Coding with Agent Teams

Lee Harrington

Introduction

You have been using AI agents for a while. You know how to write a prompt. You know what a good response looks like and what a bad one looks like. You have gotten real work done with these tools.

And at some point you hit a wall.

Not a dramatic failure — nothing crashed, nothing was obviously wrong. The output just started coming back flat. Generic. Almost right but not quite. You asked for something complex, you got something complete, and you knew the difference even if you could not name it. You tried better prompts. Longer prompts. More specific prompts. The ceiling stayed where it was.

This book is about what is on the other side of that ceiling.

The Problem With One Agent

A single agent session has one context, one perspective, and one job: produce the output you asked for. On simple tasks — write this function, summarize this document, answer this question — that is enough. The task fits in the context. The agent does it. You move on.

Complex tasks do not fit in a single context. Not because the model lacks capability, but because complex work has structural properties that one agent cannot satisfy. Research and writing are different cognitive tasks — when they happen simultaneously, the research gets compromised. Editing and authorship require different perspectives — the writer cannot be the adversarial reader of their own work. Planning and implementing have a hard dependency — a plan must be complete before implementation begins, or the implementation is built on guesses.

You can write longer prompts to try to simulate all of this. It does not work reliably. The same model that produces brilliant output on a focused task blends concerns, loses constraints, and drifts when asked to be everything at once.

The solution is not a better prompt. It is a team.

What This Book Teaches

This book teaches you to build and run teams of AI agents — specialized roles, working in sequence, passing structured output between them, coordinated by a skill file you design.

You do not need external orchestration infrastructure to do this. You do not need to write a routing layer or manage API calls between agents. Everything in this book runs in Antigravity, using documents and skill files as the coordination layer. The human — you — runs the coordinator and confirms the gates.

That is both the power and the limit of what this book teaches. You will build something genuinely useful. You will also learn exactly where it breaks down and what the next level looks like when you are ready for it.

Part 1 builds a content pipeline: a team that takes a topic and produces a finished article. You start by running a single prompt and saving the output. Over five chapters you build the team that does the same task properly. In Chapter 5, you compare both outputs. The difference is not subtle.

Part 2 applies the same pattern to software development. You build a coding team — Planner, Implementer, Reviewer, Tester, Documenter — with a coordinator skill that sequences them through a full feature sprint. The project is a CLI tool called `git-summary`. By Chapter 11, you run the complete pipeline from requirement to documented, tested code.

What You Need

A Google Antigravity account (free tier is sufficient throughout — no paid features are required). Some familiarity with using AI agents for development tasks. And a complex task you have been avoiding because you suspected one agent was not going to cut it.

That last one is the most important. The system in this book is not a demonstration — it is a working method. The best way to learn it is to use it on something real.

One More Thing

This book was written using the system it teaches.

The chapters you are about to read were produced by a pipeline of specialized agent roles: one that defined the principle to be taught, one that designed the hands-on scenario, one that reviewed the Antigravity instructions for accuracy, one that read each draft as a confused first-time reader to find the gaps, one that checked continuity across chapters. A coordinator skill sequenced all of them. The gate ran before every chapter draft was written.

The pipeline did not run perfectly every time. Stages collapsed occasionally. Roles drifted. Some runs were better than others. The gate caught most of what needed catching.

That is an accurate description of what you are about to build. Not a perfect system — a visible one, where the failures are predictable and the improvements are traceable. That is the point.

Let's build it.

Chapter 1: Run the Prompt

You have done this before. You opened an AI agent, described what you wanted, and got back something usable. Maybe you iterated once or twice. Maybe the first output was good enough. Either way, you got the thing done.

That pattern — one agent, one context, one output — is how most developers use AI tools. It works on tasks with clear scope and modest complexity: write this function, summarize this document, generate a test for this class. The agent does it, you check it, you move on.

This chapter asks you to use that pattern on something harder. Not because it will fail spectacularly — it won't. Because the output you get is the most useful thing in this book.

You are going to write an article on a topic that matters: how to write documentation your future self will actually use. It is the kind of task where quality is obvious when it is there and invisible when it is not. You will save what you get, and you will come back to it in Chapter 5. By then you will have built something to compare it to. The difference will be specific and nameable.

For now: run the prompt.

The Task

Documentation is a solved problem in theory and a persistent failure in practice. Every developer knows they should document decisions. Most documentation is either missing, outdated, or present but useless — accurate enough that someone wrote it, not specific enough that anyone reads it twice.

That gap — between documentation that exists and documentation that works — is exactly the kind of problem that looks straightforward to an AI agent. The topic is clear, the audience is defined, the format is familiar. You will get a complete article.

Keep that observation in mind when you read what comes back.

The Prompt

Open a new agent session with a clean context — no prior conversation, no previously loaded files. Then give the agent this prompt, word for word:

```
Write a 1000-word article titled "How to write
documentation
your future self will actually use." Write it for
developers.
Be practical and direct. Include specific techniques
and at
least two concrete examples. Do not use a listicle
format —
write in paragraphs.
```

> **Antigravity:** Open Antigravity. Click **New Session** (the + icon near your session list) to start a fresh conversation — you should see an empty input field with no prior context. Type or paste the prompt above into the input field and press Enter. Let the agent complete the article without interruption. Do not send follow-up messages.
>
> **Watch For:** A complete article — introduction, several body sections, conclusion — in a single response. If the agent asks a clarifying question instead of writing, reply "just write it" and let it continue. If it produces a bulleted list despite the instruction, note that; the instruction was clear, and a list anyway is itself information about what single-prompt output does with constraints it finds inconvenient.

A note on why the prompt is written this way: length, audience, tone, and format are all specified. A vague prompt produces a vague output, and a vague output is hard to compare to anything. You want something complete enough to be a real artifact. The "no listicle" instruction exists for the same reason — a list hides quality problems behind visual organization.

Reading What You Got

Read the article once, straight through, without editing it.

Then create a new file called `article-v1.md`, paste the full text in, and add this comment block at the very top:

```
<!--
First impressions — one sentence each:
  Voice:
    Does this sound like a specific person wrote it,
    or like anyone could have?
  Examples:
    Are they concrete enough to recognize,
    or generic enough to fit anything?
  Misconception:
    Does this address something developers get wrong,
    or just explain the topic?
  Send test:
    Would you send this to someone whose opinion
    you respect, unchanged?
-->
```

Write one honest sentence in response to each question. Not a critique — an impression. You are not deciding whether the article is good. You are noticing what you notice before you know what to look for.

If "voice" or "misconception" feels vague right now, treat them plainly. "Voice" means whether the article sounds like a specific person wrote it. "Misconception" means whether it addresses something developers actually get wrong, rather than only describing the topic.

> **Antigravity:** Copy the full article text from the agent response. In the Explorer pane (left sidebar), right-click your project directory and choose **New File**. Name it `article-v1.md`. Paste the article text, then add the comment block above it — at the very top of the file, before the article begins. Fill in your one-sentence impressions and save (Cmd+S or Ctrl+S).

> **Watch For:** `article-v1.md` appears in your file tree with the comment block at the top followed by the article text. Your four impressions are filled in. The file is saved.

One rule: do not iterate on this article. Do not ask the agent to improve it, tighten the tone, or add better examples. Every change reduces the comparison you will run in Chapter 5. The version you have right now — first output, unedited — is the artifact. Its value is in being exactly what one agent produces in one pass.

What You Have

You have a complete article written by one agent in one session. It covers the topic. It has structure. It may have sections you find genuinely useful.

What you may have noticed — or not yet been able to name — is whether it has a voice. Whether the examples are grounded in something specific enough to recognize, or plausible enough to feel real without actually being real. Whether it addresses what developers actually get wrong about documentation, or whether it explains documentation the way documentation articles always explain documentation.

These are not small distinctions. They are the difference between an article someone reads once and an article someone sends to a colleague. You will be able to name them precisely in Chapter 5. Right now you have an impression, four sentences of it, and a file.

That is exactly what this chapter needed to produce.

> **Key Takeaway:** Complete is not the same as good. You have a complete article. Keep it exactly as it is — you will need it in five chapters.

Chapter 2: The Grounding Problem

The article you saved in Chapter 1 has examples in it. Read them again.

They sound specific. There may be a named tool, a named situation, a reference to something a developer would recognize. Now ask yourself: how do you know those examples are real?

You probably cannot check without effort. And you probably did not try — because the prose read with confidence, and confident prose does not usually flag itself as potentially invented. The AI wrote specific-sounding examples because you asked for specific examples, and it has no reliable way to distinguish between "I know this" and "this sounds right." Neither mode produces a different signal in the text.

That is the grounding problem. It is not unique to AI — human writers blur memory and invention too — but AI does it at a scale and fluency that makes it invisible. This chapter shows you how to design around it.

What One Agent Does When It Writes and Researches at Once

Ask any AI agent to write a practical article and it will blend what it knows with what it fills in, because writing and research are the same action when no one separates them. The agent does not have a research phase — it has a generation phase, and generation produces text, whether that text is grounded in something specific or assembled from plausible-sounding components.

This is not a model problem. It is an architecture problem. One agent doing everything has no moment where it commits to what it knows before it starts constructing. You can add "use specific examples" to your prompt and the agent will produce specific-sounding examples. Whether they are specific in the way you meant depends on whether it actually knows them.

