted it as a pull request and you had to approve or reject it.

Read slowly. When you finish each function, ask: *could I write this from scratch without looking at the file?*

Most readers at this point find at least one of the following:

**A function that does more than one thing.** The `add` command handler might parse the command-line arguments, validate them, construct the task object, and write to the file — all in one block. That is four responsibilities in one place.

**Variable names that are not descriptive.** The AI often uses single-letter variables (`t` for task, `d` for data, `f` for file handle) in loops and comprehensions. The code runs. The intent is not immediately clear.

**Missing error handling for invalid input.** What happens when you run `python tasks.py done 99` and task 99 does not exist? Try it now. Does the code print a clear error message and exit cleanly? Or does it crash with a Python traceback?

You probably found at least one of these. Possibly all three. None of them caused a test failure — the tests cover your requirements, which did not specify internal structure. But all of them will slow you down when you modify this code next chapter.

## The PR Review Checklist

Pull request review has standard questions. If you have done code review before, this is the same process applied to a different author. If you have not, here is the checklist:

**1. Can you explain what every function does in one sentence?**

If not, the function probably does too many things, or its name does not match its behavior.

**2. Does each function do one thing?**

A function that parses input *and* writes to disk *and* handles errors is three functions that happen to share a name. The AI defaults to this pattern because it produces working code faster. You have to decide if it is acceptable to maintain.

**3. Are variable and function names descriptive?**

`task` is better than `t`. `task_data` is better than `d`. `tasks_file_path` is better than `fp`. You will read this code again. Make it readable.

**4. Are error cases handled explicitly?**

Every command that takes an ID (`done`, `delete`) can receive an ID that does not exist. Every function that reads a file can receive a file that is malformed. If these cases are not handled explicitly, the user gets a Python traceback instead of a useful error message.

**5. Is there anything you would not want to explain to a colleague?**

This is the gut-check question. If there is a section of code you find yourself hoping no one looks at too closely, that section needs attention.

Go through the checklist on `tasks.py`. Write down every finding — even minor ones. Then prioritize: what must be fixed before you add another feature?

## Three Findings, Three Fixes

Here are three findings that commonly appear in the task manager after three chapters of AI generation. Yours may differ; apply the same approach.

### Finding 1: A Function Doing Two Things

The `add` command handler likely does argument parsing, task construction, and file writing in one place. This makes it harder to test the business logic independently and harder to read.

Stay in your project directory and start a fresh Claude Code session. Issue this targeted prompt:

```
In tasks.py, the function that handles the "add" command mixes argument
with task creation and file writing. Refactor it into two parts:
- A pure function add_task(tasks, description, priority) that creates a
  appends a task to the list and returns the updated list
- The CLI handler that calls add_task and handles file I/O

Do not change any behavior. All existing tests must still pass.
```

> **What the Agent Will Do:** The agent will extract the business logic into a separate function and update the CLI handler to call it. The interface to the user — the command syntax and output — will not change.

> **Watch For:** The agent creating a function named `add_task` (or similar) that takes only data parameters and returns data — no file I/O inside it. Then run:

```
pytest tests/ -v
```

> **Watch For:** All nine tests still passing. If any fail, the refactor changed behavior — issue a targeted correction identifying the specific test that failed and what it expected.

### Finding 2: Unclear Variable Names

Find any single-letter or abbreviated variable names in `tasks.py`. Common examples: `t` in a loop over tasks, `d` used for a dict, `f` for a file handle.

Issue a targeted rename prompt — do not ask the agent to "improve readability" (too vague):

```
In tasks.py, rename these variables for clarity:
- Any loop variable named "t" that iterates over tasks → rename to "ta
- Any variable named "d" used for task data → rename to "task_data"
- Any file handle variable named "f" → rename to "file"

Do not change any logic. Run the tests to confirm nothing broke.
```

> **What the Agent Will Do:** The agent will perform the renames throughout the file. This is a mechanical change — it should not affect any behavior.

> **Watch For:** The agent completing quickly (this is a simple change). The agent may run the tests itself — but run them yourself in the terminal panel to confirm:

```
pytest tests/ -v
```

### Finding 3: Missing Guard on Nonexistent ID

Run this in the Claude Code terminal panel (not the Claude Code session):

```
python tasks.py done 99
```

If the output is a Python traceback (`IndexError`, `KeyError`, or similar) rather than a clean error message, the error case is not handled.

Write the requirement for the error behavior, then issue the fix prompt:

```
In tasks.py, the "done" and "delete" commands crash with a Python excep
when given an ID that does not exist in tasks.json.
Fix both commands to instead print a clear error message and exit with
```

```
non-zero exit code when the given ID is not found. Example error messag
"Error: task 99 not found."

Do not change behavior for valid IDs. Run the tests to confirm nothing
```

> **What the Agent Will Do:** The agent will add a guard at the start of each command handler that checks whether the given ID exists before proceeding. If not found, it prints the error message and exits.

> **Watch For:** Running `python tasks.py done 99` now prints `Error: task 99 not found.` instead of a traceback. Then run the full test suite:

```
pytest tests/ -v
```

> **Watch For:** All nine tests still passing. Optionally, add a tenth test for the nonexistent-ID case — you now have a requirement for this behavior.

## Debrief

Three findings. Three targeted prompts. Nine tests still passing after each fix.

Notice what the process was not: it was not "rewrite this to be cleaner." Vague improvement requests produce unpredictable changes. Each finding was specific, each prompt was specific, and the test suite confirmed that the behavior was preserved throughout.

This is the difference between code review as a discipline and code review as an editorial opinion. Discipline means: find a specific problem, specify the fix, verify the behavior did not change.

**The AI is a fast implementer. You are the permanent maintainer.** Those roles have different incentives. The AI's incentive is to produce working code quickly. Your incentive is to produce maintainable code you can understand, explain, and modify for the next six chapters. Review is where you enforce your incentive on the AI's output.

One more thing worth noting: you just did what a senior developer does when reviewing a junior developer's pull request. You did not rewrite it. You did not take it over. You asked targeted questions, identified specific issues, requested specific changes, and verified the result. The skill is identical — the colleague is just a different kind of intelligence.

In Part 2, the code review skill formalizes this checklist into a repeatable step in the Development Loop, run automatically after every implementation sprint. The autonomy modes you will learn about also reflect this ownership question: how much trust you extend to the AI, and when you pull the work back for human review. You have been calibrating that instinctively. Part 2 makes it explicit.

*Understanding the code is not optional — you are the one who has to change it next.*

# Chapter 5: Iterate Deliberately

AI makes iteration cheap. You can prompt, get a result, prompt again, and repeat. This is genuinely useful — you can explore and refine quickly.

It is also a trap.

When iteration is cheap, the temptation is to iterate recklessly — pile on changes in a single prompt, accept whatever seems better, never quite stop to verify. You move fast. The code accumulates. Something breaks three iterations later and you cannot trace it back, because you were not tracking what changed when.

**The cost of reckless iteration is not speed — it is traceability.** When something goes wrong, you need to know which change caused it. If you made five changes in one prompt, you have no way to isolate the cause without untangling everything.

Deliberate iteration solves this by making the same rule explicit: one change at a time, verified, before the next. Not because it is slower — it often is not. Because it keeps the change history readable and the

debugging tractable.

## The Reckless Iteration Trap

You have nine passing tests, a clean implementation, and a clear next feature: due dates. Tasks can optionally have a due date, `list` shows it, and a new `overdue` command shows tasks that are past due.

This is three changes: a data model update, a display update, and a new command. The temptation is to ask for all three at once.

Open Claude Code and start a new agent session. Type:

```
Add due dates to the task manager. Tasks can optionally have a due date
Show due dates in the list command. Add an overdue command that shows
tasks past their due date that aren't done yet.
```

> **What the Agent Will Do:** The agent will modify the `add` command to accept a due date, update the `tasks.json` schema, change the `list` output format, and add the `overdue` command. It will make decisions about the date format, how to display missing due dates, and what "past due" means for tasks with no due date.

> **Watch For:** How many files changed. Run the tests immediately:

```
pytest tests/ -v
```

> **Watch For:** How many tests fail. With a combined change of this scope, you will likely see two or three failures — the list format changed, the add signature may have changed, and possibly a test for the `done` display breaks because the list output format is different.

Now try to fix the failures. Issue corrections for the failing tests.

> **Credit Note:** Each correction is another prompt. Each prompt may fix one test and break another. You are debugging a multi-change diff without a clear view of which change caused which failure.

Stop after two or three correction attempts. Count how many prompts you have used. Count how many tests are still failing. Notice: you cannot easily answer "which change broke the list display tests?" because three changes landed together.

This is the traceability problem. Not a catastrophe — but a debugging session that did not have to happen.

## The Increment Plan

The solution is to plan your increments before you open Claude Code, the same way you write a requirement before you build a feature.

An increment plan answers:

• What changes in this increment?

• What does NOT change?

• What does "verified" mean before moving to the next increment?

Here is the increment plan for due dates:

**Increment 1: Data model only**

• Change: `add` accepts `--due YYYY-MM-DD` (optional). Stores `due_date` in `tasks.json` (null if not provided).

• Does not change: `list` output format, `overdue` command does not exist yet

• Verified when: `add "task" --due 2026-04-01` stores `"due_date": "2026-04-01"` in JSON. `add "task"` stores `"due_date": null`. All 9 existing tests pass.

**Increment 2: List display**

• Change: `list` shows `[2026-04-01]` after the task description when a due date exists. Shows nothing extra when `due_date` is null.

• Does not change: `add` signature (already done), `overdue` command still does not exist

• Verified when: list output matches the new format. Update 1 existing test that checks list output. 9 tests pass.

**Increment 3: Overdue command**

• Change: `overdue` command lists tasks where `due_date` is before today's date and `done` is false. Tasks with no due date are never shown as overdue.

• Does not change: anything in increments 1 or 2

• Verified when: `overdue` shows the correct tasks. Add 2 new tests. 11 tests pass.

**Increment 4: Final check**

• Run the full suite. Manually test the feature end-to-end. Confirm the whole thing behaves as intended.

This plan took ten minutes to write. It will save more than that in debugging time.

## Four Increments, Four Verifications

Start from your Ch 4 end state: nine passing tests, reviewed `tasks.py`. If you followed the failure path at the start of this chapter and your code is now in a messy state, the simplest recovery is to open `tasks.py` and revert it manually to the last clean state, or re-run the Ch 4 success path if you did not save a copy. If you have been committing your work with git, `git checkout` to your last clean commit.

### Increment 1: Data Model

Start a fresh Claude Code session. Provide the full context of the current state, then the increment spec:

```
I have a Python CLI task manager in tasks.py with these commands:
add, list, done, delete (plus --priority flag on add).
Storage: tasks.json — array of task objects.

Increment 1 of 3: Add due date storage only.

Changes:
```

```
- add command accepts an optional --due flag: python tasks.py add "desc
- Date format: YYYY-MM-DD only. Invalid format: print error, exit non-z
- tasks.json stores "due_date": "2026-04-01" when provided, "due_date":

Do NOT change:
- list output format (no due date display yet)
- Any existing command behavior
- All 9 existing tests must still pass
```

**What the Agent Will Do:** The agent will add the `--due` flag to the `add` command's argument parser, add basic date format validation, and update the task object construction to include `due_date`. It will not touch `list` or any other command.

**Watch For:** The agent making a small, targeted change. Open `tasks.py` after — the diff should be 10-20 lines, not a rewrite.

Verify:

```
# Test storing a due date
python tasks.py add "Dentist appointment" --due 2026-04-01
python tasks.py list
# Expected: task appears, list format unchanged (no due date shown yet)

# Check tasks.json directly — open it in the editor
# Expected: "due_date": "2026-04-01" in the new task's JSON object

# Test no due date
python tasks.py add "Buy groceries"
# Check tasks.json — Expected: "due_date": null for this task

# Run the full test suite
pytest tests/ -v
```

**Watch For:** All 9 tests passing. If any fail, the increment changed something it should not have. Issue a targeted correction before moving to Increment 2.

## Increment 2: List Display

Start a fresh Claude Code session. Give the agent the current state and increment 2:

```
The task manager now stores due_date in tasks.json ("YYYY-MM-DD" or nul

Increment 2 of 3: Update list display only.
Changes:
```

```
- list shows due date after description when present: [1] [medium] Buy
- list shows nothing extra when due_date is null: [1] [medium] Buy groc
- Completed tasks: [1] [medium] [done] Buy milk [due: 2026-04-01]

Do NOT change:
- add command (already done)
- overdue command (does not exist yet)
- Any other behavior
```

> **What the Agent Will Do:** The agent will modify the `list` command's output formatting to append the due date when present. One function, a few lines.

Update the test that checks `list` output format — it now needs to account for the `[due: YYYY-MM-DD]` suffix. In the same agent session, issue a follow-up prompt if the agent did not update the tests automatically:

```
Update test_tasks.py: the test that checks list output format should no
expect [due: 2026-04-01] appended when a task has a due date, and nothi
appended when due_date is null.
```

Verify:

```
pytest tests/ -v
```

> **Watch For:** 9 tests passing (the updated list-format test now reflects the new output).

## Increment 3: Overdue Command

Start a fresh Claude Code session. Give the agent the full current state and increment 3:

```
The task manager has add (with --priority and --due), list (shows due d
done, and delete commands. tasks.json schema:
{"id": int, "description": str, "done": bool, "priority": str, "due_dat

Increment 3 of 3: Add the overdue command.

Specification:
- python tasks.py overdue — lists tasks where due_date is before today'
  AND done is false
- Output format: same as list — [ID] [priority] description [due: date]
- Tasks with due_date: null are never shown as overdue
- If no tasks are overdue, print "No overdue tasks."
- Today's date comparison uses the local system date (specifying this p
Add exactly 2 new tests:
```

```
1. overdue shows tasks with due dates in the past and done=false
2. overdue does not show tasks with due_date: null, done=true, or futur

Do NOT change any existing commands or tests.
```

> **What the Agent Will Do:** The agent will add an `overdue` command handler that filters `tasks.json` by due date relative to today. It will add 2 new tests to `test_tasks.py`.

