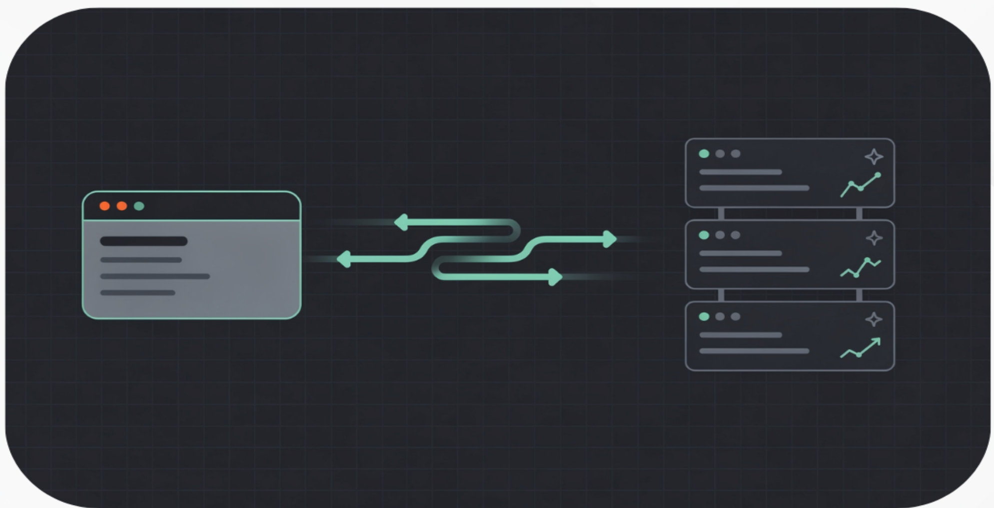
PRODUCTION AI AGENTS WITH PYDANTICAI

ENGINEERING DISCIPLINE FOR AGENTS YOU CAN LEAVE RUNNING OVERNIGHT



Production AI Agents with PydanticAI

Engineering discipline for agents you can leave running overnight

Nuno Bispo

This book is available at
https://leanpub.com/production-ai-agents-with-pydantic-ai

This version was published on 2026-07-03

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Nuno Bispo

Tweet This Book!

Please help Nuno Bispo by spreading the word about this book on Twitter!

The suggested tweet for this book is:

Just picked up Production AI Agents with PydanticAI – not another prompt tricks book. Real engineering: contracts, eval gates, cost SLOs, and a maintainer agent you ship chapter by chapter. #AgentsAreSoftware

The suggested hashtag for this book is #AgentsAreSoftware.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#AgentsAreSoftware

Also By Nuno Bispo

pip install yours
Practical Pydantic
Python's Magic Methods

Contents

Preface 1
Who this book is for 1
The project you will build 2
How to read this book 2
Who this book is not for 2

Chapter 01 – Your PoC Is Not in Production 4
Failure mode 1: The silent retrieval miss 4
Failure mode 2: The unversioned prompt change 6
Failure mode 3: The cost blowup nobody saw 7
What the three failures have in common 9
Why Pydantic AI 10
What you will build 11

Chapter 02 – The First Agent 13
The output contract 13
Setup 14
The agent 15
Fetching issues 16
Running the classification 17
What the disagreement tells you 20
What you have, and what you do not 22
Ship it 22

Chapter 03 – Shipping the Bad Version 24
What the deploy looks like 24
Setup 25
Posting a comment 26
The polling loop 27
The crontab 30
The first comment 31

Why minimal beats elaborate 31
The embarrassing inventory 32

Preface

Your PoC works on your laptop. The classifier hits the label most of the time, the model returns the structured output you asked for, the stakeholders nodded at the demo. The PoC has a deployment URL now. Your manager calls it production.

It is not production.

Production is what happens after the demo. Production is the morning you change the system prompt and quality drops 8% on traffic you cannot see. Production is the agent that worked yesterday and refuses to escalate today, and you cannot tell which is the bug. Production is the model that quietly cost $4,300 last month because nobody put a budget on it. Production is the on-call engineer at 11:42pm trying to roll back a prompt change with no rollback plan, no version history, and no idea which deploy made the agent confidently wrong.

None of these are AI problems. They are engineering problems. You already know how to solve them, for APIs. Your team versions API responses. Your team writes contract tests. Your team has SLOs, canaries, rollback procedures, a postmortem template, and someone whose job it is to own the alert at 3am. That discipline is what is missing from how teams ship LLMs today, and the gap is not because agents are exotic. The gap is because teams forget, the moment a model is involved, that everything they already know about shipping software still applies.

This book closes that gap. Agents are software. The engineering discipline you already have for APIs is the discipline you need for agents - translated, not replaced. Pydantic AI is the framework where that translation is natural, because typed outputs and dependency injection are the seams that let you test, version, and roll back the parts of the system that other frameworks leave loose.

Who this book is for

This book is for Sam. Sam is a senior engineer - Python fluent, comfortable with Docker, on the Slack channel where the on-call alerts arrive - who has shipped

a couple of LLM proof-of-concepts. They know what an embedding is, what a percentile is, what an SLO is. They have called the OpenAI or Anthropic SDK enough times to have an opinion about both. They have just been told, by a tech lead or by their own product instinct, that the next agent has to be real. Real customers, real traffic, real on-call rotation. Sam is one bad incident away from a difficult conversation, and the question they need answered is not "how do I build an agent" but "how do I build one I would trust to leave running while I sleep."

If that is you, the book is for you.

The project you will build

The project that runs through every chapter is an open-source maintainer assistant: a Pydantic AI agent that triages incoming issues on a GitHub repository. Issue data comes from `**triage-lab**`, a companion repository of curated fixtures (50 closed issues, golden datasets for eval and retrieval). Chapter 2 classifies those fixtures on your laptop. Chapter 3 ships the agent to your fork of `triage-lab` via a CLI cron job - posting comments on real issues, minimally, and on purpose. The classification is rough. Every new issue gets the same public comment whether confidence is 0.51 or 0.97 - no escalation, no draft reply, no tools. It will mislabel duplicates and gray-zone cases, that is the point. Every subsequent chapter fixes one specific issue with a test, a metric, and a rollback plan attached.

How to read this book

Chapter 1 is the argument. Chapter 2 is the smallest possible Pydantic AI agent, running on your laptop against closed issues from `triage-lab`. Chapter 3 ships that agent to your fork via a CLI cron job - minimally, and on purpose.