The Verification Test

Before you build anything, do this: open a fresh agent session — no context from Chapter 1 — and send this prompt:

```
Give me five specific, real examples of documentation
that failed its author. For each one: name the
project
or context, describe what the documentation said, and
explain what the author needed six months later that
the documentation did not provide.
```

> **Antigravity:** Click **New Session** to open a fresh conversation. Paste the prompt and send it. Read the five examples the agent produces.

Read what you get. Each example will be specific — named projects, specific situations, plausible failures. Now pick two of them and try to verify them. Search for the project, the context, the specific incident described.

You do not need to prove they are false. You only need to notice how much effort it takes to find out whether they are real. That effort is the gap the grounding problem lives in.

> **Watch For:** At least one example that you cannot verify with a quick search, where the agent's prose gave you no indication of uncertainty. That is an invented example written with the same fluency as a real one.

You are not going to fix this by asking the agent to "only use real examples." That instruction does not change what the agent has access to. It changes how it presents what it generates.

What a Researcher Role Does Differently

A role is not a prompt. A prompt tells the agent what to produce. A role defines what the agent *is* during this task — its perspective, its constraints, and its output format.

The researcher role does one thing: it commits to what it knows before writing starts. It does this by separating two outputs that one-agent generation blends together:

- Claims the agent can ground — things it knows are real, with enough specificity to verify
- Claims the agent is uncertain about — things it believes are plausible but cannot confirm

That separation is the design. The agent can still produce both kinds of claims in the researcher role. But it has to label them. And the act of labeling forces a kind of honesty that generation alone does not.

The researcher produces `research-notes.md`. Not an article. Not a list of ideas. A structured set of claims, each tagged with its confidence level, ready for a writer who will build something on top of them.

Building `agents/researcher.md`

Create `agents/researcher.md`. If you do not already have an `agents/` directory in this project, create it first. This is your researcher role definition. It should contain:

What the researcher receives: A topic and a question. In this case: the topic is "documentation your future self will actually use" and the question is "what do developers actually get wrong, and what evidence supports that?"

What the researcher produces: `research-notes.md` — a structured document with this format:

```
# Research Notes: [topic]

## Grounded Claims
Claims I can support with specific, verifiable
        examples.
- [claim] — [specific example or source]

## Plausible Claims
Claims I believe are likely true but cannot ground
        specifically.
- [claim] — [why I believe this; what would confirm
        it]

## Uncertain Claims
Things I've seen stated but cannot evaluate.
- [claim] — [why this is uncertain]
```

The researcher's constraint: Do not write prose. Do not draft article sections. Produce only the structured notes above. If you do not have a grounded example for a claim, it goes in Plausible, not Grounded.

Here is what `agents/researcher.md` should say:

```
# Agent: Researcher

You receive a topic and a research question.
You produce research-notes.md — structured notes in
        the
format provided, with explicit confidence markers.

You do not write prose. You do not draft the article.
Your job is to commit to what you know before writing
        begins.

Rules:
- Grounded claims require a specific, verifiable
        example
```

```
- Plausible claims require a reason for the belief
- Uncertain claims require an honest flag
- If you cannot ground a claim, it does not go in
      Grounded
```

> **Antigravity:** In the Explorer pane, right-click your project directory and choose **New Folder** if `agents/` does not exist yet. Name it `agents`. Then right-click `agents/`, choose **New File**, name it `researcher.md`, paste the role definition above, and save it.

Running It

Open a new agent session. Load the researcher role by pasting the full contents of `agents/researcher.md` at the start of the conversation, then follow it with:

```
Topic: How to write documentation your future self
will actually use
Question: What do developers actually get wrong about
documentation,
and what evidence supports that?
```

> **Antigravity:** Start a **New Session**. In the input field, paste the full text of `agents/researcher.md`, then add a blank line, then add the topic and question above. Send it. The agent will produce structured research notes rather than article prose.

> **Watch For:** Output organized into the three sections — Grounded, Plausible, Uncertain. If the agent produces flowing prose instead, the role constraint did not hold. You may need to add an explicit instruction at the top of your role definition: "Your entire output must use the structured format below. Do not write prose paragraphs."

When the output arrives, save it as `research-notes.md`.

Now read it and answer this honestly: which claims in the Grounded section could you verify right now? Pick two and try. Compare this to what you found in the verification test.

The Handoff Contract

`research-notes.md` is not just the researcher's output. It is the input for the next stage — the writer who will build the article on top of it. The format you defined matters because of what comes next.

This is a handoff contract: the output of one role is the input specification of the next. The researcher produces something structured because the writer needs to consume something structured. If the researcher produced free-form notes, the writer would have to interpret them, and interpretation introduces the same blending problem you just designed around.

Every role in your pipeline will have a handoff contract. You will design them deliberately, starting now.

> **Key Takeaway:** The researcher's job is not to write — it is to commit. Separating that commitment from the writing changes what the writing can trust.

Chapter 3: The Reader Problem

You wrote the draft. You read it back. It makes sense.

Here is what that means: it makes sense to you. You generated it. You know what each section is trying to say. You filled the gaps automatically as you read because you already had the context to fill them. The prose is coherent to the person who produced it, which is the least useful measurement of whether it will be coherent to someone who didn't.

This is not an AI problem. Human writers have the same blind spot. AI makes it worse because the agent writes from a position of maximal context — it generated every preceding paragraph, it knows exactly where the piece is going, it has no experience of arriving at this text cold. When it reads back what it wrote, nothing is unclear. That guarantees nothing about whether it will be unclear to you.

The solution is a role whose job is to not know what you know.

Writing From the Research Notes

Before you can test the draft against a reader, you need a draft. Open a new agent session and give it the contents of `research-notes.md` along with this instruction:

```
Using only the claims in the Grounded and Plausible
sections
of these research notes, write a 900-word article
draft titled
"How to write documentation your future self will
actually use."
Write for developers. Use the Grounded claims as the
backbone.
Use Plausible claims only where they support a
Grounded claim.
Do not add examples or claims that are not in these
notes.
```

> **Antigravity:** Start a **New Session**. Paste the full contents of `research-notes.md` first, then add a blank line, then add the instruction above. Send it. The agent will write a draft using only what the researcher committed to.
>
> **Watch For:** The draft should stay closer to the research notes than the Chapter 1 article did. If the agent adds new examples that were not in your notes, it has stepped outside its instructions. Note where that happened — it is the same invented-example pattern from Chapter 2, now visible because you have the notes to compare against.

Save the output as `draft.md`. This is a working draft, not a final article. It does not need to be polished. It needs to be complete enough for the next step.

The Gap That Writing Creates

Read `draft.md` once. Now ask yourself: if you arrived at this article having never thought about the documentation problem before, would every paragraph follow from the previous one? Would you understand the examples without background? Would you know what the first section assumed you already knew?

You probably cannot answer those questions accurately. Not because you are a poor reader — because you are the wrong reader. You have the context. Gaps in a text are invisible to the person who filled them while writing.

This is structural, not stylistic. You can edit a draft for clarity and still not see the gaps, because editing from the writer's perspective does not change whose context you are reading with. The only way to find the gaps is to read with someone else's context — or to build a role that simulates not having yours.

Building `agents/adversarial-reader.md`

Create a new file: `agents/adversarial-reader.md`. The adversarial reader has one job: arrive at the draft cold and report where the experience breaks down.

This role is not an editor. It does not improve the prose. It does not suggest rewrites. It finds places where a reader without the writer's context would lose the thread — and reports them as

specifically as possible: where in the draft, what assumption was made, what the reader would need to know that the draft does not provide.

Here is what `agents/adversarial-reader.md` should say:

```
# Agent: Adversarial Reader

You are reading this draft for the first time.
You do not know what the writer was trying to say.
You only have the words on the page.

Your job is to find where the reader's experience
silently breaks down — not where the prose is rough,
but where a reader without context would lose the
thread and not know they'd lost it.

For each problem you find, report:
-
        Location: where in the draft (quote the
        sentence or phrase)
- Problem: what assumption the text makes that the
        reader may not share
- Impact: what the reader will do wrong or
        misunderstand as a result

Do not suggest rewrites. Do not give style feedback.
Find the silent gaps and name them precisely.
```

> **Antigravity:** Create `agents/adversarial-reader.md` in your project directory. Paste the role definition above. Save it.

Running It

Open a new agent session. Load the adversarial reader role, then give it `draft.md` to read:

```
[paste full contents of agents/adversarial-reader.md]

---

[paste full contents of draft.md]
```

> **Antigravity:** Start a **New Session**. Paste the full text of `agents/adversarial-reader.md`, then a blank line with `---`, then the full text of `draft.md`. Send it.
>
> **Watch For:** A list of located problems — each one citing a specific place in the draft, a specific assumption, and a specific impact. If the output is general ("this section could be clearer"), the role constraint did not hold. Add this line to the role definition and run it again: "Do not give general feedback. Every item must cite a specific sentence or phrase."

Read each item the adversarial reader found. For each one, notice whether your reaction is "yes, that's a real gap" or "that's obvious, any reader would know that." The second reaction is the signal — it means you knew something the reader did not, and you did not know you knew it. That is the gap the role was designed to find.