Verify:

```
pytest tests/ -v
```

> **Watch For:** 11 tests passing. Then manually verify the command:

```
# Add an overdue task (use a past date)
python tasks.py add "Overdue task" --due 2024-01-01
python tasks.py overdue
# Expected: the overdue task appears

# Add a future task
python tasks.py add "Future task" --due 2099-12-31
python tasks.py overdue
# Expected: future task does not appear
```

### Increment 4: Final Verification

Run the complete suite one more time:

```
pytest tests/ -v
```

> **Watch For:** 11 tests passing, clean output, no warnings.

Then do a full end-to-end manual test — add tasks with and without due dates, mark some done, run `overdue`, run `list`. Make sure the feature behaves as intended across realistic usage, not just the test cases.

## Debrief

Three increments. Three new agent sessions. Eleven passing tests, up from nine. At every point in this process, you could answer: which change am I currently verifying?

Compare this to the failure path, where you could not answer that question after two prompts.

**Traceability is the product of deliberate iteration.** When something goes wrong — and in a long project, something will — you need a clear record of when and where it was introduced. One change per verified increment gives you that record. Everything-at-once does not.

Notice also what you just wrote before you opened Claude Code: an increment plan. It is not a formal document. It is a sequenced list of changes with verification criteria. In Part 2, you will recognize this as the structure of a sprint plan — the same discipline applied to larger units of work coordinated across multiple AI sessions and agents. Every sprint has a defined scope and a definition of done. Every increment has the same. The scale differs; the pattern is identical.

*One change, verified, before the next. This is how you stay in control of a project the AI is building.*

# Chapter 6: Manage Scope

Every feature you ask an AI to build creates an opportunity. Not just for that feature — for all the features adjacent to it that the AI thinks you probably also want.

Search is a good example. Ask the AI to add search, and it will implement search. It will also likely add a `--case-sensitive` flag, output highlighting, and possibly a `--limit` flag. All useful. None of what you asked for.

This is gold-plating — the AI adding polish and adjacent features beyond what was specified. It is the AI being helpful based on what search usually means in a real application. The problem is that each addition is untested, undocumented, and potentially in conflict with what you actually wanted. You specified nothing, so the AI filled in.

**Gold-plated features are liability, not value.** They cost more to verify than to skip, introduce surface area for bugs you never wrote

requirements for, and expand a codebase you are trying to keep small.

The fix is the same as the fix for bad requirements: make the scope explicit before you prompt. Not just what the feature does — what it does not do.

## The Gold-Plating Problem

Your task manager now has eleven passing tests and a growing feature set: add with priority and due date, list with formatted output, done, delete, and overdue. A reasonable next feature: search. Users want to find tasks by description text.

Before you write the requirement, do the failure path. Start a fresh Claude Code session in your project directory. Type:

```
Add a search command to the task manager.
```

> **What the Agent Will Do:** The agent will read `tasks.py` and add a `search` command. It will also make decisions you did not make — about case sensitivity, about whether to search completed tasks, about how matches are displayed, and possibly about result limits. Watch how many of these decisions it makes without being asked.

> **Watch For:** The agent adding flags or options you did not request. Specifically: `--case-sensitive`, `--include-done` or similar filter, color or bracket highlighting of matched terms, or a `--limit` flag.

When the agent finishes, run a quick manual test in the terminal panel:

```
python tasks.py search "buy"
```

Look at the output. If the agent added highlighting, you will see markers around the matched text. If it added flags, the help text may show them. Now run the tests:

```
pytest tests/ -v
```

> **Watch For:** Failing tests. The agent may have changed the `list` output format to add match highlighting — which breaks tests that

check output format. Or it may have added flags that conflict with how the argument parser handles existing commands.

**Credit Note:** You are now in a debugging session you did not plan for. Each correction is another prompt. The corrections may fix one test and surface another issue. You are untangling decisions the agent made without a specification.

Stop here. Note what the agent added that you did not ask for. Then start over with a requirement.

## Scope as Part of the Requirement

The fix is to treat scope as a first-class part of every requirement — not an afterthought.

In Chapter 2, you saw the scope boundary as one of the four required properties: what is explicitly *not* part of this feature. You used it to tell the agent not to sort by priority, not to allow editing priority, not to filter by priority. Those constraints prevented three plausible features the agent might have added.

For search, the scope boundary does more work. Search is a feature with obvious extensions — case options, filters, highlighting, sorting by relevance. Each one is adjacent enough that an agent building a real application would consider it in scope. You need to be explicit that you are not building those yet.

A scope boundary works because it gives the agent a constraint it cannot argue with. Without it, the agent defaults to *what would a complete search feature look like?* With it, the agent defaults to *what exactly did the user specify?*

Here is the requirement for search with an explicit scope boundary:

*Feature:* Search command

*Specific input:*

• `python tasks.py search "text"` — case-insensitive substring match against task descriptions

*Specific output:*

• Matching tasks displayed in the same format as `list`: `[ID] [priority] description [due: date]`

• If no tasks match: `No tasks match "text".`

*Edge cases:*

• Search is always case-insensitive — no flag needed or allowed

• Completed tasks are included in results

• Tasks with no description match are excluded

*Scope boundary:*

• Does NOT highlight or mark matched text in output

• Does NOT filter by priority, done status, or due date

• Does NOT sort results by relevance

• Does NOT support regex — substring match only

Four explicit "Does NOT" statements. Each one closes off an extension the agent might otherwise add.

Notice the format is the same as Chapter 2: specific input, specific output, explicit edge cases, explicit scope. The scope section grew because search has more natural extensions than priority did. The work you do in the scope boundary is proportional to how tempting the feature is to over-engineer.

## The Success Path: Scoped Search

Start a fresh Claude Code session. Give the agent the full state and the scoped requirement:

```
I have a Python CLI task manager in tasks.py with 11 passing tests.
Current commands: add (--priority, --due), list, done, delete, overdue.
tasks.json schema: {"id": int, "description": str, "done": bool, "prior
```

```
Add a search command with the following specification:

Feature: Search tasks by description
Input:
- python tasks.py search "text" — case-insensitive substring match agai

Output:
- Matching tasks in the same format as list: [ID] [priority] descriptio
- If no matches: No tasks match "text".

Edge cases:
- Always case-insensitive (no flag)
- Completed tasks included in results
- Non-matching tasks excluded

Does NOT:
- Highlight matched text in output
- Filter by priority, done status, or due date
- Sort results by relevance
- Support regex — substring match only

Add exactly 1 new test:
- search returns matching tasks in list format, case-insensitive

All 11 existing tests must still pass.
```

**What the Agent Will Do:** The agent will add a `search` command that filters tasks by substring match and formats output identically to `list`. The explicit scope boundary should prevent it from adding flags or altering the output format.

**Watch For:** The agent's implementation after it finishes — open `tasks.py` and look at the `search` command. If it added `--case-sensitive` or any other optional flag despite the scope boundary, the implementation drifted from the requirement. Issue a targeted correction quoting the "Does NOT" line before moving on.

Verify:

```
pytest tests/ -v
```

**Watch For:** 12 tests passing. Then manually verify the command:

```
# Add tasks if tasks.json was cleared
python tasks.py add "Buy groceries" --priority medium
python tasks.py add "Fix critical bug" --priority high
python tasks.py add "Update the readme" --priority low
# Test basic search
```

```
python tasks.py search "readme"
# Expected: [3] [low] Update the readme

# Test case insensitivity
python tasks.py search "BUY"
# Expected: [1] [medium] Buy groceries

# Test no match
python tasks.py search "xyz"
# Expected: No tasks match "xyz".

# Test that completed tasks appear in results
python tasks.py done 1
python tasks.py search "groceries"
# Expected: [1] [medium] [done] Buy groceries
```

> **Watch For:** Output format matching `list` exactly — same brackets, same priority label, same `[done]` indicator. If anything differs, the agent added its own formatting. Issue a targeted correction with the exact expected output.

## Debrief

One command. One new test. Twelve total. The agent stayed in scope because the scope was specified.

Compare this to the failure path, where the agent's interpretation of "search" included features you did not ask for. None of those were wrong — they were reasonable decisions about what a complete search feature should do. They just were not your decisions. The agent made them in the absence of explicit guidance.

**Scope is a product decision, not a cleanup task.** Every "Does NOT" statement in a requirement is a decision you made before the agent could make it for you. The agent cannot know which adjacent features you want and which you do not — unless you tell it.

The discipline generalizes. Every feature you add from here has a natural set of extensions — the next obvious steps an agent would consider if you left scope undefined. Your job before each prompt is to identify those extensions and explicitly exclude the ones you are not

building yet.

Notice what happened to the features you excluded: they did not disappear. Relevance sorting, regex support, done-status filtering — those are all potential future features, now named and parked. In Part 2, that list becomes a shared backlog — plannable as a sprint, assignable to parallel agents, and readable by any session that picks up the project. The scope boundary is not a way to kill features. It is a way to defer them on your terms, with visibility, instead of building them without planning.

*Specify what you are not building. The AI will build it anyway if you don't.*

# Chapter 7: Document Decisions

The AI that helped you build this task manager has no memory of it.

Not a figure of speech — literally none. Every new agent session starts blank. The session that added the `--priority` flag has no knowledge of the session that added the `overdue` command. The session you open today cannot remember that you decided invalid priority values should error rather than silently default, or that list output should always show tasks in creation order, or that search should always be case-insensitive with no flag.

You remember these things. For now. But projects outlast working memory.

**Code records what was built. It does not record why.** Open `tasks.py` right now and find the line that handles an invalid priority value. It errors and exits non-zero. Why? Because you wrote a requirement six sessions ago that said so. Is that visible in the code? Only if someone reads the error message carefully — and even then, they do not know that the alternative (silent default) was explicitly ruled out.

A decision log captures the why. Not for documentation's sake. For operational reasons: every new agent session that works on this project needs to know what you decided, or it will make its own decisions — plausibly, confidently, and incorrectly.

## The Amnesia Problem

Here is how the problem surfaces. You are adding a `--format` flag to the `list` command. Users want to export tasks as JSON for other tools. Reasonable feature, small change.

Start a fresh Claude Code session in your project directory. Type:

```
I have a Python CLI task manager in tasks.py with 12 passing tests.
Current commands: add (--priority, --due), list, done, delete, overdue,

Add a --format flag to the list command:
- python tasks.py list --format json outputs all tasks as a JSON array
- Default (no flag) keeps the existing text format unchanged
```

> **What the Agent Will Do:** The agent will add a `--format` flag to the `list` command and implement JSON output. It will make decisions you have not specified: which fields to include in the JSON, what happens when an invalid format is passed, and whether the JSON output matches the schema in `tasks.json` exactly.

> **Watch For:** How the agent handles an invalid format value. Run this after it finishes:

```
python tasks.py list --format csv
```

The agent will likely either print an error or silently fall back to the default text format. Neither is wrong by itself. But one of them contradicts a decision you made in Chapter 2.

Now run the tests:

```
pytest tests/ -v
```

> **Watch For:** The tests may all pass — the new flag does not necessarily break existing behavior. But run this check manually: does `list --format csv` produce an error message and exit

non-zero? Or does it silently output the default format?

If it silently defaults, the agent made a different call than the one you established in Chapter 2: invalid input should print an error and exit non-zero. The agent had no way to know this. It was not told. The decision lived in your head and in a requirement you wrote six sessions ago in a different context.

**Credit Note:** This is a subtle failure. The tests pass. The feature works. The implementation just contradicts a principle you established earlier, and you will only discover it when a user passes an invalid flag and gets silence instead of an error.

Stop here. Do not fix it. You are going to write the decision log first.

## Writing decisions.md

A decision log is not a changelog and it is not documentation. It is a short list of choices that constrain future behavior — the calls you made that a new AI session would not know about unless you told it.

The format is minimal on purpose. For each decision:

• **Decision**: the call you made

• **Reason**: why you made it

• **Rules out**: what this decision explicitly excludes

Three fields, one line each. Here is decisions.md for the task manager after six chapters:

```
# Task Manager: Decision Log

## Error behavior
- **Decision**: Invalid input (bad priority, bad date format, bad --for
- **Reason**: Silent defaults make errors invisible to users and script
- **Rules out**: Silent fallback to defaults on invalid input.

## Missing fields
- **Decision**: Tasks in tasks.json missing optional fields (priority,
- **Reason**: Backward compatibility — old tasks should not break new f
- **Rules out**: Erroring on missing fields; auto-migrating stored data
```

```
## Output format
- **Decision**: list output always shows all fields for a task in a fi
- **Reason**: Consistent output makes tests reliable and user expectati
- **Rules out**: Optional fields, abbreviated output, fields varying by

## Sort order
- **Decision**: list always shows tasks in creation order (by id ascend
- **Reason**: Predictable ordering makes manual verification easier.
- **Rules out**: Any automatic sorting.

## Date format
- **Decision**: All dates use YYYY-MM-DD. Invalid format prints error,
- **Reason**: Single format, no ambiguity between MM/DD and DD/MM.
- **Rules out**: Other date formats, locale-specific parsing.

## Search behavior
- **Decision**: search is always case-insensitive. No flag.
- **Reason**: Users should not need to think about case to find their c
- **Rules out**: Case-sensitive search mode; --case-sensitive flag.
```

Create this file yourself — not by prompting the agent, but by hand. Use your editor or `touch decisions.md`, paste the content above, and save it.

This took about ten minutes to write. The benefit is not just the document itself — it is the act of writing it. Reviewing six chapters of decisions reveals which ones you would forget to mention in a prompt and which ones would surprise a new AI session if it had to guess.