From chapter 4 forward, every chapter fixes one specific way the v0 agent is wrong. Tools and dependency injection. Retrieval that does not lie. The versioned output contract. Unit tests for the deterministic seams. The eval suite - the chapter you probably bought the book for. Observability. Cost engineering. Human-in-the-loop and the kill switch. Canary rollout and quality SLOs. And the runbook that turns the agent from deployed into operated.

Who this book is not for

This book is not for everyone. It is not for first-time LLM users; if you have never sent a structured-output request and parsed the response, the first 50 pages of the Pydantic AI documentation will serve you better than this book will. It is not for readers looking for prompt-engineering tricks. The book treats prompts as deployable artifacts with rollback plans, not as the place where the magic happens. It is not for fiction writers, founders evaluating AI as a category, or managers shopping for an AI strategy. It is for the engineer who has the agent in a notebook and has been told it needs to be in production by the end of the quarter. If that is the conversation you are having this week, turn the page.

Chapter 01 - Your PoC Is Not in Production

The preface made a claim. This chapter defends it.

The claim is that agents are software. The engineering discipline you already have for APIs - versioned schemas, contract tests, SLOs, canaries, rollback procedures, on-call rotations - is the discipline that is missing from how teams ship LLMs today. The claim is easy to nod at and harder to act on. The gap between nodding and acting is the gap this book exists to close.

This chapter explains three failure modes, drawn from systems already shipped and quietly degraded, each one showing what the gap looks like in the room. The first is a retrieval failure that produces wrong answers without anyone noticing. The second is a prompt change that breaks an SLA three days later. The third is a cost blowup that nobody saw until the invoice arrived.

The examples use different surfaces - a financial-document chatbot, a customer-facing classifier, an internal support agent. From chapter 2 onward, you apply the same discipline to a fourth: an open-source maintainer agent triaging GitHub issues in `triage-lab`. Different corpus. Same failure mechanics.

These are not the only failure modes that matter. They are the three that map most cleanly onto the discipline you already have, and the discipline this book asks you to bring. After all three, the pattern is impossible to miss. After the pattern, the rest of the book stops feeling like a sequence of opinions and starts feeling like a checklist.

If you have shipped an LLM proof-of-concept and are now being asked to ship something real, at least one of these failure modes is already happening to your team. You may not know yet.

Failure mode 1: The silent retrieval miss

A RAG agent over a corporate financial document. It has been live for two months. Stakeholders are happy. The chatbot returns answers that sound

authoritative.

A user asks a reasonable question - "How many employees does the company have"? The answer is in the document, on a single page in the Human Capital Resources section. The agent retrieves three chunks from the index, none of which contain the figure. It refuses to answer: "I could not find this in the provided document".

The refusal looks correct. It is not. The information is in the document; it is on page 16. The retriever ranked adjacent pages higher because naive fixed-size chunking split the relevant paragraph across the page 15–16 boundary. The model received chunks about compensation programs and operations centers. It correctly declined to invent a number from the wrong context. From the user's perspective, the system simply does not know.

Nothing in the production system noticed this. No exception was thrown. No alert fired. The agent responded in 1.9 seconds with a fluent, well-formatted refusal. The user shrugged, asked something else, and within a week stopped using the system. They did not complain. They did not file a ticket. Their trust eroded silently.

This is a complete failure of the production system. It is also a failure that is invisible to every metric the team is currently watching. Latency is fine. Error rate is fine. Token throughput is fine. The only signal that would have caught this lives in a field that the team does not log: the retrieval score of the top-ranked chunk. In this case, that score was 0.176. The threshold below which a retrieval is suspect is around 0.75. The system answered with full confidence on a request whose retrieval was four times below the floor.

The fix is not complicated. It is one structured log field per request, plus a moving average over a rolling window, plus an alert when the moving average of low-confidence retrievals exceeds 10%. It is also a small golden dataset of (query, expected page) pairs that runs in CI, asserting that a chunking change does not drop the retrieval hit rate below 75%. None of this is novel. It is the same discipline that already applies to your API: log the field you care about, set a threshold, alert when the threshold is crossed, and write the test that prevents the regression in the first place.

What was missing was not a tool. It was the engineering posture that says "I cannot ship this without measuring it". Without that posture, the team measured everything except the thing that decided whether the system was useful. With it, the failure mode is impossible to deploy in the first place.

On a document corpus, the miss was a page boundary. On an issue corpus, the same miss is a chunk boundary - title in one chunk, the duplicate issue ID in a comment thread the retriever never indexed - or a low similarity score on genuinely related closed issues while the model still answers with confidence. The maintainer agent in chapter 2 has no retrieval layer yet; when it mislabels a duplicate, that is often why.

Chapter 5 builds the retrieval layer of the maintainer agent and includes the golden dataset and the CI gate. Chapter 8 makes the retrieval-quality assertion part of the eval suite. Chapter 9 makes the retrieval score a first-class observability metric in production. Three chapters; one failure mode closed.

Failure mode 2: The unversioned prompt change

A team has a customer-facing classification agent running in production. It has been stable for three months. On a Tuesday, an engineer opens a pull request titled “small prompt clarity improvement”. The diff is fifteen lines of system prompt - clarifying a category definition, adjusting a few examples in the few-shot block. The eval suite passes. CI is green. The change ships.

By Thursday, a customer-success representative posts in Slack: “Has the agent changed? Tickets that used to escalate to a human are auto-resolving in a way that’s wrong”. The team checks the deploy history and sees nothing relevant - no model change, no library upgrade. They check the Sentry dashboard. Error rate is fine. Latency is fine. The agent is responding successfully to every request. From every dashboard the team owns, the system is healthy.

By the next Monday, three more customer-success messages have arrived. A senior engineer pulls the prompt change off the audit trail and reads the diff. The clarification of the category definition has shifted the agent’s threshold for “this is a billing question” downward; it is now classifying ambiguous requests as billing-resolvable when previously it would have escalated. The eval suite did not catch this because the golden dataset for “billing vs. escalate” had three examples and none of them sat near the new threshold.

The fix takes ninety minutes - revert the prompt to its prior version, redeploy, write a retroactive incident note. The damage takes longer. Several dozen customers got responses they should not have. The customer-success

team spends a week unpicking the ones they can identify. The ones they cannot identify, they do not.

This was not a model regression. This was a code change shipped without the test that mattered. The team had treated the system prompt as a comment - text that lives next to code, not text that *is* code. There was no version of the prompt pinned in the eval suite. There was no canary that would have routed 5% of traffic through the candidate prompt and surfaced the threshold shift before the other 95%. There was no quality SLO that would have alerted on the escalation-rate change on Thursday morning. The prompt was treated as a configuration tweak; in fact it was a behavioral change to a customer-facing system, shipped without any of the controls the team would have required for an equivalent code change.