Save the adversarial reader's output as `reader-notes.md`.

Addressing the Gaps

Go through `reader-notes.md` and fix the located gaps in `draft.md`. Not a rewrite — targeted additions. Each gap the adversarial reader found needs either a sentence of context or a more grounded example. Nothing more.

> **Antigravity:** Open `draft.md`. For each item in `reader-notes.md`, make the specific, minimal change that closes the gap. Save the updated file.
>
> **Watch For:** Fixes that are one or two sentences, not paragraphs. If you find yourself rewriting a whole section to address one gap, the gap was structural — the section was missing a foundation, not just a clarification. Note it; that kind of gap is worth understanding.

When you are done, you have a draft that was written from grounded research and has been read by someone who did not share your context. That is a different artifact than what Chapter 1 produced.

> **Key Takeaway:** Clarity is invisible to the person with context. You need a role whose entire job is to not have it.

Chapter 4: Voice and Sequence

You now have three roles: a researcher, a writer, and an adversarial reader. Each one does its job. What you do not have yet is a team.

A collection of roles is not a team. A team has a shared register — it sounds like it came from the same place. A team has a sequence — each member knows what they receive and what they hand off. A team has a coordinator that enforces both.

This chapter builds two things: a voice constraint that gives your team a shared register, and a pipeline coordinator that sequences the stages in the right order every time. After this chapter, you have a team. Chapter 5 is where you run it.

Why Voice Drifts

Each role you have built so far runs in its own session. The researcher's session has no knowledge of the writer's session. The writer's session has no knowledge of the adversarial reader's session. They share the research notes and the draft as files, but they do not share a register.

Register is the combination of tone, formality, sentence rhythm, and assumed relationship to the reader. When different sessions produce different parts of the same article, the register shifts — subtly, in ways that do not cause obvious errors but cause the finished piece to feel like it was assembled rather than written. The introduction sounds direct; the middle section sounds cautious; the conclusion sounds like a different article entirely.

You can add "write in a direct, developer-facing tone" to every prompt. That instruction will be interpreted differently by every session. What you need is a single, specific voice definition that every stage loads before it starts — a constraint, not a suggestion.

Building `skills/voice.md`

Voice is not a style preference. It is a set of specific constraints that apply to every word the writer produces. The constraints answer questions that "write with a direct tone" leaves open:

- Who is the reader, and what do they already know?
- What is the relationship between writer and reader?
- What sentence structures are preferred? What are off-limits?
- What does "specific" mean in practice for this audience?

Create `skills/voice.md`. If you do not already have a `skills/` directory in this project, create it first. Here is a starting definition you will adapt:

```
# Skill: Voice

## The Reader
A developer who has shipped code. Comfortable with
        tools.
Not a beginner. Has run into the problem this article
        addresses.
Does not need to be convinced the problem is real.

## The Register
Peer-to-peer. Write as a developer talking to another
        developer
who is slightly behind on this specific thing. Not a
        teacher.
Not a consultant. Someone who figured this out and is
        sharing it directly.

## Constraints
- Short sentences when the point is sharp.
  Expand only when the concept requires it.
- Specific examples over general principles wherever
        possible.
- Name the problem before offering the solution.
- No hedging. If something is true, say it.
- No throat-clearing. Start with the point.

## What to Avoid
- Academic register: "it can be observed that" →
        "notice that"
- Cheerleading: do not tell the reader how useful
        this will be
- False precision: do not use numbers or statistics
        you cannot source
- Generic examples: if the example could apply
  to anything, it applies to nothing
```

> **Antigravity:** In the Explorer pane, right-click your project directory and choose **New Folder** if `skills/` does not exist yet. Name it `skills`. Then create `skills/voice.md`, paste the definition above, adjust the reader description to match your intended audience, and save it. For example, if you are writing for junior developers instead of experienced ones, change the first lines of **The Reader** section to say so explicitly.

This file will be loaded at the start of every stage that produces prose. It does not change what the stages do — it changes what register they do it in.

Why Sequence Matters

The roles you have built have dependencies. The writer depends on `research-notes.md`. The adversarial reader depends on `draft.md`. If you run the writer before the researcher is done, the writer has nothing to work from. If you run the adversarial reader before the draft exists, the same problem.

These dependencies are obvious. The less obvious constraint is this: earlier stages change what later stages can do. If the researcher runs without the voice constraint loaded, its output will be in whatever register the base model defaults to. The writer will have to translate that register, introducing inconsistency. If the adversarial reader runs before the draft

has been voice-checked, it will find register problems mixed in with content problems — and the feedback will be harder to act on.

Sequence is not just about prerequisites. It is about what each stage can reasonably be asked to produce given what preceded it.

Building `skills/article-pipeline.md`

The coordinator skill defines the stages, their order, what each stage receives, and what each stage produces. It is a playbook. When you load it at the start of a session, the AI knows what stage it is on, what role to adopt, what input to expect, and what output to produce.

Create `skills/article-pipeline.md`:

```
# Skill: Article Pipeline

This skill coordinates writing one article. Load it
        at the
start of a session and follow the stages in order.
Do not proceed to the next stage until the current
        one is complete.

## What You Need to Start
- A topic and a target reader
- `skills/voice.md` loaded and available
- An empty project directory for this article

---

## Stage 1: Research
```

```
Role: load `agents/researcher.md`
Input: topic + research question
Output: `research-notes.md` (grounded, plausible,
        uncertain claims)

Produce research notes only. Do not write prose.

---

## Stage 2: Draft

Role: load `agents/writer.md` and `skills/voice.md`
Input: `research-notes.md`
Output: `draft.md`

Write from the Grounded and Plausible claims only.
Apply the voice constraints throughout.
Do not add examples or claims not present in
        research-notes.md.

---

## Stage 3: Adversarial Read

Role: load `agents/adversarial-reader.md`
Input: `draft.md`
Output: `reader-notes.md`

Find located gaps. No style feedback. No rewrites.

---

## Stage 4: Revise

Role: load `agents/writer.md` and `skills/voice.md`
Input: `draft.md` + `reader-notes.md`
Output: `final.md`
```

```
Address each located gap in reader-notes.md.
Minimal targeted changes. Do not rewrite sections
that do not have a located problem.

---

## ⛔ Gate (Optional)

You may stop after any stage to review the output
        before
proceeding. Stopping after Stage 1 (before writing
        begins)
is the most valuable gate — it is the last moment to
        catch
a research problem before prose is built on top of
        it.
```

> **Antigravity:** Create `skills/article-pipeline.md` inside `skills/`. Paste the coordinator above. Save it.

You will notice Stage 2 references `agents/writer.md` — a role you have not built yet. That is intentional. The coordinator names the full sequence first; you will add the writer role next. It is a simple role that receives research notes and voice constraints and produces a draft. The writer role does not need to be complex; the researcher and adversarial reader are doing the heavy lifting. The writer's constraint is straightforward: use the research notes, use the voice, produce prose.

```
# Agent: Writer

You receive research-notes.md and skills/voice.md.
You write a draft article using only the claims in
        the
```

```
Grounded and Plausible sections of the research
        notes.
You apply the voice constraints throughout.
You do not add examples or claims not present in the
        research notes.
You produce draft.md.
```

> **Antigravity:** Create `agents/writer.md` with the role definition above. Save it.

Two Kinds of Files

You now have four files in your project directory:

```
agents/researcher.md
agents/writer.md
agents/adversarial-reader.md
skills/voice.md
skills/article-pipeline.md
```

They look the same — markdown files with instructions — but they do two different things.

Agent files define a role. `agents/researcher.md` tells the AI who it is for one stage: what it receives, what it produces, what it is not allowed to do. It is a specialist identity, adopted for the duration of a stage and then set aside. Every stage in your pipeline uses an agent file.

Skill files define a rule that persists. There are two kinds: `skills/voice.md` is a shared constraint — a set of specific instructions loaded by every stage that produces prose. It does

not define a stage; it shapes what all stages do. `skills/article-pipeline.md` is a coordinator — it defines the full sequence, which stage comes next, which agent role to adopt, and what each stage hands off. It does not do any of the work; it tells you who does.

The directory names are not decorative. When you see `agents/`, you are looking at a role — a job that gets done once at a specific point. When you see `skills/`, you are looking at a durable rule that either coordinates the whole sequence or applies throughout it.

This distinction will matter more as your teams grow. A new role becomes a new agent file. A new coordination sequence becomes a new skill file. You will never put a coordinator in `agents/` or a role definition in `skills/`.

The Gate

The coordinator includes an optional gate after Stage 1. This is the most important stopping point in the pipeline: the moment before writing begins, when the research is committed and the voice is loaded, but no prose exists yet.

If the research notes look thin, or the voice definition feels wrong for the topic, stopping here is cheap. Stopping after Stage 3 — after the adversarial reader has run — costs a full draft. Stopping after Stage 4 costs a revision. The earlier you catch a problem, the less work you discard.

Whether to stop at the gate depends on how well you know the topic and how much you trust the researcher's output. For familiar topics with a mature researcher role, running the full pipeline without stopping is reasonable. For new topics or new audiences, the Stage 1 gate is worth taking.