## The Success Path: Format with Context

Start a fresh Claude Code session. This time, give the agent your decisions.md as part of the context. You do not need to paste the entire file — pull out the decisions relevant to this feature (error behavior and output format). The prompt below shows exactly what to include:

```
I have a Python CLI task manager in tasks.py with 12 passing tests.
Current commands: add (--priority, --due), list, done, delete, overdue,
tasks.json schema: {"id": int, "description": str, "done": bool, "prior

Project decisions (from decisions.md):
- Invalid input prints an error and exits non-zero. Does not silently c
- JSON output includes all task fields: id, description, done, priority
- list always shows all fields; no optional or abbreviated output.

Add a --format flag to the list command:
```

```
Input:
- python tasks.py list — unchanged text output
- python tasks.py list --format json — outputs tasks as a JSON array

Output (json format):
- A JSON array of task objects, one per line, all fields included
- Example: [{"id": 1, "description": "Buy groceries", "done": false, "p
- If no tasks: []

Edge cases:
- python tasks.py list --format csv or any unsupported format: print er
  "Error: unsupported format 'csv'. Supported formats: json" and exit n

Add exactly 1 new test:
- list --format json outputs valid JSON containing all task fields

All 12 existing tests must still pass.
```

> **What the Agent Will Do:** The agent will add the `--format` flag and implement JSON output. Because the decision log explicitly stated that invalid input must error and exit non-zero, the agent will implement the error case correctly — it has a clear constraint.

> **Watch For:** The invalid format test — run this after the agent finishes:

```
python tasks.py list --format csv
```

Expected: `Error: unsupported format 'csv'. Supported formats: json` and a non-zero exit code. If you see this, the decision was respected. If the command silently outputs text, the agent missed the constraint.

Verify:

```
pytest tests/ -v
```

> **Watch For:** 13 tests passing. Then manually verify JSON output:

```
python tasks.py add "Buy groceries" --priority medium
python tasks.py add "Fix critical bug" --priority high --due 2026-04-01
python tasks.py list --format json
# Expected: JSON array with both tasks, all fields present including du
```

> **Watch For:** Every field present — `id`, `description`, `done`, `priority`, `due_date`. If `due_date` is missing from the JSON,

> the agent did not follow the "all fields" decision. Issue a targeted correction referencing the decision log.

## Debrief

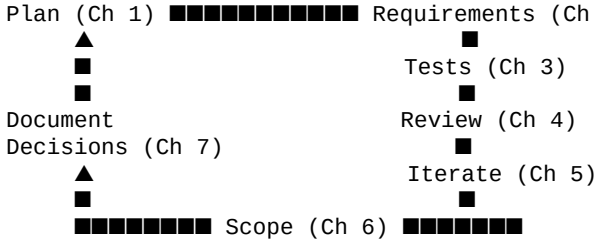
Compare the two sessions. In the first, the agent made its own call on invalid format — plausibly, but inconsistently with your established behavior. In the second, the agent had the decision log and implemented correctly on the first try.

The decision log did not add complexity. It removed ambiguity. The agent did not have to guess what you wanted for invalid input because you had already decided that, six chapters ago, and written it down.

**A decision log is insurance against your own forgetfulness as much as the AI's amnesia.** Midway through a long project, you will not remember why you chose YYYY-MM-DD over other formats, or why search has no case-sensitive flag. The log is the record. The AI is just the most obvious beneficiary.

The discipline generalizes to collaborators — human and AI alike. If another developer joins this project, the code tells them what exists. The decisions.md tells them what constraints those choices were made under. One file, one purpose: preserving intent across time.

You have now built a task manager through seven chapters of deliberate practice: planning, requirements, testing, review, iteration, scope, and decision tracking. Each discipline compounds. The requirements from Chapter 2 informed the test cases in Chapter 3. The tests from Chapter 3 caught the scope drift in Chapter 6. The scope decisions from Chapter 6 are now in the decision log you will carry into Part 2.



*Each new feature enters at Plan, informed by documented decisions from the previous feature. The loop is the practice.*

In Part 2, decisions.md becomes the seed of a larger artifact — the project context file that makes multi-session AI development tractable at scale. You built the habit here. Part 2 gives it a formal structure.

*The AI forgets everything when the session ends. decisions.md doesn't.*

# Chapter 8: The AgentFlow Loop

Look back at what you did in Part 1.

Before each feature, you wrote a requirement. Before you opened a new session, you read your code. After each change, you ran the tests. When something went wrong, you issued a targeted correction. When you finished, you updated decisions.md.

You were not following a random set of tips. You were running a loop.

Every session had the same structure: establish what you know, specify what you want, implement with a scoped agent, verify the result, review what changed, record what was decided. Seven disciplines, one sequence. **AgentFlow is the name for that sequence made explicit** — a repeatable structure that puts the disciplines in order so you do not have to remember when to apply them.

Part 2 is about running that loop deliberately, at scale, across a project that grows over weeks. This chapter introduces the loop. The chapters that follow introduce the tools that make it faster.

## The Five Stages

The AgentFlow Development Loop has five stages. You have run all of them. Here they are with their names:

**Stage 1: Start**

Load the context for this session. What is the current state of the project? How many tests are passing? What schema does the data have? What decisions have been made? This is where decisions.md earns its keep — instead of reconstructing context from memory, you read the file.

Then write the requirement for this session's work. Specific input, specific output, edge cases, explicit scope boundary. Nothing new here — this is Chapter 2.

**Stage 2: Implement**

Open a new agent session. Give the agent the context (state) and the requirement. Let it build. Do not interrupt unless it asks a question or does something visibly wrong.

This is one session, one requirement, one increment — Chapter 5.

**Stage 3: Verify**

Run the test suite. Read the output. If tests fail, issue a targeted correction. If tests pass, run the manual checks from your requirement's verification section.

The tests come from your requirements, not from the agent's interpretation of its own code — Chapter 3.

**Stage 4: Review**

Read the diff. Not to find bugs — to transfer understanding. Can you explain every function that changed? Are the variable names readable? Is there anything you would not want to explain to a colleague?

This is Chapter 4, applied after every implementation sprint rather than once at the end.

**Stage 5: Update**

Record what changed. Add a new entry to decisions.md for any new decisions made during this session. Note the end state: how many tests pass, what commands exist, what the schema looks like. This is the Start material for the next session.

Then close the agent session. The next session begins at Stage 1.

The loop is not bureaucracy. Each stage takes a fraction of the time the implementation takes. The overhead is the requirement (ten minutes), the test run (seconds), the diff read (two minutes), the decisions.md update (two minutes). The total is fifteen minutes of structure per hour of implementation. The payoff is a project you can pick up in two weeks and know exactly where it stands.

## The Failure Path: Skipping the Loop

Your task manager has thirteen passing tests and a decisions.md. A reasonable next feature: a `stats` command that shows a quick summary — total tasks, how many are done, how many are overdue.

Skip the loop. Start a fresh Claude Code session in your project directory. Type:

```
Add a stats command to the task manager.
```

> **What the Agent Will Do:** The agent will add a `stats` command that counts tasks. It will make choices you have not specified: what to count, what format to display, what to show when tasks.json is empty, and how to count "overdue" — does it use the same logic as the `overdue` command, or its own?

> **Watch For:** The agent's output format. Run it after the agent finishes:

```
python tasks.py stats
```

Note the format. Is it one line? Multiple lines? Does it show counts as numbers or percentages? Does it mention overdue? Now run the tests:

```
pytest tests/ -v
```

> **Watch For:** The test count. If the agent wrote a test for stats, it wrote it against its own implementation — not against a requirement you specified. And the session left no trace: no updated decisions.md, no documented output format, no note of what was

> added.

Stop here. The feature may work. But if you close this session and open a new one tomorrow, you cannot answer: what does `stats` output when there are no tasks? The agent decided. You do not know its reasoning. It is gone.

This is what the loop prevents — not bad code, but undocumented decisions.

## Running the Loop: Stats

Start over. Close the agent session. Now run the loop explicitly.

### Stage 1: Start

Before opening a new agent session, establish context and write the requirement.

**Current state:**

• 13 passing tests

• Commands: add (--priority, --due), list (--format json), done, delete, overdue, search

• tasks.json schema: `{"id": int, "description": str, "done": bool, "priority": str, "due_date": str|null}`

• decisions.md: error behavior, missing fields, output format, sort order, date format, search behavior

**Requirement:**

*Feature:* Stats command

*Specific input:*

• `python tasks.py stats` — no arguments

*Specific output:*

• One-line summary: `Total: N tasks (X done, Y overdue)`

• If tasks.json is empty or does not exist: `No tasks.`

*Edge cases:*

• Overdue count uses the same logic as the `overdue` command: due_date before today AND done=false

• Done count: tasks where `done` is true, regardless of due date

*Scope boundary:*

• Does NOT show individual tasks

• Does NOT break down by priority

• Does NOT show percentages

## Stage 2: Implement

Start a fresh Claude Code session. Give the agent context and the requirement:

```
I have a Python CLI task manager in tasks.py with 13 passing tests.
Current commands: add (--priority, --due), list (--format json), done,
tasks.json schema: {"id": int, "description": str, "done": bool, "prior

Project decisions (from decisions.md):
- Invalid input prints an error and exits non-zero. Does not silently c
- Output format: fixed, one-line per summary. No optional fields.
- Overdue logic: due_date before today's local date AND done=false.

Add a stats command with the following specification:

Feature: Stats command
Input: python tasks.py stats (no arguments)
Output:
- Total: N tasks (X done, Y overdue)
- If no tasks or tasks.json missing: No tasks.

Edge cases:
- Overdue count: same logic as overdue command (due_date before today A
- Done count: tasks where done=true, any due date

Does NOT:
- Show individual tasks
- Break down by priority
- Show percentages
Add exactly 1 new test:
```

```
- stats shows correct total, done, and overdue counts

All 13 existing tests must still pass.
```

**What the Agent Will Do:** The agent will add a `stats` command that reads tasks.json and computes the three counts. It will reuse the date comparison logic from the `overdue` command. One new test.

**Watch For:** The agent completing without clarifying questions. A well-specified requirement should not require agent questions. If it asks something, the answer is in the requirement — point to the relevant line.

### Stage 3: Verify

```
pytest tests/ -v
```

**Watch For:** 14 tests passing. Then verify manually:

```
# Add test tasks if needed
python tasks.py add "Buy groceries" --priority medium
python tasks.py add "Fix critical bug" --priority high --due 2024-01-01
python tasks.py add "Future task" --priority low --due 2099-12-31
python tasks.py done 1

python tasks.py stats
# Expected: Total: 3 tasks (1 done, 1 overdue)

# Verify with no tasks
# Open tasks.json in your editor and replace its contents
# with an empty array: []
# Save (Cmd+S or Ctrl+S), then:
python tasks.py stats
# Expected: No tasks.
```

**Watch For:** Both cases matching the requirement exactly. If the overdue count is wrong, check whether the agent used today's local date (correct) or UTC (which may differ by timezone).

### Stage 4: Review

Open `tasks.py` and find the `stats` command handler. Read it. Can you answer:

• How does it compute the overdue count? Is it the same logic as `overdue`?

• What does it do when tasks.json does not exist?

• Is the output format constructed correctly for singular vs. plural? (This edge case is easy to miss: "1 tasks" instead of "1 task".)

If the singular/plural case is wrong — `Total: 1 tasks` instead of `Total: 1 task` — issue a targeted correction now, before moving to Stage 5.

> **Watch For:** Any code path the agent added that you cannot explain in one sentence. If found, ask the agent to explain it in a follow-up message before moving on.

### Stage 5: Update

Add a new entry to decisions.md. Open `decisions.md` in your editor and add:

```
## Stats format
- **Decision**: stats output is one line: "Total: N tasks (X done, Y ov
- **Reason**: Matches the one-line summary pattern established for othe
- **Rules out**: Multi-line breakdown, percentage display, priority bre
```

Also note the new end state at the bottom of decisions.md:

```
## Session end state
Tests: 14 passing
Commands: add, list, done, delete, overdue, search, stats
```

Close the agent session. The loop is complete.

## Debrief

You just ran a named, documented session. In thirty years, when you open this project after a long break, you will know exactly where it stands — not because you have a good memory, but because the loop left a trail.

**The loop is the discipline container.** Without it, the seven disciplines from Part 1 are habits that fire when you remember them. With the loop,

they fire because the loop has stages and the stages have names: Start requires a requirement (Chapter 2), Verify requires tests (Chapter 3), Review requires a diff read (Chapter 4), Update requires decisions.md (Chapter 7). The loop does not add discipline — it schedules it.

The next three chapters introduce the tools that make the loop faster. Chapter 9 covers skills — prompts that automate specific stages, so you do not have to write the same context setup from scratch each session. Chapter 10 covers context files — the formalized version of decisions.md that makes Stage 1 nearly instantaneous. Chapter 11 covers multi-agent coordination — running multiple loop sessions in parallel, which is how Claude Code subagents become useful.

All of that builds on what you just did: one loop, five stages, one feature, fourteen tests, decisions documented.

*The loop does not make you faster. It makes you traceable — which is what lets you go faster for longer.*

# Chapter 9: Skills and Autonomy Modes

Count how many times you have typed a variation of this at the start of an agent session:

```
I have a Python CLI task manager in tasks.py with N passing tests.
Current commands: add (--priority, --due), list, done, delete, overdue,
tasks.json schema: {"id": int, "description": str, "done": bool, "prior
```

Seven times across Part 1. Plus the decisions. Plus the scope of the specific feature. Every session, reconstructed from memory.

This is the boilerplate problem. The context setup is identical across sessions — the same project, the same schema, the same decision constraints — but you write it from scratch each time. Small errors accumulate: a forgotten decision, a wrong test count, an outdated schema. The agent works with what you give it. If the context is incomplete, the implementation reflects the gap.