The discipline that would have prevented this is not exotic. Pin the prompt as a deployable artifact with a version. Add the boundary examples - the ones near the threshold - to the eval suite, not just the obvious ones. Route prompt changes through a canary the way you route schema changes. Set an SLO on escalation rate and alert when it drifts more than two percentage points over a rolling window. Each of these is a one-day task for an engineer who has shipped APIs. None of them are present in the team's current operating model for the agent.

What was missing was not technology. It was the recognition that a prompt is a deployable artifact that affects customer-facing behavior, and therefore deserves the same treatment as any other customer-facing artifact. Without that recognition, prompts get edited like comments and shipped like config tweaks. With it, the failure mode is caught at PR time on Tuesday morning, not on Thursday afternoon when a customer notices.

The maintainer agent you ship in chapter 3 has the same shape: a system prompt, a five-label enum, and gray-zone fixtures your eval set does not cover until chapter 8. A "small clarity improvement" to that prompt is the same class of change as this Tuesday PR - customer-facing behavior with no boundary tests.

Chapter 6 establishes the output contract that pins the prompt's expected behavior. Chapter 7 writes the prompt-snapshot test that fails CI when the prompt changes without an intentional eval update. Chapter 8 makes the eval suite the gate. Chapter 11 builds the rollback procedure. Chapter 12 puts the canary in front of every prompt change. Five chapters; one failure mode closed.

Failure mode 3: The cost blowup nobody saw

A team has a Pydantic AI agent running an internal customer-support tool. Cost-per-month at launch was forecast at $1,200, based on token-cost estimates from the eval set. After three months, the OpenAI invoice for the previous month is $4,800. Finance asks why. Engineering does not have an answer ready.

It takes the team four days to reconstruct what happened. The team had shipped two changes in the previous month: a new tool that fetches context from a third-party API, and a system-prompt update that encourages the agent to “verify your reasoning before responding”. Either change in isolation would have been fine. Together, they caused the agent to enter a longer tool-and-reasoning loop on roughly 30% of requests. Average tokens per request went from 1,800 to 6,400.

Chapter 2 already prints dollars for a fifty-issue classification run. This failure is what happens when nobody treats that line as a budget. Two weeks after the invoice arrives, a different signal surfaces: one customer is responsible for 18% of total agent traffic. They are not a power user - they have a buggy retry loop in their integration that triggers the agent four times for every actual user request. The agent gets called 40,000 times a month from one customer’s broken client. None of those calls produce billable outcomes for the team; they are pure cost. The team has no per-user cost telemetry, so this customer was indistinguishable from the rest of the traffic in every dashboard.

The two signals together explain the invoice. Each is an instance of the same gap: cost was treated as a number to read at the end of the month, not a metric to observe in real time.

This is the failure mode that destroys budget approval for the next agent. The CFO does not see “the team learned something”. The CFO sees “the team shipped a system whose costs they could not predict or control”. The next agent proposal goes through a different approval process, with different scrutiny, often with different ownership. The damage compounds beyond the immediate invoice.

The fix is not about being clever. It is about measuring the right thing. Cost-per-request is a first-class metric, derived from token counts and per-model pricing, exported on every span the same way latency is. Cost-per-user is a

derived metric on top of cost-per-request, bucketed by whatever identifier the system already uses for authentication. A cost SLO - "cost-per-request must not exceed $0.04 at the 95th percentile" - runs in CI on the eval set, fails the PR when a change crosses it, and runs in production as a rolling alert. None of this is new engineering. It is the same observability discipline that already applies to latency and error rate, extended one more axis.

What was missing was not insight. It was the act of treating cost as a metric on equal footing with latency and error rate. Latency has a budget. Latency has alerts. Latency has a place in every postmortem. Cost should have all three. Without that, the system's economics are a surprise every month - sometimes a pleasant one, often not.

Chapter 9 makes cost-per-issue a structured log field on every agent run and a span attribute on every trace. Chapter 10 builds the cost SLO and the cost regression gate in CI. Chapter 12 wires the cost SLO into production alerting. Three chapters; one failure mode closed.

What the three failures have in common

These three failures are not the same incident. The retrieval miss is a measurement gap. The prompt change is a versioning and gating gap. The cost blowup is an observability gap. They look like three different problems.

They are not three different problems. They are three instances of the same problem: an artifact that affects production behavior was deployed without the discipline that would have applied to any other artifact that affected production behaviour.

The retriever's chunk boundaries determined the system's accuracy. They were treated as an implementation detail rather than as a configuration with a measurable quality and a versioned shape. The system prompt determined the agent's classification threshold. It was treated as a piece of explanatory text rather than as a customer-facing artifact with a behavioral contract. The cost-per-request determined whether the system could be afforded at all. It was treated as a monthly report-out rather than as a metric with a budget.

In each case, the team had every tool they needed already. They had structured logging. They had a CI pipeline. They had dashboards and alerts. They had a postmortem template they had used for API regressions. They had pull-request reviews and version control. What they did not have was the

recognition that all of these tools apply to the parts of the system that involve a model, with no modification beyond what they already ran for the parts that did not.

This is the entire thesis of this book. An LLM call is a non-deterministic IO boundary, no different in shape from a flaky third-party API. The discipline that already applies to flaky third-party APIs - versioned contracts, canaries, SLOs, alerts on tail latency, rollback procedures, postmortems with timelines - applies to LLM calls, translated into the right metric and the right artifact. Agents are software. The discipline you already have for shipping software is the discipline you need for shipping agents.

There is one framework where this translation is more natural than in any other framework available today. The rest of the chapter is about why.

Why Pydantic AI

In production engineering, a *seam* is the place in a system where you can substitute one implementation for another at test time without changing the rest of the system. Seams are how you write fast tests against a slow database. Seams are how you assert that your service handles a 5xx from a downstream API without ever making the downstream call. A system without seams is a system that can only be tested end-to-end, which is to say, a system that cannot be tested at all on a developer's laptop.

Most LLM frameworks have very few seams. The agent is constructed dynamically, the prompt is templated at runtime, the tools are bound to the agent in a way that makes them hard to substitute, and the output is a string that the calling code parses with a regex or a try/except. None of those parts has a stable shape that a test can assert against. To test the system, you have to run the system. To run the system, you have to call the model. CI either becomes slow and flaky, or the team stops testing the parts that matter.