This is the same calibration question you will face with every pipeline you build. More on it in Chapter 6.

> **Key Takeaway:** A coordinator turns a collection of roles into a pipeline. Sequence is a constraint, not a preference — and voice is a constraint you define before writing starts, not a style that emerges during it.

Chapter 5: Run the Pipeline

You have everything you need.

A researcher that commits to what it knows before writing begins. An adversarial reader that finds the gaps the writer cannot see. A voice constraint that applies the same register to every stage. A coordinator that sequences the whole thing and holds the gate.

This chapter has one job: run it.

The Run

Load `skills/article-pipeline.md` in a new agent session. The topic is the same one you used in Chapter 1. This run will produce new versions of `research-notes.md`, `draft.md`, and `reader-notes.md` — these will replace the files from Chapters 2 and 3, which served their purpose as practice. That is expected.

```
Topic: How to write documentation your future self
will actually use
Target reader: Developers who have written
documentation that became
useless within six months
```

Follow the stages in the coordinator. At each stage, let the role complete before moving to the next. Use the optional Stage 1 gate — read the research notes before the writer starts. If anything in the notes looks uncertain, address it before prose exists.

> **Antigravity:** Start a **New Session**. Paste the full contents of `skills/article-pipeline.md` at the start, then add the topic and target reader above. The agent will walk through the stages. At Stage 1, read `research-notes.md` before confirming it should proceed to Stage 2. When you are ready, type: `Stage 1 complete. Proceed to Stage 2.` At subsequent stages, let the pipeline run.
>
> **Watch For:** The stages producing distinct outputs — research notes that stay in the structured format, a draft that stays within the research notes' claims, adversarial feedback that is located and specific, a final article that differs from the draft only at the located gaps. If stages blend — if the writer starts adding new claims, or the adversarial reader starts rewriting — the coordinator did not hold. Note where it broke; Chapter 6 addresses this directly.

When `final.md` is ready, read it once through. Save a copy as `article-v2.md` — this is the artifact you will compare against `article-v1.md`. Like `article-v1.md`, leave it unchanged after saving so the comparison stays honest.

> **Antigravity:** Copy the content of `final.md`. Create a new file in your project directory named `article-v2.md`. Paste the content and save. Do not keep editing `article-v2.md` after that; it is your comparison artifact.

Then open `article-v1.md` — the article you saved in Chapter 1.

The Comparison

Place both articles side by side. Read them on the same five dimensions you noted your first impressions against in Chapter 1:

> **Antigravity:** Open both `article-v1.md` and `article-v2.md` in the editor pane. If your editor uses tabs, switch between them as you compare. You are not looking for perfect prose; you are looking for visible differences you can name.

Voice. Does `article-v1.md` have a consistent register throughout? Does `article-v2.md`? Can you tell where in `article-v1.md` the tone shifts? Can you find a shift in `final.md`?

Examples. Pick one example from each article. For the `article-v1.md` example: could it have come from anywhere, or is it specific to a real situation? For the `article-v2.md` example: is it in `research-notes.md`? Can you trace it to a grounded claim?

Misconception. Does `article-v1.md` address something a developer actually gets wrong about documentation — a specific wrong belief — or does it explain what good documentation looks like? Does `final.md` do the same? Is the difference visible?

Silent gaps. Find one place in `article-v1.md` where a reader without background might lose the thread without knowing it. Now find the same kind of place in `article-v2.md`. Were the gaps in `article-v2.md` smaller? Were they in different places?

Send test. Read the conclusion of each article. Which one would you send to a colleague who asked you "how do I write better documentation"? Why?

The comparison is the argument this book has been building. Not "the pipeline produces better output" — you can see whether it does. What you should be able to name now is *why*: which stage produced which quality. The grounded examples came from the researcher. The addressed gaps came from the adversarial reader. The consistent voice came from the voice constraint. The fact that those things exist in sequence, each building on the last, came from the coordinator.

What the Skills Produced

The five qualities you just evaluated each trace back to a specific design decision:

Consistent voice — `skills/voice.md` loaded at every prose-producing stage. Not a prompt instruction that gets interpreted differently each time. A shared constraint that applies to every word.

Grounded examples — `agents/researcher.md` separating the commitment to what is known from the act of writing. The writer cannot add a claim the researcher did not commit to.

Addressed misconception — the researcher role was given a research *question*, not just a topic. "What do developers actually get wrong?" produces different output than "write about documentation." The question aimed the research.

Located gaps closed — `agents/adversarial-reader.md` finding specific places, not general impressions. The writer addressed those places, not a vague directive to "be clearer."

Sequence — `skills/article-pipeline.md` ensuring the researcher ran before the writer, and the adversarial reader ran after the draft. The coordinator is not adding capability; it is preventing the capability that exists from being wasted by running in the wrong order.

What the Pipeline Cannot Guarantee

The pipeline ran. The output is better than Chapter 1's in the specific ways the design intended. It is not perfect.

Look at `article-v2.md` again. Is there a place where the adversarial reader's feedback was correct but the revision did not quite close the gap? Is there a claim in `research-`

`notes.md`'s Plausible section that made it into the final article in a way that feels shakier than the Grounded claims? Did the voice hold perfectly, or did one section drift?

You are not looking for failures. You are calibrating your trust. The pipeline produces consistent improvement, not guaranteed quality. The gap between those two things is what the next chapter is about.

> **Key Takeaway:** Each quality in the final article traces back to a specific role. That traceability is what a team gives you that one agent cannot — not just better output, but output you can understand and improve.

Chapter 6: What You Cannot Trust

The pipeline works. You have evidence of it in `final.md`. The comparison with `article-v1.md` showed specific, traceable improvements — not "better in general" but better in the ways the design intended.

Now for the part the pipeline does not advertise.

Run the pipeline again on a new topic. Read the output. Is it as good as the first run? Maybe. Maybe not. Run it a third time on the same topic as the first run. Is it identical? It isn't. Different claims in the research notes, different examples in the draft, different gaps found by the adversarial reader. The coordinator ran the same stages in the same order with the same roles. The output is different.

This is not a bug. It is a property of how language models work. And it has practical consequences for how much you can trust the pipeline to produce consistent quality without your attention.

What Can Go Wrong

Stage collapse. The coordinator instructs the agent to run Stage 1, produce research notes, then stop. Sometimes it runs Stages 1 and 2 together — research notes and draft in the same response — because the agent anticipates what comes next and does not wait for a boundary you defined in the skill file. The stages blend, the handoff contract is skipped, and the writer's output is not constrained by the research notes in the way you intended. This happens more often in long sessions where the coordinator has already explained the full pipeline.

Role drift. The researcher produces three Grounded claims and then, in the same response, starts writing article prose because it can see where the research is going. The adversarial reader gives general style feedback alongside the located gaps because it has opinions about the prose and cannot fully suppress them. The roles you defined are constraints, not walls. The base model is still there, underneath, and it will surface when the role constraints are underspecified or when the task is long enough that the agent loses track of which stage it is on.

Output variance. The same researcher role, same topic, run twice, produces different Grounded claims. Not because one run is wrong — both may be accurate — but because the model's sampling process is not deterministic. Some runs will be more thorough than others. Some adversarial readers will find three gaps; others will find six. You do not control which run you get.

Context accumulation. A full pipeline run is a long session. By Stage 4, the agent is working in a context that contains the full coordinator skill, the research notes, the draft, the

adversarial reader's feedback, and whatever stage instructions preceded the current one. Later stages are influenced by earlier stages in ways you did not design. The adversarial reader in Stage 3 may be gentler because it has seen the effort that went into the draft. The revising writer in Stage 4 may be more conservative because it has seen what the adversarial reader flagged.

The Gate Is Your Main Instrument

Every one of these failure modes has the same mitigation: read the output before proceeding to the next stage.

The Stage 1 gate — before writing begins — is the most valuable stopping point because it is the cheapest. If the research notes have drifted into prose, or the Grounded section is thin, catching it before the writer runs saves a full draft. The gate after Stage 3 is the second-most valuable: if the adversarial reader produced style feedback instead of located gaps, address it before the revision runs, or the revision will be fixing the wrong things.

Whether to stop at each gate depends on how well you know the topic, how mature your roles are, and how much your last run looked like your previous one. For a pipeline you have run many times on familiar topics, running straight through is reasonable. For a new topic or a role you just wrote, use the gates.

The gate works only if you know what good output looks like at each stage. A research notes file that is all Plausible claims and no Grounded ones is a signal. An adversarial reader report that runs to ten items for a 900-word article probably caught some noise alongside the signal. A draft that introduces examples not present in the research notes means the writer stepped outside its constraint. These are the things to check. The gate is useless if you confirm it without looking.

What an External Orchestrator Solves

The failure modes above have a common structure: they happen because the AI is also the executor. The coordinator skill is read by the same model that runs the stages. The role constraints are instructions to the model, not walls around it. When the model loses track of a stage boundary or surfaces its base behavior through a role constraint, there is no independent layer to catch it.

An external orchestrator is that independent layer. Instead of a skill file that the AI reads and follows, it is code that programmatically routes work between stages — calling an AI for Stage 1, collecting the structured output, validating it against the expected format, then calling an AI for Stage 2 with that output as input. Each stage runs in an independent context: no accumulated session, no model anticipating what comes next, no role drift across a long conversation. The output of each stage is validated before it becomes the input of the next.