**A skill is the fix.** Write the context setup once, save it as a file, reference it instead of retyping. The agent reads the skill file and has everything it needs to start. The session begins at the feature, not at the boilerplate.

## What a Skill Is

A skill is a markdown file that encodes a repeatable prompt. It is not code. It is not documentation. It is a saved prompt template — the same thing you have been typing manually, written once and stored.

A skill for the Start stage of the AgentFlow loop would contain:

• The current project state (test count, commands, schema)

• A reference to decisions.md for the full decision list

• Any standard instructions that apply to every session (the requirement format, the "Does NOT" convention, the test count to preserve)

The format is simple:

```
# Task Manager: Session Start

## Project state
- Tests: [CURRENT_COUNT] passing
- Commands: add (--priority, --due), list (--format json), done, delete
- tasks.json schema: {"id": int, "description": str, "done": bool, "pri

## Decisions
See decisions.md for the full decision log. Key constraints:
- Invalid input: print error, exit non-zero. Never silently default.
- Missing fields: display defaults, never error.
- Output format: fixed, all fields shown, no optional output.
- Date format: YYYY-MM-DD only. Overdue uses local system date, not UTC
- Sort order: creation order always. No automatic sorting.

## Standard instructions
- New features require a requirement before implementation.
- Each session adds at most 2 new tests.
- All existing tests must pass after any change.
```

Three sections. The only part that changes session to session is the test count. Everything else is stable until the project changes significantly.

## Writing start-session.md

Create the skill file in your task manager project. Use your editor or `touch start-session.md`, then copy the template above and update `[CURRENT_COUNT]` to `14` (your current test count after Chapter 8).

Save with Cmd+S or Ctrl+S.

> **Note:** The skill file lives in your project directory alongside `tasks.py` and `decisions.md`. It is not a special Claude Code file type. It is a plain markdown file you can reference at the start of each session with `@start-session.md`, or copy into the prompt if you prefer.

Now see the failure path, to understand what the skill prevents.

## The Failure Path: Without the Skill

The next feature is `clear` — a command that removes all completed tasks from tasks.json. Simple feature, one new test. Add it without using the skill.

Start a fresh Claude Code session in your project directory. Type from memory:

```
I have a Python CLI task manager with 14 tests. Add a clear command tha
removes done tasks. Also make sure the overdue count in stats is right.
```

> **What the Agent Will Do:** The agent will implement `clear`. It will also attempt to verify the `overdue` count in `stats`. But "overdue count is right" is ambiguous — the agent may use UTC rather than local system time, which is what your decisions.md specifies. You did not include that decision, so the agent guesses.

> **Watch For:** After the agent finishes, run:

```
python tasks.py stats
```

The count may look correct. But check the implementation — open `tasks.py` and find where `stats` computes overdue tasks. Does it use `datetime.now()` (local time) or `datetime.utcnow()` or `datetime.now(timezone.utc)`? If it uses UTC, it contradicts the decision you made in Chapter 8, and users in certain timezones will see wrong overdue counts.

You would not catch this from the test suite. The tests run on your machine, in your timezone. The inconsistency is invisible until a user in a different timezone files a bug report.

This is the cost of incomplete context: the agent filled the gap with a plausible assumption. The assumption was wrong.

> **Credit Note:** Run the tests anyway to see the count:

```
pytest tests/ -v
```

> **Watch For:** Tests passing, but the implementation containing a decision you did not specify. Stop here — do not fix it. You will use the skill in the success path.

## The Success Path: Clear with the Skill

Start a fresh Claude Code session. This time, reference `@start-session.md` at the top of your prompt, or paste its contents if you prefer. Then add the feature requirement below:

```
# Task Manager: Session Start

## Project state
- Tests: 14 passing
- Commands: add (--priority, --due), list (--format json), done, delete
- tasks.json schema: {"id": int, "description": str, "done": bool, "pri

## Decisions
See decisions.md for the full decision log. Key constraints:
- Invalid input: print error, exit non-zero. Never silently default.
- Missing fields: display defaults, never error.
- Output format: fixed, all fields shown, no optional output.
- Date format: YYYY-MM-DD only. Overdue uses local system date, not UTC
- Sort order: creation order always. No automatic sorting.

## Standard instructions
```

```
- New features require a requirement before implementation.
- Each session adds at most 2 new tests.
- All existing tests must pass after any change.

---

Feature: Clear completed tasks

Input: python tasks.py clear (no arguments)
Output:
- Cleared N completed tasks. (where N is the count removed)
- If no completed tasks: No completed tasks to clear.

Edge cases:
- Task IDs are NOT renumbered after clearing
- Tasks with done=false are not affected

Does NOT:
- Clear by priority, due date, or any other filter
- Ask for confirmation before clearing
- Support undoing the clear

Add exactly 1 new test:
- clear removes done tasks and prints the correct count

All 14 existing tests must still pass.
```

**What the Agent Will Do:** The agent will implement `clear` with full knowledge of your decisions — it will not touch the overdue logic, it will not renumber IDs, and it will follow the established error pattern. The skill context prevents the gap the failure path exposed.

**Watch For:** The agent not asking clarifying questions. Complete context means no guessing required.

Verify:

```
pytest tests/ -v
```

**Watch For:** 15 tests passing. Then manually verify:

```
python tasks.py add "Buy groceries" --priority medium
python tasks.py add "Fix critical bug" --priority high
python tasks.py done 1

python tasks.py clear
# Expected: Cleared 1 completed tasks.
python tasks.py list
```

```
# Expected: Only task 2 remains. Task 1 is gone. IDs are not renumbered

python tasks.py clear
# Expected: No completed tasks to clear.
```

Update decisions.md with a new entry for `clear`:

```
## Clear behavior
- **Decision**: clear removes all done=true tasks permanently. IDs not
- **Reason**: Matches delete behavior (no confirmation, no undo). Keeps
- **Rules out**: Confirmation prompt, undo capability, selective clear
```

Also update the test count in `start-session.md`: change `14` to `15`.

## Autonomy Modes

A skill makes the Start stage fast. Autonomy modes decide how much of the remaining loop the AI runs before pausing for human input.

**Mode 1: Stage-by-stage**

The AI completes one loop stage and stops. You review, then invoke the next stage explicitly. Maximum oversight — every implementation, verification, and review step has human eyes before proceeding. Appropriate when: the task is unfamiliar, the requirement is uncertain, or you want to build confidence in a new skill.

**Mode 2: Implement and verify**

The AI runs Implement and Verify automatically — it builds the feature and runs the tests. It stops before Review and Update, which remain human steps. Appropriate when: the requirement is tight, the test suite is comprehensive, and you trust the implementation quality. This is the mode you effectively used for most of Part 1 — you wrote the requirement, let the agent implement, then ran tests yourself.

**Mode 3: Full loop**

The AI runs all five stages — Start (using the skill), Implement, Verify, Review, and Update — and delivers a complete session artifact. You review the entire output at the end. Appropriate only when: your skills are mature, your context file is complete (Chapter 10), and the test suite is reliable enough to catch regressions. Mode 3 is fast and high-risk;

save it for routine features on a well-established project.

For the `clear` command you just built, Mode 1 was appropriate — first time using a skill, first session with the new loop structure. As you build confidence in your skills and context file, you will naturally shift toward Mode 2 for routine features. Mode 3 is something to earn, not a default.

## Debrief

A skill is saved process. The context setup that took five minutes every session now takes thirty seconds — copy, paste, append the feature requirement, send. The overdue-uses-local-date decision travels with every session automatically, not just when you remember to include it.

**Skills reduce the cost of consistency.** The disciplines from Part 1 are only valuable if applied every session. Skills apply them without relying on memory.

The next chapter formalizes this further. Rather than embedding the project state directly in `start-session.md`, you will learn to maintain a dedicated context file — a single source of truth for current project state that your skills reference. When the test count changes, you update one file. All skills automatically reflect the new state.

*A skill is a discipline made repeatable. Write it once; apply it every session.*

# Chapter 10: Context Files and Handoffs

Open `start-session.md`. Find the line that says `Tests: 14 passing`.

It is wrong. You have 15 tests now, after Chapter 9. Every new agent session you start with this skill sends stale context — the agent thinks you have fewer tests than you do.

You could fix it. Update the number to 15. But next session, after the next feature, it will be wrong again. The test count is embedded in the skill file, which means maintaining the skill is a second job that runs in parallel with maintaining the project. Forget once, and the agent works with wrong information.

This is the stale skill problem. And it points to a separation of concerns that skills did not handle: the *instructions* for how to start a session are stable — same requirement format, same "Does NOT" convention, same test-pass constraint. The *state* of the project changes every session. They should not live in the same file.

A context file is that separation: project state in one place, session instructions in another. The skill references the context file. You update the context file. The skill stays current without touching it.

## What context.md Is (and Is Not)

`context.md` is not a README. A README explains the project to someone who wants to use it — how to install it, what commands exist, what it does. context.md explains the project to someone who wants to continue building it — what state it is in right now, what decisions constrain it, and what is open or unresolved.

The audience for context.md is not a user. It is the next developer — which is usually you, two weeks from now, or an AI session starting fresh.

What context.md contains:

**Project overview** — one paragraph. What the project is and what it does. Stable — rarely changes.

**Current state** — test count, command list, data schema. Changes every session, or whenever a feature is added.

**Decision summary** — the key constraints from decisions.md, compressed. Not a replacement for decisions.md — a quick reference so the agent does not have to read the full decision log for every session.

**Open questions** — anything undecided that might affect future sessions. Empty is fine.

**Session marker** — when it was last updated. Helps you know if the file is stale.

That is it. Five sections. The file should be readable in under two minutes.

## Writing context.md

Create `context.md` in your project directory using your editor or `touch context.md`. Copy and fill in this template:

```
# Task Manager: Project Context

## Overview
Python CLI task manager. Reads and writes tasks.json. Single-user, loca
Built across a tutorial series; feature-complete as of Chapter 9.

## Current state
- Tests: 15 passing
- Schema: {"id": int, "description": str, "done": bool, "priority": str
- Commands: add (--priority, --due), list (--format json), done, delete

## Key decisions
Full log: decisions.md
Summary:
- Invalid input → print error, exit non-zero. Never silently default.
- Missing fields → display defaults, never error or modify stored data
- All dates: YYYY-MM-DD only. Overdue uses local system date, not UTC.
- Output: fixed format, all fields always shown. No optional output.
- List order: creation order always. No automatic sorting.
- Search: always case-insensitive. No flag.

## Open questions
None currently.

## Last updated
[today's date] — 15 tests, 8 commands
```

Save with Cmd+S or Ctrl+S.

Now update `start-session.md`. Remove the embedded state section and replace it with a reference:

```
# Task Manager: Session Start

## Project state
```

```
See context.md for current project state, schema, and decision summary.

## Standard instructions
- New features require a requirement before implementation.
- Each session adds at most 2 new tests.
- All existing tests must pass after any change.
```

Three lines replaced with one reference. The test count no longer lives in start-session.md — it lives in context.md, which you update at the end of every session as part of Stage 5 of the loop. The skill does not go stale because it does not contain the state.

## The Failure Path: Thin context.md

Before testing the full context.md, see what a minimal version produces.

Start a fresh Claude Code session in your project directory. Paste only this:

```
Read this project context and tell me: (1) what the current state of th
project is, and (2) what the most natural next feature to add would be.

# Task Manager: Project Context
Tests: 15 passing
Schema: {"id": int, "description": str, "done": bool}
```

> **What the Agent Will Do:** The agent will describe the project state as best it can from the minimal context — 15 tests, basic schema. For the next feature, it will guess based on common task manager patterns: due dates, priority, tags, recurring tasks. It has no way to know that due dates, priority, and search already exist.

> **Watch For:** The agent suggesting a feature that already exists. "You could add priority levels" or "consider adding due dates" — both already built in Chapters 2 and 5. The thin context.md did not give the agent enough to work with.

Stop here. Do not fix it in this session. The thin context was a demonstration, not a starting point.

## The Handoff Test: Full context.md

This is the test of whether your context.md is complete. A new agent session should be able to read it and — without any other input from you — correctly describe the project state and propose a reasonable next feature.

Start a fresh Claude Code session. Reference `@context.md` and send this prompt — replacing the bracketed placeholder if you are pasting manually:

```
Read this project context and tell me:
1. What is the current state of this project?
2. What would be the most natural next feature to add, consistent with
   the established decisions?

[paste the full contents of your context.md file here]
```

**What the Agent Will Do:** The agent will describe the project accurately — 15 tests, 8 commands, the schema, the key constraints. For the next feature, it should propose something consistent with the decisions: perhaps a `--priority` filter on list, or a bulk `done` command, or CSV export — features that extend the project without contradicting established choices. It will not suggest due dates or priority (already built) and it should not suggest anything that violates the "no automatic sorting" or "always fixed format" decisions.

**Watch For:** Three things:

1. Does the agent name all 8 commands correctly?

2. Does the agent's proposed feature respect the key decisions — no automatic sorting, no optional output, invalid input errors?

3. Does the agent mention the date format constraint when discussing any date-related feature?

If the agent answers all three correctly, the context.md passes the handoff test. A colleague joining the project could receive this file and start a productive session.

> **Credit Note:** This exercise does not consume much of your rate limit — you are asking the agent to read and respond, not to write code. Run it deliberately; pay attention to which decisions the agent correctly recalls and which it misses.

If the agent misses a decision — proposes something that contradicts your constraints — that is signal that context.md needs another line. Add it to the decision summary, run the handoff test again.

## Debrief

The handoff test reveals something important: context.md is not primarily a convenience. It is a verification tool. When the agent correctly describes your project from the context file alone, you know the file is complete. When it misses something, you know the file has a gap.