Pydantic AI is built around two seams that change this. The first is the typed output:

```python
from pydantic import BaseModel, Field
from typing import Literal

class IssueClassification(BaseModel):
    label: Literal["bug", "feature-request", "question", "duplicate", "spam"]
    confidence: float = Field(ge=0.0, le=1.0)
    rationale: str
```

This is a Pydantic model the agent must produce. Every model response is validated against it before any of your code sees it. The seam is the model class itself: you can assert against it without running the LLM, you can version it the way you version an API response, you can write a test asserting that `result.label == 'bug'`, and you can run that test against a deterministic stub of the agent in CI. The output stops being a string you parse and starts being a contract you test.

The second seam is dependency injection. Pydantic AI's agent takes a `deps` parameter that is passed through to every tool. At production runtime, `deps` carries (for instance) a real GitHub client and a real database. In tests, `deps` carries fakes. Substituting them is one line:

```python
result = await agent.run("triage this issue",
↪ deps=FakeDeps(github=fake_github))
```

That single substitution is what makes the agent unit-testable. Without it, every test has to either hit the real GitHub or monkey-patch deep into the library. With it, a tool test is the same shape as any other test that uses a fake for a downstream service. The discipline you already have for testing API endpoints with a fake database connection is, without modification, the discipline you use for testing this agent with a fake GitHub.

If you have shipped LangChain to production, you already know which seams it does not give you. The same patterns are achievable there, with significantly more scaffolding - but the patterns sit uphill from the framework's defaults rather than being the framework's defaults themselves. The rest of this book leans on these two seams in every chapter. If they were not present, none of the chapters that follow are practical.

What you will build

The agent that runs through every chapter is an open-source maintainer assistant. By the end of chapter 2 it classifies an incoming GitHub issue into

a typed enum, running on your laptop against the last fifty issues of a real public repository. By the end of chapter 3 it is live on your own repo, posting comments on real issues via a CLI cron job. The classification will be rough. Every issue gets the same public comment whether confidence is 0.51 or 0.97. It will mislabel duplicates and gray-zone cases. None of that is a problem; it is the curriculum.

Every chapter from chapter 4 onward fixes exactly one of those reasons it is bad. Tools, the output contract, the test suite, the eval suite, observability, cost, human-in-the-loop, the canary, the runbook. Each chapter ends with the v0 agent fixed in one specific way, and a new failure mode visible that the next chapter addresses. By chapter 13, the agent is the same agent it was on chapter 3 - only now it survives traffic, prompt changes, and Monday morning.

The argument is done. Stop reading. Open your editor. Chapter 2 is waiting.

Chapter 02 - The First Agent

The argument is done. This chapter builds the thing.

By the end of this chapter, you have a Pydantic AI agent that reads a real GitHub issue and returns a validated classification: `bug`, `feature-request`, `question`, `duplicate`, or `spam`. It runs on your laptop against 50 closed issues from `triage-lab`, the book's issues repository - curated fixtures with labels that match the agent's output contract. It tells you where it disagrees with the human labels. It tells you what each classification costs.

None of that is impressive yet. It is the foundation that every subsequent chapter builds on. The agent you build here is the agent you will ship in chapter 3, observe breaking in chapter 4, and fix - systematically, one failure mode at a time - for the rest of the book.

The output contract

Before writing a single line of agent code, write the output model. This is not a habit. It is a constraint that changes how you think about the problem.

```python
# models.py
from typing import Literal

from pydantic import BaseModel, Field


class IssueClassification(BaseModel):
    label: Literal["bug", "feature-request", "question", "duplicate", "spam"]
    confidence: float = Field(ge=0.0, le=1.0)
    rationale: str
```

Three fields. `label` is a `Literal` enum - not a string the model can fill however it likes, but a closed set of five values that Pydantic will enforce. `confidence` is a float bounded between 0 and 1. `rationale` is a string: the agent's reasoning in one or two sentences, which you will use in chapter 8 when you build the eval suite.

That `Literal` is already doing production work. If the model returns `"enhancement"` or `"bug report"` or any value outside the five, Pydantic AI retries the model with a validation error - automatically, before your code ever sees the result. The output is either a valid `IssueClassification` or an exception. There is no third option. That is what a contract looks like.

You will version this model in chapter 6. You will write tests against it in chapter 7. You will run evals against it in chapter 8. All of that is possible because the output has a stable, typed shape. Start with the shape before anything else.

Setup

All runnable code for this chapter lives in `chapter-02/` at the root of the book companion repository. Pydantic AI requires **Python 3.10+**; the project pins **3.12** via `.python-version` for reproducibility.

The example repository

Issue data lives in `triage-lab` - 50 closed fixtures labeled with the book's five categories, plus golden datasets for chapters 5 and 8. The default repo is nunombispo/triage-lab (read-only eval).

Install uv if you do not have it, then bootstrap the project:

```
1  cd chapter-02
2  uv sync                    # install pinned deps from uv.lock
3  cp .env.example .env       # then fill in keys
```

If you are creating the project from scratch instead of cloning the repo:

```
1  mkdir chapter-02 && cd chapter-02
2  uv init --name chapter-02 --no-readme
3  uv python pin 3.12
4  uv add "pydantic-ai-slim[anthropic]==1.97.0" "httpx>=0.28,<1"
   ↪  "python-dotenv>=1.0,<2"
```

`pydantic-ai-slim[anthropic]` pulls in only the Anthropic provider - enough for this chapter without installing every other model backend. Versions

are pinned in `pyproject.toml` and `uv.lock` so your environment matches the book.

Create a `.env` file (or copy from `.env.example`):

```
ANTHROPIC_API_KEY=sk-ant-...
GITHUB_REPO=nunombispo/triage-lab
GITHUB_TOKEN=ghp_...              # optional for classify.py; raises rate limit
```

The GitHub token is optional for 50 issues but worth having. It takes 30 seconds to generate at github.com/settings/tokens with no scopes required for public repo read access.

The agent

```
# agent.py
from dotenv import load_dotenv
from pydantic_ai import Agent

from models import IssueClassification

load_dotenv()

SYSTEM_PROMPT = """You are an open-source issue classifier. Given a GitHub
↪  issue \
title and body, classify it into exactly one of these categories:

- bug: reports unexpected behavior or a clear defect
- feature-request: asks for new functionality or a change in behavior
- question: asks how to do something or seeks clarification
- duplicate: the same issue already exists (use only when certain)
- spam: irrelevant, automated, or off-topic content

Return your confidence as a float between 0.0 and 1.0. Return your rationale \
in one or two sentences explaining what signals drove the classification."""

agent = Agent(
    "anthropic:claude-sonnet-4-6",
    output_type=IssueClassification,
    system_prompt=SYSTEM_PROMPT,
)
```

Four things to notice.