This solves stage collapse (the router controls when stages transition), role drift (each stage is a fresh context with a single role), output variance (you can validate the format before proceeding), and context accumulation (there is no accumulated context to affect later stages).

It also requires building something. The orchestrator is code — not a skill file, not a role definition. It has dependencies, it has failure modes of its own, and it is more work to build than what you have spent the last five chapters creating.

Whether that tradeoff is worth it depends on what you are building. For a content pipeline that produces articles, the document-driven approach you have now is probably enough — the failure modes are tolerable and the gate catches most of them. For a pipeline that produces code that gets deployed, or runs in production without human review, the tradeoff shifts significantly.

That is the next level. It is not in this book. Where to find it is at the end of this chapter.

What You Have

You have a working pipeline, an honest picture of its limits, and a clear view of what comes after it.

The pipeline is useful. It produces consistent improvement over single-agent output in the specific ways it was designed to. The improvement is traceable — you can see which stage produced which quality. That traceability is what lets you improve the pipeline: if the adversarial reader is finding too much noise,

tighten the role definition. If the researcher is not producing enough Grounded claims, change the research question. The design is visible.

The limits are real but manageable at this stage. The gate is your main instrument. Use it when you need it, skip it when you don't, and know the difference.

Part 2 takes everything you just built and applies it to a software development workflow. The roles change. The files change. The coordinator changes. The design decisions — why you separate stages, what the handoff contracts look like, how the gate works — do not change at all.

> **Key Takeaway:** The pipeline works because the design is visible. It fails in predictable ways for the same reason. Know the failure modes, use the gate, and you have a system you can trust — within the limits you now understand.

For the Next Level

External orchestration frameworks provide the programmatic routing layer described above. This is one of the fastest-moving areas in software right now — frameworks that were experimental a year ago are production-ready today, and the landscape will continue to shift. Treat this list as a starting point, not a permanent reference.

As of writing, the main options worth knowing:

- **LangGraph** — graph-based workflow with explicit state management and checkpointing; strongest story for complex, stateful pipelines
- **CrewAI** — role-based multi-agent coordination with structured handoffs; the fastest path to multi-agent prototyping; broad protocol support
- **OpenAI Agents SDK** — OpenAI's production-ready orchestration layer (replaced the earlier experimental Swarm framework in 2025); simplest onramp if you are already using OpenAI models
- **OpenAI Symphony** — released March 2026; monitors a project board (currently Linear), creates an isolated workspace per issue, and spawns autonomous coding agents through each stage of the ticket lifecycle; built on Elixir/BEAM for fault-tolerant agent supervision; the closest existing thing to what this book describes — a system where you move tickets on a board instead of prompting agents directly
- **Microsoft Agent Framework** — merged AutoGen and Semantic Kernel into a single framework; natural fit for Azure-based teams; reached release candidate in early 2026

Each of these trades the simplicity of document-driven coordination for the reliability of programmatic control. If what you built in Part 1 is working for you, you do not need them yet. If the failure modes are costing you more than the infrastructure would, you do now. Search for current comparisons before committing — the relative strengths of these frameworks are changing faster than most books can track.

Chapter 7: Translating the Pattern

You built a pipeline that takes a topic and produces a finished article. Researcher, Writer, Adversarial Reader, Coordinator. The pipeline works because each role does one thing, each stage hands off a structured output, and the coordinator enforces the sequence.

Now you are going to build the same thing for code.

Not the same roles. Not the same files. The same design: specialized roles, explicit handoff contracts, a coordinator that sequences them, and a gate where you look before you commit. The pattern translates directly. What changes is what each role is protecting against — and code has failure modes that prose does not.

What Carries Over

Everything structural carries over unchanged.

Roles over prompts. The coding pipeline has a Planner, an Implementer, a Reviewer, a Tester, and a Documenter. Each one receives a specific input, produces a specific output, and operates within explicit scope constraints. You will write role definition files for each of them, exactly as you wrote `agents/`

`researcher.md` and `agents/adversarial-reader.md`. Code needs two roles prose did not: a Tester, because code has an objective pass/fail signal, and a Documenter, because the implementation and test report together can define the tool's real boundary.

Handoff contracts. The Planner produces a plan that the Implementer consumes. The Implementer produces code that the Reviewer and Tester consume. The Tester produces a report that the Documenter uses to describe what the tool does and what it doesn't. Each stage is designed with the next stage's input in mind.

A coordinator. `skills/feature-sprint.md` will do what `skills/article-pipeline.md` does: sequence the roles, define what each stage receives and produces, and hold the optional gates.

Shared memory. `requirement.md` is the equivalent of the topic and research question you gave the article pipeline — the fixed input that every stage reads from without changing. In Part 1 that was a few lines of text. In Part 2 it is an explicit document, because code requires more decisions than prose does, and those decisions need to be traceable. It does not change between stages. If the requirement changes mid-sprint, the sprint stops and restarts.

What Changes

Code has a pass/fail signal that prose does not. Tests either pass or they do not. This changes what the Tester role does compared to the Adversarial Reader — instead of simulating a reader without context, the Tester generates and runs objective checks. When tests fail, the failure is unambiguous. The Implementer did not meet the requirement, or the requirement was underspecified, or the test was wrong. You know something is wrong; you may not know which.

The Reviewer's job is different from the Adversarial Reader's. The Adversarial Reader simulates not having context. The Reviewer has full context — they read the code knowing what it is supposed to do — and asks different questions: Is this implementation correct? Is it the simplest implementation that satisfies the requirement? Does it handle the cases the requirement did not specify? Are there things this code does that the requirement did not ask for?

Scope discipline is sharper. In Part 1, the voice constraint and the research notes kept the writer from wandering. In Part 2, the requirement document keeps the Implementer from adding features that were not asked for. This is a harder constraint to enforce in code: the Implementer will see obvious improvements and want to make them. That is not its job. Its job is to implement the requirement as written. Improvements go on a backlog.

The order of stages matters more. In the article pipeline, a weak research notes file produces a weaker article — bad, but recoverable. In the coding pipeline, an underspecified plan

produces an implementation that does not match what the Reviewer and Tester are checking against. The Planner stage is where the most important gate lives.

The Project: `git-summary`

Across Part 2, you will build `git-summary` — a small CLI tool that takes a repository path and produces a human-readable summary of recent activity: commits in the last week, files changed most frequently, and the most recent commit message for each active file.

```
$ python git_summary.py /path/to/repo

Repository: my-project
Period: last 7 days
Commits: 12

Most active files:
  src/main.py       — 4 commits  — "fix: handle empty
input edge case"
  tests/test_main.py — 3 commits — "test: add edge
case coverage"
  README.md          — 1 commit  — "docs: update
usage examples"
```

This tool has enough surface area to demonstrate all five roles without requiring infrastructure. It has no external dependencies beyond the `git` command and Python's standard library. It is testable: the output for a known repository with known commits is deterministic. It has a clear

requirement that can be underspecified in interesting ways — "recent activity" is vague enough that the Planner will need to make decisions, and those decisions will matter to the Tester.

You will build it across Chapters 8 through 11, one role and one pipeline stage per chapter.

The Requirement Document

Before the pipeline starts, the requirement document exists. This is the input every stage reads from. Create `requirement.md` now:

```
# Requirement: git-summary

## What it does
A CLI tool that accepts a local repository path and
        prints a
human-readable summary of recent git activity.

## Inputs
- A path to a local git repository (required)
- Optional: number of days to look back (default: 7)

## Output format
Printed to stdout. Sections:
1. Repository name and period covered
2. Total commit count for the period
3. Top 5 most-modified files, with: filename, commit
        count,
   most recent commit message for that file

## Constraints
- Python standard library only (no pip installs)
- Must run on macOS and Linux
```

```
- If the path is not a valid git repository, print a
      clear
  error message and exit with code 1
- Output is plain text, no color codes

## Out of scope
- Remote repositories
- Branch filtering
- Commit author breakdown
- Output formats other than plain text
```

> **Antigravity:** Create a new folder for Part 2 in your workspace root. Name it `git-summary-project`. Then create `requirement.md` inside that folder, paste the requirement above, and save it. This file does not change during the sprint — if you find that the requirement is underspecified while building, you add a new version after the sprint, not during it.
>
> **Watch For:** `requirement.md` exists in `git-summary-project` with all five sections present: What it does, Inputs, Output format, Constraints, and Out of scope. Verify the Out of scope section is there before you proceed — it is the first scope boundary in the coding pipeline.

Read the requirement once. Notice what it specifies and what it leaves open. "Top 5 most-modified files" — what if there are fewer than 5? "Most recent commit message for that file" — what if the file has no commits in the period? These are the questions the Planner will answer. They are in the requirement as gaps on purpose — real requirements always have them.

> **Key Takeaway:** The pipeline pattern translates directly from content to code. What changes is what each role is protecting against — and code's pass/fail signal makes some failures unambiguous in a way that prose never is.