**A project with a complete context.md can be handed off cleanly.** Not to a specific person or a specific AI — to any developer with access to the files. The context is in the repository, not in anyone's head. The decisions are documented. The state is current. Someone new can pick up the project on day one and contribute on day two.

Update context.md at the end of every session as part of Stage 5 of the loop — same step where you update decisions.md. Two files, two minutes, consistent state going into the next session. The loop's Update stage is where operational discipline lives: if you skip it, the next session starts degraded.

In Chapter 11, multiple agents read the same context.md simultaneously — each running a different part of the loop in parallel. The shared context file is what makes that coordination possible. One file, consistently maintained, is the foundation that parallel work builds on.

*A complete context file means the next session — whoever runs it — starts with everything they need.*

# Chapter: When Credits Run Out — Resilience and Tool Portability

> *Note for continuity pass: insert as Ch 11, renumber current Ch 11–14 → Ch 12–15.*

You are halfway through implementing a feature. The agent is mid-run, editing files, running tests. Then: rate limit. Or the browser crashes. Or you realize you burned through this month's Claude Code credits faster than expected, and the next 24 hours are going to be slow.

In a normal AI-assisted workflow, this is where you lose momentum — sometimes the thread entirely. You wait for the limit to reset, or you open a new session and spend the first fifteen minutes reconstructing context that existed perfectly well before the interruption. You paste fragments from memory. You re-explain decisions you made two weeks ago. Sometimes you just start fresh and accept the cost.

In an AgentFlow workflow, this is a five-minute problem. You open a different tool, paste two files, run the handoff test, and continue. The agent changes. The project state does not.

This chapter is about why that works, what the tool landscape actually looks like at different price points, and how to structure your workflow so that no single tool's credit limit, rate cap, or outage can stop you.

## The Hidden Assumption

Most developers who use AI coding tools make one assumption implicitly: this tool is the tool. The session history, the in-context understanding, the workflow — it all lives in one place. When that place becomes unavailable, the work stops.

AgentFlow challenges that assumption at the architectural level. Everything the AI needs to continue your work is in files: `context.md` holds the project state. `decisions.md` holds the constraints. `sprint-plan.md` holds the scope. Skill files hold your

working conventions. These are plain markdown files in your repository. They are not inside any AI tool. They belong to you.

This means the AI session is interchangeable. Any agent that can read these files and understand markdown can pick up where the last one left off. The agent is an executor, not a memory. The memory is in the files.

**You are not locked into any tool. You are locked into your methodology.** And the methodology runs anywhere.

## The Failure Path: A Crash Without Context

Start a Claude Code session without `context.md` in place. Implement two or three exchanges — maybe you are adding a new command, maybe you are debugging something. The session is going well.

Now imagine the tab crashes. Or the rate limit fires. Or you come back tomorrow and open a new session.

Type:

```
What is the current state of the task manager? How many tests are passi
What was the last thing we were working on?
```

> **What the Agent Will Do:** It will apologize. It has no memory of the previous session. It cannot answer any of those questions. If you had a test count in your head, you are the backup. If you remembered the last decision, good. If not, you are either guessing or reading through the codebase to reconstruct what you already knew.

This is not a hypothetical failure. It is the default behavior of every AI coding tool in existence. Sessions are stateless. The agent's working memory ends when the session ends.

> **Watch For:** How long it takes to reconstruct a productive starting state. Count the exchanges. Count the minutes. This is the cost you are paying every time you do not maintain context.md.

## The Recovery Protocol

When a session ends unexpectedly — crash, rate limit, credits exhausted — do this:

**Step 1: Commit what exists.**

Before switching tools, make sure the current state of the code is committed to git. A half-implemented feature is fine — commit it with a descriptive message: `WIP: add export command, tests failing, see context.md`. The commit is a checkpoint.

**Step 2: Update context.md.**

If you were mid-session, context.md may be slightly stale. Spend sixty seconds updating the "Current Work" section: what was in progress, where it stopped, what the next step is. If the agent was in the middle of something when the session ended, note it.

**Step 3: Open the new tool.**

Start a new session in the replacement tool — whether that is a different AI product, a different account on the same product, or the same tool after the rate limit resets.

**Step 4: Run the handoff prompt.**

Paste the contents of `context.md` and `decisions.md` into the new session. Then send this template exactly:

| **Handoff Prompt — copy this** |
| --- |
| *Paste context.md and decisions.md above, then send:* |
| Read these two files. Then tell me: |
| 1. What is the current state of the project? |
| 2. How many tests are passing? |
| 3. What are we working on right now? |
| 4. What constraints apply to this work? |

> **Watch For:** The agent answering correctly from the files alone, without you adding anything. If it answers all four correctly, you are ready to continue. If it misses something, your context.md has a gap — add the missing information before proceeding. This step doubles as a diagnostic: a passing handoff test means your context files are complete enough to sustain the session.

**Step 5: Continue.**

Issue the next prompt — the same requirement or increment you were working on when the session ended. Reference the WIP commit if relevant. Proceed.

Five steps. Most of them take less than a minute. The handoff test (Step 4) is the only one that requires attention, and it doubles as verification that your context files are complete.

## How to Evaluate AI Tools for Tool Switching

Understanding where to switch requires knowing what to evaluate. The specific tool landscape changes — prices shift, new products appear, credit policies update. What does not change are the properties that make a tool useful for a given type of work. Evaluating tools by their properties gives you a framework that ages better than any point-in-time comparison.

**Four criteria that matter when evaluating AI coding tools for AgentFlow compatibility:**

**Reasoning quality.** How carefully does the tool approach complex code? High-reasoning tools handle edge cases, produce clean variable names, and write error handling that makes sense on first read. They produce cleaner first drafts that require less correction after review. This matters most for sessions involving intricate logic, multi-step validation, or document-and-code work where both outputs need to be correct and clear.

**Session stamina.** How many exchanges can you sustain before credit limits fire? For an implementation sprint — agent running tests, editing multiple files, iterating on output — session stamina is the primary budget constraint at the $20/month tier. A tool with excellent reasoning and limited credits may be less useful as your primary workhorse than one with generous credits and solid (if not exceptional) reasoning.

**Ecosystem integration.** Does the tool connect to software you already use? For developers in Google Workspace, AI integration into Docs, Sheets, and Drive removes friction that other tools require workarounds for. For a developer who works entirely in the terminal, ecosystem integration is less relevant — what matters is what happens in the editor.

**Context handling.** When you paste a long context.md, decisions.md, and a skill file into the session, does the tool hold and respect those constraints throughout? Some tools drift from stated constraints as the session lengthens. You want the decisions you specified at the start of the session applied consistently at exchange thirty. The handoff test (Step 4 of the recovery protocol) is your practical diagnostic for this: if the agent answers all four questions correctly from your context files alone, it is handling context well enough for the task.

**How these criteria cash out at the free and ~$20/month tier:**

*On reasoning quality* — Claude Pro (approximately $20/month) stands out. For complex logic, data transformation, and anything that mixes writing with code (documentation, spec documents, decision logs), it tends to produce cleaner first drafts that are easier to review and maintain. Its document-collaborative capabilities make it unusually strong for any session where the output needs to be both technically correct and well-written. The trade-off: its coding credit budget is more conservative than the alternatives. Reserve it for the sessions where reasoning quality pays off most.

*On session stamina* — ChatGPT Plus ($20/month) leads. For extended implementation sprints, the credit budget goes further than comparable tiers. ChatGPT Plus also offers Codex Web — the ability to kick off agent tasks from a mobile device, start a PR review from your phone,

and return to the result on your laptop. No other $20 tier offers async mobile-to-desktop workflow at this quality level. Use it as your primary workhorse for implementation sprints; use Codex Web for async work between meetings.

*On ecosystem integration* — Gemini Advanced (Google One AI Premium, approximately $20/month) leads — and leads substantially. The tier includes AI integration across Google Workspace (Docs, Sheets, Drive), NotebookLM for cross-document reasoning (ask questions across your spec files, decisions.md, and code simultaneously), and Claude Code for hands-on coding sessions — all running on the same underlying model. For teams already in the Google ecosystem, this integration removes friction that other tools cannot match at this price point.

*On context handling* — this varies by tool and task complexity. Run the handoff test every time you switch. A tool that passes it is ready for the session.

**Entry-level plans and free tiers.** None are unlimited, and none are suited for extended implementation sessions. Lower-cost access is most useful for running the handoff test, short targeted prompts, and evaluating a new tool before committing real budget. If you are on a zero-budget constraint, AgentFlow's incremental structure — one requirement, one scoped session, one verified result — is naturally compatible with those limits. Each session is bounded by design.

## The $200/Month Professional Tier

At $200/month, the math changes.

Claude Max, ChatGPT Pro, and comparable tiers at this price point offer substantially higher rate limits, priority access to the most capable models, and in some cases access to experimental features not available at the lower tier. The practical difference is not just quantity — it is the ability to run extended, uninterrupted sessions that would hit limits on the $20 tier.

For a developer working with AI tools daily on a professional project — not for an hour a week but for several hours a day — $200/month is often the right answer. The productivity gain from uninterrupted sessions, combined with access to the best models, typically pays for itself quickly in reduced friction and higher-quality output.

That said, even at $200/month, the portability principles apply. You still want context.md maintained. You still want to be able to switch if a tool has an outage. You still want your methodology to be the constant, not your vendor.

The difference at $200/month is that the credit pressure largely disappears, and the tool rotation strategy becomes a choice rather than a necessity.

## The Weekend Warrior Strategy

If you are a hobbyist, an open-source contributor, or someone building side projects on a limited budget, the free and $20/month tiers are entirely viable — with the right workflow.

The key insight: AI credits are not scarce if you are disciplined about what you do with them. The failure mode that burns credits fastest is undirected exploration — vague prompts, iterative correction, context reconstruction. AgentFlow's structure — one requirement, one scoped session, one verified result — is inherently credit-efficient. Every prompt has a purpose. Every session ends with committed, tested code.

A practical weekend warrior setup:

• **Primary tool**: Claude Code (Claude subscription) for hands-on coding sessions

• **Backup tool**: ChatGPT Plus ($20/month) for when Claude Code's rate limit fires — especially for longer implementation runs where ChatGPT's credit budget goes further

• **Reasoning tasks**: Claude ($20/month) for complex logic, document work, and anything that benefits from careful analysis

• **Total monthly spend**: $40 for two $20 subscriptions, or one paid subscription plus a lighter backup option

With AgentFlow's context files in place, the tool rotation is invisible to the project. Monday you are in Claude Code. Tuesday the rate limit fires and you switch to ChatGPT. Wednesday you come back to Claude Code. The project has no idea. The context.md knows exactly where you are.

## A Full Tool-Switch Walkthrough

Here is the scenario in concrete terms.

You are in Claude Code. You are halfway through implementing the `export` command — Increment 2 of 3, adding CSV output. The agent has modified `tasks.py` and is mid-run on the tests when Claude Code hits your rate limit for the day.

**Step 1:** Open your terminal. Commit what exists:

```
git add tasks.py tests/test_tasks.py
git commit -m "WIP: export CSV mid-implementation, tests failing on csv
```

**Step 2:** Update context.md — add one line to "Current Work":

```
WIP: Increment 2 of export command (CSV output). tasks.py modified,
tests/test_tasks.py has new tests. Tests failing — csv format not yet
matching expected output. Next: fix output format in export_csv() funct
```

**Step 3:** Open ChatGPT (or Claude, or whichever tool you are rotating to). Start a new conversation.

**Step 4:** Run the handoff prompt:

| **Handoff Prompt — copy this** |
| --- |
| *Paste context.md and decisions.md above, then send:* |
| I'm switching AI tools mid-session due to a rate limit. Here are two files |
| that describe my project — read them carefully before we continue. |
| Tell me: |

1. What are we building?
2. What is the current state of the implementation?
3. What is the next step?
4. What constraints apply to output format and error handling?

**Watch For:** The agent correctly identifying the WIP state, the failing tests, the export command context, and the key decisions (error behavior, output format consistency). If it gets these right from the files, you are ready.

**Step 5:** Continue with the increment:

```
We're mid-implementation on the export command, Increment 2: CSV output
tasks.py has been modified. tests/test_tasks.py has new tests that are
currently failing because the CSV format doesn't match the expected out

The current tasks.json schema is:
{"id": int, "description": str, "done": bool, "priority": str, "due_dat

Fix the export_csv() function in tasks.py so the CSV output format is:
id,description,done,priority,due_date
1,Buy groceries,False,medium,
2,Fix bug,False,high,2026-04-01

Run the tests and confirm they pass.
```

The new agent picks up exactly where Claude Code left off. It has the project context, the constraint history, and the specific current task. The rate limit interruption cost you less than ten minutes.

## Debrief

The tool landscape at $0–$20/month is genuinely capable. The gap between mainstream AI coding subscriptions and professional-tier AI coding is not primarily about model quality — it is about credit volume and feature breadth. The models at the $20 tier are good. The question is how many sessions you get before the limit fires.

AgentFlow changes the calculation in two ways. First, it makes every session more efficient — fewer wasted prompts, cleaner context, less

reconstruction overhead. Second, it makes tool switching seamless — because the project state lives in files, not in any agent's memory.

**The practice is the constant. The tool is the variable.** Any developer who ties their workflow to a single AI tool has a single point of failure — credit limits, outages, model degradation, pricing changes. Any developer who ties their workflow to a methodology has none of those problems. The methodology runs anywhere.

This is the full value of context.md, decisions.md, and the Update stage of the loop. Not just continuity between your own sessions — continuity across tools, across devices, across collaborators, and across whatever the AI tool landscape looks like twelve months from now.

In Chapter 11, you will use this same portability to run two agents in parallel — one in Claude Code, one potentially in a different tool — with a shared context file coordinating their work. The tool-agnostic architecture that enables crash recovery also enables multi-agent coordination. Same principle, different scale.