`"anthropic:claude-sonnet-4-6"` is the model string — Claude Sonnet 4.6, a capable mid-tier Anthropic model. Pydantic AI uses `provider:model-name` format. To switch to a cheaper model, replace it with `"anthropic:claude-haiku-4-5"`. To switch to OpenAI, use `"openai:gpt-4o-mini"`. That is the entire migration. The `agent.run()` call, the output model, the system prompt — none of it changes. The model-agnostic interface is one of the two seams chapter 1 named. This is what it looks like in practice.

`output_type=IssueClassification` is the other seam. The agent is not instructed to "return JSON with a label field." It is handed a Pydantic model and told to satisfy it. Pydantic AI handles the serialization, the validation, and the retry on validation failure. Your code works with a typed object, not with a string.

The system prompt names the five categories explicitly and distinguishes the edge cases that matter — "use `duplicate` only when certain" is doing real work. The model's behavior on ambiguous inputs depends on the specificity of these instructions. Chapter 7 will show you how to test this specificity without calling the model.

`rationale` in the output model is not decoration. You will query it in chapter 8 when an LLM judge needs to evaluate whether the classification was reasonable. Put it in the model now even though you will not use it until later. The cost is negligible. The cost of retrofitting it later is not.

Fetching issues

```
1   # fetch_issues.py
2   import os
3   from typing import Optional
4
5   import httpx
6
7   DEFAULT_REPO = "nunombispo/triage-lab"
8
9
10  def fetch_closed_issues(
11      repo: Optional[str] = None,
12      limit: int = 50,
13      token: Optional[str] = None,
14  ) -> list[dict]:
15      repo = repo or os.getenv("GITHUB_REPO", DEFAULT_REPO)
16      headers = {"Accept": "application/vnd.github+json"}
```

```
17      if token:
18          headers["Authorization"] = f"Bearer {token}"
19
20      issues: list[dict] = []
21      page = 1
22      while len(issues) < limit:
23          response = httpx.get(
24              f"https://api.github.com/repos/{repo}/issues",
25              params={
26                  "state": "closed",
27                  "per_page": 100,
28                  "page": page,
29              },
30              headers=headers,
31              timeout=10.0,
32          )
33          response.raise_for_status()
34          page_items = response.json()
35          if not page_items:
36              break
37          batch = [i for i in page_items if "pull_request" not in i]
38          issues.extend(batch)
39          page += 1
40
41      return issues[:limit]
```

The default repo is `triage-lab` – fixtures with labels that match the agent's five categories exactly.

The GitHub Issues API returns pull requests alongside issues unless you filter them – `"pull_request" not in i` drops them. The `state=closed` filter gives you issues that have human labels applied and resolved, which is what you want for comparing against.

Rate limits

The unauthenticated GitHub API allows 60 requests per hour per IP. Fetching 50 issues takes 1 request. You have headroom. For larger runs in later chapters, use the token.

Running the classification

```python
# classify.py
import asyncio
import os

from dotenv import load_dotenv
from pydantic_ai.exceptions import ModelHTTPError

load_dotenv()

from agent import agent
from fetch_issues import fetch_closed_issues

RETRYABLE_STATUS = {429, 502, 503, 504, 529}
MAX_ATTEMPTS = 5


async def run_agent_with_retry(prompt: str):
    for attempt in range(1, MAX_ATTEMPTS + 1):
        try:
            return await agent.run(prompt)
        except ModelHTTPError as e:
            if e.status_code not in RETRYABLE_STATUS or attempt == MAX_ATTEMPTS:
                raise
            wait = min(2**attempt, 60)
            print(
                f"  API {e.status_code} ({e.model_name}), retry in {wait}s "
                f"({attempt}/{MAX_ATTEMPTS})...",
                flush=True,
            )
            await asyncio.sleep(wait)


async def classify_issue(issue: dict) -> dict:
    text = f"Title: {issue['title']}\n\nBody:\n{(issue['body'] or '')[:1500]}"
    result = await run_agent_with_retry(text)
    usage = result.usage
    return {
        "number": issue["number"],
        "title": issue["title"],
        "human_labels": [label["name"] for label in issue["labels"]],
        "agent_label": result.output.label,
        "confidence": result.output.confidence,
        "rationale": result.output.rationale,
        "tokens_input": usage.input_tokens,
        "tokens_output": usage.output_tokens,
    }


async def main() -> None:
    token = os.getenv("GITHUB_TOKEN")
```

```
51      issues = fetch_closed_issues(token=token)
52      print(f"Fetched {len(issues)} issues\n")
53
54      total_input = 0
55      total_output = 0
56      for issue in issues:
57          r = await classify_issue(issue)
58          total_input += r["tokens_input"]
59          total_output += r["tokens_output"]
60          match = "✓" if r["agent_label"] in r["human_labels"] else "✗"
61          print(
62              f"{match} #{r['number']:5d}  {r['agent_label']:16s}  "
63              f"({r['confidence']:.2f})  {r['title'][:55]}"
64          )
65
66      # Sonnet 4.6 list prices: $3/MTok input, $15/MTok output
67      cost = (total_input * 3.00 + total_output * 15.00) / 1_000_000
68      print(f"\n{len(issues)} issues — {total_input + total_output:,} tokens —
        ↪  ${cost:.4f}")
69
70
71  if __name__ == "__main__":
72      asyncio.run(main())
```

Run it from `chapter-02/`:

```
1  uv run classify.py
```

Transient API errors

Fifty classifications means fifty sequential calls to Anthropic. When the API is under load, you may see HTTP **529** (`overloaded_error`) or **429** (rate limit). The script retries those with exponential backoff (up to five attempts per issue) before failing. If every attempt still returns 529, wait a minute and run again — or pass `limit=10` to `fetch_closed_issues` while you are iterating on the prompt.