Chapter 8: The Coding Team

In Part 1, your team had four members: a Researcher, a Writer, an Adversarial Reader, and a Coordinator. Each one had a role file, an input, an output, and a scope constraint.

Your coding team has five. The work is different; the structure is the same.

The Five Roles

Planner. Receives the requirement document. Produces a plan — not pseudocode, not a full implementation, but an explicit set of decisions: how the tool will be structured, which edge cases are in scope, what assumptions fill the requirement's gaps. The Planner is where vague requirements become specific ones. If the Planner cannot produce a plan from the requirement, the requirement needs work before implementation starts.

Implementer. Receives the plan and the requirement. Produces the code — nothing more. Explicitly not responsible for tests, not responsible for documentation, not responsible for improvements it noticed while implementing. Its scope is the requirement as written and the plan as specified. If the Implementer adds something the plan did not specify, it did something outside its role.

Reviewer. Receives the code and the requirement. Does not run the code. Reads it — asking whether it correctly implements the requirement, whether it handles the edge cases the Planner specified, whether it does anything the requirement did not ask for, and whether it is the simplest correct implementation. The Reviewer's output is located findings: specific lines, specific issues.

Tester. Receives the code and the requirement. Writes the tests that verify the requirement and records the results when they are run — not tests that prove the code works, but tests that prove the code meets the requirement. The distinction matters: a test that passes because the implementation is clever is not the same as a test that confirms the requirement was satisfied. The Tester's output is a test file and a report.

Documenter. Receives the code, the requirement, and the test report. Produces a README that tells a new user what the tool does, how to run it, what the options are, and what it does not do. The "does not do" section is not optional — it closes the scope explicitly and prevents the next person from implementing features that were deliberately excluded.

Building the Roles

Create an `agents/` directory in your Part 2 project. You will write five files.

agents/planner.md

```
# Agent: Planner

You receive requirement.md.
You produce plan.md.

Your job is to turn the requirement into a specific
        implementation plan.
For every gap in the requirement, make an explicit
        decision and record it.
For every edge case the requirement implies, specify
        how it will be handled.

plan.md structure:
## Implementation approach
[One paragraph: data flow, key functions, how git
        will be called]

## Data structures
[What the code will use internally]

## Edge cases and decisions
[Numbered list: each gap in the requirement, your
        decision, your reasoning]

## Out of scope
[Explicit list of things you will not implement,
        drawn from the requirement's
Out of scope section plus anything you decided to
        exclude]

Rules:
- Do not write code. Do not write pseudocode.
- Every edge case in the requirement must appear in
        your decisions.
```

```
- If the requirement is too vague to plan against,
      list the questions
  instead of guessing. The sprint stops until those
      questions are answered.
```

agents/implementer.md

```
# Agent: Implementer

You receive requirement.md and plan.md.
You produce git_summary.py.

Implement exactly what the plan specifies. Nothing
      more.
If you see an improvement the plan does not cover, do
      not make it.
If you find an edge case the plan does not address,
      note it in a
comment — do not handle it without a plan decision.

Rules:
- Python standard library only
- Follow the plan's data structures and approach
      exactly
- Add inline comments only where the logic is non-
      obvious
- Do not write tests. Do not write documentation.
- If the plan is contradictory or insufficient, stop
      and report
  the contradiction. Do not guess.
```

agents/reviewer.md

```
# Agent: Reviewer

You receive requirement.md and git_summary.py.
You produce review-notes.md.
```

```
Read the code against the requirement. Do not run it.
Your job is to find: incorrect implementations,
        missing edge case
handling, scope violations (code that does things the
        requirement
did not ask for), and unnecessary complexity.

review-notes.md structure:
## Correctness issues
[Specific lines where the implementation does not
        match the requirement]

## Missing handling
[Edge cases from the requirement that are not
        handled]

## Scope violations
[Things the code does that the requirement did not
        ask for]

## Simplicity findings
[Implementations more complex than necessary for the
        requirement]

Rules:
- Every finding must cite a specific line or function
- Do not suggest new features. Do not rewrite
        sections.
- "Looks fine" is not a finding. If a section is
        correct, omit it.
- If you cannot find issues in a category, write
        "None found."
```

agents/tester.md

```
# Agent: Tester

You receive requirement.md and git_summary.py.
```

```
You produce test_git_summary.py and test-report.md.

Write tests that verify the requirement is satisfied.
Not tests that verify the implementation is clever.
For each requirement statement, write at least one
        test that
confirms it is met and at least one test that
        confirms the
failure case is handled correctly.

test-report.md structure:
## Tests written
[List of test functions with one-sentence description
        each]

## Tests passed
[Which passed on first run]

## Tests failed
[Which failed, with exact error output]

## Requirement gaps found
[Requirements that could not be tested because they
        were underspecified]

Rules:
- Use Python's unittest or pytest (standard library
        or pytest only)
- Every test must state in its docstring which
        requirement statement it covers
- Do not test internal implementation details — test
        observable behavior
- If a requirement is untestable as written,
  report it; do not invent a proxy test
```

agents/documenter.md

```
# Agent: Documenter
```

```
You receive requirement.md, git_summary.py, and test-
        report.md.
You produce README.md.

Write documentation for a developer encountering this
        tool for the
first time. They have not read the requirement
        document.

README.md sections:
## git-summary
[One sentence: what it does]

## Usage
[Exact command with all options; output format
        described]

## Requirements
[Python version, OS, dependencies]

## What it does not do
[The Out of scope section from the requirement,
        written for a user,
plus anything the test report revealed was not
        implemented]

Rules:
- Write for a user, not for someone who built the
        tool
- Do not explain the implementation
- The "What it does not do" section is not optional
- If the test report shows a failing test, note the
        limitation in the README
```

Antigravity: In `git-summary-project`, right-click the project folder and choose **New Folder**. Name it `agents`. Then create all five role files above inside that directory. Save each one.

> **Watch For:** Five files exist in `agents/`: `planner.md`, `implementer.md`, `reviewer.md`, `tester.md`, and `documenter.md`. Open each one and confirm the first line matches the role name you intended to create.

The Handoff Chain

Read through the five roles and trace what each one receives and produces:

```
requirement.md
  └─▶ Planner ──────────────────▶ plan.md
                                     └─▶ Implementer ──▶
git_summary.py
                           ├─▶ Reviewer
                           │     └─▶ review-notes.md
                           └─▶ Tester
                                 ├─▶ test_git_summary.py
                                 ├─▶ test-report.md
                                 └─▶ Documenter
                                       └─▶ README.md
```

Reviewer and Tester both receive `git_summary.py`. They run in parallel — their outputs do not depend on each other. That is the independence test from Chapter 7: can task B start before task A completes? For Reviewer and Tester, the answer is yes. For Implementer, the answer is no — it depends on the plan.

This dependency structure determines the sprint's parallelism. Stages 1 and 2 are sequential. Stages 3 and 4 are parallel. Stage 5 depends on both Stage 3's review notes and Stage 4's test report — it cannot start until both are complete. You will run this in Chapter 10.

> **Key Takeaway:** Five roles, five files, one handoff chain. The chain is visible before a single line of code is written — and it tells you which stages can run in parallel before you decide to try.

Chapter 9: The Feature Skill

You have five roles and a requirement document. What you do not have is a coordinator that sequences them, defines what each stage receives, and holds the gate before implementation begins.

That is `skills/feature-sprint.md`. This chapter builds it.

What the Coordinator Does

In Part 1, `skills/article-pipeline.md` did three things: it defined the stages in order, it specified the input and output of each stage, and it held an optional gate before writing began.

`skills/feature-sprint.md` does the same three things for a coding workflow — with one addition. In the coding pipeline, the gate after the Planner stage is not optional. It is mandatory.

Here is why: in the article pipeline, a weak research notes file produces a weaker article. That is recoverable — you revise the article, you do not throw away the draft. In the coding pipeline, a weak plan produces an implementation that does not match what the Reviewer and Tester are checking against. When the tests fail or the review surfaces mismatches, the Implementer runs again from scratch. The work is not recoverable — it is redone. The gate prevents that.

Every other gate in the feature sprint is optional, following the same calibration from Chapter 6: use it when you need it, skip it when you don't.