*Your context file is more durable than any AI tool's memory. Build accordingly.*

# Chapter 11: Multi-Agent Coordination

Until now you have used a single Claude Code session at a time. That is the right default. It keeps the loop tight and the context manageable.

Claude Code also supports subagents with fresh, separate context windows. They are the right tool when you want parallel work without dumping every branch of the task into one conversation. You have not used it yet because you did not need to. After ten chapters, the task manager is feature-complete and you have a mature context file. Now the conditions are right.

To use subagents, either ask Claude directly to delegate part of the work to a subagent, or use `/agents` to create a reusable specialist first. Each subagent runs with fresh context and returns a summary to the main session.

Claude Code subagents are useful for exactly one situation: **genuinely independent work**. If Agent A and Agent B have no overlapping file writes, they can run in parallel without coordinating. Agent A's session does not know what Agent B is doing — which means they cannot negotiate, they cannot wait for each other, and they cannot merge their own changes. That is the human's job.

The discipline from Chapter 6 (explicit scope) is what makes parallelism safe. Without it, two agents working on the same project without shared context about who owns which file will produce conflicts — not immediately visible, but present in the merged result.

## When Parallelism Is Safe

The safety rule for multi-agent work is a single constraint:

**No two agents should write to the same file in the same session.**

Reading the same file is safe. Both agents can read `context.md` and `decisions.md` simultaneously without conflict — they are not modifying the files, just consuming them. Writing is different. If both agents write to `tasks.py`, the second agent's write overwrites or ignores the first, depending on timing and Claude Code's file handling. You will not see an error. You will see one agent's changes missing from the final result.

The way you enforce this rule is the same way you have been preventing scope creep since Chapter 6: explicit "Does NOT" statements in each agent's prompt. "Agent A modifies only tasks.py and test_tasks.py. Agent B creates only COMMANDS.md." Each agent knows its boundary. Neither wanders.

Two additional conditions make parallel work tractable:

• **Independent requirements**: Each task is specifiable without reference to the other. If Agent B needs to know what Agent A built before it can start, they are not independent.

• **Shared context**: Both agents read from the same context.md. They may not know what the other is doing, but they know the same project state and constraints.

When these conditions hold, parallel agents are an efficiency gain. When they do not, serialization is safer.

## The Failure Path: Unscoped Parallel Agents

Stay in your main Claude Code session and explicitly delegate to two subagents with non-overlapping write scopes. Tell one subagent it owns the code change. Tell the other it owns the documentation change.

Send each agent the same vague prompt:

**Subagent 1:**

```
Update the task manager to add filtering to the list command.
```

**Subagent 2:**

```
Update the task manager to improve its output formatting.
```

> **What the Agents Will Do:** Both agents will read `tasks.py`. Both will make changes. Agent 1 may add a `--filter` flag. Agent 2 may reformat the `list` output. Both changes touch the same function. When Agent 2 writes its changes to `tasks.py`, it may overwrite Agent 1's filter flag — or Agent 1 may not have finished yet, writing on top of Agent 2.

> **Watch For:** Open `tasks.py` after both agents indicate they are done. Check whether both changes are present. One or both may be missing. Run:

```
pytest tests/ -v
```

> **Watch For:** Unexpected test failures. The file may be in a state that neither agent intended — one agent's output plus a fragment of the

> other's.

Stop here. This is the conflict that scope discipline prevents.

## The Success Path: Scoped Parallel Agents

Two tasks, genuinely independent:

• **Agent A**: Add `--priority` filter to `list` (e.g., `list --priority high` shows only high-priority tasks). Modifies `tasks.py` and `test_tasks.py`.

• **Agent B**: Create `COMMANDS.md` — a reference document of all commands with usage and examples. Creates a new file only.

Stay in the main session and create two subagent tasks.

**Send Agent A this prompt:**

```
[paste your full context.md here]

Task for Agent A: Add --priority filter to list command.

SCOPE: Modify ONLY tasks.py and test_tasks.py. Do not create or modify

Feature: --priority filter on list
Input: python tasks.py list --priority high (or medium or low)
Output: Only tasks matching that priority, same format as regular list.
Default (no --priority flag): unchanged — shows all tasks.

Edge cases:
- Invalid priority value: print error, exit non-zero (consistent with a
- --priority can be combined with --format json
- If no tasks match the filter: No tasks with priority 'high'.

Add exactly 1 new test:
- list --priority high shows only high-priority tasks

All 15 existing tests must still pass.
```

**Send Agent B this prompt:**

```
[paste your full context.md here]

Task for Agent B: Create COMMANDS.md reference documentation.

SCOPE: Create ONLY a new file named COMMANDS.md. Do not modify tasks.py

Write a concise command reference for the task manager covering all 8 c
```

```
add, list, done, delete, overdue, search, stats, clear

For each command include:
- Command name and usage syntax
- Brief description (one sentence)
- Example with expected output

Format: markdown, one section per command. Keep it short — this is a re
```

> **What the Agents Will Do:** Agent A will add the `--priority` flag to the argument parser, add a filter step to the `list` handler, and write one new test. Agent B will write COMMANDS.md from scratch, reading context.md to get the command list and schema. Neither agent touches the other's files.

> **Watch For:** Both subagents running in parallel. You do not need to alternate between them; let them run, then review the summaries and changed files.

When both agents indicate completion, verify each independently.

**Verify Agent A:**

First confirm Agent A stayed in scope — open `tasks.py` and check that the formatting of existing functions is unchanged from what you expect. Agent B was not supposed to touch this file, but a quick sanity check takes ten seconds.

```
pytest tests/ -v
```

> **Watch For:** 16 tests passing. Then manually:

```
python tasks.py add "Buy groceries" --priority medium
python tasks.py add "Fix critical bug" --priority high
python tasks.py list --priority high
# Expected: Only [2] [high] Fix critical bug

python tasks.py list --priority urgent
# Expected: Error message, non-zero exit
```

**Verify Agent B:**

Open `COMMANDS.md` in your editor. Read it. Check:

• All 8 commands covered?

• Usage syntax matches actual commands?

• Examples are realistic?

• Nothing in the document contradicts a decision in decisions.md?

## The Merge Review

Both agents produced correct output independently. Now you review them as a pair before the session is complete.

The merge review has one question that single-agent review does not: **do the two outputs conflict?**

In this case they should not — one modified code, one created documentation. But check:

• Does COMMANDS.md document the `--priority` filter on `list`? It should, since Agent B had context.md and the filter now exists in the project. If Agent B finished before Agent A and did not know about the filter, the documentation is already incomplete. Note this for the Stage 5 update.

• Do the tests Agent A wrote conflict with anything in the existing test suite? Run the full suite one more time to confirm.

Update context.md to reflect the new state:

```
- Tests: 16 passing
- Commands: add (--priority, --due), list (--format json, --priority),
```

Add a decisions.md entry:

```
## Priority filter
- **Decision**: list --priority filters by exact value; invalid priorit
- **Reason**: Consistent with add --priority validation (Ch 2 decision)
- **Rules out**: Fuzzy match, multiple priority values, silent filterin
```

If COMMANDS.md did not include the `--priority` filter, return to a single Claude Code session and issue a targeted correction:

```
Add --priority to the list command's entry in COMMANDS.md.
Usage: python tasks.py list --priority high|medium|low
Description: Filter tasks by priority. Invalid value prints error.
```

## Debrief

Two agents, two tasks, one session. The throughput advantage is real — both tasks completed in the time one would have taken. The coordination cost is also real: you had to specify scope explicitly for each agent, monitor both sessions, and review both outputs as a pair before updating context.md.

**Parallel agents multiply throughput when scopes are clean. They multiply problems when they are not.** The discipline that makes this work is not multi-agent-specific — it is the same scope discipline from Chapter 6 and the same review discipline from Chapter 4. The only thing that changes at scale is that you apply those disciplines to two agents at once instead of one.

Claude Code subagents become useful when you have a backlog of independent tasks. In Chapter 12, you will see how to identify those tasks systematically — which is what sprint planning is for. A sprint is not just a time box. It is an analysis of what work is truly independent, so you know which tasks can run in parallel and which must serialize.

*Parallelism is a throughput multiplier, not a coordination shortcut. The discipline still runs once per agent.*

# Chapter 12: The Sprint Cadence

You have been running sprints without calling them that.

Look at how this book is structured. Each chapter is a unit of work with a goal, a scenario, a definition of done, and a commit at the end. You planned chapters before writing them. You reviewed them before committing. You updated the sprint plan after each one. That is a sprint cadence — a recurring cycle of plan, execute, review, repeat.

Now apply it to software. A session is one loop: Start, Implement, Verify, Review, Update. A sprint is several sessions grouped around a coherent goal. The sprint cadence is the rhythm that governs which sessions run, in what order, and when the sprint is done.

**Without a cadence, you optimize sessions but not the project.** Each session is locally fine — good requirement, clean implementation, tests passing. But there is no global view: features accumulate without a theme, decisions that should have been made together get made in isolation, and the backlog grows unchecked because nothing ever closes.

The sprint cadence closes things. Plan the sprint, execute the sprint, review the sprint, update context. Then plan the next one.

## What Sprint Planning Produces

A sprint plan is not a Gantt chart. It is a one-page analysis that answers five questions before you open Claude Code:

**1. What is the sprint goal?**

One sentence. Not a list of features — the reason this set of features belongs together. "Export capabilities" is a goal. "Add --format csv, export command, and fix the priority filter bug" is a task list, not a goal.

**2. What features are in scope?**

List the specific features. Each one maps to a requirement you will write before implementing. Anything not on the list is out of scope for this sprint — it goes to the backlog.

**3. What are the dependencies?**

Which features depend on other features being done first? A `--format csv` flag on `list` depends on nothing. An `export` command that produces the same format as `--format csv` depends on the CSV format decision being made first — but not necessarily on `--format csv` being fully built.

**4. Which tasks can run in parallel?**

If two features have no shared file writes and no dependency between them, they are parallelism candidates. Identify them explicitly. This is the sprint planning analysis that makes Chapter 11's Claude Code

subagents useful at scale.

**5. What is "done" for this sprint?**

State the exit criteria before starting. Not "it feels finished" — a specific count of tests, a specific set of commands, a specific update to context.md. If you cannot state done before starting, the sprint scope is not clear enough.

## The Failure Path: No Plan

The task manager needs export capabilities. Two features: `--format csv` on `list`, and an `export` command that writes tasks to a CSV file.

Jump straight in. Ask Claude to use two subagents immediately, giving each only a rough description:

**Agent A:**

```
Add CSV output to the task manager.
```

**Agent B:**

```
Add an export command to the task manager.
```

> **What the Agents Will Do:** Agent A may add `--format csv` to `list`. It will choose a CSV format — fields, order, header row. Agent B will add an `export` command. It will also choose a CSV format. They will almost certainly choose differently, because neither agent knows what the other decided.

> **Watch For:** After both agents finish, compare their CSV output formats. Run:

```
python tasks.py list --format csv
python tasks.py export tasks.csv && cat tasks.csv
```

> **Watch For:** Different field orders, different header rows, or different handling of null fields (`due_date: null` becomes an empty string? The word "null"? Omitted?). Two agents, two CSV formats, one broken sprint.

Stop here. The conflict is in the format decision — which should have been made once, in the sprint plan, before either agent started.

## Writing the Sprint Plan

Before opening Claude Code, write a `sprint-plan.md` in your project using your editor or `touch sprint-plan.md`. Use this structure:

```
# Sprint: Export Capabilities

## Goal
Give users two ways to get task data out of the system in a consistent

## Features
| Feature | Description | Test count |
|---------|-------------|------------|
| --format csv | Add CSV option to list command | +1 (total: 17) |
| export command | Write all tasks to a named CSV file | +1 (total: 18)

## Dependency analysis
- CSV format (fields, header, null handling) must be decided before eit
- --format csv and export are otherwise independent — no shared file wr
- Both can run in parallel after the format decision is made.

## CSV format decision
- Header row: id,description,done,priority,due_date
- Fields: all fields, always (consistent with JSON format decision)
- Null due_date: empty string in CSV
- Invalid format value: print error, exit non-zero (consistent with exi

## Parallelism plan
- Agent A: --format csv on list. Modifies tasks.py and test_tasks.py or
- Agent B: export command. Modifies tasks.py and test_tasks.py only.
- Note: both agents modify the same files. Run sequentially, not in par
  (The format decision is shared; same-file writes require serializatio

## Definition of done
- 18 tests passing
- list --format csv and export produce identical CSV format
- context.md updated
- decisions.md updated with CSV format decision
```

Notice what the dependency analysis revealed: both agents would modify `tasks.py` and `test_tasks.py`. They cannot run in parallel. The sprint plan caught a would-be conflict before any code was

written. Run Agent A first; then Agent B reads the completed implementation as context. Because both tasks write to the same files, use one Claude Code session at a time rather than parallel subagents for this sprint.

Save `sprint-plan.md` with Cmd+S or Ctrl+S.

## Executing the Sprint

### Session 1: --format csv (Agent A)

Start a fresh Claude Code session. Give the agent context and the requirement:

```
[paste your full context.md here]

Sprint context: Adding CSV export capabilities. See sprint-plan.md for

Feature: --format csv on list
Input: python tasks.py list --format csv
Output: CSV with header: id,description,done,priority,due_date
        One row per task; null due_date = empty string; done = true/fal

Does NOT:
- Sort output
- Filter output
- Change the default (no --format flag) behavior

Add exactly 1 new test:
- list --format csv produces correct CSV output with header row

All 16 existing tests must still pass.
```

> **What the Agent Will Do:** The agent will add CSV to the `--format` flag handler in the `list` command. It will format the output consistently with the JSON format: all fields, all tasks, no filtering.