The output will look roughly like this (token counts and cost depend on the model and responses; this sample is from a full 50-issue run on Sonnet 4.6):

```
Fetched 50 issues

✓ #   50  question          (0.46)  Field marked optional still required at
↪  runtime?
✓ #   49  feature-request   (0.82)  Document OpenAPI export workflow end-to-end
...
✓ #   35  bug              (0.95)  UUID4 field accepts UUID1 values without error
✓ #   34  duplicate         (0.72)  EmailStr validation fails for IDN email
↪  addresses
✓ #   33  bug               (0.38)  EmailStr rejects plus-addressing with
↪  unicode local par
✓ #   32  duplicate         (0.92)  JSON parsing fails on trailing comma
✗ #   31  feature-request   (0.60)  Cannot parse JSON with trailing comma in
↪  strict mode
✗ #   30  bug               (0.82)  Behavior differs from documentation for
↪  validate_assign
✓ #   29  question         (0.52)  Validation 3x slower after upgrade — is this
↪  a bug?
✓ #   28  question         (0.95)  Migrating `@validator` with `always=True` to
↪  v2?
✓ #   27  question          (0.93)  Best practice for generic BaseModel
↪  subclasses?
✓ #   26  question          (0.93)  Are async field validators supported?
✓ #   25  question          (0.95)  model_config extra='forbid' vs Field on
↪  every attribute
...
✓ #    2  bug              (0.97)  Segfault when validating deeply nested model
↪  (>200 leve
✓ #    1  question          (0.52)  model_validator runs twice when using
↪  inheritance

50 issues — 51,534 tokens — $0.2331
```

About twenty-three cents for the full run – Sonnet 4.6 at $3 per million input tokens and $15 per million output tokens. Your token split will differ run to run; the total line will move with it. Keep that order of magnitude in your head; you will come back to it in chapter 10.

Scroll the output and find the □ lines before you read the next section. On this run there are two.

What the disagreement tells you

On a full 50-issue run, most lines show □. Two □ lines are worth stopping on – they disagree for different reasons.

Docs vs code (#30)

```
1 ✗ #   30  bug              (0.82)  Behavior differs from documentation for
  ↪ validate_assign
```

The fixture `gray-zone-docs-mismatch` title: "Behavior differs from documentation for validate_assignment." The body: "Docs say assignment validates. Setting attribute on frozen model raises ValidationError but value still changes in `__dict__`. Documentation bug or implementation bug?"

The agent labeled `bug` at 0.82. The human labeled `question`.

The agent read "behavior differs from documentation" as a defect report – unexpected behavior against the spec. The human read the same issue as a triage question: the reporter is not sure whether the bug is in the docs or the code. The closing sentence is literally asking which bucket it belongs in.

Both are defensible. This is the gray zone between "I found a bug" and "I need a maintainer to tell me what kind of problem this is." At 0.82 confidence, the agent is overclaiming. An honest read would land closer to 0.55 and let escalation logic route it to a human.

Feature request vs bug (#31)

```
1 ✗ #   31  feature-request  (0.60)  Cannot parse JSON with trailing comma in
  ↪ strict mode
```

The fixture `dup-original-trailing-comma` title: "Cannot parse JSON with trailing comma in strict mode." The body: "Input `{\"a\": 1,}` fails validation. Standard JSON forbids trailing comma. Need opt-in lenient JSON mode."

The agent labeled `feature-request` at 0.60. The human labeled `bug`.

The agent latched onto "Need opt-in lenient JSON mode" – that is a request for new behavior. The human labeled `bug` because the report starts from a concrete failure: strict parsing rejects input the reporter's API already sends. Same underlying issue as #32 (`duplicate`), which the agent got right at 0.92.

Here the agent is closer to the right confidence – 0.60 signals uncertainty – but the label disagreement is structural, not ambiguous. Your enum forces

every issue into one of five categories; "fix the parser" and "add a lenient mode flag" are different maintainer actions. A human triager might file this as `bug` first and open a design discussion for lenient JSON as a follow-up.

What both teach you

This is the classification problem no system prompt will fully solve. Some issues are genuinely ambiguous (#30). Others sit on a boundary between categories that your contract treats as mutually exclusive (#31). The agent's job is not to eliminate that tension – it is to classify consistently and use confidence to flag when a human should look.

That is what the eval suite in chapter 8 is for. You will build a golden dataset, which marks both fixtures as `gray_zone` or duplicate pairs – and measure whether the agent's confidence is calibrated. Not "does it get the right label" but "does it know when it does not know."

For now, note the disagreements. On #30 the agent may have the wrong label but the wrong confidence is the bigger problem. On #31 the label fight is real and the 0.60 score is doing its job. Those are different failures with different fixes.

What you have, and what you do not

The four files total about 80 lines of Python. What they give you:

A working classification agent that validates its output against a typed contract, runs against a real repository's issues, and tells you exactly what it costs to operate.

What they do not give you, yet: tests, tools, observability, or a way to run outside your laptop. That is intentional.

Ship it

The agent classifies issues. On this first full run it agreed with human labels on 48 of 50 fixtures – the two □ lines above. That number will move when you change the model, tune the system prompt, or point at a noisier repo. Even two disagreements are enough to see where the agent overclaims or picks the

wrong category. You do not need a high error rate before chapter 3; you need real failures you can name.

Open chapter 3. Ship it.

Chapter 03 - Shipping the Bad Version

The agent classifies issues. It is not good enough to leave running unsupervised. Ship it anyway.

This is not recklessness. It is the same instinct that drives good API development: deploy early to a real environment, observe real behavior, fix the specific things that break rather than the hypothetical things that might. The agent you ship at the end of this chapter is the agent every subsequent chapter improves. Without this deploy, the rest of the book has no target.

By the end of this chapter, the agent is live on **your fork of `triage-lab`**, posting comments on real issues. Every new issue gets a comment showing the classification, confidence score, and rationale. It is embarrassing. That is the point.

What the deploy looks like

No GitHub App. No webhook endpoint. No container. No secrets manager. Those come back in chapter 12.

What you have: a Python script, a GitHub personal access token in a `.env` file, a `.last_run` timestamp file that tracks which issues have already been seen, and a crontab entry that runs the script every five minutes. Total new infrastructure: none. Total new dependencies: none beyond what chapter 2 already installed.

This deploy is not production-grade by any metric. It does not handle GitHub rate limits. Retries on transient Anthropic errors (429, 502, 503, 504, 529) come from the same helper `classify.py` uses — not a production retry policy with jitter, circuit breaking, or alerting. It has no monitoring beyond a log file in `/tmp`. It has no kill switch. If you change the system prompt and the agent starts misfiring, the only way to stop it is `crontab -r`.

crontab -r deletes everything

`crontab -r` removes your entire crontab — not just the triage entry. If you have other cron jobs, use `crontab -e` instead and delete only the triage line manually.

All of that is acceptable. The goal of this deploy is not reliability. It is reality: the agent runs against real issues from your real repository, and you can see exactly how it behaves before you invest in making it production-grade. You cannot observe a system that does not exist.