Building `skills/feature-sprint.md`

Create `skills/feature-sprint.md`:

```
# Skill: Feature Sprint

This skill coordinates implementing one feature from
        requirement
to documented, tested code. Load it at the start of a
        session
and follow the stages in order.

## What You Need to Start
- requirement.md — complete and reviewed
- agents/ directory with all five role files
- An empty implementation directory

---

## Stage 1: Plan

Role: load `agents/planner.md`
Input: requirement.md
Output: plan.md

Turn the requirement into a specific implementation
        plan.
Fill every gap. Specify every edge case decision.
If the requirement cannot be planned against, stop
        and report
what is missing. Do not proceed to Stage 2 without a
        complete plan.
```

```
---

## ⛔ Gate — Required

Stop here. Review plan.md before proceeding.

Check:
- Does every requirement statement have a
        corresponding plan decision?
- Are all edge cases from the requirement handled in
        the plan?
- Is the Out of scope section explicit?
- Do you agree with the Planner's decisions?

Do not proceed to Stage 2 until this gate is
        confirmed.
A bad plan produces work that gets thrown away.

---

## Stage 2: Implement

Role: load `agents/implementer.md`
Input: requirement.md, plan.md
Output: git_summary.py

Implement exactly what the plan specifies.
Do not add features not in the plan.
Do not write tests.

---

## Stage 3: Review (runs in parallel with Stage 4)

Role: load `agents/reviewer.md`
Input: requirement.md, git_summary.py
Output: review-notes.md
```

```
Read the code against the requirement.
Produce located findings only.

---

## Stage 4: Test (runs in parallel with Stage 3)

Role: load `agents/tester.md`
Input: requirement.md, git_summary.py
Output: test_git_summary.py, test-report.md

Write tests that verify the requirement.
Report all results — passing and failing. If you
        cannot run the
tests in your current session, make that explicit in
        test-report.md.

---

## Stage 5: Address Review and Test Findings

Role: load `agents/implementer.md`
Input: git_summary.py, review-notes.md, test-
        report.md
Output: git_summary.py (revised), updated test
        results

Fix correctness issues and failing tests from Stages
        3 and 4.
Do not add features. Do not refactor beyond what
        fixes require.
If a review finding and a test failure point to the
        same issue,
fix it once.

---

## ⛔ Gate — Optional
```

```
Review the revised code and updated test results
        before documenting.
If tests still fail, return to Stage 5.
If review findings were not addressed, return to
        Stage 5.
Proceed to Stage 6 only when tests pass and review
        findings are resolved.

---

## Stage 6: Document

Role: load `agents/documenter.md`
Input: requirement.md, git_summary.py, test-report.md
Output: README.md

Write documentation for a developer who has not seen
        the requirement.
Include what the tool does not do.

---
```

> **Antigravity:** In `git-summary-project`, create `skills/` if it does not exist yet. Then create `skills/feature-sprint.md`, paste the coordinator above, and save it.
>
> **Watch For:** `skills/feature-sprint.md` exists with all six stages present. Verify the ⛔ `Gate — Required` block appears between Stage 1 and Stage 2 — that gate is load-bearing.

Stages 3 and 4: Parallel

The coordinator marks Stages 3 and 4 as running in parallel. In the article pipeline, every stage ran sequentially — each one used the previous stage's output as its input. In the coding pipeline, the Reviewer and Tester both receive the same input (`git_summary.py`) and produce independent outputs. Neither needs to wait for the other.

Running them in parallel is not a requirement — you can run them sequentially. But it is an option, and Chapter 10 shows you how to take it using two simultaneous sessions. For now, note the dependency structure in the coordinator: Stage 5 needs both `review-notes.md` and `test-report.md`, so it cannot start until both Stages 3 and 4 are complete.

The Mandatory Gate

Reread the gate between Stages 1 and 2. It is not marked optional. The checklist is specific:

- Does every requirement statement have a corresponding plan decision?
- Are all edge cases handled?
- Is the Out of scope section explicit?
- Do you agree with the decisions?

That last question matters. The Planner made decisions to fill the requirement's gaps. Those decisions are now part of the spec. If you disagree with them — if you think the Planner's

edge case handling is wrong, or its Out of scope decisions are too conservative — the place to address that is here, not after the Implementer has run.

After Stage 1 is the last cheap moment. Everything after it is built on the plan.

> **Key Takeaway:** The coordinator sequences the team and holds the gates. The mandatory gate after Stage 1 is the most important line in the whole skill file — it is the last cheap moment before work that cannot be recovered.

Chapter 10: Running a Sprint

Load `skills/feature-sprint.md`. You are going to run `git-summary` from requirement to documented code.

This chapter is a walkthrough, not a tutorial. You have done every step before — you built all the roles, you built the coordinator, you know what each stage produces. What you have not done is run them together. That is what this chapter is.

Stage 1: The Planner

Open a new session. Load the feature sprint coordinator. The coordinator will tell you to start with the Planner.

```
[paste full contents of skills/feature-sprint.md]

Ready to begin Stage 1.
Topic: git-summary
```

> **Antigravity:** Start a **New Session**. Paste the full contents of `skills/feature-sprint.md`, then a blank line, then "Ready to begin Stage 1. Topic: git-summary". The agent will load the Planner role and begin producing `plan.md`.

> **Watch For:** Structured output in the format specified in `agents/planner.md`: an Implementation approach section, Data structures, Edge cases and decisions, and Out of scope. The plan should address at minimum: what happens with fewer than 5 active files, what the tool does when the git command is not available, how "most recent commit message" is determined when a file has multiple commits in the period, and what constitutes a "valid git repository."

When `plan.md` is ready, stop. Read it before proceeding. This is the mandatory gate.

The Mandatory Gate

Read `plan.md` against `requirement.md`. Work through the checklist:

Does every requirement statement have a plan decision? The requirement specifies "Top 5 most-modified files." The plan should say what happens if fewer than 5 files were modified — does it show fewer, does it pad with zeros, does it show all files up to 5? If the plan does not specify, add it now.

Are edge cases handled? What does the tool do if the repository has no commits in the period? What if the `--days` flag is 0? What if the path exists but is not a git repository? Each of these should be in the plan's Edge cases section.

Is the Out of scope section explicit? The requirement's Out of scope list should appear in the plan, unchanged. If the Planner added anything, check whether you agree.

Do you agree with the decisions? This is the judgment call only you can make. If the Planner decided that fewer than 5 files shows a "No activity found" message instead of the files that do exist — is that what you want? Change the plan now if not.

> **Antigravity:** Save the Planner's output as `plan.md`. Read it thoroughly. Add or adjust any decisions before proceeding. The plan is the spec from here.

When you are satisfied with `plan.md`, confirm to the session that Stage 1 is complete and Stage 2 is beginning. Use the exact transition message: `Stage 1 complete. Proceed to Stage 2.`

Stage 2: The Implementer

> **Antigravity:** In the same session, paste the full contents of `requirement.md` and `plan.md`, then instruct the agent to proceed to Stage 2. It will load the Implementer role and produce `git_summary.py`.
>
> **Watch For:** Python code that references only the standard library. The implementation should follow the plan's data structures and approach. If the Implementer

> adds error handling beyond what the plan specified, or imports a library not in the standard library, note it — that is a scope violation the Reviewer will flag.

Save the output as `git_summary.py`. Read it once before proceeding to Stages 3 and 4.

Stages 3 and 4: Running in Parallel

Stages 3 and 4 are independent — the Reviewer and Tester both need `git_summary.py` but not each other's output. This is where running two simultaneous sessions pays off.

What "parallel" means here is simple: both stages start from the same implementation file, but in separate contexts. They do not share memory, they do not wait on each other, and neither stage can silently bias the other. Each produces its own artifact. Stage 5 begins only after you have both.

> **Antigravity:** Click the grid icon (upper right) to open **Manager View**. You will see your current session as a card. Create a second agent card by clicking the + or **New Agent** button on the canvas.
>
> You now have two agent cards. In Card 1, load `agents/reviewer.md` and paste `requirement.md` and `git_summary.py`. In Card 2, load `agents/tester.md` and paste `requirement.md` and `git_summary.py`. Send both. They will run simultaneously.

> **Watch For:** Card 1 producing `review-notes.md` in the structured format (Correctness, Missing handling, Scope violations, Simplicity findings). Card 2 producing test code and a `test-report.md`. Neither card should be producing output that depends on the other.

If you prefer to run sequentially — one session for the Reviewer, then a second for the Tester — the output will be identical. Parallel is faster; sequential is simpler. Use whichever you are comfortable with.

Wait for both cards to complete before proceeding.

Reading the Outputs

Before Stage 5, read both outputs together.

From the Reviewer: Are the findings located? Does each one cite a specific line or function? If the Reviewer produced general impressions ("this could be clearer"), the role constraint did not hold. Set that finding aside and act only on the located ones.

From the Tester: Which tests passed on first run? Which failed? For the failing tests: is the failure because the implementation is wrong, or because the test's assertion is wrong? Both are useful information. A test that fails because the implementation misses an edge case identified in the plan is a correctness failure. A test that fails because the tester wrote an assertion that contradicts the requirement is a test failure — fix the test, not the code.

Save `review-notes.md`, `test-report.md`, and `test_git_summary.py`.

Stage 5: Addressing Findings

> **Antigravity:** Open a new session (or return to your Stage 2 session if the context is still clean). A session is still clean if it has only the Stage 2 exchange in it and the agent is still following the original implementation scope without commentary drift. Load `agents/implementer.md`. Paste `git_summary.py`, `review-notes.md`, and `test-report.md`. Instruct the agent to fix the correctness issues and failing tests.
>
> **Watch For:** Changes that are targeted — fixes for the specific issues raised, not rewrites of working sections. If the Implementer rewrites a section that had no findings, it has stepped outside its scope. Note it; that is scope drift.

When the revised `git_summary.py` is ready, run the tests again yourself. If the tester wrote `unittest` tests, run `python -m unittest test_git_summary.py`. If the tester wrote `pytest` tests and `pytest` is available in your environment, run `pytest test_git_summary.py` instead. They should pass. If they do not, the Stage 5 Implementer missed something — run Stage 5 again with the updated test output.

Check the optional gate: tests pass, review findings resolved, no new scope violations introduced. If yes, proceed.