Verify:

```
pytest tests/ -v
```

> **Watch For:** 17 tests passing. Then manually verify:

```
python tasks.py add "Buy groceries" --priority medium
python tasks.py add "Fix bug" --priority high --due 2026-04-01
```

```
python tasks.py list --format csv
# Expected:
# id,description,done,priority,due_date
# 1,Buy groceries,false,medium,
# 2,Fix bug,false,high,2026-04-01
```

### Session 2: export command (Agent B)

Start a fresh Claude Code session. Agent B runs after Agent A is verified:

```
[paste your full context.md here]

Sprint context: --format csv is already implemented and produces CSV wi
header: id,description,done,priority,due_date. Null due_date = empty st

Feature: export command
Input: python tasks.py export <filename>
Output:
- Writes all tasks to the specified file in the same CSV format as list
- Prints: Exported N tasks to <filename>.
- If no tasks: No tasks to export.
- If file already exists: overwrite without warning

Does NOT:
- Filter by priority, done status, or due date
- Ask for confirmation
- Support different output formats

Add exactly 1 new test:
- export writes correct CSV file with expected content

All 17 existing tests must still pass.
```

> **What the Agent Will Do:** The agent will add an `export` command that reuses the CSV formatting logic from `list --format csv`. Because the sprint plan defined the format explicitly and Agent A already implemented it, Agent B has a concrete reference for the exact output format.

Verify:

```
pytest tests/ -v
```

> **Watch For:** 18 tests passing. Then manually verify consistency:

```
python tasks.py export tasks-export.csv
cat tasks-export.csv
# Expected: same format as list --format csv — same header, same fields
```

```
# (Or open tasks-export.csv in your editor to view the contents)
```

> **Watch For:** `list --format csv` and `export tasks.csv` producing identical format for the same task data. If they differ, the sprint goal is not achieved — two formats is the exact problem the sprint plan was written to prevent.

## Sprint Review

The sprint is done when:

• 18 tests pass ■

• `list --format csv` and `export` produce identical CSV format ■

• context.md is updated ■

• decisions.md has a new CSV format entry ■

• sprint-plan.md is marked complete or archived ■

Update context.md:

```
- Tests: 18 passing
- Commands: add (--priority, --due), list (--format json/csv, --priorit
```

Add to decisions.md:

```
## CSV format
- **Decision**: CSV header is id,description,done,priority,due_date. Nu
- **Reason**: Consistent with JSON format (all fields, always). Matches
- **Rules out**: Optional fields, different field order, null as "null"
```

Look at the backlog — features excluded from this sprint that were named and parked. What is the most coherent next sprint goal? That is the starting question for sprint planning, next cycle.

## Debrief

The sprint plan took fifteen minutes. It prevented a format conflict that would have taken longer than fifteen minutes to untangle. It also caught the parallelism trap — two agents that looked independent but were not,

because they shared output files.

**Sprint planning is not overhead. It is front-loaded problem discovery.** The dependency analysis and format decision happened on paper, not in the middle of a debugging session. The definition of done made the sprint review a five-minute check rather than an open-ended question.

The cadence scales. A solo developer running one sprint per week accumulates coherent features, a living context.md, and a backlog that reflects deliberate deferral rather than forgotten ideas. A small team can run the same cycle — one shared context file, shared sprint plan, parallel agents when scopes are clean.

In Chapter 13, the cadence runs with reduced human involvement — higher autonomy modes, mature skills, complete context files. The discipline does not change. What changes is how much of each loop stage the AI runs versus the human.

*A sprint is the unit of coherence. Plan before you execute. Review before you close.*

# Chapter 13: Autonomy at Scale

Mode 1, Mode 2, Mode 3. You have the options. The question Chapter 9 left open is: how do you choose?

The temptation is to pick one mode and stick with it. Always Mode 1 if you are cautious. Always Mode 3 if you are optimistic. Neither is right. The correct mode is a function of the task — specifically, of three factors that change from task to task and from project to project.

**Factor 1: Task definition.** How well-specified is the requirement? A flag with explicit behavior, exact output format, and a "Does NOT" list is tightly defined. A schema change that touches stored data and backward compatibility is inherently riskier — even with a good requirement, the implementation has more ways to go wrong.

**Factor 2: Context maturity.** Is context.md current? Is decisions.md complete? Does the agent have everything it needs to make correct choices without asking? A mature context means the agent starts informed. A stale or thin context means the agent fills gaps with guesses.

**Factor 3: Test coverage.** If the agent makes a mistake, will the tests catch it? A high-coverage test suite is a safety net for high-autonomy sessions. If tests only cover happy paths, Mode 3 will eventually produce an error that passes all tests and ships to users.

All three factors high → Mode 3 is appropriate. Any one low → dial back. This is the calibration, not a rule.

## The Calibration Heuristic

Before each session, answer three questions:

Map the result to a mode:

• **All three high** → Mode 3 (full loop, human reviews output)

• **Two of three high** → Mode 2 (implement + verify auto, human reviews diff)

• **Any one low** → Mode 1 (human at every stage)

• **Multiple low** → Mode 1 with extra care; consider breaking the task into smaller pieces first

This is a heuristic, not an algorithm. A schema change almost always pushes toward Mode 1 regardless of the other factors, because the cost of a bad schema migration is high and tests rarely cover the full migration edge case space.

## Mode 2 in Practice: --done Filter

**Task**: Add `--done` filter to `list`. Shows only completed tasks. One new test. 19 tests total.

**Calibration:**

• Task definition: tight. Existing `--priority` filter is the exact pattern; same output format.

• Context maturity: context.md is current (18 tests, all commands listed).

• Test coverage: the `--priority` filter test covers the filter pattern; this is the same logic.

**Verdict**: Mode 2. Let the agent implement and run tests. Review the diff at Stage 4.

Start a fresh Claude Code session. Paste your context.md and add the requirement:

```
[paste your full context.md here]

Mode: Implement and verify. Run tests after implementing. I will reviev

Feature: --done filter on list
Input: python tasks.py list --done (shows only done=true tasks)
Output: Same format as list; only completed tasks shown.
        If no completed tasks: No completed tasks.

Edge cases:
- --done can combine with --priority and --format flags
- --done with --priority: show completed tasks with that priority
- Invalid combination still respects error behavior

Does NOT:
- Change default list behavior
- Add any other filter flags

Add exactly 1 new test:
- list --done shows only completed tasks

All 18 existing tests must still pass.
```

**What the Agent Will Do:** Because you specified "Implement and verify," the agent will interpret this as an instruction to proceed through Stages 2 and 3 without pausing — it will build the feature, run the test suite, and report back the results without stopping for confirmation.

**Watch For:** The agent's test report. It should show 19 tests passing. If it reports failures, it will describe them — issue a targeted

correction.

When the agent reports success, run the tests yourself to confirm:

```
pytest tests/ -v
```

**Watch For:** 19 tests passing. Then Stage 4 — read the diff. Open `tasks.py` and find the `--done` filter implementation. Is it the same pattern as `--priority`? Are there any extra flags or conditions you did not specify?

Update context.md: `Tests: 19 passing`.

## Mode 1 in Practice: Tag Field

**Task**: Add optional `tag` field to tasks. Tasks can be tagged with a string (e.g., "work", "personal"). Schema change.

**Calibration:**

• Task definition: reasonably tight, but it is a schema change. The backward compatibility requirement (existing tasks with no `tag` field should display without tag — same as missing `priority` from Ch 3) adds implementation complexity.

• Context maturity: good.

• Test coverage: the missing-field default test from Ch 3 covers the pattern, but a schema change expands the test surface.

**Verdict**: Mode 1. Human present at every stage. Review the schema change before proceeding.

### Stage 1: Start and Requirement

Write the requirement before opening a session:

*Feature:* Tag field

*Input:*

• `add "description" --tag work` stores tag

• `add "description"` (no flag) stores `"tag": null`

*Output:*

• `list` shows `[tag: tagname]` after description when tag present: `[1] [medium] Buy milk [tag: personal]`

• `list` shows nothing extra when `tag: null`

• Completed tasks: `[1] [medium] [done] Buy milk [tag: personal]`

• Due date + tag: `[1] [medium] Buy milk [due: 2026-04-01] [tag: personal]`

*Edge cases:*

• Existing tasks with no `tag` field: display without tag (no error, no migration)

*Scope boundary:*

• Does NOT filter by tag (not yet)

• Does NOT search by tag

• No tag editing after creation

### Stage 2: Implement

Start a fresh Claude Code session. Paste context.md and the requirement. Add explicitly:

```
Mode: Stage by stage. After implementing, stop before running tests.
I will review the schema change before proceeding.
```

**What the Agent Will Do:** The agent will add `--tag` to the `add` command, update the task object construction, and modify the `list` display. It will stop before running tests because you specified Mode 1.

**Watch For:** The agent's proposed schema change. Before moving on, open `tasks.py` and answer:

| |
|---|
| 1. Is `tag` stored as `null` when not provided (not as an empty string or omitted entirely)? |
| 2. Does the `list` display use `.get("tag")` or equivalent — not `task["tag"]` — to handle missing fields safely? |
| 3. Is the display order correct: description, then due date, then tag? |

If all three are correct, send a follow-up message in the same session: "Everything looks good — please run the test suite now."

### Stage 3: Verify

Tell the agent to run the tests (or run them yourself):

```
pytest tests/ -v
```

| |
|---|
| **Watch For:** 20 tests passing. Then manually verify backward compatibility: |

```
# Open tasks.json in your editor
# Find an existing task entry and remove the "tag" field entirely
# Save (Cmd+S or Ctrl+S), then:
python tasks.py list
# Expected: the task displays without any tag — no error, no "[tag: nul
```

### Stage 4: Review

Read the diff. Specifically: does `list` handle `due_date` before `tag` in the output? Is the display consistent with the requirement's examples?

### Stage 5: Update

Add to decisions.md:

```
## Tag field
- **Decision**: tag is stored as null when not provided; displayed as |
- **Reason**: Consistent with due_date and priority patterns — null = r
- **Rules out**: Empty string for no tag, auto-migration of existing ta
```

Update context.md: `Tests: 20 passing`. Add `tag` to the schema line.

## Mode 3: When and Why

Mode 3 — the full loop, all five stages, minimal human checkpoints — is appropriate when all three calibration factors are high and the task is low-risk.

A documentation update is a good example. Ask the agent to update COMMANDS.md to include the `--done` filter and the `--tag` flag you just added.

Open a new agent session. Give it context.md and the instruction:

```
[paste your full context.md here]

Update COMMANDS.md to document two new options:
1. list --done: shows only completed tasks
2. add --tag <name>: optional tag for a task; list shows [tag: name] wh

Mode: Full loop. Complete the update, verify the file looks correct,
and report back when done.
```

**What the Agent Will Do:** The agent will read COMMANDS.md, add the two new entries, review its own output for accuracy, and report completion. You review the final COMMANDS.md.

**Watch For:** The entries matching your requirements exactly — correct syntax, correct output format shown. If any entry is wrong, issue a targeted correction. The bar for Mode 3 on documentation is that the output is reviewable in under two minutes.

**The Mode 3 constraint:** Documentation is safe for Mode 3 because no test can fail and no stored data changes. For code changes, Mode 3 requires test coverage that would catch any mistake the agent might make. If your test suite does not test the behavior the agent might break, Mode 3 shifts the risk to the user.

**Invisible drift** is the failure mode specific to Mode 3: the agent runs all five stages, all tests pass, and the implementation quietly violates a decision from decisions.md that no test enforces. This is why decisions.md must be in the context and why the calibration factors matter. Mode 3 is not "let the AI do whatever it wants." It is "I trust that the context, the tests, and the requirement are complete enough that the AI can run the loop safely."

## Debrief

Three tasks, three modes. The modes did not change the discipline — the requirement was still written before implementation, the tests still ran, the diff was still read. What changed was the distribution of work between human and AI across the five stages.

**Autonomy is earned, not granted.** The path to Mode 3 is:

• Build complete context.md and decisions.md (Chapters 7 and 10)

• Write requirements that leave no room for guessing (Chapter 2)

• Build a test suite that covers error cases, not just happy paths (Chapter 3)

• Run enough Mode 1 sessions to develop confidence in the agent's judgment on this project

When all four are in place, Mode 3 is safe for routine, well-understood tasks. It is not safe for schema changes, new commands, or anything that expands the project's decision surface.

Chapter 14 brings all of this together — not as a new technique, but as a synthesis. The task manager, the loop, the skills, the context files, the sprint cadence, the autonomy modes — one system, working as a whole.

*Autonomy is calibrated, not maximized. The right mode is the one that matches the task's risk to your safety net's coverage.*

# Chapter 14: Putting It All Together

This is what you have built across fourteen chapters:

A **development loop** with five stages: Start, Implement, Verify, Review, Update. Every session runs the loop. Every stage has a discipline behind it.

A **skill** (`start-session.md`) that encodes the Start stage, so each session begins informed rather than reconstructed from memory.

A **context file** (`context.md`) that holds project state separately from session instructions, updated at the end of every session, readable by any agent or developer picking up the project.

A **decision log** (`decisions.md`) that records what was decided, why, and what each decision rules out — the answer to "why does this code work this way?"

A **sprint cadence**: plan, execute, review, repeat. Each sprint has a goal, a dependency analysis, a parallelism decision, and an explicit definition of done.

**Autonomy calibration**: three factors (task definition, context maturity, test coverage) that determine whether the session runs Mode 1, 2, or 3.

**Multi-agent coordination** via Claude Code subagents: parallel agents when scopes are clean, serialization when they are not, human merge review before any session closes.

That is AgentFlow. Not a tool — a practice. These pieces work together, which is why the chapters were sequenced the way they were. Skills are only useful when you have a loop to invoke them in. Context files only matter when sessions are stateless. The sprint cadence only works when you have requirements discipline behind it.