Setup

Runnable code for this chapter lives in `chapter-03/` at the repo root. That folder is the **full chapter 2 project** plus two deploy files. You should see everything from chapter 2:

```
chapter-03/
  models.py          # output contract (chapter 2)
  agent.py           # classifier agent (chapter 2)
  fetch_issues.py    # closed-issue fetcher (chapter 2)
  classify.py       # batch eval against triage-lab closed fixtures (chapter 2)
  comment.py         # NEW — format and post triage comments
  poll.py            # NEW — cron-friendly polling loop
```

If you built chapter 2 in `chapter-02/`, copy those four files into `chapter-03/` before adding `comment.py` and `poll.py`. The repo's `chapter-03/` already contains the complete tree.

Seed your fork

First, starting by forking the book's repository.

GitHub forks copy code, not issues. Before `poll.py` can run, populate your fork:

```
cd triage-lab
uv sync
export GITHUB_REPO=you/triage-lab
export GITHUB_TOKEN=ghp_...    # Issues: Read and write
python scripts/seed_issues.py
```

This creates the 50 closed fixtures from `./issues/issues.json`. Then point the agent code at your fork:

```
cd chapter-03
uv sync
cp .env.example .env   # fill in keys — see below
```

`ANTHROPIC_API_KEY` is the same key from chapter 2. `uv run python classify.py` still works.

Set `GITHUB_REPO=you/triage-lab` (your seeded fork) for deploy. `GITHUB_TOKEN` was optional in chapter 2; it is **required** for `poll.py` because this chapter posts comments.

A classic PAT needs the **`public_repo`** scope (or **`repo`** for private repos). A fine-grained token needs **Issues: Read and write** on that repository.

Never point `poll.py` at a repo you do not control.

Posting a comment

Add `comment.py`:

```
# comment.py
import httpx
from models import IssueClassification


COMMENT_TEMPLATE = """\
**[Maintainer Bot]** Automated triage:

- Classification: `{label}`
- Confidence: {confidence:.0%}
- Rationale: {rationale}

*If this classification is wrong, relabel the issue and I will learn.*\
"""
```

```
15
16
17  def format_comment(classification: IssueClassification) -> str:
18      return COMMENT_TEMPLATE.format(
19          label=classification.label,
20          confidence=classification.confidence,
21          rationale=classification.rationale,
22      )
23
24
25  def post_comment(repo: str, issue_number: int, body: str, token: str) -> None:
26      response = httpx.post(
27          f"https://api.github.com/repos/{repo}/issues/{issue_number}/comments",
28          headers={
29              "Accept": "application/vnd.github+json",
30              "Authorization": f"Bearer {token}",
31          },
32          json={"body": body},
33          timeout=10.0,
34      )
35      response.raise_for_status()
```

The comment shows the confidence as a percentage, not a float. `0.88` reads as noise to a maintainer; `88%` reads as a signal. The rationale is the sentence from `IssueClassification.rationale` – the same field you put in the model in chapter 2 for exactly this reason.

The "if this classification is wrong, relabel the issue" line is not decoration. Chapter 11 builds the escalation logic that reads the issue's labels after a human has corrected them. Put the instruction in the comment now so the feedback loop has somewhere to close.

The polling loop

Add `poll.py`:

```
1   # poll.py
2   import asyncio
3   import os
4   from datetime import datetime, timezone
5   from pathlib import Path
6
7   import httpx
8   from dotenv import load_dotenv
9
10  load_dotenv()
11
12  from classify import run_agent_with_retry
13  from comment import format_comment, post_comment
14
15  REPO = os.getenv("GITHUB_REPO", "nunombispo/triage-lab")
16  TOKEN = os.environ["GITHUB_TOKEN"]
17  LAST_RUN_FILE = Path(".last_run")
18
19
20  def parse_github_ts(ts: str) -> datetime:
21      if ts.endswith("Z"):
22          ts = ts[:-1] + "+00:00"
23      return datetime.fromisoformat(ts)
24
25
26  def read_since() -> str:
27      if LAST_RUN_FILE.exists():
28          return LAST_RUN_FILE.read_text().strip()
29      return datetime.now(timezone.utc).replace(
30          hour=0, minute=0, second=0, microsecond=0
31      ).isoformat()
32
33
34  def write_since(ts: str) -> None:
35      LAST_RUN_FILE.write_text(ts)
36
37
38  def fetch_new_issues(since: str) -> list[dict]:
39      response = httpx.get(
40          f"https://api.github.com/repos/{REPO}/issues",
41          params={"state": "open", "since": since, "per_page": 100},
42          headers={
43              "Accept": "application/vnd.github+json",
44              "Authorization": f"Bearer {TOKEN}",
45          },
46          timeout=10.0,
47      )
48      response.raise_for_status()
49      since_dt = parse_github_ts(since)
50      return [
```

```
51            i
52            for i in response.json()
53            if "pull_request" not in i
54            and parse_github_ts(i["created_at"]) > since_dt
55        ]
56
57
58    async def process_issue(issue: dict) -> None:
59        text = f"Title: {issue['title']}\n\nBody:\n{(issue['body'] or '')[:1500]}"
60        result = await run_agent_with_retry(text)
61        comment_body = format_comment(result.output)
62        post_comment(REPO, issue["number"], comment_body, TOKEN)
63        print(f"  #{issue['number']} → {result.output.label}
      ↪   ({result.output.confidence:.0%})")
64
65
66    async def main() -> None:
67        since = read_since()
68        now = datetime.now(timezone.utc).isoformat()
69        issues = fetch_new_issues(since)
70        print(f"Found {len(issues)} new issue(s) since {since}")
71        for issue in issues:
72            await process_issue(issue)
73        write_since(now)
74
75
76    if __name__ == "__main__":
77        asyncio.run(main())
```

`.last_run` is a single-line file containing the ISO timestamp of the last successful run. On the first run, it defaults to midnight UTC today – issues opened earlier today are skipped. On every subsequent run, it advances to the moment the run **started**, not the moment it finished, so an issue that arrives during a slow run cannot fall between two polling windows.

GitHub's `since` query parameter filters by **last updated**, not created. That is a problem here: posting a comment updates the issue, so the next poll would fetch it again. The list call still passes `since` to keep the response small, but the client-side filter keeps only issues where `created_at` is after the watermark. Old issues that someone relabels will not get a second comment.

One failure mode remains: if the script crashes after processing three issues but before writing `.last_run`, those three issues get processed again on the next run and the agent posts duplicate comments. For chapter 3, that is acceptable – you are the only user of your own repo and you can delete a duplicate comment. Chapter 12 replaces this entirely with an idempotent

webhook handler that cannot double-process.