Stage 6: The Documenter

> **Antigravity:** Open a new session. Load `agents/documenter.md`. Paste `requirement.md`, the final `git_summary.py`, and `test-report.md`. Let the Documenter produce `README.md`.
>
> **Watch For:** A README with all required sections, including "What it does not do." The limitations should match the requirement's Out of scope section. If the test report showed a failing test that was not resolved, the README should note that limitation.

Save `README.md`. Read it as a user who has not seen the requirement or the implementation.

What Just Happened

You ran six stages across five roles. The Planner filled the requirement's gaps before code was written. The Implementer built exactly what the plan specified. The Reviewer found the implementation issues the Implementer could not see from inside the code. The Tester confirmed which requirements were met with objective checks. The Implementer fixed what was found. The Documenter described the result for someone who was not in the room.

Each stage did one thing. Each handoff was a structured file. The coordinator held the sequence. The mandatory gate caught the underspecification before it became wrong code.

That is the pipeline. One more chapter to close the loop.

> **Key Takeaway:** Running the pipeline is following the sequence the coordinator defines, reading each output before confirming the next stage, and trusting that specialized roles produce better results than one agent doing everything — because you have now seen both.

Chapter 11: Your Coding Team at Work

You have a working pipeline for `git-summary`. A requirement document, a plan, an implementation, a test suite, review notes, and a README — produced by five specialized roles in a defined sequence, with a mandatory gate at the most expensive decision point.

Before you extend it, there is something you should know about where this system came from.

The Reveal

This book was written using the same system you just built.

Not metaphorically. The chapters you read were produced by a pipeline with specialized agent roles: one that defined the principle to be taught, one that designed the hands-on scenario, one that reviewed Antigravity instructions for accuracy, one that read the draft as a confused first-time reader, one that checked continuity across chapters, one that connected Part 1 principles to Part 2 mechanisms. A coordinator skill — `skills/ebook-writer.md` — sequenced all of them, defined what each stage received and produced, and held a gate before prose was written.

The coordinator did not always follow the stages faithfully. Roles drifted. Stages collapsed. Some runs produced better output than others. The gate caught most of the problems before they propagated.

Sound familiar?

The system that wrote this book is the system the book teaches. The failure modes in Chapter 6 are the failure modes the pipeline ran into. The mandatory gate in Chapter 9 is the gate that prevented the most expensive rework. You were not being taught an abstract methodology — you were watching it operate on itself.

What You Now Have

Look at your Part 2 project directory:

```
requirement.md          — the spec that every stage
read from
plan.md                 — the Planner's decisions,
filling the gaps
git_summary.py          — the implementation
test_git_summary.py     — tests that verify the
requirement, not the code
test-report.md          — what passed, what failed,
what was missing
review-notes.md         — located findings from
someone
                          who read without running
README.md               — documentation for someone
who was not in the room
agents/
  planner.md
```

```
    implementer.md
    reviewer.md
    tester.md
    documenter.md
  skills/
    feature-sprint.md
```

This is a team. Not a team that meets, not a team that negotiates, not a team that requires coordination tooling. A team defined in files, sequenced by a coordinator, executable by you with a session and a skill file.

The system is portable. Take it to a different feature, a different codebase, a different language. Change the requirement document. Change the plan's edge case decisions. Run the pipeline again. The roles do not change. The coordinator does not change. The handoff contracts do not change. The team is already built.

Extending the Team

The five roles you built are sufficient for `git-summary`. They are not exhaustive for every feature you will ever build. Some features need roles these five do not cover.

A **Security Reviewer** receives the code and looks specifically for security implications the general Reviewer may have deprioritized: input validation, command injection risk, file path traversal, credential handling. `git-summary` runs a git command with a user-provided path — a Security Reviewer

would flag that `subprocess.run` with shell=True would be dangerous here (use shell=False with a list), and check that the path is validated before being passed to git.

A **Dependency Auditor** receives the requirement and checks whether the standard library constraint is met, and whether any third-party library would make the implementation significantly simpler or safer — and whether that tradeoff is worth revisiting the constraint.

A **Compatibility Checker** runs the code in an environment different from the one the Implementer produced it in. macOS and Linux have different git output formats. The Tester may not have caught that.

These are roles you add when you need them. They follow the same structure: input, output, constraints. They slot into the coordinator as additional stages. The pipeline grows with the requirements.

The Next Feature

Pick one from the natural extensions of `git-summary`:

Option A: `--author` filter — show only commits by a specified author. Low risk. New flag, existing data flow. The Planner's main decision: what happens when the author has no commits in the period?

Option B: `--format json` — output the summary as JSON instead of plain text. Medium risk. New output format, same data. The Documenter will need to describe the JSON structure.

Option C: `--since` date flag — instead of days, accept an ISO date string. Medium risk. New argument type, date parsing, edge cases around timezone handling.

Write the requirement for whichever you choose. Run the pipeline. The team is already staffed.

Where the Path Continues

You have the document-driven coordination layer. You know its limits from Chapter 6. The next level — programmatic orchestration, independent contexts, validated handoffs — is the subject of a future volume.

That path is worth taking when the failure modes from Chapter 6 cost more than the infrastructure would. When you are running pipelines in production without human review. When output variance is a product defect, not an inconvenience. When you need the team to run without you.

For now, you have a system you can run, a team you understand, and an honest picture of where it stops. That is a complete starting point.

> **Key Takeaway:** The pipeline is portable. The team you built for `git-summary` is the team for the next feature, and the one after that. Add roles when you need them. The coordinator grows with the work.

Conclusion

You built two pipelines. One produces articles. One produces code. They use different roles, different file names, different handoff contracts. They use the same design.

That is the thing worth keeping: the design is portable. Specialized roles. Explicit inputs and outputs. A coordinator that enforces the sequence. A gate at the most expensive decision point. Shared memory that every stage reads from.

You can take that design to any complex, repeatable task you do regularly. The roles change. The coordinator changes. The pattern does not.

What You Have

After Part 1:

A content pipeline with three agent roles — Researcher, Writer, Adversarial Reader — and a coordinator skill that sequences them. Not just what each does, but what breaks when it is absent. The researcher prevents invented examples from slipping in unnoticed. The adversarial reader finds the gaps the writer cannot see. The voice constraint prevents drift across sessions. The coordinator enforces the sequence that makes all of it work together.

And `article-v1.md` and `article-v2.md` side by side — the argument the book did not make, but let you observe.

After Part 2:

A coding team with five roles — Planner, Implementer, Reviewer, Tester, Documenter — and a coordinator that runs them through a full feature sprint. A working `git-summary` tool, a test suite that verifies the requirement, and a README that describes what the tool does not do. The mandatory gate after the Planner, which you now understand costs nothing to take and potentially saves a full implementation run.

What the System Cannot Do

Chapter 6 named these, but they are worth restating as you close the book.

The coordinator runs inside the same context it is coordinating. Stages collapse. Roles drift. Output varies between runs. The gate is your primary quality instrument, which means quality depends on you knowing what good looks like at each stage.

This is not a reason to put the book down. It is a reason to use the gates seriously and calibrate your trust against actual results, not against what the pipeline is supposed to produce.

When these failure modes cost more than the infrastructure of a proper orchestration layer, you will know. The frameworks in Chapter 6 are there for that moment. They are not the starting point — this book is.

The Next Feature

The most common question after finishing a book like this is: what do I build next?

The answer is whatever you have been avoiding because it was too complex for one agent.

You know the structure now. Write the requirement. Identify the roles the work actually needs. Write the handoff contracts. Build the coordinator. Run the mandatory gate. The team is built the same way every time — the domain is the only thing that changes.

The roles in this book are a starting point, not a complete set. A Security Reviewer. A Compatibility Checker. A Dependency Auditor. An API Contract Validator. Every team you build will need a slightly different roster. What stays constant is the structure of each member: what they receive, what they produce, what they are not allowed to do.

Where the Path Goes

This book teaches the first level: document-driven coordination, human-executed, with Antigravity as the runtime. It is useful at this level and honest about its ceiling.

The second level is programmatic orchestration — code that routes work between agents, validates outputs against schemas, runs stages in independent contexts, and does not require your attention at every gate. That is the work of the frameworks named in Chapter 6, and of a future volume.

The third level — which frameworks like OpenAI's Symphony are beginning to approach — is the system where you manage the work, not the agents. You move a ticket to "Ready." The pipeline picks it up, plans it, implements it, tests it, reviews it, and opens a pull request. You review the PR. That is the loop.

We are early in that story. The infrastructure is being built now. The teams that get there first will be the ones who already understand what the pipeline needs to do — because they built one by hand.

That is what you just did.

A Note on This Book

This book was written by a team of agents coordinated by a skill file. The roles are named in `skills/ebook-writer.md`. The coordinator ran nine stages per chapter, with a mandatory gate after the outline and before prose was written. The pipeline ran on itself — the book that teaches agent teams was produced by one.

That is not a disclaimer. It is evidence. The system works well enough to produce something you just read from cover to cover. It also ran into every failure mode described in Chapter 6, and the gates caught most of them.

Build the team. Run the gates. Keep the output.

The companion volume, covering programmatic orchestration and autonomous pipeline execution, is in progress.