## Choosing Your Next Feature

Look at your backlog — the features you explicitly excluded from previous sprints. Some of them are named. Some are implied. Here are four candidates from the task manager's natural next steps:

**Option A: Tag filter** — `list --tag work` shows only tasks with `--tag work`. Low risk. Follows the exact same pattern as `--priority` filter (Ch 11) and `--done` filter (Ch 13). Good Mode 2 candidate.

**Option B: Bulk done** — `done 1 2 3` marks multiple tasks done in one command. Medium risk. New argument parsing pattern (multiple IDs), but no schema change. Mode 1 or 2 depending on your test suite's ID-handling coverage.

**Option C: Edit description** — `edit 1 --description "new text"` changes a task's description. Medium risk. Mutation of existing data — no undo. The decisions.md precedent is that we do not ask for confirmation (see `clear`), but that decision needs to be applied consciously here. Mode 1.

**Option D: Import from CSV** — `import file.csv` reads a CSV file and adds tasks. Higher risk. File parsing, error handling for malformed files, ID assignment for imported tasks. Mode 1 with careful increments.

Choose the one that interests you. The system works on all of them. Avoid choosing something that would require major refactoring of `tasks.py` — the synthesis session should demonstrate the methodology working smoothly, not debugging a structural change. If you are unsure, Option A is the safest demonstration.

## Running the Full Session

This is a checklist, not a tutorial. You have done every step before. The point of this chapter is to do them all in sequence, consciously, as one system.

### Step 1: Write the sprint plan

Before opening Claude Code, create or update `sprint-plan.md`. Answer the five questions:

• What is the sprint goal? (one sentence)

• What features are in scope?

• What are the dependencies?

• Which tasks can run in parallel?

• What is "done" for this sprint?

Reference: Chapter 12.

### Step 2: Calibrate the autonomy mode

Apply the three-factor heuristic to your chosen feature:

• Task definition: tight requirement with existing pattern → higher autonomy. New behavior or schema → lower.

• Context maturity: is context.md current with your 20-test, 10-command state?

• Test coverage: do your tests cover the error cases this feature might produce?

Assign Mode 1, 2, or 3 before writing the first prompt.

Reference: Chapter 13.

### Step 3: Write the requirement

Before opening a new agent session, write the requirement for your feature:

• Specific input (exact command syntax)

• Specific output (exact format)

• Edge cases (what happens at the boundaries)

• Scope boundary (at least two "Does NOT" statements)

Reference: Chapter 2.

### Step 4: Invoke start-session.md

Start a fresh Claude Code session. Reference `@start-session.md` at the top of your first message, or paste its contents if you prefer. Append the requirement below it. Specify the autonomy mode explicitly — add a line like "Mode: Implement and verify" or "Mode: Stage by stage" so the agent knows whether to pause.

### Step 5: Implement and verify

Let the agent work. If Mode 2 or 3, it will run tests automatically. If Mode 1, review the diff before telling it to proceed with tests.

```
pytest tests/ -v
```

> **Watch For:** All existing tests still passing, plus the new test you specified. If tests fail, issue a targeted correction identifying the specific failing test and the expected behavior.

### Step 6: Review the diff

Open `tasks.py` and read what changed. Ask the five questions from Chapter 4:

• Can you explain what every changed function does in one sentence?

• Does each function do one thing?

• Are variable names descriptive?

• Are error cases handled?

• Is there anything you would not want to explain to a colleague?

If yes to any, issue a targeted correction before moving on.

### Step 7: Update artifacts

Add a decisions.md entry for any new decision made. Update context.md with the new test count, new commands, updated schema if applicable. If you used sprint-plan.md, mark the sprint complete.

Close the agent session.

### Step 8: Verify the handoff

Open a new agent session with only context.md. Ask it to describe the current state of the project. If the answer is accurate — test count, commands, key decisions — the session closes cleanly and the next one opens informed.

## What You Now Have

The task manager started in Chapter 1 as a four-command CLI: `add`, `list`, `done`, `delete`. It stores tasks in a JSON file. A developer wrote a planning brief before building it, which is why it was built correctly the first time.

Fourteen chapters later:

**Code:** 20+ tests. 10+ commands. Validated input. Explicit error handling. Backward compatibility across schema changes. A codebase you can explain to a colleague because you reviewed every increment before it landed.

**Artifacts:** `context.md` — operational snapshot of the project, accurate as of the last session. `decisions.md` — the full decision log, from priority validation in Chapter 2 to CSV format in Chapter 12. `COMMANDS.md` — user-facing reference documentation. `start-session.md` — the skill that makes every session start in thirty seconds instead of five minutes. `sprint-plan.md` — the record of how the last sprint was planned and what was deferred.

**Process:** A loop you can run on autopilot. A calibration heuristic for autonomy. A sprint structure that prevents feature incoherence. A handoff test that verifies the artifacts are complete.

Hand this project to a colleague right now. They open context.md. They read the commands, the schema, the key decisions. They open decisions.md to understand why the search is case-insensitive and why the date format is YYYY-MM-DD. They run the tests — 20+ pass. They pick a feature from the backlog and start a session. No onboarding meeting. No "let me walk you through the code." The artifacts do the talking.

That is the outcome. Not a faster way to write code — a disciplined way to build projects that outlast any single session.

## The Closing Argument

Here is what this book claimed at the start, and what you have now seen demonstrated: **software engineering discipline did not disappear in**

**the AI era. It moved up a level.**

Before AI coding tools, discipline meant: write requirements before you build, test your code, review the diff, iterate deliberately, manage scope, document decisions. You did those things by doing the implementation — the discipline was embedded in the craft.

Now the AI does much of the implementation. Which means the discipline has to happen somewhere else — in the requirement before the session, in the test cases before the code, in the diff review after the implementation, in the decisions.md entry after the feature lands. The work of discipline did not go away. It migrated from implementation to specification.

This is why a developer who relies on AI without the discipline produces code that accumulates — features without requirements, tests that pass but do not cover, implementations that contradict each other session to session. And why a developer who applies the discipline gets something that compounds — each session building on documented decisions, each sprint advancing a coherent goal, each handoff cheaper than the last.

The AI is a fast implementer. You are the permanent owner. Those roles have different jobs. The fast implementer produces working code. The permanent owner produces maintainable projects. This book was about the permanent owner's job — which turns out to be the same job it always was, done with different tools.

## What Comes Next

The methodology scales. Everything in this book applies to a team project — shared context.md, shared decisions.md, parallel agents with explicit scope boundaries, sprint plans that coordinate multiple developers. The artifacts are the coordination layer. The loop is the shared process. The autonomy modes calibrate trust across team members the same way they calibrate trust in a solo session.

Your task manager is feature-complete for this book's purposes. Your backlog is not. The next sprint is yours to plan.

Pick a feature. Write the requirement. Run the loop.

*The discipline did not change. The medium did.*

# Conclusion

You have run fourteen chapters of deliberate practice. You wrote requirements before you built. You designed test cases from those requirements before you touched the AI. You reviewed code you did not write as if you were going to maintain it for the next decade. You scoped features explicitly, documented the decisions that are not in the code, and organized work into verified increments.

If you followed the project thread, you have a working task manager CLI with more than ten commands, more than twenty tests, and a decision log that captures six chapters of product choices. You did not just read about these disciplines. You applied them.

The thesis of this book was: software engineering discipline does not disappear in the AI era. It moves up a level.

Fourteen chapters later, you have evidence for that claim.

## What You Can Do Now

Before this book, you probably knew the disciplines existed. Most experienced developers do. What changed is that you now know how to apply them in an AI-assisted workflow — specifically, in the workflow where the AI is doing the implementation and you are doing everything else.

Concretely:

**You can start a project correctly.** Before you open any AI tool, you write a planning brief. Five questions, one file, five minutes. The AI's

first response will reflect your judgment, not its assumptions.

**You can specify what "done" means before the AI starts.** A verifiable requirement has specific input, specific output, defined edge cases, and an explicit scope boundary. You know how to write one.

**You can test AI output against your requirements, not the AI's implementation.** You design the test cases. You let the AI write the test code. The distinction matters because it is the only way to catch the bugs the AI cannot see in its own work.

**You can review code you did not write.** Five questions. Find the understanding gaps before they become the next chapter's technical debt. Issue targeted fixes, verify with the test suite.

**You can manage scope at the prompt level.** "Does NOT" is a first-class part of every requirement. You can name what you are not building and force the AI to respect that boundary.

**You can maintain a decision log that travels with the project.** Any new session — your own, a colleague's, a different AI — starts with context. The decisions that are not in the code are in the file.

**You can run a full AgentFlow sprint.** You know the five stages. You know how to write a skill, maintain a context file, set your autonomy mode, coordinate parallel agents, and close the loop with an update pass.

These are not theoretical capabilities. You have exercised every one of them.

## The Disciplines, Restated

Across the fourteen chapters, the same argument appeared in seven different forms:

*The AI executes. You decide what to execute.* Planning is how you make that decision before the AI starts. Requirements are how you make that decision verifiable. Tests are how you confirm the decision was implemented. Review is how you take ownership of the result. Iteration is how you stay in control across multiple changes. Scope is how you

prevent the AI from expanding the decision on your behalf. Documentation is how you make the decision durable.

The AgentFlow system in Part 2 did not add new disciplines. It gave the Part 1 disciplines a structure that persists across sessions, scales to parallel agents, and can be handed off without loss of context. The skill system is the planning discipline, encoded. The context file is the decision log, formalized. The sprint cadence is the iteration plan, made repeatable. The autonomy modes are the review discipline, calibrated.

Every Part 2 mechanism maps back to a Part 1 pain point. That is not a coincidence. It is the point.

## AgentFlow Beyond This Book

This book applied AgentFlow to solo development on a single project with one AI tool. That is the simplest case. The system scales further.

**Multiple developers.** AgentFlow's artifacts are designed to be shared. A context file read by two developers gives both the same starting point. A decision log prevents conflicting choices in different branches. A skill library encodes a team's working conventions so that any member — or any AI — produces output that matches the team's standards.

**Multiple AI tools.** The methodology is not tied to Claude Code. The same context file, skill system, and sprint structure work with other coding harnesses too. When you switch tools, the decisions travel with you.

**Multiple agents in parallel.** Chapter 11 introduced parallel agent coordination at a basic level. The same discipline scales: tighter scope boundaries for each agent, a merge review step, a shared context file that resolves conflicts. The coordination overhead is real, but so is the speed. The key is that the overhead is disciplined — not chaotic.

**Other domains.** The project thread in this book was a CLI tool. The disciplines apply to web applications, data pipelines, infrastructure-as-code, and any other domain where you are specifying what to build and the AI is building it. The artifact formats may change.

The underlying structure — requirement → test → implement → review → document — does not.

## What This Book Does Not Cover

Honesty about scope is a discipline too.

**Production infrastructure.** This book taught you to develop with discipline. It did not cover the engineering required to deploy and operate what you build — CI/CD pipelines, monitoring, scaling, security review. Those are distinct domains with their own practices.

**Advanced multi-agent patterns.** Chapter 11 covered the basics of parallel agent coordination. The full problem — agent-to-agent communication protocols, shared state management, failure recovery across parallel runs — is substantially more complex and still an evolving area.

**AI-generated tests as a QA strategy.** This book argued that you should design your tests from your requirements and let the AI write the test code. It did not address the broader question of when AI-generated tests are or are not sufficient for production-quality software. That question does not have a clean answer yet.

**Model-specific behavior.** The techniques in this book are designed to work regardless of which AI model is doing the implementation. But different models have different strengths, and tuning your workflow to a specific model is a real optimization that this book deliberately sidestepped.

These are not gaps to apologize for. They are the scope boundary of this book, stated explicitly — as every scope boundary should be.

## The Tools Will Change

Claude Code will be updated. The UI will change. The models behind it will improve. Some of the specific workflows described in this book will become outdated.

The disciplines will not.

Planning before prompting is not a workaround for AI limitations. It is good engineering. Writing verifiable requirements is not a response to AI inaccuracy. It is how you define done for any implementation effort. Designing tests from requirements is not a defense against AI-generated bugs. It is the correct way to build a test suite. Reviewing code for understanding is not skepticism about AI quality. It is what it means to be the permanent maintainer.

These practices predate AI coding tools by decades. They will outlast whatever tools replace Claude Code. If you learned them as AI-era habits, they will serve you in whatever era comes next.

## Adapt the System, Do Not Adopt It Blindly

The AgentFlow structure in this book is one way to organize AI-assisted development. It is not the only way, and it is probably not the optimal way for every team or project.

Treat it as a starting point. Run it as described for a few projects. Notice where it adds value and where it creates friction. The friction is information — it tells you either that the discipline is genuinely unnecessary for your context, or that your implementation of the discipline needs adjustment.

Cut what does not serve you. The five-question planning brief may become three questions for the projects you know well. The full decision log may be overkill for a short-lived prototype. The important thing is that you are making deliberate decisions about your process, not drifting into whatever feels easiest in the moment.

The goal is not to run AgentFlow. The goal is to ship software that does what you intended, maintained by a team that understands what they built, using AI that amplifies your judgment rather than substituting for it.

## What Comes Next

The companion repository for this book contains the complete task manager CLI as it exists at the end of each chapter, the AgentFlow skill and agent templates used to write this book, and a starter template for your own AgentFlow setup.

If you have feedback — a discipline that did not land, a scenario that did not match your tools, an argument you found unconvincing — the repository is the right place to file it. This methodology will improve through real use. Your experience is part of that.

The next thing to do is straightforward: take the project you are currently working on and apply Chapter 1. Write a planning brief before your next AI session. Not a long one. Five questions, five minutes. See what changes.

It will.

*The AI increased execution speed. You increased the quality of what gets executed. That is the whole game.*