The crontab

Find the path to your virtual environment's Python:

```
cd chapter-03
which python   # inside the activated venv
# /home/you/projects/ai-agents-pydantic-ai/chapter-03/.venv/bin/python
```

Open your crontab:

```
crontab -e
```

Add one line:

```
*/5 * * * * cd /path/to/ai-agents-pydantic-ai/chapter-03 &&
↪  /path/to/ai-agents-pydantic-ai/chapter-03/.venv/bin/python poll.py >>
↪  /tmp/triage.log 2>&1
```

The `cd` is required – `poll.py` reads `.last_run` relative to the current directory and loads `.env` from the current directory. Without it, both will be missing and the script will fail on every run – errors land only in `/tmp/triage.log`, not in your terminal.

Verify it is running:

```
tail -f /tmp/triage.log
```

After five minutes, you should see either `Found 0 new issue(s)` or the first classification. Open a **new issue** on your fork and watch the log.

```
Found 1 new issue(s) since 2026-05-22T00:00:00+00:00
  #51 → bug (95%)
```

And on the issue:

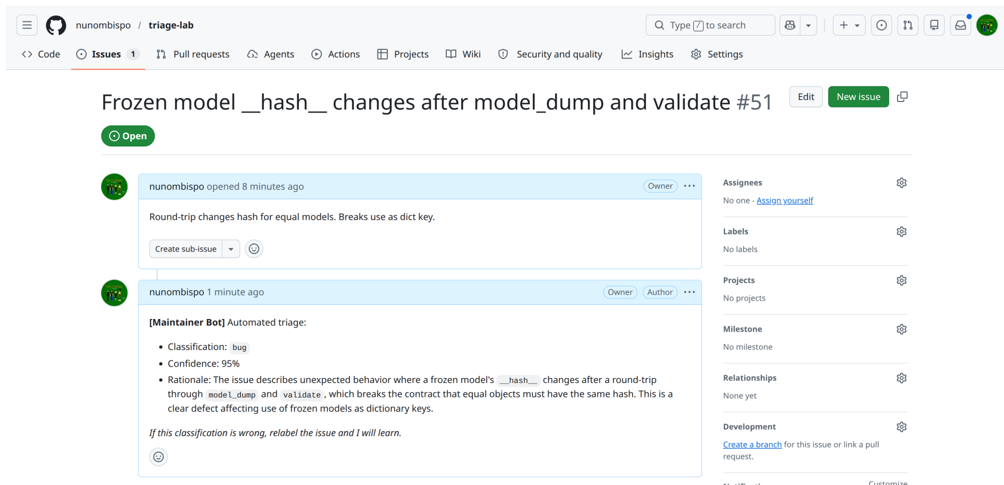


Figure 1. Classified Issue

Test on your fork first

Run against `you/triage-lab` – a repo you control with seeded fixtures – before pointing the bot at a production repo you maintain. Open a real issue, watch the agent comment, read the comment as a maintainer. Once that works, optionally repoint `GITHUB_REPO` at a repo you actually maintain. That is when the stakes become real.

The first comment

When the agent comments on a real issue for the first time, something shifts. The classification numbers from chapter 2 become a specific comment on a specific issue that a real person could read. The 0.88 confidence score is no longer an output in a terminal; it is a claim the agent is making in public.

This is the shift the book is trying to induce. An agent that runs on your laptop is a script. An agent that posts to a real repository is a system with stakeholders. The stakeholder in chapter 3 is you. From chapter 4 onward, the question is: how do you make this system reliable enough that the stakeholder is someone else?

Why minimal beats elaborate

The instinct to wait – to add the GitHub App auth, the container, production-grade retries and monitoring before shipping – is the instinct that produces agents that never leave a notebook.

Every production system you have shipped started as something worse than what it became. The difference between a shipped system and a perfect design is that the shipped system accumulates real feedback. The prompt that looked correct in chapter 2 will misclassify an open issue on your fork that the closed fixtures did not cover. The confidence threshold that seemed reasonable will turn out to be wrong in a specific, observable way. The comment template will need adjusting once you have seen a dozen of them in context.

None of that feedback is available until the system is live.

Minimal-and-live also reframes every subsequent chapter. Chapter 4 is not "add tools because tools are a feature." Chapter 4 is "the agent you shipped in chapter 3 cannot look up whether a similar issue exists, and that is why it keeps misclassifying duplicates." Every improvement has a specific motivation rooted in a failure you have already observed, on your own repository, with your own issues. The book has a target.

The elaborate-but-local alternative has no target. It has a design.

The embarrassing inventory

This is a full accounting of what the v0 agent cannot do. Read it as a table of contents for the rest of the book.

1. **No tools.** The agent sees only the title and body. It cannot look up whether a similar issue was filed last month, whether the label taxonomy your repo uses matches the five categories in the system prompt, or whether the reporter has context in a linked PR. **Chapter 4.**
2. **No retrieval.** Finding genuinely similar past issues requires semantic search over the issue history, not keyword matching. The agent has no access to that history. **Chapter 5.**
3. **No versioned output contract.** The comment format will need to change. There is currently no migration plan for the issues that were commented on before the change. **Chapter 6.**

4. **No unit tests.** The only way to verify that the system prompt change you made this morning did not break the classification logic is to run the agent against real issues. CI does not exist. **Chapter 7.**
5. **No eval suite.** The agreement rate from your chapter 2 `classify.py` run is not an SLO. It is one measurement against `triage-lab` fixtures — not yet a CI gate on your fork. There is no way to know if it is getting better or worse over time. **Chapter 8.**
6. **No observability.** The only record of what the agent did is `/tmp/triage.log`, which rotates on restart and contains no structured data. There is no way to answer "why did the agent label that issue `spam` last Tuesday?" **Chapter 9.**
7. **No cost accounting.** The agent spends money on every run. There is no per-issue cost metric, no budget, and no alert if something causes average token usage to triple. **Chapter 10.**
8. **No escalation path.** The agent comments on every issue, including the ones it should flag for human review. A confidence score of 0.51 and a confidence score of 0.97 produce identical behavior. **Chapter 11.**
9. **No canary.** The next system prompt change ships to 100% of your incoming issues immediately. There is no gradual rollout, no way to compare the new prompt against the old one on real traffic, and no automatic rollback. **Chapter 12.**
10. **No runbook.** When the agent starts misfiring at 2am — and it will — there is no documented procedure for stopping it, diagnosing it, or reverting it. There is only `crontab -r` and the hope that the damage is limited. **Chapter 13.**

Ten items. Ten chapters. Every one of them is a failure mode the preface named. Every one of them has a fix that applies the engineering discipline you already have for APIs. The agent you just shipped is the worst version of itself.

That is the whole point. Open chapter 4.